Linköping Studies in Science and Tecnology
Dissertation No. 530

Declarative Debugging for Lazy Functional Languages

Henrik Nilsson

Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden

Linköping 1998

Abstract. Lazy functional languages are declarative and allow the programmer to write programs where operational issues such as the evaluation order are left implicit. It is desirable to maintain a declarative view also during debugging so as to avoid burdening the programmer with operational details, for example concerning the actual evaluation order which tends to be difficult to follow. Conventional debugging techniques focus on the operational behaviour of a program and thus do not constitute a suitable foundation for a general-purpose debugger for lazy functional languages. Yet, the only readily available, general-purpose debugging tools for this class of languages are simple, operational tracers.

This thesis presents a technique for debugging lazy functional programs declaratively and an efficient implementation of a declarative debugger for a large subset of Haskell. As far as we know, this is the first implementation of such a debugger which is sufficiently efficient to be useful in practice. Our approach is to construct a declarative trace which hides the operational details, and then use this as the input to a declarative (in our case algorithmic) debugger.

The main contributions of this thesis are:

- A basis for declarative debugging of lazy functional programs is developed in the form of a trace which hides operational details. We call this kind of trace the Evaluation Dependence Tree (EDT).
- We show how to construct EDTs efficiently in the context of implementations of lazy functional languages based on graph reduction. Our implementation shows that the time penalty for tracing is modest, and that the space cost can be kept below a user definable limit by storing one portion of the EDT at a time.
- Techniques for reducing the size of the EDT are developed based on declaring modules to be trusted and designating certain functions as starting-points for tracing.
- We show how to support source-level debugging within our framework. A large subset of Haskell is handled, including list comprehensions.
- Language implementations are discussed from a debugging perspective, in particular what kind of support a debugger needs from the compiler and the run-time system.
- We present a working reference implementation consisting of a compiler for a large subset of Haskell and an algorithmic debugger. The compiler generates fairly good code, also when a program is compiled for debugging, and the resource consumption during debugging is modest. The system thus demonstrates the feasibility of our approach.

To my parents

Acknowledgements

First of all, I would like to thank my supervisor Prof. Peter Fritzson who initiated this work and who has shown a great deal of patience and flexibility as to letting me pursue my interests. I am also indebted to him for thorough proof-reading of this document at various stages of incompleteness and many constructive comments. Remaining errors are, of course, mine alone. I would also like to thank the two other members of my advisory group, Prof. Mariam Kamkar and Prof. Jan Małuszyński, who have contributed with their expertise and experience during my years as a PhD student.

Then I would like to thank my other friends and colleagues at PELAB, past and present, for being around and creating a friendly but, at times, also challenging atmosphere, where no issue is deemed too insignificant for a heated debate. In particular I would like to thank Mikael Pettersson, Lars Viklund, Niclas Andersson, Patrik Nordling, Patrik Hägglund, and Tommy Persson. Tommy Persson has also provided prompt technical support, and Gunilla Norbäck has sorted out administrative mysteries.

A very special thank you to the people in the Multi Group, who made me feel at home during my stay at Chalmers during the late autumn 1995 and early winter 1996, and who I have had the pleasure to meet on numerous other occasions. It was a most fruitful time, and the coffee was, indeed, very good. I would especially like to mention Jan Sparud, my colleague and co-author; Thomas Johnsson, for many interesting discussions and suggestions; Lennart Augustsson, for HBC and for answering innumerable technical questions regarding HBC; Magnus Carlsson, for a most useful set of parsing combinators; Thomas Hallgren; and Urban Boquist.

Special thanks also to Prof. Nahid Shahmehri, who believed it was possible, and, finally, to my family, without whom it would not have been possible.

Thank you!

Henrik Nilsson
Linköping, April 1998

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

Contents

Chapter 1

Introduction

This thesis contributes to the field of debugging lazy functional programs. Currently there are no good, general purpose debuggers readily available for lazy functional languages. Since this type of language is declarative, we argue that lazy functional programs should be debugged declaratively, and we present a technique for doing so along with an efficient implementation. As far as we know, this is the first implementation of a declarative debugger for a lazy functional language which is sufficiently efficient to be useful in practice. The emphasis is on Haskell-like languages and implementations based on graph reduction.

1.1 Motivation and objectives

The distinguishing feature of declarative programming languages is that they allow the programmer to leave the exact evaluation order in a program unspecified, at least to some extent. This is an advantage for the programmer, who can concentrate on the logic of the problem at hand rather than on a detailed description of exactly how to solve it. The result is often very succinct yet clear programs. This aspect of declarative languages can also be an advantage from an implementation perspective since the evaluation order may be chosen more freely. This can be exploited to gain efficiency; one oft-cited possibility is parallel execution, for instance.

Declarative languages also have implications as regards programming errors and debugging. On the one hand, programmers using a declarative language often find that they make fewer programming mistakes than when using conventional languages. This is partly a consequence of the declarative aspects: there are typically fewer lines of code to write and less details to be concerned with, so there are fewer opportunities for making mistakes and it is easier to get an overview of the code one is writing. The other part of the explanation lies in that declarative languages invariably have automatic memory management (garbage collection) and quite often also sophisticated type systems. The former eliminates a whole class of common bugs (which often are difficult to find), whereas the latter tend to catch simple mistakes early. (Of course, automatic memory management and good type systems are not unique to declarative languages.)

On the other hand, declarative languages do certainly not eliminate the possibility of making programming errors, so there is still a need to debug code when it turns out that it does not behave as intended. This can be problematic. The reason is that conventional debugging techniques are based on observing execution events as they occur. Consider tracing, for example, or a conventional debugger for C or Pascal supporting breakpoints, single stepping and inspection of the procedure call stack. Such techniques are not really useful unless the the programmer fully understands what is going on operationally, which is exactly what a declarative programmer usually prefers not to be too concerned with. At any rate, requiring a declarative programmer, expert and novice alike, to fully understand all the subtleties of some particular evaluation scheme just to be able to debug a program, when such details supposedly were of subordinate importance, casts a shadow of doubt over the purported virtues of declarative languages. It ought to be possible to debug at the conceptual level at which a program is written.

One class of declarative languages are the lazy functional ones, with Haskell [HPJW$^+$92] currently being the most important example. They are characterized by demand driven evaluation: a function argument is evaluated only when its value is needed to compute part of the output. Moreover, no argument is evaluated more than once. (This is why they are known as 'lazy'.) Functional programming languages, including the lazy ones, are today in many respects a mature technology. In a recent article, Wadler [Wad98] presents a selection of six Real-World applications of functional programming, ranging from industrial-strength theorem provers to an expert system in use at the Orly and Charles de Gaulle airports in Paris which generates invoices and explanations of the services used for all flights.

However, Wadler also points out that there is one respect in which these languages are not so mature: there are still few good debuggers for functional languages. This is true in particular for the lazy functional languages. To this authors knowledge, the only readily available debugging tools which are sufficiently mature for practical use, are either low-level operational tracers (such as the tracing facilities offered by HBC [Aug93a]), or specialized tools with a limited scope (for example, Hazan's and Morgan's tool for finding the call path which led to a run-time error [HM93], or Sparud's stream programming debugger [Spa96]).

Why are there no good, general purpose, 'lazy debuggers' available? Lazy debugging is difficult for precisely the reasons discussed above: even though the evaluation principle is simple, programmers nevertheless find the ensuing evaluation order 'surprising' [Mor82, Toy87, OH88]. Because of this, conventional debugging techniques such as operational tracing are only of limited use. As a more suitable alternative, a number of researchers have proposed that some form of trace reflecting the *logical* structure of the computation should be constructed, thus allowing debugging to take place at an appropriate level (see chapter 11). However, what were proposed were in most cases *complete* traces. Such traces require storage in proportion to the size of the computation. The result is severe performance problems as soon as realistic programs are being traced.

Thus, conventional debugging techniques are cheap, but not really suitable for lazy languages. On the other hand, general debugging techniques appropriate for lazy languages seemed too expensive. The objective of this thesis is to show that the latter need not be the case. Specifically, we develop techniques which allow lazy functional programs to be debugged declaratively, and we show how to implement a debugger based on these techniques efficiently. Moreover, we do this for a modern, functional language, essentially a large subset of Haskell. We call our compiler *Freja*[1] and we will also refer to the Haskell subset it currently implements as Freja when the distinction is important. Appendix A gives the details.

1.2 Approach

Our philosophy regarding lazy functional debugging is embodied by the following three tenets:

(i) Lazy functional programs should be debugged declaratively.

(ii) Debugging should not change the semantics of the program being debugged.

(iii) Performance is vital.

The motivation for the first tenet was discussed in the previous section. The point is that programs should be written and debugged at the same conceptual level. Now, perhaps we should not refer to this as a 'tenet' since we concede that there are exceptions (e.g. when it comes to performance debugging or programs written in an operational style), but we do believe this is a good basic approach.

The second tenet may seem superfluous, but there have been proposals for lazy debuggers where debugging can cause evaluation of expressions

[1] Freja is a goddess in Nordic mythology.

which normally would have remained unevaluated. This risks turning a terminating computation into a non-terminating one, thus changing the semantics of the the program being debugged, the *target*. That debugger-induced non-termination is undesirable should be obvious.

The importance of the third tenet should also be clear, at least if the aim is a debugger which is useful in practice. Our goal is that it should be possible to debug *any* program using only a fixed amount of extra storage. Furthermore, the amount needed should be reasonable in comparison with the size of the primary memory of a typical computer. We are willing to pay for this by increased computation time, as long as the response times during debugging does not get unduly large.

Several declarative debugging techniques which could be used in a lazy context have been suggested, e.g. Shapiro's algorithmic debugging [Sha82], or Ducassé's trace analysis techniques [Duc92]. Both types of debuggers perform debugging on an execution record, or trace, of the target program. However, the usefulness of the debugger is very much dependent on the structure of this record. A straightforward construction of a lazy execution record results in a structure where operational aspects feature prominently. This makes declarative debugging difficult [NF92].

Our solution to this problem is to construct the execution record in such a way that the demand driven aspect of lazy functional languages is effectively hidden (while respecting the second tenet). We call this particular kind of execution record *Evaluation Dependence Tree* (EDT). Once the execution has finished (or the user aborts it), the EDT is used as input to an algorithmic debugger. From the user's perspective, the effect of hiding the lazy evaluation order is that the evaluation appears to have been carried out eagerly, except in cases where the lazy evaluation strategy successfully avoided some computation. Since eager evaluation reflects the syntactic structure of the source code quite well, this makes it relatively easy to debug the target declaratively in accordance with the first tenet.

As to the third tenet, our approach is to construct the EDT *piecemeal*, one part at a time, on demand, by re-executing the target program. The user can specify the bounds on the size of the stored EDT portion which allows him or her to strike a convenient balance between execution time and memory consumption.

1.3 Contributions

The main contributions of this thesis are:

- A basis for declarative debugging of lazy functional programs is developed in the form of a trace which hides operational details. We call this kind of trace Evaluation Dependence Tree (EDT).

- We show how to construct EDTs efficiently in the context of implementations of lazy functional languages based on graph reduction. Our implementation shows that the time penalty for tracing is modest (typically within a factor of 2 to 4), and that the space cost can be kept below a user definable limit by storing one portion of the EDT at a time.
- Techniques for reducing the size of the EDT are developed based on declaring modules to be trusted and designating certain functions as starting-points for tracing.
- We show how to support source-level debugging within our framework. A large subset of Haskell is handled, including list comprehensions.
- Language implementations are discussed from a debugging perspective, in particular what kind of support a debugger needs from the compiler and the run-time system.
- We present a working reference implementation consisting of a compiler for a large subset of Haskell and an algorithmic debugger. The compiler generates fairly good code, also when a program is compiled for debugging, and the resource consumption during debugging is modest. The system thus demonstrates the feasibility of our approach.

1.4 Structure of the thesis

The thesis is organized as follows.

Chapter 2 gives a brief introduction to functional programming, lambda calculus and implementation of lazy functional languages. The chapter is included to make the thesis reasonably self contained. Any reader familiar with these topics can skip the chapter without loss of continuity.

Chapter 3 starts with a general discussion on declarative languages and debugging where we argue that declarative programming calls for declarative debugging. We also introduce a version of algorithmic debugging which is the declarative debugging technique used in this thesis. We then turn to the problem of debugging lazy functional programs, explain why conventional debugging techniques in our view are inadequate for this purpose, and identify the main problems which a successful lazy debugging technique must address.

Chapter 4 lays the foundation for our debugging method by introducing the EDT and defining what it is. A general, high-level, definition is given, as well as a formal one for a specific, small, lazy functional language. The latter definition is given in the form of an EDT-generating denotational semantics for the language.

Chapter 5 deals with EDT construction in practice. We show how to construct EDTs efficiently in the context of language implementations based on compiled graph reduction. We also show how to limit the storage requirements by building one portion of the EDT at a time on demand.

Chapter 6 presents a number of improvements of the basic EDT construction algorithm. In particular, the chapter gives two methods for reducing the number of nodes in an EDT which can be used when the user has some idea about where the bug might be located. The chapter also explains how to handle non-terminating programs.

Chapter 7 addresses source-level debugging. Two common language features are considered: local functions and list comprehensions. The chapter shows how to translate these constructs and annotate the resulting code with debugging information in a way which makes source-level debugging possible.

Chapter 8 discusses user interfaces and presents the one which is provided as part of our system. The chapter also contains a large example demonstrating how our debugger works. The chapter has been written to be reasonably self-contained, so the reader who quickly would like to get a feeling for what it is like to use the debugger, could skip directly to this chapter.

Chapter 9 discusses the implementation of our system and in what respects there is special support for debugging. To some extent it collects information spread throughout the thesis.

Chapter 10 evaluates the performance of our system.

Chapter 11 is a survey of related work.

Chapter 12 sums up and discusses future work.

1.5 Relation to our earlier work

We first presented an algorithmic debugger for lazy functional languages based on an EDT-like structure in Nilsson & Fritzson [NF92]. Even though that implementation was very inefficient, it was, to our knowledge, the first working algorithmic debugger for a lazy functional language.[2] The basic ideas for a realistic implementation were then outlined in Nilsson & Fritzson [NF93, NF94]. The EDT in its current incarnation was first presented in Sparud & Nilsson [SN95]. A significantly reworked version of this article later appeared as Nilsson & Sparud [NS97], where a more detailed description of the EDT structure was given and the current method for EDT construction outlined. Nilsson & Sparud [NS96] presented the EDT construction method in greater depth. The method for handling list comprehensions was first given in Nilsson [Nil94]. This thesis includes parts from several of these publications and reports, but in most cases the material has been substantially revised and extended.

[2] Others had suggested using algorithmic debugging both for strict [Sha82] and non-strict [OH88] functional languages earlier.

Chapter 2

Preliminaries

This chapter gives a brief technical background to the rest of the thesis, covering the areas of functional programming, lambda calculus, and implementation of lazy functional languages. A short introduction to the syntax and features of Haskell is also given. The aim is to make the thesis reasonably self contained, not to treat any of the subjects in depth. For a comprehensive survey of the functional programming field, the reader is referred to Hudak [Hud89]. The books by Bird & Wadler [BW88] and Thompson [Tho96] are good introductions to lazy functional programming and, in the latter case, Haskell. The books by Field & Harrison [FH88] and Peyton Jones [PJ87] cover implementation of functional languages and also give adequate introductions to lambda calculus.

2.1 Functional programming

Over the years, numerous functional or nearly functional languages have emerged, for example ML [GMW79, Mil84], Hope [BMS80], Lazy ML (LML) [Aug84], Miranda [Tur85], and Haskell [HPJW+92, PHA+97]. These and other languages have all contributed to the set of features that are now thought to characterize a modern functional language. The most important of these are:

- First-class functions.
- Static, strong, polymorphic type system.
- Pattern matching.
- Algebraic types.
- Automatic memory management (garbage collection).

Taken together, these features yield languages of remarkable power and expressiveness.

Of the languages mentioned above, the last three are lazy whereas ML and Hope are strict. (Different evaluation orders and laziness are discussed in sections 2.3.3 and 2.4.1.) Haskell is the most recent and currently the most important of the lazy functional languages. This thesis is concerned with lazy functional languages of the kind typified by Haskell. Our compiler, which is called Freja, currently implements a large subset of Haskell (see appendix A). The implemented Haskell subset will also be referred to as Freja when the distinction is important.

While lazy functional languages have many attractive features, how to implement them efficiently remained an open question for quite a long time. This spurred a number of attempts to build specialized hardware [Veg84, PJ89, Szy91]. However, in their break-through G-machine papers, Augustsson [Aug84] and Johnsson [Joh84] showed that lazy functional languages could be implemented with good performance 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 (see section 2.4.4). Since then the techniques have been refined (see [PJ92] and [PJL91] for instance), and today the performance of state-of-the art implementations is typically within a factor of 3 or so of C [H^+96, Wad98].

A functional program consists of a set of function-defining equations. Conceptually, all computations take place by applying functions to arguments and rewriting the applications using the equations in a directed manner as rewrite rules. There is no implicit state being manipulated by executing a sequence of commands. There are no updatable variables and no side-effects. In principle, the only way to bind a variable to a value is by means of function application, even though there in practice may be other syntactic binding constructs for the convenience of the user or for efficiency reasons.

Since there are no side-effects, functions in a functional language become functions in the mathematical sense: 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 remain unchanged. This property is called *referential transparency*, and makes it comparatively easy to formally reason about and manipulate functional programs. 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. As a consequence of referential transparency, functional languages are also deterministic.

Some functional languages actually have imperative, updatable variables, and it would perhaps be more appropriate to label them 'almost functional'. ML belongs to this category, for instance. Functional languages which really are referentially transparent are therefore sometimes called *pure* to emphasize this fact. This thesis is only concerned with pure languages since lazy functional languages generally are pure (imperative variables and laziness do not go very well together). Referential transparency also makes it a lot easier to apply declarative debugging techniques, even though it is possible to debug declaratively in the presence of side-effects [Sha91].

2.2 A short introduction to Haskell

The following points on Haskell's syntax and features might be helpful for readers who are not familiar with Haskell or some similar functional language. It covers most of the constructs that are used in this thesis. A thorough introduction to Haskell may be found in Hudak *et al.* [HFP96] or in Thompson [Tho96]. A definition of Haskell version 1.4 can be found in Peterson *et al.* [PHA$^+$97]. (See also appendix A.)

Function application is denoted by juxtaposition, so `f (1+1) 2` means the function `f` applied to the arguments `(1+1)` and `2`. Function application has higher precedence than infix operator application, which is why the first argument to `f` is enclosed in parentheses. Also, application is left associative, so `f (1+1) 2` is really interpreted as `(f (1+1)) 2`. Conceptually, `f` is first applied to the argument `(1+1)` which results in a new function (expecting one argument less than `f`). This function is then applied to `2`. Handling application of functions of more than one argument in this manner is known as *currying*. An application of a function to too few arguments is known as a *partial application*.

Function definition follows the juxtaposition pattern, so the function `f` above might be defined as `f x y = x * y`. In general, a function may be defined by a series of equations, where *patterns* and *guards* are employed to decide which equation applies when the function is applied to some specific arguments.

As an example, here is a definition of the factorial function:

```
fac 0         = 1
fac n | n > 0 = n * fac (n - 1)
```

The first equation states that $0! = 1$. The pattern `0` constrains the equation to be applicable only when `fac` is applied to 0. The second equation states that $n! = n \cdot (n-1)!$. The pattern `n` is a variable pattern. A variable matches anything and is then bound to the matched argument. In this case, however, the guard `n > 0` constrains the equation to be applicable only when the argument is greater than 0. Should the patterns happen to be overlapping so that more than one equation apply, the semantics of Haskell states that the first (textually) of these equations should be picked.

Local definitions may be introduced by means of `where`-clauses or `let`-expressions. For example:

```
foo x y = let a = 2 * x in fie a
    where
        fie z = z * y
```

Given this definition, the value of `foo 3 4` is 24.

Haskell has a module system. Each definition belongs to some module. There is one distinguished module called `Main` in which a variable `main` should be defined. This variable is bound to the expression which is evaluated and printed in order to execute the program. Haskell has a monadic I/O-system [PHA+97], so the value of `main` is actually a value in the I/O-monad. This value can be understood as a computation which should be carried out. However, Freja does currently not support monadic I/O, so in this thesis we will usually assume that `main` is bound to some expression yielding a value which can be printed directly (such as a list of integers or a string).

So called *lambda-abstractions* are used to introduce functions without first giving them names. In Haskell, '\' is used to denote λ. Lambda abstractions have the general form `\`x `->` *exp*, where x is the formal argument and *exp* is the body of the function. For example, `\x -> 2*x` is a function that when applied to a number yields that number multiplied by two, and the expression `(\x -> x*x) 3` evaluates to 9.

Tuples are written enclosed in parentheses and lists enclosed in square brackets. Thus `(1,'a',3)` is a three-tuple and `[1,2,3]` is a list of three elements. The latter is just syntactic sugar for `1:2:3:[]`, where `:` is the (right associative) list construction operator (pronounced 'cons') and `[]` is the empty list. Note that the elements of a tuple can be of mixed types, whereas the elements in a list must be of a single type.

Pattern matching also works for tuples and lists. The first of the two functions below extracts the first component of a pair (two-tuple) whereas the second computes the length of a list.

```
fst (a,_) = a

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

Note the use of the *wild-card* pattern `_` in the definition of `fst`. The wild-card is like a variable pattern in that it matches anything, but unlike a variable it will not be bound to the matched argument which thus cannot be referred to in the body of the function. In the definition of `length`, the pattern `[]` matches only the empty list, whereas the pattern `(x:xs)` matches any non-empty list, binding `x` to the first element of that list (the *head* of the list) and `xs` to the remainder of the list (its *tail*). Since `x` is not used in the right-hand side, it could equally well be replaced with a wild-card.

Sometimes it is convenient to name a pattern for use in the right-hand side. This can be achieved using an *as-pattern*. The following function duplicates the first element in a non-empty list:

```
f (x:xs) = x:x:xs
```

Using an as-pattern, it could be defined as follows:

```
f s@(x:_) = x:s
```

Haskell has a *polymorphic* type system (see Cardelli & Wegner [CW85] for instance). Since the functions `fst` and `length` defined above do not make any assumptions regarding the types of the elements in the pair and list, respectively, they are polymorphic, meaning that `fst` can be applied to pairs of elements of any types, and that `length` can be applied to lists of elements of any type.

Type declarations are introduced by `::`. Function types are written using `->`. Assuming that the function `g` expects two integers and returns a character, the type of `g` is written `Int -> Int -> Char` and the fact that `g` has this type is expressed as `g :: Int -> Int -> Char`. The type constructor `->` is right associative, matching the left associative function application. Thus the type of `g` is really `Int -> (Int -> Char)`, meaning that when `g` is applied to an integer we get a function from integer to character back. The types for tuples and lists have a special syntax that is reminiscent of values of that type, e.g. the type of the tuple `(1,'a',3)` is `(Int,Char,Int)` and the type of the list `[1,2,3]` is `[Int]`. Type declarations are optional. Types are inferred automatically if omitted.

Implicitly universally quantified type variables are used to specify polymorphic types. Thus `fst :: (a,b) -> a` means that for all types `a` and `b`, `fst` is a function that maps a pair of elements of types `a` and `b` to values of the type `a`. Further, `length :: [a] -> Int` means that for all types `a`, `length` is a function that maps lists of elements of type `a` to integers.

New types are created by `data` declarations. There is also a facility for defining type synonyms. Here are some examples:

```
data Colour = Red | Green | Blue
type Point = (Float,Float)
data Object = Rectangle Point Point | Circle Point Float
data NewList a = Null | Cons a (NewList a)
```

`Colour` is a simple enumeration type with three values. `Point` is a type synonym, i.e. just a shorthand notation for a tuple of two floating point numbers. `Point` is then used in definition of the data type `Object`. An `Object` is either a `Rectangle` consisting of two points (the coordinates of opposite corners) or a `Circle` consisting of a point and a floating point number (the origin and the radius). The final example illustrates a *recursive* type definition, isomorphic to the built-in list type. Thus, a `NewList` with elements of type `a` is either `Null` (the empty list) or a `Cons`-cell consisting of an element of the type `a` (the head of the list) and a `NewList` with elements of type `a` (the tail of the list). The kind of types defined by the `data`-declarations above are known as *algebraic types* since they are formed as sums (discriminated unions) of product types (tuples).

`Red`, `Green`, `Blue`, `Rectangle`, `Circle`, `Null` and `Cons` are called *constructors* and work as functions (or constants) for constructing values of the corresponding type. Thus we have

```
Red :: Colour
Cons :: a -> NewList a -> NewList a
```

for instance.

Constructors also work as tags, distinguishing various kinds of objects from one another within a type, and can be used to take objects apart by pattern matching. For example, a function to compute the area of an `Object` and a function to compute the length of a `NewList` may be defined as below:

```
area (Rectangle (x1,y1) (x2,y2)) = abs ((x2-x1)*(y2-y1))
area (Circle _ r)                = pi * r * r

newLength Null        = 0
newLength (Cons _ xs) = 1 + newLength xs
```

2.3 The lambda calculus

The functional languages rest on solid mathematical foundations. At the very core is the lambda calculus, which is not only important for giving a well-defined meaning to functional programs, but also plays an important role for implementation. We will therefore give a short account of the basics of lambda calculus here. For a comprehensive treatment, see the book by Hindley and Seldin for instance [HS86].

2.3.1 Basics

The lambda calculus was invented by the logician Alonzo Church in the 1930's. Originally developed to give operational semantics to mathematical functions, it turned out to be an ideal vehicle for both description and implementation of functional languages. What makes the lambda calculus

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

Figure 2.1: Syntax of the lambda calculus with constants.

so attractive, is that it is a very simple language, yet sufficiently powerful to express any computable function; that is, any program that can be written in any programming language can 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 it for granted that they can be expressed in the basic language. The interested reader is referred to the literature for details.

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*, and a lambda expression which can be reduced in this way is known as a reducible expression, or *redex* for short. An expression without any redexes is said to be on *normal form*.

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 the outer abstraction's 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`.

Note that an argument could contain free variables in which case there is a risk of *name capture*. For example, consider:

```
(λx.(λy.(λx.+ x y)) x)
```

The expression contains just a single redex. If this is β-reduced according to the rules above, we get

```
(λx.(λx.+ x x))
```

which is clearly wrong. To avoid such problems, it is sometimes necessary to rename variables. We omit the details.

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 principle 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

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

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 2.4.3).

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.:

$$(\lambda\mathtt{x.+\ x\ 1}) \overset{\alpha}{\Leftrightarrow} (\lambda\mathtt{y.+\ y\ 1})$$

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

$$(\lambda\mathtt{x.F\ x}) \overset{\eta}{\Rightarrow} \mathtt{F}$$

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*.

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 order

Consider the following lambda expression:

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

Here, `if` is a function of three arguments (a new constant). If the first evaluates to `True` (another new constant), the expression as a whole will reduce to the second argument, otherwise it will reduce to the third argument. Furthermore, we assume that `=` is a function which reduces to `True` if its two arguments are equal, and that `*` and `/` denote integer multiplication and integer division respectively.

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`. 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.

It can be proved that if there is some reduction order that terminates (i.e. succeeds in reaching a normal form) 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:

Theorem 2.1 (Church-Rosser Theorem I) *If $E_1 \Leftrightarrow E_2$, then there exists an expression E such that $E_1 \Rightarrow E$ and $E_2 \Rightarrow E$.* □

Theorem 2.2 (Church-Rosser Theorem II) *If $E_1 \Rightarrow E_2$, and E_2 is in normal form, then there exists a normal order reduction sequence from E_1 to E_2.* □

Normal-order reduction states that the outermost, leftmost redex should be reduced first. The effect is *call-by-need* or *non-strict* semantics. In our case, this means that `(/ 2 0)` never would be evaluated. The outer redex is the application of `if`, which is thus reduced first resulting in the third argument being discarded before it is evaluated. (To see the details, we would have to replace the involved constants with the corresponding lambda expressions.) *Lazy evaluation* is an implementation technique for implementing languages with non-strict semantics efficiently. It is discussed further in section 2.4.1.

Innermost (leftmost) reduction order is called *applicative-order* 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). Note that the example above would lead to failure if eager evaluation were used. Normal-order reduction terminates whenever applicative-order does, but not vice versa.

The term *strict* deserves a brief explanation. In order to reason about computations, it is useful to introduce a symbol denoting failure. Often the symbol $\perp$ ('bottom') is used. It is also known as the *improper value*. In this way, it is possible to formally assign a value to computations which fail to terminate or are undefined for some other reason (division by 0, perhaps).

Returning to the term strict, a function which has the property that it will fail if the computation of its argument fails is said to be strict in its first (and in this case only) argument. This property can be expressed concisely using $\perp$ as

$$\texttt{f}\ \perp = \perp$$

If a function is strict in all its arguments, it is simply referred to as strict. Under an eager reduction strategy, all functions become strict since arguments are evaluated before a function is called. That is why this kind of languages are said to have strict semantics. Under a normal-order reduction scheme, functions are not necessarily strict in all arguments. Hence non-strict semantics.

2.3.4 Recursion

Since the lambda calculus is sufficiently powerful to express any computable function, it must somehow be possible to express recursive computations. But at first sight, there is no sign of any recursive constructs in the basic calculus. However, the effect of recursion can be achieved by repeated 'code duplication'. This is accomplished by the famous `Y`-combinator, which is defined as

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

`Y` can be used to solve fixed point equations of the type

```
x = f x
```

since `x = Y f` is a solution to the equation, i.e.

```
Y f = f (Y f).
```

Thus `Y` is useful for defining recursive functions 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 provided the free variable `fac` is bound to the factorial function (i.e. to the very function we are trying to define):

```
F = (λn.if (= n 0) 1 (* n (fac (- n 1))))
```

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 since the factorial function is a fixed point of `(λfac.F)`.

2.4 Implementation of functional languages

This section covers the basics of lazy functional language implementation. The reader is referred to the standard book by Peyton Jones [PJ87] for details. The focus is on graph reduction and the G-machine since this is what the current Freja implementation is based on and thus the context in which our debugging techniques have been developed. However, notice that graph reduction in some form is the dominating implementation technique for lazy functional languages.

2.4.1 Laziness and graph reduction

As we saw in section 2.3.3, normal-order reduction has the interesting property that it yields the best possible termination properties. This is attractive from a programming perspective since it makes a host of useful programming techniques available, and often allows programs to be written at a more declarative level without having to pay too much attention to the evaluation order.

However, normal-order reduction is not always the most efficient reduction order in terms of the number of reductions performed. In fact, it quite often leads to duplicated work. For example, suppose that we have `f x = x * x` (which is just another way of writing $\mathtt{f} = (\lambda\mathtt{x}.\mathtt{*}\ \mathtt{x}\ \mathtt{x})$) and consider the following normal-order reduction sequence:

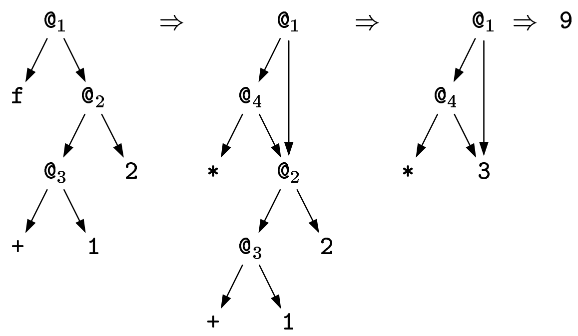
$$\mathtt{f\ (1+2)} \Rightarrow \mathtt{(1+2)\ *\ (1+2)} \Rightarrow \mathtt{3\ *\ (1+2)} \Rightarrow \mathtt{3\ *\ 3} \Rightarrow \mathtt{9}$$

Note that the argument `(1+2)` was evaluated twice. A *lazy* evaluation strategy stipulates that function arguments should not be evaluated unless needed, as for normal-order reduction. But in addition, it also requires that an argument should be evaluated at most once. This is crucial for efficient implementation, and languages with non-strict semantics are thus usually lazy in this sense.

Laziness is usually implemented through graph reduction. In an implementation of a lazy language based on graph reduction, expressions and data are represented in a uniform manner as pieces of graph (stored on the heap). As a matter of fact, the entire program is represented as one graph which is successively rewritten (or reduced or evaluated), on demand, until a graph representing the final answer is obtained. A key point is that whenever a redex (reducible expression), is evaluated, that redex is physically overwritten with the result. Thus no expression is evaluated more than once.

The uniform representation is convenient from an implementation point of view. For example, no special actions need to be taken when delaying the evaluation of a argument to a function: an argument is just a piece of graph, and it does not matter whether it represents some expression or some data. On the other hand, whenever a function argument is used, it must be checked that the argument is not a redex. If it is, it must be evaluated before it is used. Representing expressions as pieces of graph also permits sharing of subexpressions and handling of recursive definitions by means of circular graphs. The former is important for efficiency reasons, since it means that there is no need to duplicate subexpressions. Compare the normal-order reduction example above, where, as we saw, normal-order reduction led to performing the same computation several times. The latter is a more efficient and practical way to handle recursion than using the combinator `Y`.

The following example shows how duplication of work when reducing `f (1+2)` is avoided by using graph reduction. For this kind of graph, it is conventional to represent an application by `@`, with the graph representing the function down to the left and the graph representing the argument down to the right. Here, the application nodes have also been numbered to show which are the same. Using this convention we get the following reduction sequence. Note how the argument `(1+2)` becomes a shared subexpression. Since it is physically overwritten when evaluated, it is only evaluated once.



When evaluating an expression, how far should the evaluation be taken? The usual approach is to evaluate an expression until there is no outer redex because then there is something which can be printed or inspected by a pattern-matching construct. Then, if necessary, the computation can continue by evaluating subparts of the expression equally far, and so on. Thus the computation becomes demand driven. The notion of an expression with no outer redex is made precise by the following definition:

Definition 2.1 (Weak head normal form) An expression E is said to be in *weak head normal form* (WHNF) if:

(i) E is a constant,

(ii) E is an expression of the form $(\lambda\mathtt{x}.E')$ for *any* E',

(iii) E is of the form $f\ E_1\ E_2\ \ldots\ E_n$ for any constant function f of arity $k > n$. □

2.4.2 Template instantiation and compilation

When we described how to evaluate an application of a lambda abstraction in section 2.3.2, 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 run-time. This leads to poor performance.

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 run-time. This is the basic idea behind compilation of functional languages.

Free variables, however, pose an additional problem. Consider the following lambda abstraction:

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

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* inner lambda abstraction, for which no instantiation code could have been compiled in advance. 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 run-time. Hence a fixed code sequence can be compiled since the code is parameterized with respect to its free variables. 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. 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. The lambda abstraction as a whole has no free variables. This leads to *supercombinator reduction*, which is the topic of the next section.

2.4.3 Supercombinators and lambda-lifting

Let us first define those lambda abstractions that are amenable to multiple-argument application as outlined in the previous section:

Definition 2.2 (Supercombinator) A *supercombinator* S of arity n is a lambda expression of the form

$$\lambda x_1.\lambda x_2.\cdots\lambda x_n.E$$

where E is not a lambda abstraction, such that

(i) S has no free variables,

(ii) any lambda abstraction in E is a supercombinator,

(iii) $N \geq 0$. □

If reduction only takes place once all arguments are available, and if all arguments then are substituted into the supercombinator body simultaneously, 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. 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 but distinct) names to the supercombinators that are created by the lambda-lifting algorithm. This also permits direct implementation of recursion (rather than going via `Y`) since lambda abstractions then can be referred by name.

We 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 combinator interpreter from each program. This is the basis of the G-machine, as described in the next section.

2.4.4 The G-machine

The G-machine [Aug84, Aug87, Joh84, Joh87b] is an abstract machine 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,

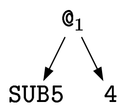
(iv) D is the dump, a stack on which S and C are stored to preserve them across 'subroutine' calls.

The G-machine and its instruction set, the G-code, can be formally defined by specifying the state transition for each of its instructions. Well shall not go into details here, but see below for a G-code example.

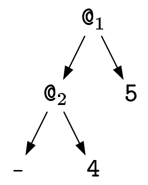
As explained in section 2.4.2, executing a compiled code sequence should build an instance of the corresponding supercombinator body. Suppose that we have the following supercombinator:

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

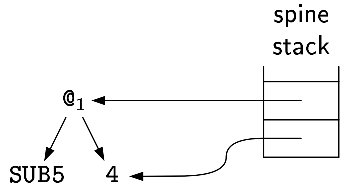
Suppose further that the application `SUB5 4`, which corresponds to the graph



should be reduced. Then the following graph should be built:



Again, the application nodes have been numbered for purpose of identification. Note thus that the root of the redex and the root of the result are the same physical node ($@_1$), i.e. the root of the redex has been overwritten with the root of the result (which happened to be an application node as well). 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, the spine stack would thus look as follows when the code sequence for `SUB5` is invoked:



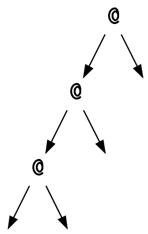
The G-code sequence in figure 2.2 constructs the desired graph. The instruction `FUNSTART` marks the beginning of a code sequence for a supercombinator. The arguments are the name of the supercombinator and its arity. Conceptually, it does not affect the state of the G-machine, but in a concrete implementation it may carry out menial tasks such as checking for stack overflow or zapping redexes (see section 2.4.6). The numbers after `PUSH` and `UPDATE` are offsets from the stack top. The number after `POP` is the number of elements to remove. The contents of the stack is shown in the rightmost column with the top of the stack to the left. In practice, as indicated above, the stack only contains references to the built structures. The structures themselves are part of the graph. R indicates the reference to the redex root.

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 spine stack. If this is a data object (e.g. an integer) then evaluation is complete (WHNF has been reached). 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 now follows the left branch of the application node, pushing further application nodes onto the spine stack as they are found. This is known as unwinding the spine, hence spine stack. A chain of application nodes looks a bit like a spine. The application

`FUNSTART SUB5 1`	Start of code for `SUB5`	[4, R]
`PUSHINT 5`	Push the integer 5	[5, 4, R]
`PUSH 1`	Push copy of argument	[4, 5, 4, R]
`PUSHFUN -`	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	[`@ (@ - 4) 5`]
`UNWIND`	Continue execution	

Figure 2.2: G-code for `SUB5 x = - x 5`. The rightmost column shows the effect of the instructions on the spine stack. Here it is assumed that `SUB5 4` is being reduced. R indicates the reference to the redex root.

nodes themselves are called the vertebrae and the arguments are known as the ribs:



When a supercombinator is encountered, the G-machine checks if enough arguments are available. If not, evaluation is complete (the expression is in WHNF) 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 the books by Peyton Jones [PJ87] or Field & Harrison [FH88].

2.4.5 Optimizations

One important aspect of the G-machine is that it opens up a wide spectrum of optimization possibilities. For example, it is wasteful to construct a redex on the heap if it is known that the next step is to unwind it onto the spine stack and enter the code for the applied supercombinator. It would be much more efficient to put the arguments on the stack directly and call (or, in case of a tail call, jump) to the code for the supercombinator in question. It might be even better to inline the code, e.g. in case of arithmetic operations.

Good quality implementations perform this type of optimizations. Even our Freja compiler tries to inline calls to known functions (e.g. arithmetic operations) when applicable. However, in the discussions of our debugging techniques, we will usually assume that unoptimized graph reduction as outlined in sections 2.4.1 to 2.4.4 is being performed. This is justified by noting that, in the worst case, it is always possible to revert to unoptimized graph reduction for debugging purposes.[1] On the other hand, as long as the calls are not interesting for debugging (e.g. arithmetic operations), they may be optimized in any way the compiler is capable of. Moreover, it should not be a major problem to develop optimized calling conventions also for debuggable functions, but we have not yet done this.

2.4.6 Zapping and black holes

Some implementations overwrite the redex root with a special 'zap node' as soon as reduction of that redex has begun. Once the reduction is complete, the redex, and hence the zap node, will be overwritten with the result. This makes it possible to detect so called *black holes*: self-dependent expressions. For example, `x = x + 1` is a black hole. Unless detected statically by the compiler, a zapping implementation would zap `x` as soon as the code for constructing its body (`x + 1`) was invoked. When executing this code, the reference to `x` would be encountered and an evaluation of `x` thus initiated. However, it would immediately be discovered that the node has been zapped, i.e. that it currently is being evaluated. Thus the evaluation must be self-dependent, a black hole, and the execution can be terminated with a suitable error message. Had zapping not been employed, the implementation would have entered an infinite loop instead in this case.

As we will see, zapping is also useful for debugging purposes. Whenever an execution terminates prematurely, because of a run-time error or the user interrupting it, redexes that were being evaluated when the execution was aborted remain zapped. This is useful for debugging since redexes being reduced when some error was encountered correspond to the undefined value, $\perp$. Thanks to zapping $\perp$ has an explicit representation on the heap.

[1] Note that it not always the case that optimizations can be turned off without changing the semantics of the program being debugged. The semantics of the language might be loose, for instance, so that a program could have more than one meaning [Cop94]. However, we assume that the semantics of the languages with which we are concerned are sufficiently well specified so that correctly implemented optimizations never change the meaning of a program.

Chapter 3

Lazy Functional Debugging

In this chapter, we first discuss declarative programming languages and the consequences declarative programming has for debugging. We argue that declarative programs are best debugged by declarative means, and introduce one such technique: algorithmic debugging. Finally we examine the particular debugging problems posed by lazy functional languages.

3.1 Declarative languages and debugging

3.1.1 Declarative languages

Lloyd [Llo94] defines declarative languages as languages where the program can be considered to be a theory and computation is deduction from this theory. This definition is quite general and includes both logic languages such as (pure) Prolog, where a program is a finite set of formulas in first order logic, and (pure) functional languages such as Haskell, where a program is a finite set of function defining equations which are given a formal meaning through λ-calculus. In the case of Prolog, computation is performed by drawing conclusions from the predicates using SLD-resolution; in the case of functional languages, computation can be regarded as proving some initial term equal to the final result by using the equations in a directional manner to rewrite (or reduce) the former into the latter.

In more popular terms, declarative programming is often explained as focusing on *what* to compute rather than *how* to compute it. In a famous paper advocating the virtues of declarative programming, Kowalski [Kow79] expressed this division between what and how by the equation

$$Algorithm = Logic + Control$$

where *Logic* refers to the knowledge to be used in solving problems, and

	a	b	c
1	c3+c2		
2	a3*b2	2	a2+b2
3	7		a2+a3

s

$\Longrightarrow$

	a	b	c
1	37		
2	14	2	16
3	7		21

r

Figure 3.1: Evaluation of a simple spreadsheet.

Control refers to the problem-solving strategies employed to make use of that knowledge. In this sense of the words, declarative programming means stating the logic component of a program but leaving the control component unspecified. This in turn implies that it is the responsibility of the language implementation to perform the deduction, i.e. to decide on how to solve the specified problem by supplying the missing control part which dictates the order in which the necessary computations are to be carried out so as to arrive at an answer. This is necessary since a computer needs very precise instructions regarding what to do and in which order to to it.

The possibility of leaving the control component unspecified is what makes declarative programming attractive since it allows the programmer to write concise and clear programs without having to worry too much about operational issues. Not specifying control also has implications for code reuse since the same piece of code often can be used in many ways. This is a well known property of logic languages, where a variable in a clause head operationally can serve either as an in-parameter or as an out-parameter depending on usage, but it is also true for other declarative languages. For instance, in a lazy functional program it is often possible to exploit the fact that data structures are constructed lazily on demand to write more reusable code [Hug89].

The following example should make this discussion more concrete. It shows how a simple spreadsheet evaluator can be specified declaratively in Haskell. Figure 3.1 illustrates the task. The spreadsheet itself is represented as an array `s` of cells, where each cell may contain an expression or be empty. In order to compute the result array `r` the expressions in `s` must be evaluated, but since the expressions contain references to the values of other expressions, this must be done in an order determined by the dependences between the expressions.

In a strict language, the evaluator would have to specify a suitable order explicitly. Such an order could be found by first analysing the dependences between the expressions. Alternatively, an order which is always suitable, such as performing all computations iteratively until the results stabilize (or an upper bound on the number of iterations is exceeded), could be used.

In a lazy functional language, the evaluator can be specified declaratively by simply stating the the relationship between `s` and `r`, e.g. as follows:

```
r = array (bounds s) [ ((i,j), eval r (s!(i,j)))
                       | (i,j) <- indices s ]
```

Without going into details, the code above says that `r` is an array with the same dimensions as `s` (the `(bounds s)` bit), where the value at index `(i,j)` is given by evaluating the expression at the same index in `s`. Evaluation of a single expression is done using the auxiliary function `eval` which takes `r` as an environment where the values for the references may be looked up. Since, under a lazy evaluation scheme, nothing is computed unless needed, the necessary computations will be carried out in dependence order yielding a fully evaluated `r`, as long as there are no circular dependences.[1]

Note that the above example, given a slightly different problem formulation, could be seen as a declarative specification of an algorithm employing *dynamic programming* [AHU83]: a number of of computations are to be performed in an order dictated by data dependences, and as the results become available, they are stored in an array to avoid having to re-compute them should they be needed more than once. However, in the dynamic programming case, it is normally known that there are no circular dependences which makes the above technique much more practical. For another example, see section 8.3.1 which shows how an evaluator for an attribute grammar can be implemented declaratively in a lazy functional language [Joh87a].

Unfortunately, the implicitness as regards the control component is also the main weakness of declarative languages. The evaluation strategy built into the language implementation must be quite general, and it could therefore be the case that the algorithm that results when the declarative program is combined with the built in evaluation strategy is rather inefficient. Furthermore, the programmer might actually wish to express some clever algorithm or to convey some information about how a program is going to be used with the aim of making the program more efficient.

Now, in the words of Lloyd, declarative programming languages come in two flavours: *strongly declarative* and *weakly declarative*. Strongly declarative languages do not require the programmer to convey any control information at all, whereas weakly declarative languages need some control information. Languages of practical interest tend to be weakly declarative thus making it possible to perform computations in a reasonably efficient manner. Typically, weakly declarative programs, in addition to their declarative reading, also have a procedural reading, which is how control information is conveyed. There might even be means whereby a programmer explicitly can

[1] If there are circular dependences, the implementation would either report that a *black hole* (see section 2.4.6) had been encountered, or the program would loop.

supply additional control information (e.g. through annotations) to further enhance performance.

Syntactical limitations also play a role here. In a functional language, for instance, the programmer is constrained to write equations which can be used as rewrite rules. This makes it possible to use fairly efficient evaluation strategies, but on the other hand the programmer is required to express himself in a way that rather strongly suggests an algorithm to be used.[2]

3.1.2 Debugging declarative programs

Either way, leaving control information implicit, all of it or just a part, leads to problems when it comes to debugging. Debugging has traditionally been based on monitoring execution events as they occur. Consider inserting `print`-statements into the source code of the target program (the program being debugged) for example. Or take a typical debugger for imperative languages like C or Pascal, where debugging amounts to setting break-points at strategic locations, single stepping through parts of the code, inspection of the call stack, and monitoring the values of important variables as the execution proceeds. To be useful, such debugging techniques require a good operational understanding of the target program. This is usually not a problem for imperative programs[3] since the operational behaviour mirrors the structure of the source code.

In a declarative setting, on the other hand, it is often the case that the programmer does *not* have a detailed operational understanding of the target simply because he did not specify the evaluation order and may not care much about it, indeed has *chosen* not to care. It could be that the programmer is not even *aware* of all details regarding the built in evaluation strategy or the full implications of it. Being exposed to all the operational details during debugging is then usually not very helpful.

As regards lazy functional languages, this problem has been observed many times over the years [HO85, TR86, OH88]. Morris [Mor82] eloquently puts it as follows:

[2] In the case of (pure) strict functional languages, the evaluation strategy is trivial (call-by-value or eager evaluation), so an algorithm *is* actually given.

[3] An exception is debugging heavily optimized code which may have been subjected to procedure integration (inlining), loop unrolling, constant folding, dead-store elimination, and so on. A common approach is to turn off the optimizations when compiling for debugging. However, since even *correctly implemented* optimizations could change the meaning of the target, turning off optimizations is not always a viable option. There are two reasons for this. First, the semantics of the language might be loose so that there are several possible interpretations of a program. Second, a correctness-preserving optimization does not necessarily preserve the meaning of an incorrect program. And then there is always the possibility of an incorrect compiler. Thus there is a wide body of work on debugging optimized imperative code [Cop94].

> How does one debug a program with a surprising evaluation order? Our attempts to debug programs submitted to the lazy implementation have been quite entertaining. The only thing in our experience to resemble it was debugging a multi-programming system, but in this case virtually every parameter to a procedure represents a new process. It was difficult to predict when something was going to happen; the best strategy seems to be to print out well-defined intermediate results, clearly labelled.

We therefore argue that when writing declarative programs, one is for debugging purposes best served by a declarative debugger since coding and debugging then takes place at the same conceptual level, much as one is normally better off using a source-level debugger than an assembly-level debugger when debugging programs written in conventional high-level languages such as Pascal, C, or C++.

3.1.3 Errors in declarative programs

Declarative programmers often find that they make fewer programming mistakes than when using conventional languages. This is partly a consequence of the declarative aspects: there are typically fewer lines of code to write and less details to be concerned with, so there are fewer opportunities for making mistakes and it is easier to get an overview of the code one is writing.

The other part of the explanation lies in that declarative languages invariably have automatic memory management and quite often also sophisticated type systems.[4] The former eliminates a whole class of common bugs (which often are difficult to find), whereas the latter tend to catch simple mistakes early. In addition, a strong type system encourages the programmer to think about the structure of the problem at hand before starting to implement a program which solves it. This is probably also beneficial as regards the number of mistakes made.

Nevertheless, mistakes are made also when programming declaratively. As always, this results in programs which output erroneous results, terminate abnormally or get stuck in infinite loops. A good debugger is then a valuable and time-saving aid.

Another class of problems concerns performance: the target program runs unexpectedly slow or needs much more memory than one would think. This is sometimes referred to as *performance bugs*. These problems are inherently related to the operational behaviour of a program, and because of this probably more common in a declarative setting than in an imperative one since the operational aspects are less obvious in the former case.

Because performance bugs have nothing to do with the declarative meaning of a program, they are outside the scope of this thesis. Performance

[4] Of course, these advantages are also available in many other languages.

debugging of lazy functional programs has been addressed by other researchers, and excellent tools for time and heap profiling are readily available [RW93, SPJ95, RR96].

3.1.4 Algorithmic debugging

As we saw in section 3.1.1, computation in a declarative language can be seen as drawing conclusions from a given set of facts and rules, the program. Should the conclusions be wrong, then there is something wrong in the program. By by keeping track of the dependences between drawn conclusions, the *proof tree*, and by classifying the drawn conclusions as either true or false, it is possible to deduce where in the program the bug is located, and it may even be possible to infer what a correct program should look like. This is called declarative debugging.

We will now introduce a version of algorithmic debugging [Sha82], which is a semi-automatic, declarative debugging technique, originally developed for Prolog. Under algorithmic debugging the debugger keeps track of the proof tree. By using the proof tree and information regarding the validity of the conclusions, which may be obtained by interactively asking the programmer or by referring to some kind of formal specification, the debugger infers where in the program the bug is located. This lets the programmer focus entirely on declarative issues and disregard the operational deduction process. Thus there is a good match between the conceptual levels at which programming and debugging take place.

The basis for algorithmic debugging is the fact that if a conclusion is wrong, then there is either something wrong with the rule that was used to draw this particular conclusion, or there is something wrong with (at least) one earlier conclusion on which the current one depends. It is thus easy to state an algorithm (hence the name) for finding the erroneous rule. One possibility is to start from the root of the proof tree: presumably there is something wrong with it, there is a *visible bug symptom*, or debugging would not have been initiated in the first place. Then do the following:

(i) For each child of the current node in the proof tree, i.e. for each conclusion on which the current one depends, check whether it is valid or not. If not, make it the current conclusion and restart from (i).

(ii) If there are no children, or if all children are valid conclusions , then there is something wrong with the rule used to draw the current conclusion.

It should be pointed out that the aims of Shapiro to some extent were different from ours. For instance, he saw the method more as a basis for mechanization than as debugging tool in itself. The idea was that a (partial)

```
insert x []                   = [x]
insert x (y:ys) | y > x       = y : (insert x ys)
                | x < y       = x : y : ys
                | otherwise = y : ys

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

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

Figure 3.2: Erroneous insertion sort program.

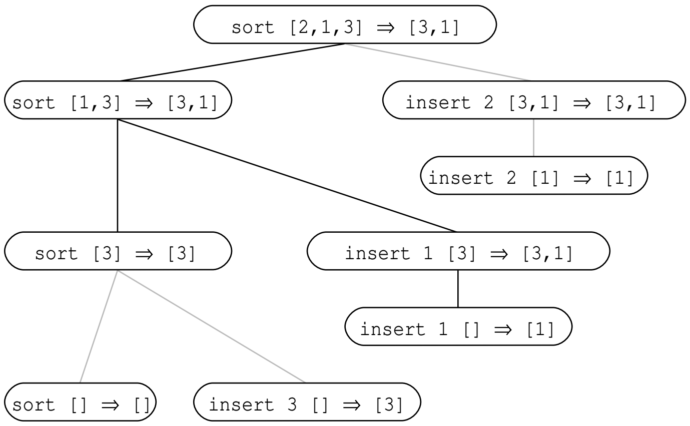
specification of the program to be debugged would be available (perhaps in the form of an unoptimized version of the program, or as a database of queries and answers from earlier debugging sessions), and that most questions could be answered by referring to the specification. Thus the number of questions asked were no major concern, and a bottom-up query order was used which meant that no extensive tracing was needed to keep track of the proof tree. Another suggestion was automatic program correction. While many of Shapiro's suggestions could work also in our setting, we mainly regard algorithmic debugging as a systematic and convenient way of navigating through the proof tree.

As an illustration of algorithmic debugging in a functional setting, consider the erroneous insertion sort program in figure 3.2 for sorting lists of integers. The bug is in the function `insert`: the arguments to `(>)` are the wrong way round.

Figure 3.3 illustrates how this program is debugged algorithmically. First, the program is run and the proof tree, as shown in the figure, is constructed. Then debugging starts at the root of the tree. By answering no the the first question, the user indicates that the conclusion of the root node, i.e. that `sort [2,1,3]` can be rewritten to `[3,1]` is wrong. The debugger then proceeds by asking about the children of the root node. Since there is something wrong already with the first child, this node becomes the current one.

The debugger now continues by asking about the children of the new current node. According to the user, the conclusion that `sort [3]` can be rewritten to `[3]` is correct, so the sibling to the right is investigated next. This conclusion happens to be wrong, but the only conclusion on which it depends, that `insert 3 []` yields `[3]`, is correct. The bug must therefore be in the rules that were used to draw the erroneous conclusion that `insert 1 [3]` can be rewritten to `[3,1]`, i.e. the bug must be in the function `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




```
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 3.3: Proof tree and debugging session for the insertion sort program. Explored paths are black, unexplored ones grey.

the 'full picture' for each question, i.e. there must be no auxiliary conditions that the user is not aware of, nor any implicit side effects of the deduction step. This of course applies in a declarative setting, or it would be difficult to regard computation as deduction in the first place.

3.2 The lazy functional debugging problem

This section identifies the key problems lazy functional languages pose for debugging. We also argue that declarative debugging is only a partial solution: the *structure* of the proof tree must also be appropriate.

3.2.1 Why conventional debugging techniques are unsuitable

While the demand driven execution of a lazy functional language has many advantages, it also has a number of drawbacks. One of them is that it makes it difficult to use conventional debuggers and debugging techniques for debugging lazy functional programs. As discussed in section 3.1.2, the fundamental problem is that conventional debugging techniques are based on observation of execution events as they occur. This is usually not very helpful when debugging lazy functional programs since the order of events appear somewhat random and since values in general are partially evaluated expressions which might be very large and difficult to understand.

We will illustrate this by two examples. First, consider the following functional program, where `foo` is a function that clearly must not be applied to the empty list:

```
foo xs | head xs < 0 = 0
       | otherwise   = hd xs

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

main = fie [-1]
```

The problem with this program 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]`.

For the sake of comparison, let us suppose that we have strict semantics. The execution of the above program can then be illustrated by the execution tree shown in figure 3.4. The nodes in this tree correspond to reduction steps, and the parent-child relationship denotes that the parent caused the child reduction step. In operational terms, each node records a function call and its result, and the edges show who made the call.

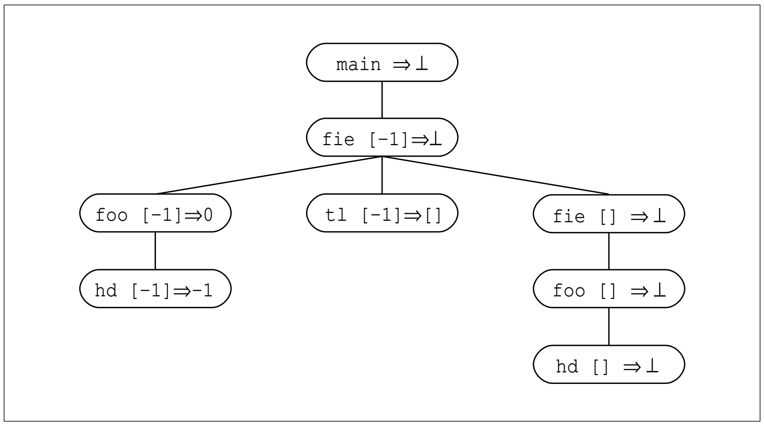


Figure 3.4: Strict execution tree.

There are few surprises here. The structure of the tree reflects the structure of the source code in a fairly obvious way. Arguments and results are simple values. In the rightmost branch, we see how the application of `hd` to the empty list provoked a run-time error, represented by the symbol $\perp$ ('bottom'). In order to find out what caused the run-time error, we simply have to follow the edges from the leaf node towards the root: moving up two levels we find that `fie` has applied `foo` to the empty list, `[]`. In practice this is easily achieved by inspecting the run-time *call* stack.

Figure 3.5 shows the tree that results in the case of a lazy language. The interpretation of nodes and edges is as above, though it should be emphasized that function calls now happen on demand, and that the function that demands the result of a function application in general is not the same function as in which the function application syntactically occurs.

As above, executing the program results in a run-time error. But here it is much more difficult to get 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 run-time 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 function arguments and results 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. If a user is faced with such expressions during debugging, the work of

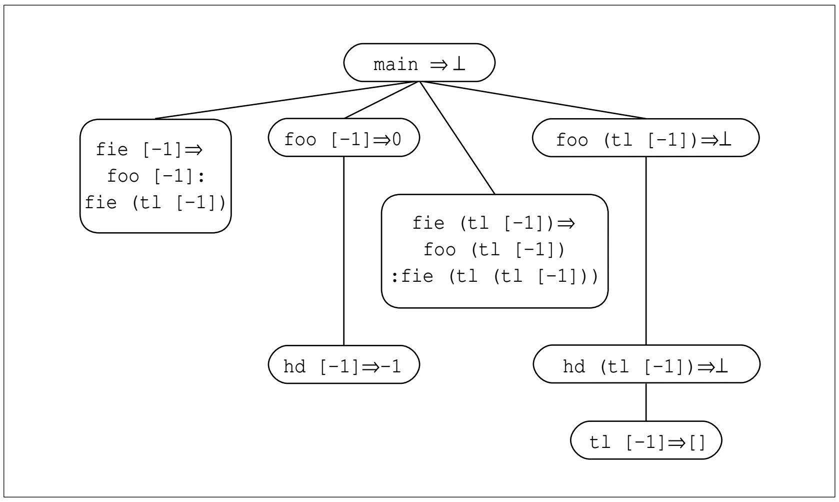


Figure 3.5: Lazy execution tree.

interpreting them could make the debugging process considerably harder.

The tracing facility of the HBC compiler [Aug93a] provides another illustration of the problem of applying conventional debugging techniques to lazy functional languages. This mechanism is built around a special function `trace` that takes two arguments: a string and an argument of an arbitrary type.[5] The semantics is that the second argument of the function is returned as the result of the application, but only after the string has been printed as a side effect. Thus it is possible to debug programs in a way that resembles the debugging technique of inserting print-statements in imperative programs.

Unfortunately, the order of events might be such that the output becomes very difficult to interpret. An even more serious problem is that the only way to output values of interesting variables, is to convert them to string form and embed them in the string argument to the trace function. This will of course force the evaluation of the values of these variables which in turn could change the termination properties of the program being debugged. Furthermore, the act of forcing these expressions could cause `trace` to be invoked recursively which would result in very confusing output.

The following example, borrowed from Sparud [Spa96], illustrates how tracing might change a terminating program into a non-terminating one. In the program below, calls to `trace` have been inserted into the functions `f` and `g` to make it possible to verify that they are invoked.

[5]There is also a low-level tracer which allows the graph reduction process to be scrutinized.

```
f xs = trace "Entering f" (head xs + 1 : g (tail xs))
g xs = trace "Entering g" (head xs - 1 : f (tail xs))
main = take 4 (f [1..])
```

While `f` is applied to the infinite list of positive integers, `[1..]`, the fact that only the first four elements of the resulting infinite list are used ensures that the program will terminate. Running it results in the following output:

```
Enter trace(0):
Entering f
Exit  trace(0)
[2Enter trace(0):
Entering g
Exit  trace(0)
, 1Enter trace(0):
Entering f
Exit  trace(0)
, 4Entering trace(0):
Entering g
Exit  trace(0)
, 3]
```

In order to show the arguments to `f` and `g` in the output, the program would have to be changed as follows. The function `show` converts a value of a printable type to a string:

```
f xs = trace ("Entering f, xs = " ++ show xs)
             (head xs + 1 : g (tail xs))
g xs = trace ("Entering g, xs = " ++ show xs)
             (head xs - 1 : f (tail xs))
main = take 4 (f [1..])
```

When the new version of the program is executed, the following output ensues:

```
Enter trace(0):
Entering f, xs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, ^C
```

That is, the printing of an argument has turned a terminating program into a non-terminating one and the execution must be aborted manually. Note that any debugging technique that forces evaluation of potentially unevaluated expressions will face similar problems.

3.2.2 Declarative debugging in a lazy functional context

The problems discussed in the previous section are partly due to conventional debugging techniques focusing on the order of operational execution events and on an implicit assumption that the event order is closely related to the syntactic structure of the source code. If this assumption does not hold, the interaction with the debugger could be confusing. By using a declarative debugging technique, such as algorithmic debugging, where order is not important, it should be possible to overcome these problems.

While this in principle is true, it is also the case that the structure of the proof tree, i.e. the execution record, which is the input to a declarative debugger, to a large extent determines the character of the resulting debugging process. If the proof tree really is rather operational, the debugging will take place at an operational level as well, unless significant measures are taken to abstract from the operational execution record within the debugger.

As a case in point, suppose that the execution record in figure 3.5 is used for algorithmic debugging. A question that then would be asked is whether

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

is a correct reduction or not, i.e. whether it should be possible to prove the term on the left of the arrow equal to the term on the right using the equations given in the program.

While this is a declarative question, it is too closely related to the operational aspects of the reduction process which makes it unnecessarily difficult to understand and answer. The function arguments on the left-hand side are shown as unevaluated expressions since that happened to be how they were represented at the time of the reduction. This is a problem in the general case because functions are defined using pattern matching and guards. Since patterns syntactically look like values, and since it normally is easier to see whether a guard applies when the involved arguments are evaluated, not showing the arguments in evaluated form makes it difficult to see which equation that applies, which in turn makes it difficult to answer the question. The term on the right-hand side is also an unevaluated expression which could make the question harder to understand. Note that forcing the evaluation of unevaluated expressions is not a good solution since this may result in a non-terminating computation.

Furthermore, the dependence structure between the reduction steps is also operational since a dependence just indicates that the parent happened to be the first reduction step that needed to rewrite the left-hand side of the child reduction. Under algorithmic debugging, this is not really a problem in itself since the only task of the user is to reply to the questions asked: the order of the questions should be of no concern. But a seemingly strange order could still be puzzling and detract from the questions themselves.

3.3 Summary

In this chapter we examined the lazy evaluation process and explained why conventional debugging tools are not suitable in a lazy context. The reasons are the somewhat surprising evaluation order and the presence of large, unevaluated expressions. We made a general argument for the use of declarative debugging for declarative languages, but at the same time we noted that declarative debugging only constitutes a partial solution for lazy functional languages. The problem is that the employed proof strategy itself is quite low-level and operational. Concretely, in an algorithmic debugging context, this manifests itself in the form of questions involving large, unevaluated expressions (the same problem as for conventional debugging techniques), and possibly in a surprising query order, even though the latter in theory should not be a problem. In order to be successful, a lazy debugger must address these problems.

Chapter 4

A Basis for Lazy Functional Debugging

This chapter presents a basis for lazy functional debugging in the form of a tree-structured, declarative execution record or trace. It abstracts from operational details such as evaluation order, emphasizing the syntactic structure of the program instead. We argue that this is the key to successful debugging in a lazy context. The structure is declarative in the sense that it essentially is a proof tree relating terms to terms through the equations defining the target program. From this perspective, the syntactic structure of the tree reflects a proof strategy where terms as soon as possible are simplified exactly as much as needed for obtaining the final result of the program; that is, an eager reduction strategy, but only up to a point.

We call this kind of trace *Evaluation Dependence Tree* (EDT). In the following we define what an EDT is, justify its structure, and demonstrate how to use it as a basis for declarative debugging. The focus is on the key ideas: exactly what a real EDT looks like depends to some extent on the features of the language in question as well as various implementation choices. The chapter therefore provides two definitions. One is a general, high-level definition which captures the essential aspects of the structure. The other is a formal, detailed definition for a small example language in the form of a denotational semantics which takes the meaning of a program to be its trace in the form of an EDT. The latter illuminates the high-level definition as well as important implementation aspects.

4.1 The Evaluation Dependence Tree

This section introduces the ideas behind the EDT and provides a high-level definition.

4.1.1 Hiding operational details

As discussed in the previous chapter, the demand driven evaluation of lazy functional languages causes problems both for conventional debugging techniques and declarative ones. We identified two main problems.

The first problem is the structure of the computation as such. Though simple in principle, a lazy evaluation strategy is in practice difficult to follow at a detailed level since the structure of the computation does not reflect the structure of the source code. Because programs are written at the declarative level, it should ideally not be necessary to have a detailed understanding of what is going on operationally to be able to debug a program.[1]

The second problem is that suspending the computation of values means that the run-time representation of a value often is an unevaluated expression. Since a programmer arguably understands the program in terms of what functions do to values – at least that is the view implied by defining functions using pattern matching – this is often not the right representation for debugging purposes.

Our solution is to construct a declarative trace that addresses these problems by hiding the problematic, operational aspects of the language implementation. We call this trace the Evaluation Dependence Tree (EDT). Debugging is then performed on this trace. Note that this is post-mortem debugging: the program being debugged is first run and a representation of the computation is constructed. Then, once the execution of the program has finished, debugging can begin.

The insight that lazy functional languages require special tools and techniques for debugging is not new. Some of the earliest work in the area known to us is Hall & O'Donnell [HO85, OH88], where they identify the difficult-to-predict evaluation order as the main problem of debugging lazy functional programs using conventional techniques, and propose a number of possible solutions. A review of related work can be found in chapter 11; here we content ourselves with noting that most works in the area suggest using some form of tracing to handle the problem. Kamin [Kam90] even argues that tracing might well be inevitable in the context of debugging lazy functional programs.

The EDT is essentially a proof tree, where each node is a conclusion of the form: 'Given the equations in the program, it was possible to prove term x equal to term y.' Since we are concerned with functional languages, x will be a reducible function application (redex), and proving equality means rewriting (or reducing) x to y by using the equations defining the applied function in a directed manner as rewrite rules. Debugging is then just a matter of looking for erroneous conclusions and following the chain of reasoning

[1] Reality is unfortunately not always ideal, and an operational understanding of the evaluation process is sometimes necessary, e.g. to deal with performance problems such as space leaks.

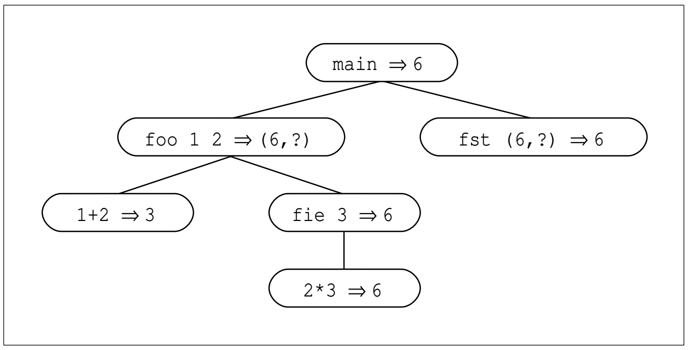


Figure 4.1: A small EDT. '(?)' stands for expressions which were never evaluated.

down the tree until the mistake can be identified, for instance through algorithmic debugging as explained in section 3.1.4. Operationally, each rewrite step is performed by calling the applied function and overwriting the redex with the result, so each node in the EDT can also be seen as representing a function call.

To abstract the operational details, the EDT should suggest a proof or reduction strategy where expressions as soon as possible are simplified as much as is needed to compute the final answer of the program. This corresponds to an eager reduction strategy which somehow, 'miraculously', stops as soon as the result of a reduction would not be used. This has two effects. First, the structure of the tree will reflect the syntactic structure of the target program: if a function in its definition makes use of a reducible application of another, then a reduction involving the former function will depend on the reduction involving the other function if it takes place. Second, values will be shown as evaluated as possible in the tree since reductions seemingly are performed as soon as possible. Furthermore, (sub)expressions left unevaluated can be abstracted to a special value meaning 'unevaluated, assume it is correct' since they cannot have influenced the computation in any way. Thus both debugging problems discussed above are addressed.

Figure 4.1 shows the EDT for the following program.

```
foo x y = (fie (x+y), fie (x/0))

fie x = 2*x

main = fst (foo 1 2)
```

Note that the structure of the EDT reflects the structure of the source code, but that only nodes corresponding to reduced redexes are present (e.g. `foo` 'calls' `fie` only once). Also note that some values were never needed, leaving them represented as unevaluated expressions. These are shown as '?'.

We emphasize that we believe that debugging a program should not affect its semantics; at least, this should be the standard behaviour of the debugger. In a lazy functional context this means in particular that the debugger must not cause evaluation beyond what normally is needed. The reason is that too much evaluation could turn a terminating program into a non-terminating one (as exemplified by the tracing facility described in section 3.2.1), or cause a non-terminating program to be non-terminating for the wrong reason. An EDT-based debugger respects the semantics of the target in this sense since an EDT is constructed by recording the normal execution events.

Disregarding the termination problems, it is also interesting to note that a more conventional debugger which tries to solve the problem of unevaluated expressions simply by evaluating them risks deceiving the user. (Conversely, an EDT *must* contain values in their most evaluated form.) Consider the following example, where we assume that the definition of `h` is wrong.

```
f x = h x x
g x = x + x
h x x = 0
main = g (f 2)
```

Stepping through the reduction sequence, the user would encounter the reduction `(f 2)` $\Rightarrow$ `(h 2 2)` which our hypothetical debugger would simplify to `(f 2)` $\Rightarrow$ `0`. The user may then believe that the bug is in `f` or in a function called from `f`. However, setting a break-point on `f`, for instance, and single-stepping through its code will not directly result in the bug being found since `h` in fact is not called during the call to `f`.

4.1.2 The EDT definition

We now solidify the discussion in the preceding section into a high-level EDT definition. We start by defining the sense in which the terms *saturated application*, *redex* and *partial application* are used in the following. Functions in languages like Haskell are curried, which means that a function of arity $n > 1$ conceptually is understood as a function of arity 1 which returns a (curried) function of arity $n - 1$; that is, the function takes its arguments one at a time. However, this is not how implementations normally work, nor is it the most suitable view for debugging since the function definition syntax allows more than one formal parameter. It will thus be understood that the arity of functions may be greater than one, and that no computation can be performed unless all arguments are available. The arity is typically the syntactic arity implied by the function definition, but since simple transformations could affect the arity, this is somewhat implementation dependent. For example, consider:

```
f x = \y -> x + y
```

An implementation may well decide to treat `f` as having arity 2. (See section 7.1.4 for a discussion of this point from a debugging perspective.)

Definition 4.1 (Saturated application) A function application is *saturated* if the number of arguments is equal to the arity of the applied function. □

Note that it may not be statically known whether a particular syntactic application is saturated or not; the applied function could be a formal parameter of an enclosing function, for instance, and the actual parameters may even have varying arity. Saturation is thus a dynamic property.

In the context of the kind of language implementations considered in this thesis, it is, as indicated above, the case that only saturated applications can be reduced. This is embodied in the following two definitions.

Definition 4.2 (Redex) In the rest of this thesis, the term *redex* (reducible expression) is defined to mean a saturated function application. □

Definition 4.3 (Partial application) An application where the number of arguments is fewer than the arity of the applied function is *partial*. □

Note that a partial application is in WHNF and thus not a redex.

The main EDT definitions can now be stated. Compare the discussion in the preceding section.

Definition 4.4 (Direct evaluation dependence) Let $f\ x_1 \ldots x_m$ be a redex for some function f (of arity m) with arguments x_i, $1 \leq i \leq m$. Suppose

$$f\,x_1 \ldots x_m \;\Rightarrow\; \ldots\;(g\,y_1 \ldots y_n)\;\ldots$$

where $g\ y_1 \ldots y_n$ is an instance of an application occurring in f's body and furthermore a redex for the function g (of arity n) with arguments y_i, $1 \leq i \leq n$. Should the g redex ever become reduced, then the reduction of the f redex is *direct evaluation dependent* on the reduction of the g redex. □

The g redex in definition 4.4 is thus a direct descendant of the f redex (i.e. an instance of an application syntactically occurring in the body of f), and the evaluation of the latter, as far as it was taken, caused the evaluation of the former. Hence direct evaluation dependence. Notice that this is a relation between *reductions*, which also can be seen as function calls. Thus direct evaluation dependence can be understood as a generalized call dependence which does not require the function calls on which a call depends to take place during the latter call.[2] Also note that normal call dependence is subsumed by definition 4.4 as long as it is understood that a direct function call is equivalent to instantiating an application and then reducing it, only much more efficient.

Definition 4.5 (Most evaluated form) The *most evaluated form* of a value is its representation once execution has stopped, assuming a lazy language implementation. □

Consider the following code fragment:

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

x = from 1
```

Notice that `from 1` is a representation of of the infinite list of non-negative integers from 1 and upwards. Suppose the first three elements of the list bound to `x` were needed during the execution of the program to which the above code fragment belongs. Then the most evaluated form of the value bound to `x` is `(1:2:3:(from (3+1)))`.

Definition 4.6 (EDT node) An *EDT node* represents the reduction of a redex. It has the following attributes:

- the name of the applied function
- the names and values of the free variables
- the actual arguments
- the returned result

where values are represented in their most evaluated form. □

To appreciate why it is necessary to keep track of the names and values of the free variables, consider the following program fragment:

```
foo n xs = map fie xs
    where
        fie x = n * x
```

Suppose the expression `foo 2 [1,2,3]` is evaluated. The redexes reduced as part of the computation include `map fie [1,2,3]` and `fie 1`. What

[2] Maybe 'lazy call dependence' would have been a more apt description of the relation.

should these redexes reduce to? Clearly, that depends on the value of the free variable `n`. Thus, to make it possible to verify these reductions during debugging, the binding of the free variable must be presented to the user, as in the following algorithmic debugging scenario:

```
map (fie where n = 2) [1,2,3]
=>
[2,4,6]
Yes/No?

fie 1 where n = 2
=>
2
Yes/No?
```

Definition 4.7 (EDT) An *Evaluation Dependence Tree* (EDT) is a tree structured execution record abstracting the evaluation order, where:

(i) The tree nodes are EDT nodes (in the sense of definition 4.6) and a special root node.

(ii) A node p is the parent of a node q if the reduction represented by p is direct evaluation dependent on the reduction represented by q.

(iii) The special root node, which represents the evaluation of the entire program, is the parent of the EDT nodes representing reductions of top-level redexes.

(iv) The ordering of children is such that a node representing the reduction of an inner redex is to the left of a node representing the reduction of an outer redex w.r.t. the body of the applied function of the parent node. □

The nodes in an EDT may represent only a subset of the reductions which actually were performed; see section 6.3 and 6.4.

Requirement (iv) of definition 4.7 is not a prerequisite for successful algorithmic debugging, but does ensure that the user gets a chance to verify the computation of arguments before these are used in a call. This is usually helpful. On the other hand, the ordering between the arguments to a function (e.g. leftmost first or rightmost first) is less important in this respect and is thus left unspecified.

For an illustration concerning requirement (iv), suppose `foo (fie 1)` constitutes two redexes. Suppose further that `foo` is strict in its only argument and that `fie 1` erroneously reduces to $\perp$. If the node representing the outer redex were to be placed to the left of the node representing the

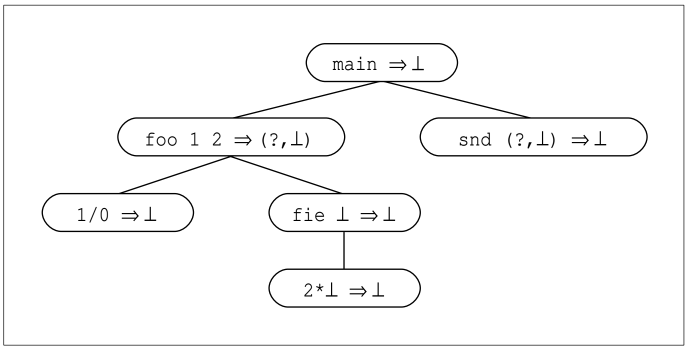


Figure 4.2: EDT with a bug. $\perp$ is the undefined value, in this case `1/0`. An attempt to use it caused an execution error. Since the second component of the result from `foo` was not supposed to be undefined, and since the nodes on which the application of `foo` depends show correct behaviour, the bug must be in the definition of `foo`.

inner redex, the user would during debugging first be asked whether `foo` $\perp$ should reduce to $\perp$ (it should, since `foo` is strict). Then the user would be asked about the erroneous reduction. This is a perfectly legitimate questioning order, but note that it requires the user to understand how `fie` is supposed to behave when applied to an erroneous argument. In real code, it is often implicitly assumed that arguments are correct (i.e. there are no special cases which deal with invalid arguments), so it is quite likely that the correct answer to such questions are not immediately obvious to the user. By requirement (iv), questions tend to be asked in terms of functions applied to verified arguments. Such questions are usually easier to answer.

4.2 EDT-based debugging

The program below is faulty; executing it results in a division by 0 whereupon the program stops.

```
foo x y = (fie (x+y), fie (x/0))

fie x = 2*x

main = snd (foo 1 2)
```

The resulting EDT is given in figure 4.2.

Debugging algorithmically, the debugger would first ask about the result of `main`, which is wrong, then about the application of `foo`, which is wrong as well, then about `1/0` yielding $\perp$, which is correct, and finally about `fie` $\perp$, also yielding $\perp$, which is correct. Given these answers, it would conclude that the bug must be in `foo`.

While the EDT is mainly intended for declarative debugging purposes, it should be pointed out that it has other uses. For example, since dependences can be viewed as function calls, an EDT can be used to produce the equivalent of a conventional stack dump in the event of an execution error. Such dumps are often very useful for quickly getting close to the bug that caused the failure.

4.3 Mini-Freja: a lazy functional language

We will now give a formal EDT definition for a simple lazy functional language, Mini-Freja. We do this in two steps. In this section, we start by giving a denotational standard semantics in continuation passing style (CPS) for the language, taking the meaning of a program to be the output of the program when it is executed. (For simplicity, we make the assumption that there is no input to the program. Otherwise its meaning would be a function mapping input to output.) Then, in the next section, we change the semantics and take the meaning of a program to be a record of its execution in the form of an EDT. Thus, the EDT definition is given in the form of a function which maps a program to its EDT. While Mini-Freja is very simple compared with real languages like Haskell, the definition still covers the key aspects of this kind of trace and will serve as a useful reference when we develop techniques to build EDTs in the context of realistic language implementations in the following chapters. The presentation was inspired by Kishon's & Hudak's work [KH95].

Note that it is important to model the operational lazy aspects of the language since we for EDT construction purposes need to know exactly which function applications that eventually became evaluated: the EDT may be declarative in the sense that it hides details regarding evaluation order, but it is still an execution record and it should log precisely those events which occurred during the normal course of lazy evaluation. This means that the notion of sharing must be captured properly by the semantics. The semantics therefore includes a state component, the *graph*, which permits redexes to be shared and eventually overwritten with the result of evaluating them, thereby ensuring that they are evaluated at most once. The CPS formulation was chosen since it facilitates handling the state component and ensures that it is used in a single-threaded manner. (Alternatively, a state monad could have been used.)

Haskell is used as the meta-language for the semantic specifications. While Haskell may lack a little in precision (all datatypes contain a bottom element, for instance), it is a convenient notation which is good enough for our purposes. It may also make the presentation a little more accessible to the average functional programmer. Above all, using Haskell as the meta-language allows a specification to be type-checked by a Haskell compiler, and we can also execute it in order to convince ourselves that it captures the intended semantics.

In this section and the following, we assume that the reader is fairly familiar with Haskell, CPS, and to some extent denotational semantics. The book by Schmidt [Sch86] is a good introduction to denotational semantics. It also covers CPS. Stoy [Sto77] is a classic text on the subject.

4.3.1 The abstract syntax

First we need a domain for the abstract syntax of Mini-Freja programs:

```
type Id = String

type Name = String

data Exp = LitInt Int          -- Literal integer
         | Prim Name           -- Primitive
         | Var Id              -- Variable
         | App Exp Exp         -- Application
         | Letrec [Def] Exp    -- Recursive definition

data Def = VarDef Id Exp       -- x = exp
         | FunDef Id [Id] Exp -- f x1 ... xn = exp
```

The intention is that a program is a single expression whose value is the result of executing the program. The `Letrec`-expression allows the definition of functions and variables that are in scope in the entire `Letrec`-expression, thus making recursive definitions possible. Note that the abstract syntax permits functions of arity greater than one to be defined, but that functions are applied to one argument at a time (currying). Expression can also be one of the built in constants (primitives), such as arithmetic functions, boolean functions, or functions operating on lists. Such primitives will be introduced as needed.

As an illustration, consider the following program. (We have invented a suitable concrete syntax.) `Cons` is the list construction primitive, and `head`. `tail` are the primitives which return respectively the head and the tail of a list.

```
letrec
    from n = Cons n (from (n + 1))
in
    head (tail (from 2))
```

The following is the abstract syntax representation of the above program.

```
Letrec
   [ FunDef "from" ["n"]
        (App (App (Prim "Cons")
                  (Var "n"))
             (App (Var "from")
                  (App (App (Prim "+") (Var "n"))
                       (LitInt 1)))) ]
   (App (Prim "head")
        (App (Prim "tail")
             (App (Var "from") (LitInt 2))))
```

Note that infix operator application such as `(x+y)` has been translated into prefix function application.

4.3.2 The semantic algebras

We now turn to the semantic domains, i.e. the objects which capture the meaning of syntactic objects in the language. We start with a general store which can be used to store elements of an arbitrary type. A store with elements of type `Node` is used to represent the graph in the following.

```
type Loc = Int

type Store a = ...

emptyStore :: Store a
newStore   :: (Store a) -> a -> (Loc, Store a)
writeStore :: (Store a) -> Loc -> a -> Store a
readStore  :: (Store a) -> Loc -> a
```

The idea is that a store is indexed by elements of the type `Loc` (location). `emptyStore` is the initial, empty store. The operations on a store are `newStore`, which adds a new element to a store and returns the location and an updated store; `writeStore`, which updates the element at the given location in a store; and `readStore`, which returns the element stored at the given location. The details are omitted.

```
type Env = Id -> Loc

emptyEnv :: Env
emptyEnv = \i -> undefined

updEnv :: [(Id, Loc)] -> Env -> Env
updEnv bs env = \i -> case lookup i bs of
                          Nothing -> env i
                          Just a  -> a
```

An environment is simply a function which maps an identifier to a location. Thus the values of all variables (formal function parameters and let-bound variables) are going to be stored in the store. This also offers an easy (though operational) way to handle recursion: since the values of all let-bound variables are in the store, any value can refer to any other value (including itself) via its location in the store.

The function `updEnv` adds new bindings to an environment, possibly shadowing old bindings of an identifier. It is slightly unusual in that it adds several bindings at once to an environment, but this formulation simplifies the handling of functions with more than one argument and the `Letrec`-construct.

```
type Graph = Store Node

type Ans = String

type Cont = Graph -> Ans

type Kont = Val -> Cont
```

The graph is just a store where the element type (i.e. the type of storable values) is `Node` (see below). The answer type `Ans` is the type of program meanings. In this case, the meaning of a program is taken to be its output encoded as a string. A command continuation is a function which maps a graph to an answer. We name this type `Cont`. An expression continuation expects the result of an expression and then continues with the evaluation. The type of expression continuations is thus `Val -> Cont` which we name `Kont`. We will also encounter other types of continuation (e.g. location continuations), but these types are not named explicitly. In the following, the variable `c` will be used for command continuation and the variable `k` for expression-like continuations, i.e. continuations of the type `T -> Cont` for some type `T`.

```
data Val = Int Int
         | Ctr Name [Loc]
         | Fun Int ([Loc] -> Kont -> Cont)
         | PAp Int ([Loc] -> Kont -> Cont) [Loc]

data Node = WHNF Val
          | Thunk (Kont -> Cont)
```

Values of type `Val` are the result of evaluating an expression to WHNF. A value can be an integer (`Int`), a constructed value (`Ctr`), a function (`Fun`), or a partially applied function (`PAp`).

The type `Node` is the type of graph nodes. A node can either be a value or a thunk. A thunk is a suspended computation. If and when the value of the thunk is needed, the suspended computation will be carried out and the thunk will be overwritten by the resulting value. This gives lazy semantics (evaluation at most once). The evaluation of a thunk is often referred to as *forcing* the thunk.

The `Ctr`-constructor is used to represent constructed data objects such as tuples, list-cells, or booleans. Note that the fields are locations and thus refer to nodes in the graph, which need not be in WHNF. As an example, the list `[42]` might be represented as `(Ctr "Cons" [59, 73])` provided the graph contains `Int 42` and `Ctr "Null" []])` at locations 59 and 73 respectively, depending on exactly which names we choose for the constructors.

Built-in and user-defined functions are represented by functions of the type `[Loc] -> Kont -> Cont`. The first argument is a list of locations where the actual parameters (which thus need not be in WHNF) can be found. The second argument is the expression continuation which when applied to the result (in WHNF) of a function application, will carry out the rest of the computation.

The integer-valued field of the constructor `Fun` holds the arity of the function. The arity is needed to be able to tell whether an application is saturated or not. Recall that an application is saturated if as many arguments as indicated by the arity are present, and that a saturated application is a redex. The representation of a partial application is similar to the representation of a function, but the constructor has an extra field which holds the arguments the function has been applied to so far.

A thunk is represented by a function which takes an expression continuation as its only argument. When applied to this expression continuation, it will perform the suspended computation and apply the expression continuation to the result. The semantics given here arranges that the first step carried out by the expression continuation is to overwrite the thunk with the result.

We now give the operations for the types above.

```
eval :: Loc -> Kont -> Cont
eval l k = \g ->
    case (readStore g l) of
        WHNF a  -> k a g
        Thunk s -> s (\a -> update l (WHNF a) (k a)) g

alloc :: (Loc -> Cont) -> Cont
alloc k = \g -> let (l, g') = newStore g undefined
                in k l g'

newWHNF :: Val -> (Loc -> Cont) -> Cont
newWHNF a k =  \g -> let (l, g') = newStore g (WHNF a)
                     in k l g'
```

```
newThunk :: (Kont -> Cont) -> (Loc -> Cont) -> Cont
newThunk s k =  \g -> let (l, g') = newStore g (Thunk s)
                      in k l g'

update :: Loc -> Node -> Cont -> Cont
update l a c = \g -> c (writeStore g l a)
```

The operation `eval` fetches a value from the graph `g` at location `l` and applies the expression continuation `k` to it. Should the location contain a thunk, `eval` first evaluates it and updates the location with the resulting value. This is done by applying the suspended computation `s` to an expression continuation which performs the update and then applies `k` to the value returned from `s`.

The operation `alloc` is used to reserve a location in the graph. It is used when elaborating recursive definitions.

The operation `newWHNF` allocates a new WHNF node. The location is passed to the supplied location continuation.

The operation `newThunk` is similar to `newWHNF`, but creates a thunk instead. It takes two arguments: a function `s` of type `Kont -> Cont` representing a suspended computation, and a location continuation `k` which is passed the location at which the thunk is stored.

The operation `update` simply overwrites the node at location `l` with a new node.

```
apply :: Val -> Loc -> Kont -> Cont
apply (Fun n f)    l k | n == 1 = f [l] k
                       | n > 1  = k (PAp (n - 1) f [l])
apply (PAp n f ls) l k | n == 1 = f (ls ++ [l]) k
                       | n > 1  = k (PAp (n - 1)
                                         f
                                         (ls ++ [l]))
apply _            _ _          = undefined
```

The operation `apply` applies a function (in WHNF) to an argument stored in the graph (and thus not necessarily in WHNF) at location `l`. Let us consider the `Fun` case first. If the arity is 1, then the application is saturated. The function `f` is thus applied to its (only) argument `l`. It is also passed the expression continuation `k` which it will apply to the value it computes. If the arity is greater than one, then the application is not saturated and thus in WHNF. A partial application is therefore created and passed to the expression continuation `k`. The case for `PAp` is similar. If the arity is one, then the application is saturated and `f` is applied to all its arguments (the accumulated ones and the last one). Otherwise a new partial application is created where `l` is added to the list of accumulated arguments and the arity is decreased by 1.

```
printResult :: Kont

run :: Cont -> Ans
run c = c emptyStore
```

The expression continuation `printResult` takes a value in WHNF and converts it to the answer type, in this case a string. The details are omitted. The important aspect for our purposes is that this creates a demand which causes all parts of the value to be evaluated and thus drives the computation. We will return to this point when we give the EDT semantics.

Finally, `run` applies a command continuation to an initial, empty graph, thus producing an answer of the type `Ans`.

4.3.3 The valuation functions

We now turn to the evaluation functions. The first is `vfE` which maps an expression to its meaning. This function is normally denoted by **E** in denotational specifications, but since Haskell does not permit function names starting with capital letters, we add the prefix `vf` for 'valuation function'. The same convention is used for the other valuation functions.

```
vfE :: Exp -> Env -> Kont -> Cont
vfE (LitInt n)    env k = k (Int n)
vfE (Prim n)      env k = vfP n k
vfE (Var i)       env k = eval (env i) k
vfE (App e1 e2)   env k = vfE e1 env   $ \f ->
                          build e2 env $ \l ->
                          apply f l k
vfE (Letrec ds e) env k = vfDs ds env $ \env' ->
                          vfE e env' k

vfP :: Name -> Kont -> Cont

build :: Exp -> Env -> (Loc -> Cont) -> Cont
build (LitInt n) env k = newWHNF (Int n) k
build (Prim n)   env k = vfP n (\a -> newWHNF a k)
build (Var i)    env k = k (env i)
build e          env k = newThunk (vfE e env) k
```

The function `vfE` takes three arguments: the expression, an environment in which to evaluate it, and an expression continuation which is applied to the value of the expression. The case for integer constants is straightforward. The case for primitives makes use of the valuation function `vfP`. Given the name of a primitive, it applies its expression continuation argument to a semantic value representing the primitive. The definition of vfP is omitted, but as an example, here is the semantic value for unary negation:

```
Fun 1 (\[l] k -> eval l (\(Int n) -> k (Int (-n))))
```

Returning to the definition of vfE, a variable is evaluated by first looking it up in the environment. This yields the location where its value is stored. It is evaluated if necessary and passed to the expression continuation k.

An application is handled by first evaluating the function. The result is bound to f. The argument is handled by the auxiliary function build. Its purpose is to delay the evaluation of the argument by creating a thunk. However, build carefully avoids creating a thunk unnecessarily, i.e. when the expression already is in WHNF (integers and primitives) or when the expression is a variable in which case its value is already present in the graph. A thunk is created by passing a function representing the suspended computation (the expression (vfE e2 env)) to newThunk. In either case, bind passes the location of the argument to the continuation supplied by vfE which binds it to l, and f can then be applied to l with the expression continuation k.

Note that '$' is the infix function application operator. Thus

```
vfE e1 env $ \f ->
build e2 env ...
```

is equivalent to

```
vfE e1 env (\f -> build e2 env ...)
```

A set of (mutually recursive) definitions is elaborated by the valuation function vfDs explained below. The result is an extended environment env' in which the expression e can be evaluated with the continuation k.

```
vfDs :: [Def] -> Env -> (Env -> Cont) -> Cont
vfDs ds env k = bind ds $ \bs ->
                let env' = updEnv bs env
                in elaborate ds env' (k env'))

bind :: [Def] -> ([(Id, Loc)] -> Cont) -> Cont
bind []                  k = k []
bind (VarDef i _    : ds) k = alloc   $ \l ->
                              bind ds $ \bs ->
                              k ((i, l) : bs)
bind (FunDef i is _ : ds) k = alloc   $ \l ->
                              bind ds $ \bs ->
                              k ((i, l) : bs)))
```

```
elaborate :: [Def] -> Env -> Cont -> Cont
elaborate []                env c = c
elaborate (VarDef i e : ds) env c =
    update (env i)
           (Thunk (vfE e env))
           (elaborate ds env c)
elaborate (FunDef i is e : ds) env c =
    update (env i)
           (WHNF f)
           (elaborate ds env c)
    where
        f = Fun (length is)
                (\ls -> vfE e (updEnv (zip is ls) env))
```

The function `vfDs` takes three arguments: a list of definitions, an environment which is to be extended with bindings for the defined variables, and an environment continuation which is applied to the resulting environment. The function works by storing the semantic representations of the right-hand sides of the definitions in the graph, and extending the environment so that it maps each defined identifier to the location where its value is stored.

However, the definitions may be mutually recursive, which means that we need the final environment in order to map each right-hand sides to its semantic meaning. `vfDs` therefore works in two stages. First it uses the auxiliary function `bind` to allocate one location for each definition and associate it with the identifier in question. The resulting association list is used to compute the new environment `env'` which maps all defined identifiers to the correct location.

Since we now have access to the resulting environment, the right-hand sides can be evaluated and then stored at the correct locations. This is the task of the auxiliary function `elaborate`. Note that `vfDs` passes the command continuation `(k env')` to `elaborate`. Thus, once `elaborate` has done its work, the environment continuation `k` will be invoked on the new environment.

The elaboration of a variable definition is straightforward. It is just a matter of creating a thunk and storing it at the correct location, i.e. `(env i)`. In the case of a function definition, the semantic value is a function accepting a list of locations where the actual parameters are stored. When this function is invoked, the formal parameters are associated with the location of the corresponding actual parameter (`zip is ls`), the environment is updated with these bindings, and the body finally evaluated in the resulting environment.

This completes the definition of the standard semantics for Mini-Freja. The denotation of a program `prog` is given by

```
run (vfE prog emptyEnv printResult)
```

If `prog` is bound to the abstract syntax tree of the example in section 4.3.1, for instance, the denotation would be `"3"`.

4.4 The EDT semantics for Mini-Freja

This section gives a formal EDT definition for Mini-Freja. The definition is given in the form of a semantics which maps a Mini-Freja program to the corresponding EDT. The EDT semantics is developed by modifying the standard Mini-Freja semantics from the previous section. The abstract syntax is unchanged, so we start with the new versions of the semantic algebras.

4.4.1 The semantic algebras

The types `Loc`, `Store` and `Env` and the operations on them are as given in section 4.3.2. The definition of the type `Graph` is also unchanged, but note that the type `Node` is changed below. The answer type is changed since the denotation of a program now is an EDT.

```
type Ans = EDT

data EDT = EDTNode Function [EDTValue] EDTValue [EDT]

data EDTValue = EVUneval
              | EVInt Int
              | EVConstr Constructor [EDTValue]
              | EVClosure Function [EDTValue]

type Constructor = String

type Function = (String, [String])

makeRoot :: EDTValue -> [EDT] -> EDT
makeRoot res edts = EDTNode ("<root>", []) [] res edts
```

EDTs are represented by the type `EDT`. This type has only one constructor, `EDTNode`, which has four fields: an abstract representation of the applied function (see below), a list of values of any free variables and the actual parameters, the result, and a list of children.

The type `EDTValue` represents values. There are four kinds of value. The constructor `EVUneval` stands for a value which was never needed and thus left unevaluated. Objects built using `EVInt` and `EVConstr` represent integers and constructed values respectively. `EVClosure` objects, finally, represent functional values. The value list contains the values of any free variables and, in case of a partial application, the actual parameters.

Note that there is no representation of $\perp$, the undefined value. While the semantics could be written to catch some kinds of error (e.g. division by zero) by introducing a proper error value (which then could be presented to the user as $\perp$), this would complicate the definitions. Furthermore, there is no way to write a semantics where the denotation of an arbitrary non-terminating program is this proper error value. Thus no attempt is made to handle $\perp$ in this semantics, but we will return to this topic in section 5.2.2 and 6.2 where it is discussed in the context of practical implementations.

The type `Function` is an abstract function representation suitable for debugging purposes. It is a pair of the function name and a list of the names of its free variables. The values of the free variables are understood to be the first elements of the free variable and argument list of a closure. For example, consider:

```
letrec
    z = 17
    f x y = x + y + z
in
    ...
```

Note that `z` is free in the body of `f`. Hence the partial application `(f 3)` would be represented as:

```
EVClosure ("f", ["z"]) [EVInt 17, EVInt 3]
```

The operation `makeRoot` is used to build the root of the EDT. It is passed the result of a computation together with the EDTs for the top-level redexes.

The type `Val` must also be changed to make it possible to construct an EDT. Two issues must be addressed. First, when a function is applied, it must be possible to find its name and the names and values of the free variables. Second, it must be possible to interpret functional values, i.e. to find the name of the function and the names and values of the free variables. This is accomplished by adding a function information field of type `FunInfo` to the constructors `Fun` and `PAp`. The function information consists of a pair of the function name and a list associating the name of each free variable with the location where its value can be found.

```
data Val = Int Int
         | Ctr Name [Loc]
         | Fun FunInfo Int ([Loc] -> Kont -> Cont)
         | PAp FunInfo Int ([Loc] -> Kont -> Cont) [Loc]
```

```
type FunInfo = (Id, [(Id, Loc)])
```

The next step is to change the semantics so that the valuation functions return EDTs in addition to values. Note that the evaluation of an expression could result in several EDTs since an expression may contain several redexes. This suggests that an expression continuation should have a type like the following:

```
type Kont = (Val, [EDT]) -> Cont
```

However, evaluation only takes an expression to WHNF; there may be inner redexes left which are not evaluated until much later, if at all. If such redexes are evaluated later, then the corresponding EDTs belong in the list of EDTs passed to the expression continuation since the redexes *syntactically* are part of the evaluated expression. There is a similar problem for the EDT values. The EDT definition stipulates that a value should be present in the EDT in its most evaluated form, which means that it cannot be constructed until the evaluation has terminated.

The EDT is therefore constructed in two steps. First a tentative EDT is built. In addition to ordinary tree nodes, it contains nodes which act as placeholders for subtrees which have to be inserted if unreduced redexes later are reduced. Each placeholder refers to the thunk in question (which may contain unreduced redexes) by its location in the graph. Furthermore, when a thunk is evaluated, it is overwritten with the resulting value *and* any resulting tentative EDTs. Thus, given the tentative EDTs for the body of the top-level `Letrec` and the final graph, the definite EDT can be constructed by substituting EDTs for the placeholders in case the thunk in question eventually was forced. Values are also handled by referring to their position in the graph and looking them up in their most evaluated form in the final graph when building the definite EDT.

```
data TEDT = TEDTNode FunInfo [Loc] Val [TEDT]
          | MaybeTEDTs Loc

data Node = WHNF Val [TEDT]
          | Thunk (Kont -> Cont)
```

`TEDT` is the type representing tentative EDTs. The constructor `TEDTNode` is similar to the constructor `EDTNode`, but note that the arguments are represented by locations whereas the result is a value. The constructor `MaybeTEDTs` is used to construct placeholders. The constructor `WHNF` of the type `Node` is changed so that it can hold the tentative EDTs which may result when a thunk is forced.

The new expression continuation type is thus:

```
type Kont = (Val, [TEDT]) -> Cont
```

The operations `eval`, `newWHNF`, `newThunk`, and `apply` must be changed according to the above modifications.

```
eval :: Loc -> (Val -> Cont) -> Cont
eval l k = \g ->
    case (readStore g l) of
        WHNF a _ -> k a g
        Thunk s  ->
            s (\(a, ts) -> update l (WHNF a ts) (k a)) g

newWHNF :: Val -> (Loc -> Cont) -> Cont
newWHNF a k = \g -> let (l, g') = newStore g (WHNF a [])
                    in k l g'

newThunk :: (Kont -> Cont) -> ((Loc,TEDT)->Cont) -> Cont
newThunk s k = \g -> let (l, g') = newStore g (Thunk s)
                     in k (l, MaybeTEDTs l) g'

apply :: Val -> Loc -> Kont -> Cont
apply (Fun fi n f) l k
    | n == 1 = f [l] $ \(a, ts) ->
               k (a, [TEDTNode fi [l] a ts])
    | n > 1  = k ((PAp fi (n - 1) f [l]), [])
apply (PAp fi n f ls) l k
    | n == 1 = f (ls ++ [l]) $ \(a, ts) ->
               k (a, [TEDTNode fi (ls ++ [l]) a ts])
    | n > 1  = k ((PAp fi (n - 1) f (ls ++ [l])), [])
apply _ _ _ = undefined
```

The type of `eval` is unchanged since its purpose is to fetch a value. Whether or not a thunk must be forced to compute that value is of no interest to the consumer of the value; specifically, the consumer should *not* be passed any tentative EDTs which may result from forcing a thunk. These tentative EDTs are instead stored in the graph, together with the resulting value, in place of the forced thunk.

The `newThunk` operation is changed to return a placeholder in addition to the location of the created thunk. As explained earlier, if this thunk has been evaluated in the final graph, then the EDTs which resulted from that evaluation (which the `eval` operation stores together with the resulting value) will be substituted for the placeholder.

There are two cases to consider for the `apply` operation. The simple case is partial application (i.e. $\texttt{n} > 1$). In this case, the application is in WHNF and there is nothing to evaluate. Thus an empty list of tentative EDTs is passed to the expression continuation `k`. Otherwise the application is a redex and the applied function is called. This results in a value and a list of tentative EDTs. These tentative EDTs become the children of a single tentative EDT node which records this particular function call, i.e. the function information (function name and free variables), the arguments and the result. A singleton list containing this node is then passed to the expression continuation together with the resulting value.

Two new operations are also introduced. The first one is a replacement for `printResult`. This time, the result in textual form is not of interest, but it is still necessary to create the corresponding demand in order to drive the computation exactly as far as printing the result would have done. Thus an operation `force` is introduced which takes a value to normal form.

```
force :: Val -> Cont -> Cont
force (Ctr _ ls) c = forceLocs ls c
force _          c = c

forceLocs :: [Loc] -> Cont -> Cont
forceLocs []       c = c
forceLocs (l : ls) c =
    eval l (\a -> force a (forceLocs ls c))
```

The second operation, `makeEDT`, builds the definite EDT given the forced result of the program, the tentative EDTs for the top-level redexes, and the final graph. Its definition is straightforward. A placeholder is replaced by an empty list of EDT nodes if the corresponding thunk is still unevaluated. Otherwise it is replaced by the EDTs which resulted from the evaluation of the thunk. EDT values are constructed by looking up nodes in the final graph. If a thunk is found, the corresponding EDT value is `EVUneval`. Otherwise it is a value with an obvious EDT value representation.

```
makeEDT :: Val -> [TEDT] -> Cont
makeEDT a ts g = makeRoot (mkEDTVal a)
                          (concatMap mkEDT ts)
    where
        mkEDT :: TEDT -> [EDT]
        mkEDT (TEDTNode fi ls a ts) =
            [EDTNode (mkFunction fi)
                     (freeVarVals fi
                      ++ (map mkEDTValLoc ls))
                     (mkEDTVal a)
                     (concatMap mkEDT ts)]
        mkEDT (MaybeTEDTs l) =
            case (readStore g l) of
                WHNF _ ts -> concatMap mkEDT ts
                Thunk _   -> []
```

```
mkEDTVal :: Val -> EDTValue
mkEDTVal (Int n)        = EVInt n
mkEDTVal (Ctr n ls)     =
    EVConstr n (map mkEDTValLoc ls)
mkEDTVal (Fun fi _ _)   =
    EVClosure (mkFunction fi) (freeVarVals fi)
mkEDTVal (PAp fi _ _ ls) =
    EVClosure (mkFunction fi)
              (freeVarVals fi
               ++ (map mkEDTValLoc ls))

mkEDTValLoc :: Loc -> EDTValue
mkEDTValLoc l = case (readStore g l) of
                  WHNF a _ -> mkEDTVal a
                  Thunk _  -> EVUneval

mkFunction (fn, fvbs) = (fn, (map fst fvbs))

freeVarVals (_, fvbs) = map (mkEDTValLoc . snd)
                            fvbs
```

4.4.2 The valuation functions

We can now give the new valuation functions.

```
vfE :: Exp -> Env -> Kont -> Cont
vfE (LitInt n)    env k = k (Int n, [])
vfE (Prim n)      env k = vfP n (\a -> k (a, []))
vfE (Var i)       env k = eval (env i) (\a -> k (a, []))
vfE (App e1 e2)   env k = vfE e1 env   $ \(f, ts1) ->
                          build e2 env $ \(l, ts2) ->
                          apply f l    $ \(a, ts3) ->
                          k (a, ts1 ++ ts2 ++ ts2)
vfE (Letrec ds e) env k = vfDs ds env $ \(env', ts1) ->
                          vfE e env'  $ \(a, ts2)    ->
                          k (a, ts1 ++ ts2)

build :: Exp -> Env -> ((Loc, [TEDT]) -> Cont) -> Cont
build (LitInt n) env k = newWHNF (Int n) $ \l ->
                         k (l, [])
build (Prim n)   env k = vfP n     $ \a ->
                         newWHNF a $ \l ->
                         k (l, []))
```

```
build (Var i)     env k = k (env i, [])
build e           env k = newThunk (vfE e env) $
                          \(l, t) -> k (l, [t])
```

Neither integers nor primitives are redexes. Evaluating them does thus not result in any tentative EDTs. The same is true for variables.

Applications are evaluated by first evaluating the function expression. The resulting function value is bound to `f`, and any resulting tentative EDTs to `ts1`. The argument is handled by `build`, as before, which delays its evaluation if necessary. The location of the argument is bound to `l` and any tentative EDTs (at most one, a placeholder) are bound to `ts2`. Finally, the function is applied. This results in a value, which is bound to `a`. Any tentative EDTs (at most one) are bound to `ts3`. The expression continuation is passed the result of the function application and a list where all tentative EDTs have been collected in an order corresponding to innermost-first, left-to-right evaluation.

The auxiliary function `build` must be modified since a tentative EDT results when a thunk is created. Thus `build` now passes a pair of the location and a list of tentative EDTs to its continuation argument. The list is empty in the integer, primitive, and variable cases since these are not redexes. Otherwise it is a singleton list containing a placeholder referring to the delayed expression.

Elaboration of definitions is again handled by the valuation function `vfDs` and results in a new environment and in a list of tentative EDTs which are bound to `env'` and `ts1` respectively. The body of the `Letrec` is then evaluated in the new environment, and the resulting value and tentative EDTs are bound to respectively `a` and `ts2`. The result is finally passed on to the expression continuation `k` together with the list of all tentative EDTs.

The reason `vfDs` must return tentative EDTs in addition to a new environment is that the right-hand sides of variable definitions may contain redexes. Elaboration of function definitions, on the other hand, does not result in any tentative EDTs since the body of a function is not evaluated until the function is applied. We give the new versions of `vfDs` and `elaborate` below; `bind` is exactly as before.

```
vfDs:: [Def] -> Env -> ((Env,[TEDT])->Cont) -> Cont
vfDs ds env k = bind ds $ \bs ->
                let env' = updEnv bs env
                in elaborate ds env' $ \ts ->
                   k (env', ts)

elaborate :: [Def] -> Env -> ([TEDT] -> Cont) -> Cont
elaborate []                  env c = c []
```

```
elaborate (VarDef i e    : ds) env c =
    update (env i)
           (Thunk (vfE e env))
           (elaborate ds env c')
    where
        c' = \ts -> c (MaybeTEDTs (env i) : ts)
elaborate (FunDef i is e : ds) env c =
    update (env i)
           (WHNF f [])
           (elaborate ds env c)
    where
        fvns = freeVarsExp is e
        fvvs = map env fvns
        fvbs = zip fvns fvvs
        f = Fun (i, fvbs)
                (length is)
                (\ls -> vfE e (updEnv (zip is ls) env))
```

The main changes concern `elaborate`. It now expects to be passed a continuation to which the resulting list of tentative EDTs should be passed. Whenever a variable definition is elaborated, a placeholder is created and prepended to the list of tentative EDTs. Function definitions are handled almost exactly as before. The main difference is that the function information tuple must be created. The name of the function is directly available from the definition (`i`). The names of the free variables are computed from the abstract syntax of the body of the function (`e`) and the list of the formal parameters (`is`) by the auxiliary function `freeVarsExp`, whose definition we omit. The locations of the free variables are then looked up in the environment, and a list of pairs of variable name and location bound to `fvbs`. Finally, the semantic representation of the function can be constructed, where the desired function information is given by `(i, fvbs)`. Notice that the WHNF node in which the function representation is stored contains an empty list of tentative EDTs; a function is a constant, not a redex.

4.4.3 The EDT denotation of a small program

The EDT denotation of a Mini-Freja program `prog` can now be obtained in the following way:

```
run (vfE prog
         emptyEnv
         (\(a, ts) -> force a (makeEDT a ts)))
```

The program is first evaluated to WHNF. The resulting value is bound to `a` and the resulting tentative EDTs to `ts`. The 'print demand' is then simulated by forcing `a`, whereupon the definite EDT is constructed by `makeEDT` applied to `a` and `ts`.

Consider the example program from section 4.3.1 again.

```
letrec
    from n = Cons n (from (n + 1))
in
    head (tail (from 2))
```

Its EDT denotation, passed through a suitable pretty-printing function, is:

```
<root> => 3
  from 2 where from=(from where from=<from>)
      => (Cons 2 (Cons 3 ?))
    + 2 1 => 3
    from 3 where from=(from where from=<from>)
        => (Cons 3 ?)
      Cons 3 ? => (Cons 3 ?)
    Cons 2 (Cons 3 ?) => (Cons 2 (Cons 3 ?))
  tail (Cons 2 (Cons 3 ?)) => (Cons 3 ?)
  head (Cons 3 ?) => 3
```

The root node represents the entire execution, and we see that the result of the program is 3. The children of the root represent the reduction of the three top-level redexes: `from 2`, `tail (from 2)`, and `head (tail (from 2))`. Note that the ordering of the children is innermost-first. (Since all involved functions have arity 1, the example does not demonstrate that the ordering is also left-to-right.)

The list returned from `from` is only partially evaluated since only the first two elements were needed. The unevaluated part is shown as '`?`'.

Note that `from` seemingly invokes itself recursively, but only as many times as is needed. The children of the first call to `from` represent the reduction of the three redexes in the body of `from`: the computation of the argument to the recursive call to `from`, the recursive call to `from`, and the call to the constructor function `Cons` which builds a list cell (also shown as `Cons`). The second call to `from` only has one child, the call to the constructor function `Cons`. The remaining two redexes in the body of `from` were of course constructed, but since they were never reduced, the corresponding placeholders were found to refer to thunks in the final graph during the construction of the definite EDT, and thus no EDT nodes resulted.

Finally, notice that `from` has a free variable, `from`, which is bound (in a `where`-clause) to a closure representing the function `from`, i.e. `from` with its free variable `from` bound to a closure representing the function `from`, and so on. The recursive function `from` is represented by a circular closure, created by the semantic function `vfDs`. Since the term-representation of a circular structure is infinite, the pretty-printer gives up after a while and prints the closure as `<from>`. Showing all formally free variables in this way may not be the best option for debugging purposes. We will return to this topic in section 7.1.3.

4.4.4 Implementation aspects

An important aspect of the EDT semantics is that it clearly shows what kind of information that must be available to a debugger in a real implementation in order to build an EDT and to interpret the run-time representation of values.

To build an EDT node, it is necessary to have access to the name of the applied function, the names and values of its free variables, the arguments and the result. The semantics also shows that in order for values to be present in their most evaluated form in the final EDT, the EDT nodes should hold references to values in the graph until the execution stops.

To convert a value to a meaningful, printable representation, it must be possible to recognize basic types like integers, to find the constructor name for a constructed object, and to find the name of a function and the names and values of its free variables. The semantics also illustrates that it should be possible to recognize thunks. In practice, it is desirable to have access to even more information such as source code references.

4.5 Limitations of EDT-based debugging

The main feature of an EDT-based debugger is that it focuses on the declarative aspects of the target program. This is also its main limitation.

First it should be made clear that there are certain types of bugs that are not of interest in a pure, functional context. Most modern functional languages are strongly typed, for instance, preventing type errors, and storage allocation and deallocation is completely automatic, eliminating another class of bugs that plagues most imperative programmers. Thus, for bugs of these kinds, there is no need to worry about whether or not an EDT-based debugger would be suitable.

The types of bugs that is addressed by an EDT-based debugger are those that are caused by errors in the program logic. It could either be various types of programming mistakes leading to erroneous results, or it could be problems like a missing case in a function definition, an array index out of bounds, or an erroneous termination condition, leading to run-time errors or a program that fails to terminate. In all these cases there is a *visible bug* *symptom* which in principle can be tracked from the root of the EDT to the place where it originates.

However, since it is possible to write programs in an operational style even in a lazy functional language, an EDT-based debugger might not always be the best tool for finding bugs in the program logic. Sparud [Spa96] gives an example of one class of such programs: reactive, stream-based ones. In an EDT-based approach, the user would have to consider complete input and output streams at once, while a more appropriate scheme may be to focus on some particular output caused by some particular input; that is, there is a time-aspect that is important but which is not recorded in any way in an EDT.

An obvious case where an EDT-based debugger is not going to be useful is for performance debugging, i.e. the process of finding out why the program runs unexpectedly slow or why it consumes surprisingly much memory and what to do about it. Here, by definition, the operational behaviour is what is of interest.

Chapter 5

Basic EDT Generation

The previous chapter introduced the EDT, a particular kind of execution record, or trace, and explained why it is a suitable basis for debugging lazy functional programs. This chapter addresses the problem of making these ideas work in practice. To achieve this, EDT generating mechanisms must first of all be integrated into a real language implementation. Then it must be ensured that the overhead of EDT generation is small enough to make it possible to debug real programs.

The approach taken in this chapter is to describe the ideas behind our EDT generator and its implementation. We think it performs well enough to qualify as a realistic implementation (see chapter 10). The ideas will be developed gradually, justifying various design decisions as we go along. Our language implementation is fairly traditional and based on compiled graph reduction and G-code. This is thus the context in which the discussion is set. The basic ideas are quite general, however, and could be used with other types of compiling implementations or interpreters.

5.1 Selecting the approach

Whenever a target program is to be traced, the appropriate trace-generating mechanisms must somehow be invoked from the running program or on behalf of it. There are many ways to achieve this. In simple cases, it might be a question of inserting print-statements into the source code of the target. Other possibilities include linking with trace-generating version of standard libraries or executing the target under the control of another program which performs the actual tracing.

Since each node in an EDT represents the reduction of a (saturated) function application, we are interested in calling the tree-building mechanisms precisely when functions are being invoked. This is most easily achieved by executing an instrumented version of the target, i.e. a version where calls to the tree-building mechanisms have been embedded in a suitable way. This can be done in two ways. The first possibility is to add the necessary instrumentation through source-to-source transformations of the target's source code. The other alternative is to delegate the work to the language implementation in question. From the user's perspective, there might be little telling these two approaches apart since the source-to-source transformer obviously could be run as a pre-pass to the compiler. But from the point of view of implementation, this decision is fundamental, in particular in a pure functional context.

The main advantage of a *pure* transformational approach is portability. Provided the transformed code is still a valid program (according to the language standard) which does not rely on any auxiliary mechanisms not definable within the language, the transformer can be turned into a tool which can be used together with any implementation of the language. This means that the debugger implementation can be made portable across language implementations as well as platforms.

In a pure functional context, the transformed source code must also be purely functional, which means that the trace cannot be constructed through side-effects on global variables. The only possibility is to transform the program in such a way that the new version returns the desired trace as a part of its result along with the result of the original program. Recent work taking this approach is Sparud [Spa96] and Naish [NB95].

However, as exemplified by both Sparud's and Naish's work, it is not possible to produce an EDT-like trace for *lazy* functional languages without the aid of specialized, impure support primitives if one desires to guarantee that the act of tracing does not affect the termination properties of the target program. The problem is that the trace may contain references to unevaluated expression, and any attempt to use such an expression (displaying it, for instance) will cause evaluation beyond what normally would have taken place, thus risking non-termination. This undermines the portability argument. To what extent depends on the difficulty of implementing the support primitives. As illustrated by Sparud's work, it is possible to go quite a long way with the aid of only a few easily added routines.

In order to implement a practical trace-based debugger, there are other issues that must be taken into account, besides just being able to generate the trace. First of all, it is vital that the tracing is not too costly in terms of space and additional execution time. It is also important to handle execution errors and looping programs. For the convenience of the user, the possibility of controlling various aspects of the tracing interactively (e.g. starting tracing from a certain function) may also be important. All of these present additional challenges to the implementor of a debugger based on source-to-source transformations. For instance, the transformations tend to increase the code size substantially, which is costly both in terms of compilation and execution time. Any substantial measurements have not been performed to this author's knowledge, but a few small tests in Nilsson & Sparud [NS96] indicate that a transformed program may well run a factor ten or more slower than the original, *not* counting additional garbage collection time. This is arguably too slow for practical purposes, in particular if the time for garbage collection is also taken into account. See chapter 11 for a more thorough discussion on transformation-based schemes.

The other approach, letting the language implementation perform the instrumentation, does not constrain the implementor in the same way. For example, in an implementation that compiles to imperative code, unconstrained side-effects can be used for trace-construction purposes, and instrumentation can be as simple as inserting procedure calls at strategic points in the code. This means that the time overhead can be made fairly small and that the trace construction process can be controlled in a very precise manner, which in turn allows the other practical aspects mentioned above to be addressed. This is thus the approach that we follow in this work.

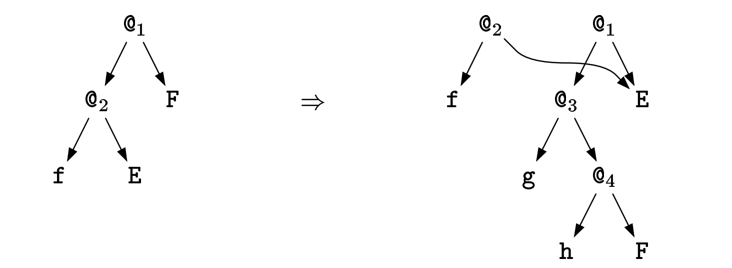
The obvious disadvantage of working at the language implementation level, is that the debugger (or at least the trace generating part of it) becomes tightly coupled to a particular compiler or interpreter. Thus portability across language implementations is sacrificed. In addition, quite a lot of support is needed from the language implementation – this is at least the case for the methods described in this thesis – and some of this support may be in conflict with design decisions in existing implementations. Retrofitting a tracer along the lines described here to an existing language implementation is thus likely to be a substantial undertaking.

It should, however, be pointed out that the advocated approach does not preclude portability across platforms. In fact, since we are not assuming anything about object file formats, or about operating system facilities for controlling the execution of the target program from a debugger, the opposite is true: the proposed scheme facilitates porting the debugger to new platforms along with the rest of the system. Furthermore, a large part of our debugger is written in ANSI C and thus fairly portable. It is also worth noting that a close coupling between a debugger and a particular language implementation (or a suite of implementations based on a common back-end) is standard practice today. Thus this work in no worse than most major debuggers in this respect.

5.2 Building the tree

5.2.1 Dependences

Let us first briefly recall the basic principles of compiled graph reduction. A code sequence is compiled for each function (supercombinator). This code



(a) Graph for `f E F`.

(b) Graph after rewrite step.

Figure 5.1: Graph reduction of `f E F`.

sequence is invoked whenever a saturated[1] application of the function, i.e. a redex, is to about be evaluated (reduced). It constructs an instance of the function body and and then physically overwrites the redex root with the root of the newly constructed graph.

For example, suppose that we have a function `f x y = g (h y) x`. The code for `f` will then perform the rewriting step illustrated in figure 5.1, where `E` and `F` are arbitrary pieces of graph, and the indices on the application nodes (`@`) identifies physical locations on the heap.

The redex root is application node 1 ($@_1$) in figure 5.1a, and it has thus been physically overwritten with the root of the instantiated function body in figure 5.1b. The application nodes 3 and 4 are new, constructed by the code of `f` in some newly allocated memory cells, whereas the remaining application node in the original graph, number 2, becomes garbage, to be collected by the garbage collector later, unless it happens to be shared.

When we talk about *calling* or *invoking* one of the functions in the target program in the following, or, equivalently, about *reducing a redex*, it is the process described above which we are referring to. On the other hand, in the context of lazy functional languages, *applying* a function does not imply that it gets invoked. A function application could be created instead, as illustrated in the example above. If this application is saturated, then it is a redex which may be reduced later causing the applied function to be called.

Referring to the EDT definition (section 4.1.2, definitions 4.4 to 4.7), recall that an EDT node represents the reduction of a redex. It records the name of the applied function, the arguments and the result. Furthermore, the evaluation of a redex depends on the evaluation of another (i.e. the EDT

[1]Recall that an application is *saturated* when the number of arguments are equal to the arity of the function.

node representing the former is the parent of the EDT node representing the latter) if the latter redex is an instance of a function application that syntactically occurs in the body of the applied function.

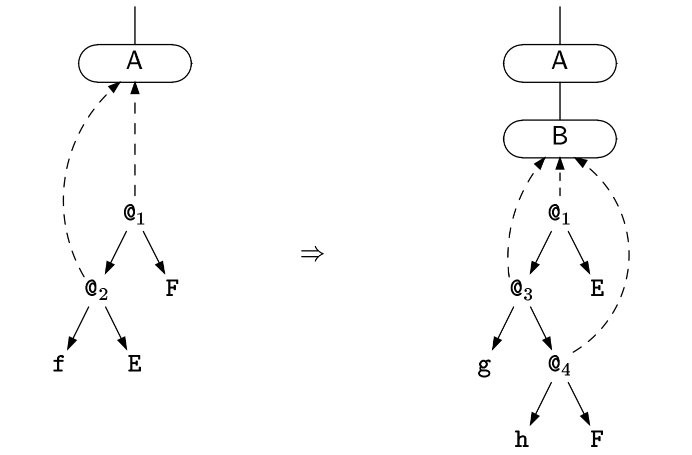
An EDT node is thus created as the result of *reducing* a redex, not as a result of constructing it. On the other hand, the parent of an EDT node is determined when the corresponding redex is *constructed*. Since the reduction of a redex can occur much later than the construction of it in a lazy language, this means that it is necessary to somehow keep track of the prospective parent until the redex is reduced (or until the program terminates, at which point it will be clear that the redex will never be reduced).

We solve this by annotating application nodes with a reference to the EDT node in question. Whenever a redex is reduced, the annotation of the application node at the redex root will refer to the node which is the parent of the EDT node which is about to be created. The application nodes are annotated as they are constructed, i.e. during the instantiation of a function body. The annotation refers to the EDT node representing the current reduction, thus capturing the syntactic aspect of the definition of EDT dependence. The EDT node referenced by a redex will sometimes also simply be called 'the parent of the redex', since this EDT node is the record of the reduction that created the redex.

Figure 5.2 illustrates the scheme outlined above. It is the example from figure 5.1, but the application nodes have been annotated with references to the EDT (the dashed arrows). As shown in figure 5.2a, the redex root of the graph is annotated with a reference to an EDT node marked A. This node is the record of the function application during which the the redex `f E F` was built, and it will thus become the parent of the EDT node representing the reduction of this redex.

The situation after the reduction is shown in figure 5.2b. A new EDT node, marked B, which is the record of the reduction of `f E F`, has been created and inserted into the EDT as a child of node A. The new application nodes, created as a result of instantiating the body of `f`, have been annotated with references to node B. Note that this is also true for application node 1 since it is a new node, even though it physically is built on top of the old redex root.

This process can be optimized by observing that only application nodes that are redexes need to be annotated. For instance, if `h` in the above example is a function of arity two or more, then `h F` is not a saturated application. An unsaturated application is not a redex, and there is thus no point in annotating that application node with a reference to the EDT node. The advantage of not annotating a node is that annotated application nodes take more time to construct, occupy more space on the heap, and take longer to garbage collect than unannotated ones.



(a) Graph for f E F.

(b) Graph after rewrite step.

Figure 5.2: Graph reduction with annotated application nodes. A and B are EDT nodes.

5.2.2 Values

The EDT definition also stipulates that function arguments and results should occur in their most evaluated form in the EDT. As explained in section 4.1.1, this is actually necessitated by the syntactic structure of the EDT as well as a desirable property in itself. Consequently, arguments and results must not be copied from the heap when an EDT node is built, something which in any event would be very expensive, but instead be shared via pointers. Once the execution has terminated, these pointers will refer to the arguments and results in their most evaluated form by the nature of graph reduction.

In the following, the term EDT will be used both to refer to the EDT as previously defined, including the heap-allocated values, and in a narrower sense referring only to the EDT nodes proper. When the distinction is important, it will be clear from the context. This impreciseness is due to the fact that many parts of the graph simultaneously belong to the EDT and to the running program, making the exact extent of the EDT difficult to decide.

Keeping references from the EDT to live pieces of graph in the heap means that the garbage collector must be made aware of the EDT. Otherwise values no longer referred to by the target program, but which are part of

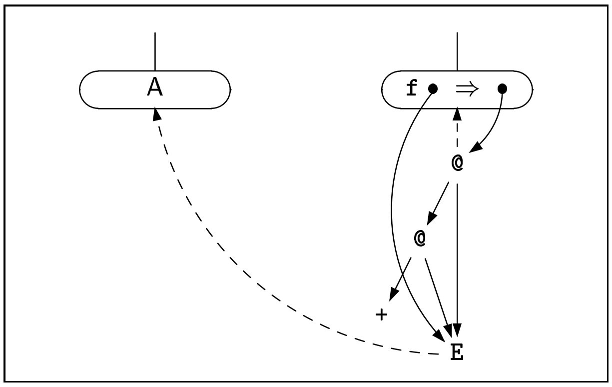


Figure 5.3: The situation after reduction of `f E`. The arguments and results in the EDT nodes are live pieces of graph, referred to via pointers. Each redex is annotated with a reference to the EDT node which will become the parent of the EDT node representing the reduction of the redex when and if this reduction takes place.

the EDT, would be lost. As a consequence, note that not only will the EDT nodes themselves occupy memory, but as new nodes are added, the EDT will also hold on to an increasing amount of heap memory which normally would be reclaimed by the garbage collector. Sometimes arguments and results may even grow as the execution proceeds, e.g. in case of (non-circular) infinite lists. We will return to these points in section 5.3.

The following example shows how arguments and results are handled. Suppose that the function `f` is defined as `f x = x + x`. Suppose also that we are evaluating `f E`, where `E` is a redex and the result of a preceding reduction step. The situation immediately after the reduction of `f E` is shown in figure 5.3. Note that the argument and the result are live pieces of graph, referred to via pointers from the newly constructed EDT node. Also note how each allocated redex refers to its parent in the tree (the dashed arrows), and that not every application is a redex.

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 forces the evaluation of `E`, yielding `7`, say.

The new situation is shown in figure 5.4. The new EDT nodes, representing the reductions of the redexes `E` and `E + E`, have been inserted as children of the redexes' parents. Note how each of these expressions has been physically overwritten by the value obtained by evaluating it. Thus, once the execution has terminated, values referred from the EDT will be in

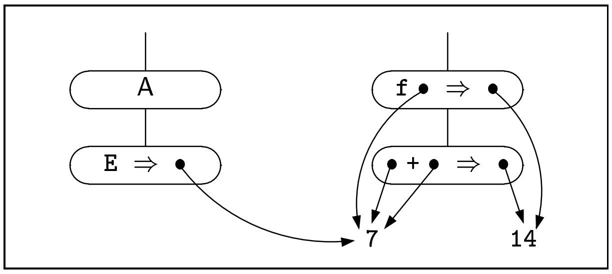


Figure 5.4: The situation after reduction of `E + E`.

their most evaluated form.

If an execution terminates abnormally, by a run-time error, it may be necessary to tidy up the graph a bit, e.g. by replacing the redex(es) that caused the error with a representation of $\perp$ (bottom). Should the execution not terminate at all, one must rely on the user eventually forcing termination, e.g. by sending a suitable signal to the process. The implementation must then regain control and overwrite all redexes that were being evaluated by $\perp$. However, if the implementation performs zapping (see 2.4.6), the zap nodes can be used as a representation of $\perp$, and no patching of the graph should be necessary. Section 6.2 considers the handling on non-terminating programs in greater detail.

5.2.3 Ordering children

Thus far, nothing has been said about the ordering of the children, should an EDT node have more than one. In section 4.1.2 it was argued that an innermost-redex-first order is suitable, whereas the ordering between the arguments to a function was left unspecified.[2] To implement this, we order the application nodes according to the aforementioned syntactic criterion as they are created by numbering them. Thus application nodes are annotated with an ordering number along with the parent reference. This allows the children of an EDT node to be sorted into order as and when the corresponding redex is reduced. Note that the chronological order in which children are inserted is determined by dynamic demand and thus in general not in agreement with the chosen syntactical order. This necessitates numbering of the application nodes or some similar scheme. Figure 5.5 illustrates this point, as well as the way in which an EDT typically grows at many places at once.

[2]For debugging purposes, being asked about the arguments of a function before being asked about the function application itself is often helpful, whereas it matters little whether the debugger asks about the leftmost or the rightmost argument first.

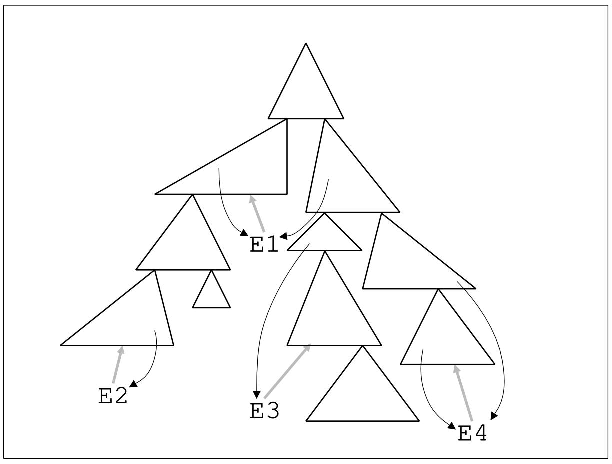


Figure 5.5: Illustration of EDT growth. Each triangle represents a tree of one or more EDT nodes. The `E`is are annotated redexes on the heap. Note that the insertion of new EDT nodes in not restricted to any particular part of the tree. Also note that the EDT node representing the reduction of `E3` belongs to the left of its sibling: the chronological order in which children are inserted does not reflect the desired ordering.

5.2.4 Optimized graph reduction

It should be pointed out that the graph-reduction process as outlined above often is side-stepped in real implementations in order to gain efficiency: if it can be determined that some value will be needed, it is always better to compute that value immediately than to construct an expression for later evaluation. Thus, a good implementation of a lazy functional language will perform a function call immediately whenever that does not change the meaning of the program. This was discussed in section 2.4.5.

Constructing an EDT with the proper structure in this case is, in principle, easy. Just note that what conceptually happens is that a redex is constructed and then immediately reduced. Thus, in order to correctly build an EDT, the implementation should behave as if the construction and subsequent reduction have taken place. It ought to be possible to do this fairly efficiently since the parent of the EDT node resulting from a direct function call is the EDT node representing the current function invocation. In the worst case, it is, for debugging purposes, always possible, at some performance cost, to resort to a naive implementation strategy where the redex is constructed explicitly and then reduced, in which case the EDT construction scheme as described above is directly applicable.

5.3 Piecemeal EDT generation

5.3.1 Storage requirements for EDT-based debugging

A problem facing the implementor of any debugger based on tracing of execution events, is that the size of the trace is a monotonically increasing function of the execution time of the target program. Since the execution time of a program in general is not bounded by any upper limit, this is also true for the size of the trace.

In some cases this may not be a problem in practice. Perhaps the logged events occur very rarely, or perhaps there are sufficient memory resources for all practical purposes, e.g. large disks. For an EDT-based debugger, however, this is a big practical problem since the logged events (reductions) are very frequent, and since the EDT must be held in primary memory.

The latter is a consequence of the fact that the EDT grows at many places simultaneously and refers to live pieces of graph which move around as a result of garbage collection. (See the example in section 5.2.2 for an illustration.) This makes it difficult to store EDT nodes on secondary storage without repeatedly having to read them back into primary memory at a large performance cost. Employing a large virtual memory may facilitate getting the EDT onto secondary storage, and it may also save the cost of explicit `read` and `write` system calls, but it does not fundamentally change anything.

Also notice (5.2.2) that the EDT holds on to values on the heap which under normal circumstances would have been reclaimed by the garbage collector. The amount of memory required to store these values can easily dwarf the amount of memory used to store the EDT nodes proper. This incurs a time penalty whenever the garbage collector runs, since the amount of live data is much larger than normally. A generational garbage collection scheme may mitigate the problem, but does not completely eliminate the extra overhead. Placing data which only the EDT refers to on secondary storage is possible, but, again, very costly, since the data may contain references to values which are still in use by the running program. Thus the garbage collector must repeatedly bring this data back into primary memory in order to find such references.

In practice, only a small fraction of the execution events are of any interest for finding a bug. Thus it is interesting to use various 'filtering' techniques so as to avoid storing uninteresting events. This is further discussed in chapter 6. However, while filtering helps combating the large trace size, and in addition speeds up the debugging process, one cannot expect such techniques to make it possible to carry out arbitrary debugging within limited memory resources: there is no more an upper limit on the number of 'interesting' events than there is one on the total number of events.

It is worth pointing out that these problems are inherent in an EDT-based debugger. For instance, one might after a cursory inspection believe that a transformational scheme would be less demanding in terms of memory consumption than our approach, since it may be possible to arrange so that the EDT nodes are generated on demand by virtue of working in a lazy language. However, the only thing this will achieve is delaying the computation of EDT nodes by creating suspended computations. These will hold on to the same heap allocated values as any other kind of EDT, and thus the magnitude of the memory consumption remains unchanged.

What is needed is a way to impose an arbitrary upper bound on the amount of memory needed for debugging. This is the topic of the next section. Admittedly, our current implementation does not impose an absolutely tight upper bound as the storage requirements intermittently do exceed the prescribed bound: it is this that activates the pruning process which keeps the size of the tree within the limits. But as long as a reasonable amount of memory is allocated for debugging (a few megabytes or more), we have found that the scheme usually works well.

5.3.2 Trading time for space

An alternative to storing a complete EDT, is to store only so much of the EDT as there is room for. Debugging is then started on this first piece of the tree. If this is enough to find the bug, all is well. Otherwise, the target is re-executed, and the next piece of the EDT is captured and stored. The idea is not new, see Plaisted [Pla84] for instance. We refer to this as *piecemeal EDT generation*. Re-executing the program is not a problem since pure functional programs are deterministic, even though, from a practical point of view, it is a bit involved since any input to the program must be preserved and reused, a forced termination of a looping program automatically re-issued at the appropriate moment, etc.

The process is illustrated in figure 5.6. The large, dashed triangle corresponds to the entire EDT, whereas the small, topmost triangle corresponds to the part of the EDT 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 EDT is reached, assuming no bug has been found, the program is re-executed and the part of the EDT corresponding to the second triangle stored (disposing of the old EDT part). 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.

Re-executing the target program repeatedly is costly. Is this really acceptable from a user's point of view? It should be kept in mind that debugging is an interactive activity. Thus the acceptability depends on how long time it takes to re-execute the program, i.e. the worst case response time,

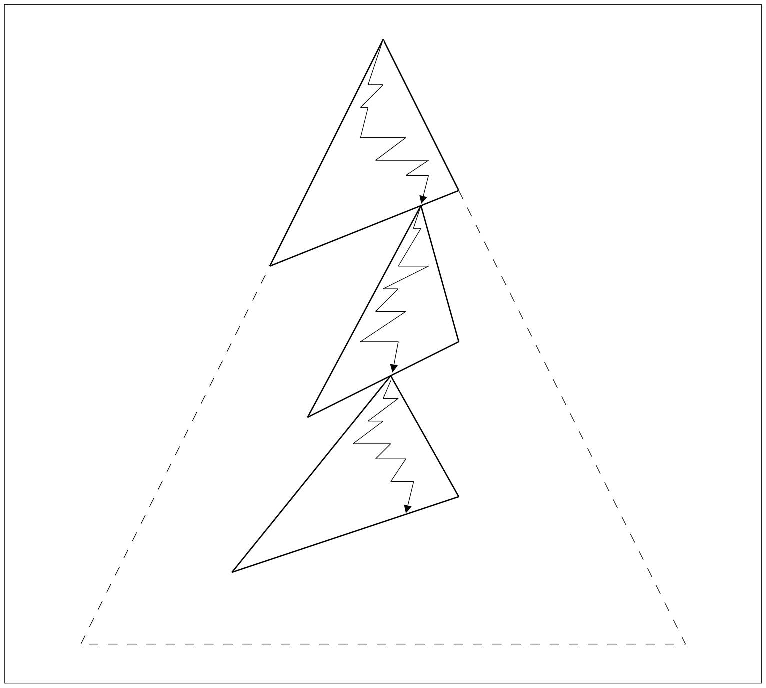


Figure 5.6: Piecemeal EDT generation. The large, dashed triangle represents the entire EDT, the smaller triangles represent the parts of the EDT which are stored during the first, second and third execution. The path going from the root downwards illustrates how the EDT is traversed during debugging.

and how frequently this happens. If re-execution happens often, then an execution time of a few seconds may be acceptable, whereas if re-execution is an infrequently occurring event, considerably worse response times might be tolerated.

The execution time obviously depends on how long it normally takes to execute the program, on the instrumentation overhead, and on the time taken to build the EDT nodes. The time for garbage collection also enters the equation since a large EDT means more live data and thus longer garbage collection times. In fact, for large EDTs, the garbage collection time tends to dominate the execution time. This means that the target program runs faster when only a small part of the EDT is kept. If many nodes are stored, it is also likely that only a fraction of these will ever be visited during debugging, so constructing them and garbage collecting the pieces of graph they retain was really work wasted. In fact, it may be *cheaper* to execute the target a few times than to execute it once and build the entire EDT.

The re-execution rate in turn depends on many factors. For starters, the more memory that is set aside to store the EDT, the less often missing parts of the EDT would have to be created by re-executing the program. It is also a question of how the EDT is traversed. If this happens in an orderly, partly predictable manner, as is the case when debugging algorithmically for instance, this can be taken into account when deciding what parts of the EDT to keep during construction and what parts to throw away, with an eye to minimizing the total number of re-executions during the debugging process. If, on the other hand, the tree is accessed in a much more unpredictable manner, a lot of re-executions would probably occur.

Thus it is not easy to say how a piecemeal scheme fare in general, but practical experience indicates that a suitable compromise between execution time on the one hand, and the frequency of re-execution on the other, can be found by adjusting the amount of memory that the EDT is allowed to occupy. Chapter 10 presents performance figures for five different benchmarks and varying size of the stored portion of the EDT.

Also note that it is not absolutely necessary to re-execute the entire program, since doing this is just a simple way of placing the right demand context on the EDT node where debugging is to be resumed. An alternative is to take advantage of the fact that the demand context can be inferred from the result of the parent of that node by re-evaluating the parent application and driving the computation until the result is exactly as evaluated as it was before [NB95]. This would however require further modifications of the graph reduction machinery and has not been implemented. Moreover, our experiments indicate that garbage collection often dominates the cost of execution when performing tracing (see section 10.4), so it may well be the case that partial re-execution would not improve performance much in practice.

5.3.3 Deciding which nodes to store

Once one has opted for a piecemeal scheme, the next question that begs an answer is how to select the nodes which should be stored. We assume that the EDT usually is going to be traversed in an orderly manner, as is the case when debugging algorithmically. This order directly induces a priority order among the nodes: to choose between two nodes, prefer the one which would be visited first. The idea is illustrated in the context of algorithmic debugging in figure 5.7. Each node is assigned a distance measure relative the root node by counting the number of questions that would have to be answered in order to get from the root node to the node in question. For obvious reasons, we call this measure the *query distance*. Nodes that are close to the root in this sense should thus be preferred over more distant

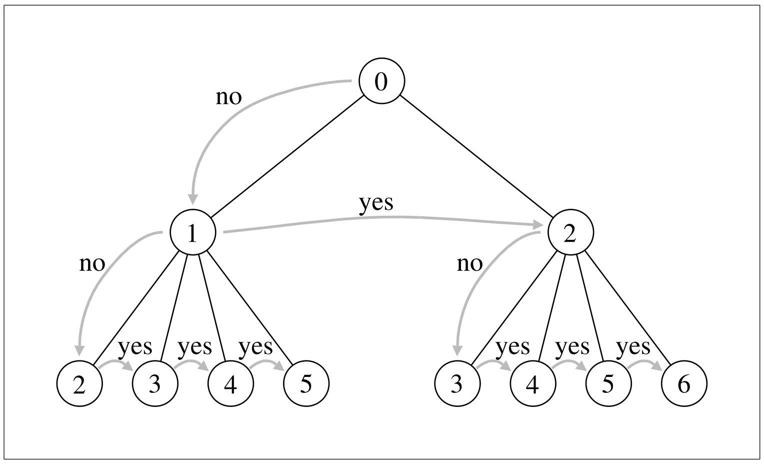


Figure 5.7: The query distance. The distance between the root and a node is obtained by counting the number of questions that would have to be answered in order to get from the root to the node in question during algorithmic debugging.

nodes.

Now the question is how many nodes that constitute a suitably large EDT piece. The answer is that the number of nodes on its own is not a good measure of the size of the EDT since a single node could refer to an arbitrarily large piece of graph on the heap. A much more robust solution is obtained by monitoring the real memory consumption[3] of the EDT. This allows the size of the EDT to be kept below a certain limit by removing the most distant nodes when the EDT grows too large. In addition to this, there is a (user-definable) upper bound on the number of EDT nodes.

It is important to realize that the size constraint cannot be maintained simply by stopping adding nodes once the size limits has been reached. Instead, the size of the tree must be constantly monitored and the tree must be pruned whenever the limit is exceeded. There are two reasons for this.

First, as has been pointed out earlier (section 5.2.3), nodes are not inserted into the EDT in an orderly manner. This means that the insertion of a node may necessitate the removal of a more distant node to keep the size of the EDT within the prescribed limits.

Second, the values referred to from an EDT node may grow *after* the node has been inserted into the EDT. For instance, suppose that we have

[3]In our implementation, the garbage collector provides this information by measuring how much extra heap memory the EDT holds on to.

the following function:

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

Suppose further that there is a redex `from 1`. Once this redex is reduced, the resulting EDT node would refer to the result `1 : from 2`, which conceptually is the infinite list of all integers from 1 and upwards. After a while, however, a much larger part of the result may have been computed, which means that the EDT node refers to a list `1 : 2 : 3 :` ... This representation of the result occupies much more space than the previous one.

In order to implement a piecemeal scheme as outlined here, we see that the tree construction process must be controlled very carefully and in close co-operation with the run-time system. This kind of control would probably be difficult to obtain if the EDT construction was to be performed at a higher level, e.g. through a transformational scheme.

5.4 Implementation details

Sections 5.1 through 5.3 outlined our EDT generation scheme and described some of the most important aspects of the implementation. In this section, the implementation is considered in greater detail.

5.4.1 Preliminaries

In order to understand the implementation at a more detailed level, it is useful to have a rough idea about how the compiler and run-time system cooperates with the debugger. The main points are thus listed below. This topic is addressed further in chapter 9.

- Generation of instrumented code. A call to *Trace*, the main EDT construction routine, is inserted at the beginning of the generated code sequence for functions which should be debugged. Application nodes are annotated when constructed. Functions thus instrumented are referred to as *traced functions*. It is usually not necessary to trace all functions; see section 6.4.

- Debugging information such as function names, source code references, arities, etc. is made available to the debugger by embedding it into the generated code.

- The representation of the graph is such that it can be understood by the debugger. All objects are tagged, and tags are distinct to make it possible to tell isomorphic objects of different types apart (the empty list and `False` have distinct tags; so have a list cell and a pair). It must also be possible to find function names and the names and values of free variables. This was discussed in section 4.1.2.

- The garbage collector is made aware of the EDT, since the EDT holds references to the graph.
- Control is passed to the debugger on run-time errors and user interrupts since it must be possible to debug non-terminating programs as well as programs which abort.
- Support for re-execution. There is an interface which allows the debugger to execute and then re-execute the generated code. This involves preserving any input to the program and automatically aborting re-executions in case of non-termination (see section 6.2).

Let us now introduce some fundamental notions. Under a piecemeal generation scheme, only a part of the complete EDT is going to be physically stored in the memory at any one point in time. Yet, for several reasons, it must be possible to refer to an arbitrary node in the complete tree. For instance, it must be possible to refer to a child of an EDT node even if that child has been removed by the pruning process. EDT references from annotated application nodes must also be valid regardless of which EDT nodes physically remain in primary memory.

Each node in the *complete* tree is therefore assigned an identity, *id*, which is independent of which part of the tree that is currently stored. One possibility (in a sequential implementation) is to count the reductions. Since the reductions will occur in exactly the same order when the program is re-executed, and since each node in the EDT represents a single reduction, the current reduction count can be used as the *id* of the node corresponding to the latest reduction. A hash table is maintained that makes it possible to quickly find an EDT node given its *id* as long as it is present.

We also need criteria for deciding how much and which part of the tree to keep. As discussed in section 5.3.3, the aim should be to keep the physical memory consumption below a user-defined limit. In our implementation, the garbage collector monitors the memory consumption of the EDT, and if it finds that the prescribed limit is exceeded, the tree is pruned. The memory consumption is defined as the amount of heap memory which is only referenced from the EDT; pieces of graph still in use by the running program do not count.

Since it is difficult to know how much memory the pruned tree occupies without performing a new garbage collection, we do not check that the pruned tree actually is small enough. However, the tree is pruned substantially at each occasion, by halving the maximum distance of stored nodes (see below), so in a troublesome situation (e.g. when the root node refers to large graphs), the tree will become small quickly. Unfortunately, this does mean that the implementation falls short of the ideal of guaranteeing a firm upper bound on the size of the EDT. Furthermore, no attempt is currently made to handle a situation where the root node alone holds on to *more* memory than what is permitted. The latter actually turns out to be a real problem in some cases, which shows that it may be necessary to prune the arguments and result of an EDT node as a last resort. See section 10.4 for further details.

As to which part to store, one particular node in the complete tree is designated the *current root*. (The current root is normally the node in which the smallest subtree known to contain the source of the bug is rooted.) Then the nodes that are closest to the current root in the subtree emanating from it are kept. For measuring the distance from the current root, we chose a simple scheme optimized for algorithmic debugging. We call this the query distance; the idea was outlined in section 5.3.3. Here is the precise definition:

Definition 5.1 (Query distance) The *query distance* measure the distance from the current root to a descendant in terms of the number of questions which have to be answered to get from the current root to the node in question when debugging algorithmically in the way described in section 3.1.4. Thus:

(i) The query distance to the current root from the current root is 0.

(ii) If the query distance to a node from the current root is d, then the children of that node are assigned increasing query distances, from left to right, starting from $d + 1$.

(iii) The query distance to nodes which are not descendants of the current root is undefined. □

Only descendants of the current root are eligible for being kept, and each such node is conceptually assigned a distance attribute, qd, with respect to the current root, through the definition above. But in practice there is a problem since the children of a node are not going to be inserted in an orderly manner from left to right when the tree is constructed. Thus the qd attributes cannot be computed until all nodes have been inserted in the tree, which defeats the purpose of the attribute!

We solve this by attributing each *redex* with an estimate of the distance attribute. This estimate can easily be computed given qd of the EDT node for the call which created the redex, and syntactic features of the source code reflecting the innermost-first ordering of the children that we have chosen. When and if a redex is reduced, we take qd of the resulting EDT node to be the estimated distance attribute of the redex. As a bonus, the redex attribute can also be used to sort the resulting EDT node into the correct place among its siblings (see section 5.2.3).

For example, consider:

```
f x y = g (h x) (f y x)
```

Suppose that a redex `f 1 2` is reduced and that its distance attribute is d. Then `(h 1)` might be attributed $d+1$, `(f 2 1)` $d+2$ and `(g (h 1) (f 2 1))` $d+3$ if it is known that these are the redexes and we decide to order the arguments from left to right. (Cf. the reduction order in a typical strict language: first `g`'s first argument, `(h 1)`, then `g`'s second argument, `(f 2 1)`, and finally the application of `g` to its arguments.) This is just an approximation since there is no guarantee that all redexes are going to be reduced. When an unknown function is applied in the body of a higher order function, it may not even be possible to statically determine which applications that are redexes. Such applications must conservatively be assumed to be redexes.

As we have seen earlier, the distance attribute is in general not directly used to a priori decide which nodes should be inserted into the EDT. Instead the construction process alternates between inserting nodes as redexes are reduced, and pruning of the tree in order to meet the size constraints, working from distant nodes towards ones closer to the root. However, an invariant is that either all or no nodes at a certain distance are present in the stored portion of the tree. This means that once nodes at distance d have been removed, only nodes strictly closer than d are eligible for insertion in the future.

5.4.2 EDT nodes

Before turning to the tree construction algorithms, let us briefly describe two important data structures: the low-level representation of the EDT nodes and *traced application nodes.*

Figure 5.8 shows a somewhat simplified version of the EDT node. The fields `id` and `qd` were described above. The field `fun_info` is (a pointer to) information about the applied function, such as its name, the names of any free variables, source code references, arity, and any other information a debugger may need. This information is generated by the compiler and embedded (in the form of a C structure) in the object code of the target. The fields `args` and `result` points to the list of arguments and the result on the heap. The fields `parent` and `leftsib` points to the parent and to the sibling to the left (if any) of the node. These fields are mainly used to facilitate pruning. The field `next` is used for implementing a hash table that maps an *id* to the corresponding EDT node (if present), and `next_same_qd` is used to link all nodes at the same distance from the current root for pruning purposes.

The remaining fields are more interesting. They refer to the sibling to the right (if any) and to the first child (if any). Thus each EDT node refers to a linked list of its children. What complicates matters slightly, is that

```
typedef struct edt_node {
   unsigned        id;
   int             qd;
   obj_info        fun_info;
   list            args;
   graph           result;
   struct edt_node *parent;
   struct edt_node *leftsib;
   struct edt_node *rightsib; /* Valid if rightsib_qd = 0 */
   int             rightsib_qd;
   struct edt_node *firstchd; /* Valid if firstchd_qd = 0 */
   int             firstchd_qd;
   struct edt_node *next;
   struct edt_node *next_same_qd;
} edt_node;
```

Figure 5.8: The EDT node (C syntax).

it must be possible to refer to nodes which exist in the complete tree but have been removed as the result of pruning. At the same time, it must be possible to quickly traverse the tree, insert new nodes, etc.

While the *id* of a node, via the hash table, could be used to refer to siblings and children, this was rejected on the grounds of efficiency considerations. Instead, two fields are used in combination as follows. If `rightsib_qd` is zero, then the field `rightsib` points to the sibling to the right of the node. A `NULL` pointer is used to indicate that there is no sibling. If, on the other hand, `rightsib_qd` is non-zero, then this indicates that the sibling to the right has been removed and that the field `rightsib` is invalid. The value of `rightsib_qd` is the *qd* of the removed sibling (which is strictly larger than zero for all nodes except the root). The fields `firstchd` and `firstchd_qd` work in a similar manner, but refer to the first child of the node.

This might seem as a rather indirect way to refer to a missing child or sibling, but the *id* of the *parent* together with the difference between the parent's *qd* and the missing child or sibling's *qd* make it easy to capture the right part of the tree once an attempt to access a missing node during debugging has triggered a re-execution. Basically the tracer first waits for the the parent *id*, which becomes the new current root, and then for the right child, identified by the *qd* difference. The point of this is that any *siblings* to the right of the desired node will be inserted into the tree (and then possibly removed), making it possible to go both right (corresponding to the answer 'yes' during algorithmic debugging) and down ('no') from the node.

(a) Application node. (b) Traced application node.

Figure 5.9: The layout of untraced and traced application nodes. `AP` and `TAP` are the node tags. The fields *fun* and *arg* are pointers to the applied function and the argument, respectively. The field *pid* is the identity of the parent, and *qd*, finally, is the estimated query distance.

5.4.3 Traced application nodes

A normal application node contains two fields (in addition to a tag that gives the node type): the function being applied and the argument it is applied to. A *traced application node* contains two extra fields: the *id* of the *parent* (*pid*), i.e. the EDT node recording the reduction step which created the traced application node (see section 5.2.1), and an estimation of *qd* as explained earlier. Figure 5.9 shows the two kinds of application node. A redex where the redex root is a traced application node is called a *traced redex.*

Note that the parent is referred to via its *id* rather than via a pointer. This is because the parent might be removed due to pruning. Had pointers been used, this would result in dangling pointers from the heap into the memory area used for storing the EDT nodes. In general it would then be impossible to determine whether the parent of a redex is present or not. The *id*, on the other hand, is always a valid reference, and the aforementioned hash table which maps *id*s to EDT nodes can be used to determine whether a node is still present whenever this is needed.

Figure 5.10 puts the two types of application nodes into context. It depicts the situation when a traced redex `f 1 2` is being reduced, just before entering the code for `f`. Note the reference from the traced application node (`TAP`) to its parent (the EDT node A), and the query distance which in this case happens to be 1. We are assuming that the function `f` has arity 2. Thus `f 1` is not a redex and the other application node is therefore untraced (`AP`). The box obj. info represents the object information record for `f`. Also note the spine stack which at this stage contains pointers to the redex root and the two arguments.

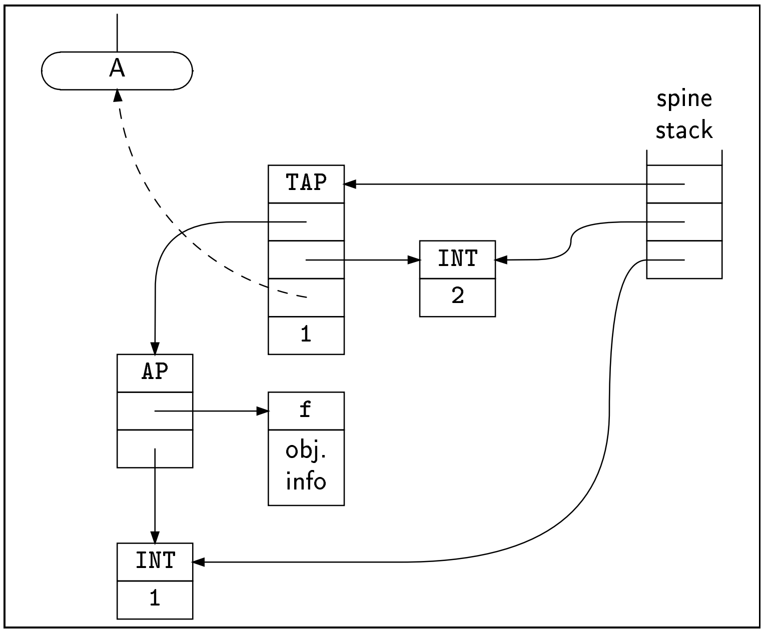


Figure 5.10: The reduction of a traced redex `f 1 2`. The figure illustrates the situation just before the code for `f` is entered. It is assumed that the arity of `f` is two. Hence only the redex root is a traced redex.

5.4.4 The Trace algorithm

Now let us consider the algorithms for constructing (a suitable portion of) the EDT. The construction process relies on a subtle interplay between instrumented code, generated by the compiler for each (traced) function (supercombinator), and the EDT construction routines. We would like to emphasize that the scheme has been designed in such a way that instrumented and uninstrumented code (i.e. code for traced functions and untraced functions, respectively) can coexist: the only effect of calling uninstrumented code is to disable tree construction below that point. This means that the complete EDT only contains nodes for calls to traced functions.

The following main steps are performed when a traced function is invoked:

1. Call *Trace*, the main EDT construction routine, with the following arguments:

    - A pointer to a record containing the *pid* and *qd* from the redex root. (In our case, this is simply a pointer to the *pid* field in the traced redex.) `NULL` if the redex root is untraced.

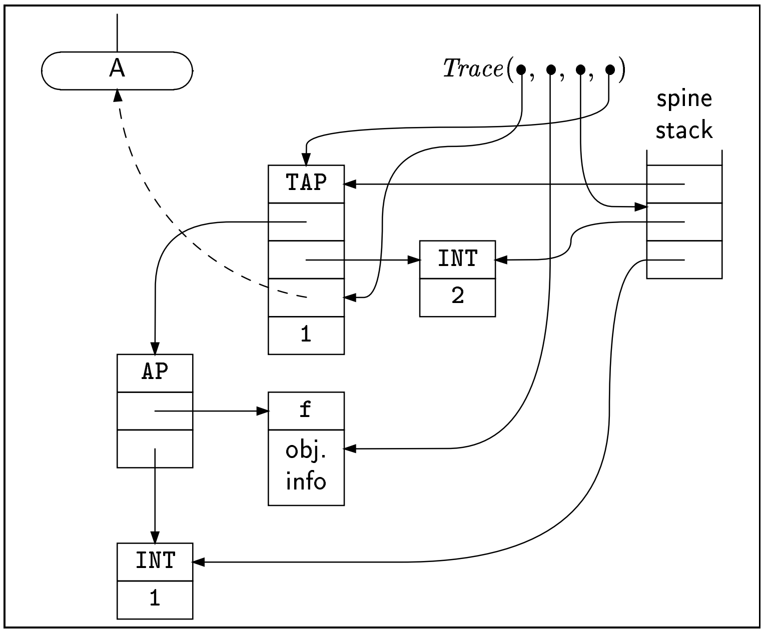


Figure 5.11: The reduction of a traced redex `f 1 2`. The figure illustrates the situation just before *Trace* is called from the code for `f`.

- A pointer to the object information record. (This pointer is what is stored in the `fun_info` field of an EDT node.)
- A pointer to the argument vector on the spine stack.
- A pointer to where the result will be located, i.e. the redex root.

Figure 5.11 illustrates a call to *Trace*. It shows the reduction of a traced redex `f 1 2`, just before calling *Trace*. Compare figure 5.10.

The layout of a CAF redex is different since there are no arguments. The outlined calling conventions ensures that *Trace* is isolated from such details. In particular, *Trace* does not have to dynamically inspect the tag of the redex root in order to find the *pid* and *qd*. This would have been necessary had the *pid*/*qd* pointer not been passed explicitly to *Trace*.

2. *Trace* now determines whether or not to build an EDT node (see below). If a node is built, the *id* of that node is returned, along with an initial estimate of *qd* for any redexes which are going to be constructed. Otherwise a special value, *nt* ('no tree'), is returned. This disables tree construction below this point, and the code then behaves exactly as if it had not been instrumented.

3. Whenever a redex, or something which might be a redex, is built (and tree construction is enabled), it is made a traced redex where *id* is set to the *id* returned from *Trace*, and *qd* is set to the initial *qd* plus an offset. This offset reflects the desired innermost-first ordering of child redexes and could be computed statically at compile time. However, it is easier to use a counter, initialized to the initial *qd* estimate, and increment this counter by one each time a (potential) redex is built. This works since redexes are constructed in a suitable order.

As was explained in section 5.2.4, it is sometimes possible to call a function directly instead of constructing a redex on the heap. In that case the above scheme would have to be changed a bit. For example, there would not be any redex to pass to *Trace*. Also note that whenever a redex is built and it can be statically determined that the applied function is untraced, there is no point in making that redex traced since *Trace* is not going to be called for the redex anyway.

Trace has two modes. It starts in waiting mode where it stays until the reduction which corresponds to the root of the desired EDT part (the current root) takes place. Then it moves to construction mode in which the actual tree building is performed. In waiting mode, *Trace* performs the following steps:

1. Increment the reduction counter.
2. Decide whether it is time to enter construction mode. This is basically a matter of waiting for the current root to show up, which can be found out by inspecting the reduction count. If it is not yet time, return *nt* to the caller.
3. Create the root node. Its *id* is set to the current value of reduction count, which is also returned to the caller along with a suitable initial *qd*.

Note that instrumented functions always call *Trace*. This means that even though returning *nt* means 'disable tree construction below this point', it does *not* prevent *Trace* from initializing the tree construction when it encounters the current root.

In construction mode, *Trace* behaves as follows:

1. Increment the reduction counter.
2. Check whether or not the redex root is a traced application node. If it is not, tree construction is disabled below this point and *nt* is returned to the caller.
3. Get *id* from the redex root and check if node *id* is still present. If not, return *nt* to caller.

4. Get *qd* from the redex root and check if it is small enough. If not, return *nt* to the caller.

5. Prune the tree if necessary in order to insert a new node. This must be done since the EDT nodes are allocated from a pool whose size is determined once and for all at the start of a debugging session.[4] Thus it may be the case that there are no free EDT nodes, and if the node which is about to be inserted is closer to the current root than the most distant node in the tree, then the tree should be pruned to make room for the new node. Do not confuse this with the pruning performed by the garbage collector.

6. Create and insert a new EDT node among the children of node *id* at the correct place as indicated by *qd* (children are sorted by *qd*). A list of arguments is built. References to the arguments are obtained from the argument vector on the spine stack. The number of arguments expected by the function is obtained from the object information record. The result is set to point to the redex root, since it is going to be updated by the result. The field `id` is set to the current value of reduction count, which is also returned as *id* to the caller along with *qd* from the redex root. The field `fun_info` is set to point to the object information record for the calling function (passed as an argument to *Trace*).

The central parts of the algorithm are listed in appendix B.

5.4.5 Integration into the G-machine

The G-machine was briefly discussed in section 2.4.4. In order to accommodate the tracing machinery within the G-machine framework, two small modifications of the machine were made. First, the instruction `FUNSTART` was changed so that it calls *Trace* (for traced functions) as explained in the previous section. It also allocates two variables (on the dump) where it stores the returned *id* and the initial *qd* estimate. Second, a new instruction `MKTAP` was introduced to build traced application nodes. It behaves as follows:

1. Check *id* (stored on the dump by `FUNSTART`). If it is the special value *nt* ('no tree'), build a normal application node (i.e. behave as the instruction `MKAP`).

[4] The scheme with a fixed pool size was chosen mainly because it is easy to implement. However, it does give an additional, user-controllable, size constraint on the EDT, which occasionally is useful.

2. Otherwise, build a traced application node. The *pid* field is set to *id* (the EDT node *id* is the parent of the redex), and the *qd* field is set to the current value of the variable *qd* (on the dump).

3. Increment the variable *qd* by one. This ensures that traced redexes are assigned increasing query distances as they are built.

Note that our compiler does not assign query distance offsets explicitly to potential redexes, but rely on creating them in a suitable order. The ordering between the children of an EDT node thus depends on the compilation strategy and may not be exactly as dictated by the EDT definition or as described in section 5.4.1. However, the constituents of an application are constructed before the application itself, so the scheme does respect the innermost-redex-first order.

To avoid complicating the translation of supercombinators into G-code by making it dependent on whether a supercombinator is traced or not, the instruction `MKTAP` is introduced by a subsequent peep-hole optimization pass. For instance, if it is known that `g` is a traced function of arity 2, the peep-hole optimizer would transform an application of `g` as follows

```
PUSHFUN g            PUSHFUN g
MKAP          ⇒      MKAP
MKAP                 MKTAP
```

since only the second `MKAP` instruction builds a redex. Had `g` been an untraced function, or had its arity been greater than 2, the peep-hole optimizer would have left the code to the left unchanged (ignoring other optimizations which may be performed), in the former case because there is no need to to build traced redexes, in latter case because there are no redexes. On the other hand, the peep-hole optimizer must be pessimistic when an unknown function is being applied, as in the following case:

```
PUSH 7              PUSH 7
MKAP          ⇒     MKTAP
MKAP                MKTAP
```

Here, an unknown function is fetched from position 7 on the stack and applied to the two topmost elements on the stack. Since nothing is known about its arity or tracedness, it must be assumed that both `MKAP`s build redexes, and that both of these must be traced.

Chapter 6

Improved EDT Generation

This chapter describes a number of important improvements of the basic EDT generation scheme. The emphasis is placed on 'filtering' techniques which reduce the size of the EDT. The idea is to make use of any suspicions the user may have regarding where the bug is likely to be located so as to avoid storing as many irrelevant nodes as possible. However, two other (unrelated) points are discussed first: how to handle constant application forms and non-terminating programs.

6.1 Handling constant application forms

Constant application forms (CAFs) are top-level constants or, equivalently, 0-arity functions. They are used fairly frequently when programming in lazy functional languages such as Haskell. Consistently with the lazy paradigm, the value of a CAF is computed on demand and then shared among its users. The run-time computation of CAFs is what sets them apart from named functions of arity one or more: the latter are always compile-time constants.[1] Here is a classic example:

```
sieve (x:xs) = x : sieve [ y | y <- xs, y `mod` x /= 0 ]

primes = sieve [2..]

main = take 100 primes
```

The code above defines two CAFs: `primes` and `main`. The CAF `primes` is the infinite list of all prime numbers from two and upwards, whereas `main` is the list of the first 100 prime numbers. The primes are computed using

[1] Very simple CAFs, like `x = 1 + 2`, may of course be computed statically.

the well known sieve of Erathostene, here embodied by the function `sieve`. This function assumes the first element in its list argument to be prime. All multiples of this prime are removed from the rest of the argument list (by using a list comprehension) and `sieve` is then recursively applied to the resulting list. The identified prime is finally consed onto the list returned by the recursive application, resulting in a list of primes in increasing order.

Now, from a debugging perspective, the question is where in the EDT the computation of `primes` belong. Is it perhaps a subcomputation of the computation of `main` since `primes` is used in the definition of `main`? But what if `primes` is shared? For example:

```
somePrimes = take 100 primes

someOtherPrimes = take 100 (drop 200 primes)

main = somePrimes ++ someOtherPrimes
```

Here, should the EDT for `primes` be part of the EDT for `somePrimes` or that for `someOtherPrimes`? Or perhaps both? (`take` n returns the n first elements of a list, `drop` n removes the n first elements from a list, and `++` is list concatenation.) Moreover, looking back at the EDT definition, we find that the EDT for `primes` does not belong anywhere since `primes` is not *applied* in the bodies of `somePrimes` or `someOtherPrimes`! Indeed, `primes` *cannot* be applied since it is a list.

A natural solution is to let the computation of each CAF result in a separate EDT. After all, the redexes in the bodies of the CAFs are allocated once and for all at the top-level. Thus the execution of a program in general yields a forest of EDTs. These are collected as the children of a special root node which conceptually represents the entire execution. In this way a single EDT is obtained, even if the root node is special since it does not record any particular reduction.

However, the CAF computations are not necessarily completely independent since a CAF can refer to other CAFs, as exemplified by `main` above. A CAF may even refer recursively to itself, possibly indirectly through a set of mutually recursive CAFs. If the value of a CAF is wrong, the reason may well be that one of the referenced CAFs is erroneous. But since the EDTs corresponding to the computation of the referenced CAFs are not subtrees of the EDT corresponding to the computation of the CAF in question, these dependences are not explicitly recorded and will not be reflected in the questions asked during algorithmic debugging. This means that the debugger may conclude that the bug is located in the wrong CAF.

There are at least two solutions two this problem. The theoretically most satisfying, at least in the context of algorithmic debugging, is perhaps to explicitly track references to free variables which are CAFs, much as free variables of local functions are handled (see section 7.1). Thus the debugger may ask questions like '`main`, where `primes = [2, 3, 5, ...]`, reduces to `[2, 3, 5, ...]`. Is this correct or not?' Indeed, the issue of handling CAFs could largely be side-stepped by actually taking out such free variables as extra parameters through a process similar to lambda lifting. But this loses CAF sharing, so the performance hit may become large.

The other approach is to topologically sort the EDTs for the CAFs into dependence order. This ensures that an algorithmic debugger asks the questions bottom-up, as it were, starting with fundamental CAFs which do not depend on other CAFs. While this may be less elegant than making the dependences explicit, it does have the advantage that the questions asked are simpler: the user may assume that any auxiliary CAF definitions are correct and can therefore focus solely on whether or not the value of the CAF in question is correct. In the other case, the user must figure out what the expected result is, given possibly erroneous input. That is much more difficult.

Note that mutually recursive CAFs cause a problem since they cannot be totally ordered while respecting all dependences. Making the dependences explicit in the asked questions would allow such CAFs to be properly debugged algorithmically, even if the user may find the asked questions somewhat awkward to answer ('Given these erroneous CAFs, does the CAF in question have the expected erroneous value?').

In our implementation, we follow the sorting approach. The problem of mutually dependent CAFs is ignored by choosing an arbitrary ordering between the CAFs in such a group. The user thus has to take the diagnosis of the debugger with a pinch of salt when debugging mutually recursive CAFs, but we do not think this is a large problem in practice.

Implementing the chosen scheme may be a little problematic in a system which permits separate compilation since it is necessary to perform a global dependence analysis, across module boundaries, just prior to linking all object files into an executable. Our system does allow separate compilation, and we handle the problem by delegating the task of analysing the dependences between CAFs to the Runtime System Generator (RSG, see 9). The extra implementation effort was however minimal since the RSG carries out this analysis anyway in order to handle garbage collection of CAFs properly and efficiently.

6.2 Handling non-terminating programs

Programs are conventionally expected to compute an answer and then terminate. Failure to terminate is regarded as an error. In practice there are, of course, programs which are meant to run indefinitely, such as various types of server or control programs, but these are in any event expected to respond to input, or perform some other kind of activity, within bounded time. We will not try to characterize harmful non-termination exactly, but instead assume that it is obvious to a user when a program is non-terminating in the wrong way.

In any case, a non-terminating target program must be stopped before debugging can begin. Since it in general is impossible to automatically detect non-termination, we will leave the task of stopping such a program to the user. In many systems it is possible to do this by pressing some special key combination such as `CTRL-C`. We assume that it can be arranged for the debugger to gain control when the user stops the target.

Semantically, the standard approach is to identify a non-terminating computation with other types of execution errors (division by 0, pattern matching failed, etc.). All of these are denoted by the special symbol $\perp$ (bottom) meaning undefined. Thus it should be clear that once a non-terminating program has been stopped, debugging it is in principle no different than debugging any other program. The symbol $\perp$ can be regarded just as any other value and it is used wherever there is something which is missing. It is then up to the user to decide whether the occurrence of $\perp$ in any particular context is correct or not.

To implement a debugger which handles non-termination as sketched above, there are two problems which must be addressed. The first is that the debugger must be able to recognize undefined values on the heap once the user has stopped the program and the debugger gains control. The second is handling re-execution.

The first problem is just a slight variant of the problem of recognizing undefined values after an execution error. The undefined values are those redexes which were being reduced when the error occurred or the user aborted the execution. Note that just being a redex is not enough to qualify as being undefined. In an implementation which performs zapping the problem is easily solved since the zapped redexes are exactly those which are being evaluated. Thus a debugger can identify a zap-node with $\perp$. If zapping is not performed, then the relevant redexes can probably be found by traversing the internal stacks and somehow mark the identified redexes. Our implementation performs zapping, in part to make it easy to identify undefined values during debugging.

What makes non-termination slightly different from other execution errors is that the latter occur at well defined points in time and then usually when the graph is in a consistent state. A user interrupt signalling non-termination, on the other hand, can occur at any time, asynchronously, e.g. during garbage collection or in the middle of a destructive update operation, and the graph may then be in a state of flux and highly inconsistent.

An easy solution to this problem is to arrange for a global flag to be set when the user signals that the program should be stopped. This flag is polled at regular intervals, when it is known that the graph is in a well-defined state, and if the flag is found to be set, the execution is terminated. In our implementation this flag is checked just prior to a garbage collection. The response is thus not immediate, but this does not matter much for the purpose of detecting non-termination. The advantage is that the overhead for checking the flag is negligible.[2]

Regarding re-execution, the user would probably find it tedious if it was necessary to hit `CTRL-C` as soon as the debugger decides that it is time to execute the target again. Moreover, in order to guarantee that the desired part of the EDT is created, the execution must not be stopped sooner during a re-execution than during the immediately preceding execution. And to give the impression of working on a stable tree, the re-execution should not be stopped much later either. It is clearly unreasonable to expect such dexterity from the user. The natural solution is to take note of the time at which the target program originally was stopped. This allows the target to be automatically stopped during re-execution at precisely the right moment. We use the reduction (EDT node identity) count as the 'clock' for this purpose, since, for obvious reasons, it is in perfect synchronization with the tree building process, regardless of variations in execution speed due to system load or other external factors.

6.3 Arbitrary starting-points

During debugging, it is quite often the case that the programmer has a fairly good idea about which function(s) that may be faulty, or at least that he has a few hypotheses regarding this which he would like to investigate. The reason may be that these functions just have been changed, apparently breaking a previously working program, or that the behaviour of the program is such that it is reasonable to suspect the functions in question.

An experienced programmer would then, guided by his intuition, focus his attention on the suspect functions, hoping that this quickly will lead him to the source of the problem. If a traditional debugger is being used, then a common method is to set breakpoints at strategic points in the code. Placing them at the start and end of suspect procedures or functions, for instance, allows single-stepping through the suspect parts of the program and the input/output relationship to be examined.

[2] The astute reader may wonder what happens if the program gets stuck in a loop where no memory allocations are performed. This is very unlikely in our implementation, maybe even impossible, but it could be more of a problem in the context of a highly optimizing language implementation. In that case, the flag would have to be checked more often. One approach could be to force a premature call to the garbage collector when an interrupt arrives by making it look as if the heap is exhausted, thus obviating the need to check a separate flag. This technique is used to handle asynchronous interrupts in the New Jersey implementation of standard ML [Rep90][App92, pp. 212-213].

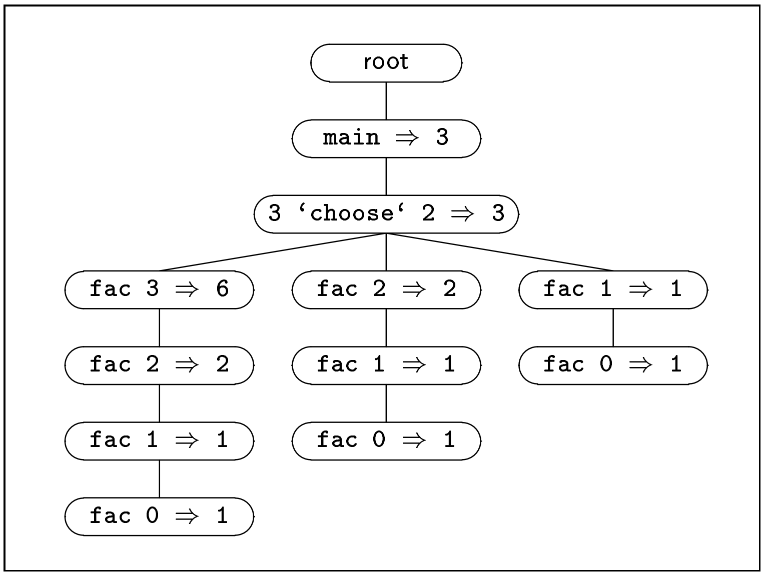


Figure 6.1: The full EDT for the 'n choose k' program.

Similar functionality would be desirable also in an EDT based debugger. Not only would it save the user the trouble of having to start debugging from the root of the entire tree in cases where only a few functions are of any interest, but there is also potential for drastically reducing the number of nodes in the EDT, thereby making better use of the allocated EDT storage and thus reducing the number of re-executions.

We have implemented such a scheme by allowing the user to designate one or more functions as *starting-points*. This has the effect of delaying tree construction until an application of one of the starting-point functions is encountered, resulting in a forest of EDTs which is collected below a special root node. Once tracing has begun, functions designated as starting-points are not treated in any special way. In particular, recursive applications of a starting-point function does not result in separate subtrees immediately below the root.

The following example illustrates the idea. The program below uses the factorial function to compute the number of ways k elements can be chosen among n elements in some set ('n choose k'). The full EDT (ignoring arithmetic operations) is shown in figure 6.1. Note that there is only one CAF, `main`, which is thus the only child of the root.

```
fac 0 = 1
fac n = n * fac (n-1)
```

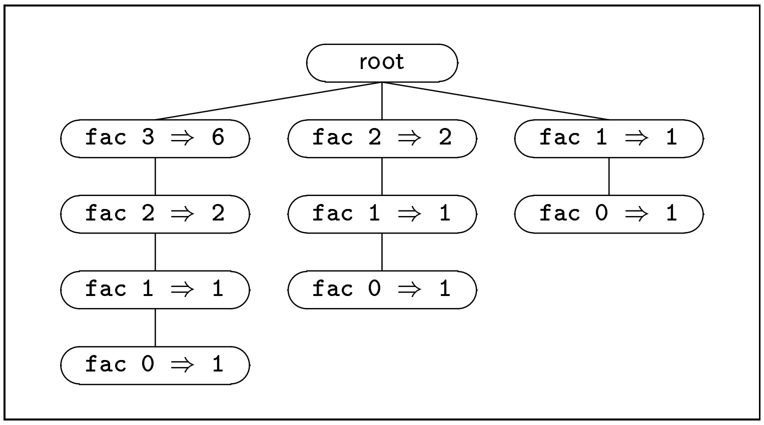


Figure 6.2: The EDT for the starting-point `fac`.

```
n `choose` k = fac n `div` (fac k * fac (n - k))

main = 3 `choose` 2
```

The result of specifying `fac` as the starting point is shown in figure 6.2. The nodes for `main` and `choose` are not present since they are not descendants of any starting-point. Note that recursive applications of `fac` are not special; generally, the subtree rooted at a starting point is identical to the corresponding subtree in the full EDT. While the partial EDT only contains two nodes less than the full EDT in this particular case, it should be clear that the size of the EDT in general can be reduced significantly by designating starting-points.

For this facility to be really useful, we think it is important that the starting-points can be set and reset dynamically, during debugging, just like breakpoints in traditional debuggers, rather than beforehand at compile-time. This allows a user to quickly test various hypotheses regarding suspect functions, without having to recompile the target each time. In our implementation, this is accomplished by regarding the EDT as a function of the user-defined starting-points. This is reflected in the interface between the EDT generator and the debugger where one of the arguments to the function which returns the EDT root is a list of starting points (see B.6).

The implementation of the starting-point mechanism is quite simple. The root node is given a special *id* to signal this mode of tracing. From the target's point of view, *Trace* then behaves as if EDT-nodes are being constructed. But in fact no nodes are constructed until a starting-point is encountered. Instead the *id* returned from *Trace* is the special root *id*. This means that all traced redexes which are not descendants of an application of one of the starting-point functions, will have the special root as their parent.

To put this another way, the special parent *id* is 'inherited' downwards in the tree. Thus, as soon as a traced application of one of the starting-point functions is encountered, and the parent of the traced redex is the special root node, then *Trace* knows that a starting-point has been reached and that this should be the root of a new sub-EDT.

Note that all of this functionality in principle could be implemented completely within *Trace*. In particular, the code generated for supercombinators remains completely unchanged. However, in order to quickly determine whether or not a function is a starting-point during tracing, it is convenient if the compiler adds an extra boolean attribute to the object information record associated with each function, which can be set dynamically by the tracer to indicate that the function is a starting-point.

Could one imagine something akin to conditional break-points? For instance, a possibility to start tree construction when a function is being applied to a certain value or a value satisfying some predicate? Yes, that would be possible. However, a straightforward implementation means that the tracer may have to force the evaluation of certain arguments. This is not desirable unless the function happens to be strict in that argument. It may be the case that the utility outweighs the disadvantage, of course, though it should be emphasized that anything which risks changing the termination properties of the target program is against the spirit of this work.

The alternative would be to somehow delay the test until the argument is evaluated for some other reason, and meanwhile speculatively construct the EDT as if the test had succeeded. Should it then be found that the test really fails, the EDT could be pruned. This seems rather complicated, however, and it is doubtful whether this would lead to any performance gains.

6.4 Avoiding tracing of trusted functions

6.4.1 Trusted functions

Up until now, save for what was said about starting-points in section 6.3, it has been assumed that every single reduction that takes place during execution should be recorded in the EDT. This is clearly wasteful. A large number of these are likely to be applications of well-known and trusted built-in functions (for instance, arithmetic operations like +, *)[3] or functions from the standard libraries (`take`, `drop`, etc.). Since these are likely to be correctly implemented, it is unlikely that the user is going to appreciate being asked

[3] A good compiler tries to inline many of these functions in order to gain efficiency. Wherever this is done successfully, tracing is made impossible and the discussion in this section is irrelevant. However, in a lazy language it is not always possible to inline a function like +. In such cases the techniques discussed here are useful.

about their behaviour. Such reductions should thus not be recorded in the EDT in the interest of making the best possible use of the allocated storage.

Moreover, large systems are usually built modularly. So, when a new bug is discovered, a common scenario is that there are a number of well-tested, trusted modules, and a few prime suspects which recently have been changed. Again it would be advantageous if tracing of functions from trusted modules could be minimized.

Shapiro also makes this observation in his seminal paper on algorithmic debugging [Sha82], and suggests that procedures that have been declared to be 'correct' should not be traced. Shapiro makes use of this to avoid tracing built-in primitives in a sample implementation. However, Shapiro worked in the context of Prolog, and since Prolog only supports first-order procedures, it is fairly easy to see how those assumed to be correct could be handled during tracing: simply ignore them, and thus also any subordinate calls made.

One of the distinguishing features of functional languages, on the other hand, is that functions have the same status as non-functional values and that functions thus can be higher-order. In fact, the latter are ubiquitous and indispensable to many advanced programming techniques. As a consequence, it is not uncommon that it is statically unknown which function is being applied. For instance, consider the following (somewhat contrived) examples:

```
foo f x = f (f x)

fie n x y = (funList !! n) x y
```

Here, `foo` is an ordinary higher-order function which applies its first argument `f` twice to its second argument `x`. In `fie`, the first argument `n` is used to select a function from a global list of functions, `funList`, which is then applied to the two remaining arguments `x` and `y`. While it in some cases, through advanced static analysis, may be possible to determine exactly which function or functions `f` could be bound to and `(funlist !! n)` could evaluate to, this is in general an undecidable problem. Note that `fie` is a first-order function and that its type thus does not reveal that an unknown function is being applied in its body.

The fact that unknown functions can be applied raises the question what it means exactly to 'trust' a function. Does the trust extend to all functions which could be applied in a function like `foo` above, for instance? Indeed, should trust extend by transitivity to *any* other function applied in the body of a trusted function? If not, then it is clear that trusting a function does not necessarily mean that it can be ignored by the tracer since it may be necessary to trace subordinate applications of functions lest some erroneous reduction should go by unnoticed.

We take the view that the trust only applies to the actual source code of a function. Thus, if a function applies untrusted or unknown functions, then it must still be traced in order that subordinate calls may be traced. Note that there is no trust by transitivity: trusting a function does not imply that functions applied in the body becomes trusted, not even if they are statically known. We think this is a natural approach. It is quite reasonable to trust a function like `map`, for example, but it is not reasonable to expect that `map` will only be applied to correct functions. Furthermore, the meaning of trust also becomes very clear: 'there is no bug in the body of this function (but there may well be bugs in functions called from it).'

6.4.2 Trace classes

We now introduce the notion of *trace classes*. A trace class prescribes how a function (more precisely, a supercombinator) should be traced, and each function in the target belongs to exactly one trace class. Since trace classes are closely related to the inner workings of the tracer, users do not have to confront them directly. Instead functions are automatically assigned to trace classes based on static properties of the function and static and dynamic assertion of trust (as discussed in the previous section) supplied by the user and, in some cases, by the compiler (see section 6.4.3).

There are four trace classes in our system, as listed below. This is not to say that there are not other possibilities, but these are the ones that we have found useful.

- *Untraced.* The function is completely untraced. Neither calls to it, nor any subordinate function calls, are recorded in the EDT.
- *Traced.* Ordinary traced function. Calls to it as well as subordinate function calls are recorded in the EDT.
- *Invisible recursion.* The function is traced, but no EDT nodes are constructed for recursive calls.
- *Invisible.* The function is traced, but no EDT nodes are constructed for calls to it. Subordinate calls are however recorded as usual.

The first two are relatively obvious. Functions belonging to the class *traced* are handled in the standard way as described in chapter 5. Functions in the class *untraced* are not traced at all, which means that entire subtrees rooted in nodes recording reductions of such function applications are missing compared with a complete EDT. The last two will be motivated and explained through two examples.

Consider the following program:

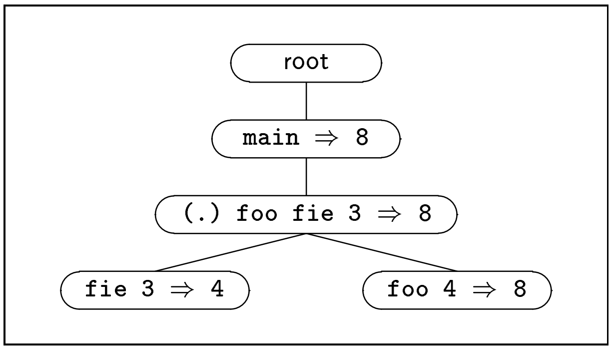


Figure 6.3: The complete EDT for the function composition example.

```
(.) f g x = f (g x)

foo x = 2 * x

fie x = x + 1

main = (foo . fie) 3
```

The function `(.)` is the function composition operator. Thus `(foo . fie)` is a function which computes $2(x + 1)$. The complete EDT for the program is shown in figure 6.3.

Now, suppose that we decide to trust the function `(.)`. (This is reasonable since it typically is one of the standard functions.) The EDT node which records the reduction of `(.) foo fie 3` is then not very interesting. Should the result be wrong, then the reason must be that `foo` or `fie` did not work as expected, and it would be better if the debugger asked directly about the applications of these two functions, i.e. if the function composition node was eliminated and its two children instead were children of its parent. This is the purpose of the trace class *invisible*. Reductions of functions in this class are not recorded in the EDT, but children of such reductions are traced in the normal way and, if necessary, adopted by their grandparent. The resulting EDT is shown in figure 6.3.

The last trace class is *invisible recursion*. As the name suggests, it is intended for tracing trusted recursive functions. The idea is illustrated by the program below.

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

foo x = 2 * x
```

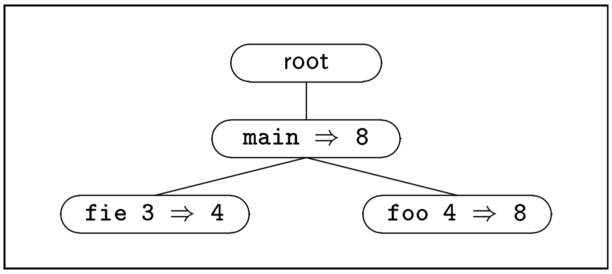


Figure 6.4: The EDT when `(.)` belongs to the trace class *invisible*.

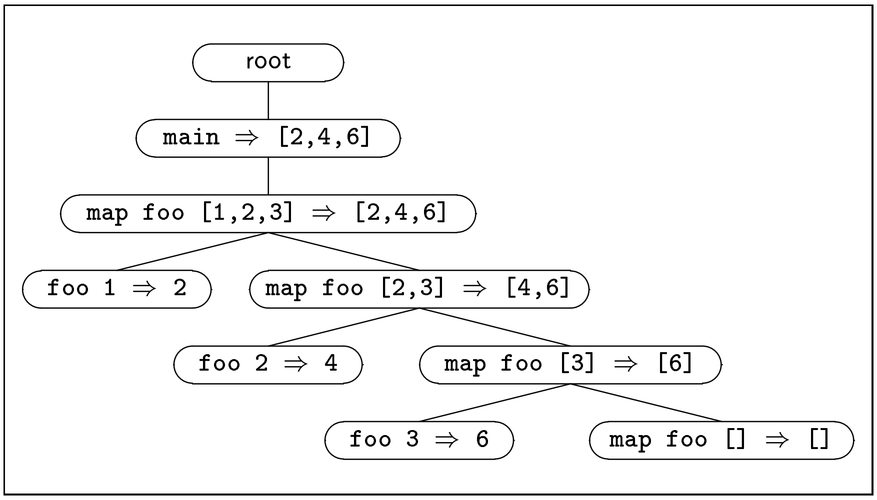


Figure 6.5: The complete EDT for the `map` example.

```
main = map foo [1,2,3]
```

In this example, the well-known function `map` is used to apply the function `foo` to each element in the list `[1,2,3]`, yielding the list `[2,4,6]`. The complete EDT, when no functions are trusted, is shown in figure 6.5.

If `map` is trusted, then, by the same reasoning as in the previous example, the nodes recording the applications of `map` are not interesting. Thus these could be eliminated and the orphaned nodes could be adopted by the node recording the reduction of `main`. In general, however, a recursive function can carry out quite a lot of work, which means that the parent of a trusted recursive function may have to adopt a lot of orphaned children. In our example, consider what would happen if the list contained 1000 elements instead of three. This is not good from a debugging perspective since a lot

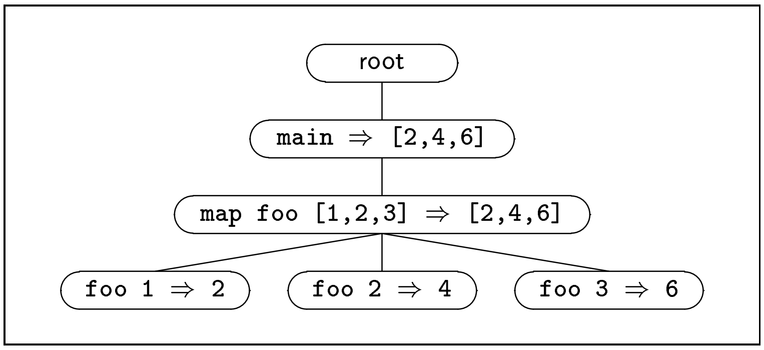


Figure 6.6: The EDT when `map` belongs to the trace class *invisible recursion*.

of unnecessary questions would be asked in case the recursive computation actually yielded the right result.

A better approach is to keep the node recording the first application of the recursive function as a kind of summary node for the entire subcomputation. Nodes corresponding to *recursive invocations*, on the other hand, are eliminated just as before, and any orphaned nodes are adopted by their closest ancestor, which in this case is the summary node. This is how functions in the class *invisible recursion* are handled. Figure 6.6 shows the resulting EDT for our example.

Note that only one question now has to be answered in order to verify that the subcomputation is correct. Should the subcomputation be incorrect, then only one extra question has to be answered in order to find the erroneous node compared with what would have been the case if there had not been any summary node. Furthermore, for this particular example, the number of nodes in the subtree for `map` is reduced by half asymptotically, and so is the number of questions that on average would have to be answered in order to find an erroneous application in this subtree.

While the two examples above concerned higher-order functions, the trace classes *invisible* and *invisible recursion* are equally useful when untrusted functions are applied in the body of a trusted function. There is no difference between these cases.

6.4.3 Trace class inference

The previous section introduced trace classes and showed how they are used to guide the tracing process. However, the concept of trace classes is likely to be unfamiliar to most users, who presumably would prefer to convey their intuition and knowledge regarding where the bug might be located in terms of whether or not they trust certain functions or modules. This section discusses how assertions of trust in combination with static properties of functions are used to attribute trace classes to functions in our system.

Our system currently allows the user to express his trust in functions in two ways. First, the user may put the pragma `{-# trusted #-}`[4] into the source code for a module to *statically* state that all functions in the module should be trusted. (There is currently no support for statically stating trust for individual functions.) Second, the user may also *dynamically*, during debugging, assert that further functions should be trusted on a per function basis. On top of this, the compiler may assert that functions which are introduced as a result of various translation steps should be trusted.

The static assertions are used by the compiler to compute the *initial trace class* for the functions as described below. This attribute is stored in the object description record associated with each function, and later used to initialize the *effective trace class* when a debugging session is about to begin. It is the effective trace class of a function that actually guides the tracing, and it is this attribute that is updated if further correctness assumptions are made while the program is being debugged. However, if the initial trace class is *untraced*, then normal, uninstrumented code is generated for this function in the interest of efficiency (see section 6.4.4). Thus it is not possible to change the effective trace class of such functions during debugging, should the user have second thoughts about his static assertions for instance.

In order to compute both the initial and effective trace class, two auxiliary boolean attributes are used: *recursive* and *must trace*. These are also stored in the object information record and thus available to the debugger for computing the proper effective trace class when additional correctness assumptions are made dynamically.

The attribute *recursive* is set to true for recursive functions. This property is manifest in the internal representation used by our compiler (non-recursive functions are `let`-bound and recursive functions are `letrec`-bound[5]), and thus does not require any additional analysis.

The attribute *must trace* is set to false only if it can be statically proved that no traced functions are applied in the body of the function. Otherwise, if the function applies an unknown function for example, it is set to true. Note that 'traced' here refers to functions that belong to any of the classes which require tracing, i.e. *traced*, *invisible recursion*, or *invisible*. In order to make this analysis possible across module boundaries, function declarations in the interface files are associated with a flag which indicates whether the function in question is traced in this sense.

For (mutually) recursive functions, it is initially assumed that all func-

[4]The tokens `{-#` and `#-}` are used do delimit pragmas in Haskell and Freja.

[5]This would not necessarily be true if mutually recursive modules are compiled separately. However, assuming a `let`-bound function to be non-recursive in that case is probably acceptable.

tions in the group are untraced. A tentative *must trace* attribute is then computed individually for each function in the group, just as above. Finally, the definitive *must trace* attribute is computed as the disjunction of the tentative *must trace* attributes, and then used for all functions in the group. Thus *must trace* is false only if no function in the group applies any traced function outside of the group.[6]

Given the attributes *recursive* and *must trace*, the initial trace class for a function is computed as follows. Note that a function is trusted if it belongs to a trusted module or if the compiler has decided that it trusts it.

if *trusted* **then**
  if *must trace* **then**
    if *recursive* **then**
      initial trace class = *invisible recursion*
    else
      initial trace class = *invisible*
  else
    initial trace class = *untraced*
else
  initial trace class = *traced*

The tracer computes the new effective trace class in the same way when the user, during debugging, asserts that a function is to be trusted. This is why the attributes *trusted* and *must trace* are stored in the object information record. Since dynamic trust assertions affect the structure of the EDT, the list of trusted functions is an argument of the tracer function which returns (the root of) the EDT, just like the list of starting-points (seeB.6).

Note that there is scope for more accurate inference. For instance, if `map` is trusted and it is statically known that `map` is applied to an untraced function, then there is no need to make this particular redex traced. This is discussed further in section 12.2.3. However, while statically avoiding creating traced redexes is certain to reduce the tracing overhead, it is not evident that the improvement would be substantial. In fact, since the creation of most of the EDT nodes in question is prevented in any case through dynamic mechanisms, it may well be that the performance improvement would not be significant.

6.4.4 Implementing trace classes

The mechanisms that are needed for implementing the trace classes are quite simple. In particular, no extra support is needed in the generated

[6] The debugger should be aware of this when dynamically assigning new effective trace classes to functions. Trusting only some of the function in a mutually recursive group can lead to somewhat obscure effects.

code, except a few extra attributes in the object description records for the functions as described in section 6.4.3 and below.

The trace class *untraced* is, unsurprisingly, the one which is the simplest to implement. In case it is statically known that a function belongs to this class (i.e., the initial trace class is *untraced*), the compiler can generate ordinary, uninstrumented code for it. There is no overhead for tracing. This has the desired effect of stopping the recording of any reductions of applications of the function, as well as completely disabling tracing of subordinate function calls since no traced redexes are built. An additional benefit of statically knowing that a function is untraced is that no applications of it need to be traced, thus reducing the tracing overhead for traced functions which call untraced ones.

In case a function is assigned dynamically to the class *untraced*, the routine *Trace* will still be called. If *Trace* finds that the invoked function is untraced, it simply does not create an EDT node for the reduction and returns the special value *nt* (no tree) that disables tracing (see section 5.4.4).

From the perspective of the instrumented code, the three other trace classes are identical and the code behaves exactly as described in section 5.4.4. The responsibility for tracing functions in these classes in the correct way rests with the tracer. *Trace* handles functions belonging to the class *trace* as described in section 5.4.4. When *Trace* encounters a function in the class *invisible* it simply avoids creating an EDT node for the reduction. Tracing is not disabled, however. Instead, the *id* of the parent of the reduction is returned. This means that any traced redexes which are constructed will refer to the parent. Conceptually, the effect is that orphaned children are adopted by their grandparent.

The final trace class, *invisible recursion*, is slightly more involved. When *Trace* is invoked for a function is this class, the parent of the redex being reduced is looked up. If the parent is an application of a function which does not belong to the same group of mutually recursive functions as the invoked one, then this must be the first invocation in a series of recursive calls. Thus the summary node should be created and *Trace* therefore behaves just as if the function had belonged to the class *traced*.

On the other hand, if the parent records an application of a function belonging to the same group of mutually recursive functions, then this must be a recursive invocation and the parent node must be the summary node. Thus no new EDT node is created and the *id* of the parent node is returned. This results in the summary node adopting any orphaned children.

In order to quickly test membership in groups of mutually recursive functions, each such group is assigned a globally unique number. This number is stored as an attribute in the object information records for the functions. By comparing these numbers it is possible to quickly determine whether two different functions belong to the same mutually recursive group or not.

Note that the groups in many cases contain only a single function.

A slight complication in a system supporting separate compilation, is that the group numbers cannot be assigned during compilation of individual modules since they must be globally unique. We solved this by letting the RSG (Runtime System Generator) assign these numbers. The RSG must perform global dependence analysis and identification of strongly connected components (i.e. groups of mutually recursive functions) in any case to handle garbage collection of CAFs (see 9). Numbering these components as they are encountered thus required minimal extra implementation effort.

See appendix B.4 for further implementation details regarding trace classes.

Chapter 7

Source-Level Debugging

In the previous two chapters, we have developed techniques for debugging lazy functional programs at the supercombinator level. However, translating Freja or Haskell source code into supercombinators requires substantial transformations. Moreover, for performance reasons, the employed transformations are often not those which are the most intuitive. Hence the resulting code is often far removed from the original source program, making it difficult to understand what goes on during execution in source-level terms. To facilitate debugging, this gap must be bridged as far as possible.

An obvious approach for dealing with the problems introduced by advanced compilation techniques, is to side-step the whole issue by employing simpler transformations and turning off optimization when compiling for debugging. This should be acceptable, as long as the simplistic compilation does not lead to such inefficiencies that the program runs unacceptably slow or not at all. Naturally, the approach presupposes that the compiler is correct, or at least that the semantics of the program to be debugged is not affected by the changed compilation strategy.

This chapter discusses two kinds of source-level constructs, local functions and list comprehensions, and shows how to translate them into supercombinators in a way which supports source-level debugging. The technique used is a combination of selecting a suitable, possibly simplified, compilation strategy, and associating the generated supercombinators with debugging information.

In order to debug full Haskell, there are additional issues to consider, e.g. the special `do`-notation for monadic programming[1] and type classes. This

[1] The `do`-notation is a layer of syntactic sugar which makes monad-structured programs easier to write and read [Wad92, PHA+97]. The main problem here is not the notation as such, but rather that monadic code often is intrinsically difficult to debug since complex functional values typically are passed around. Due to the popularity of the monadic programming style, this topic is important regardless of whether a functional language

is further discussed in sections 12.2.6 and 12.2.7. However, Freja neither have general, user-definable type classes, nor does it currently support the `do`-notation (see section A.1), so the discussion in this chapter covers Freja and languages similar to Freja in this respect (e.g. LML and Miranda) fairly well.

7.1 Local functions

One of the distinguishing features of functional languages is that functions have the same status as other kinds of value. Functions may, for example, be passed as arguments, returned as results, or embedded in data structures. Furthermore, functional languages allow local function definitions, where free variables may occur in the function bodies, and often also lambda abstractions (anonymous functions). This has far-reaching consequences both for implementation and debugging since it necessitates keeping track of free variable bindings by more sophisticated means than a call stack.

This section discusses the implications this has for debugging, and shows how a compiler can provide adequate support to an EDT-based debugger by selecting a suitable compilation strategy and embedding debug information in the generated code. While the setting of the main discussion is an implementation which handles local functions by lambda-lifting, the basic ideas are quite general.

7.1.1 Closures and debugging

Since functional values are treated just like other values in a functional language, it is essential that functional values are self-contained: there must be no references to auxiliary values which could become invalid. This is usually solved by representing functional values by an association between the function definition and a list of bindings for any free variables in the body. This is called a *closure*. While it in some cases may be possible and beneficial to choose other alternatives (for example, a Pascal-like approach, where the values for the free variables can be found on the stack via static links or similar mechanisms), the following discussion will only treat the closure representation since it is always applicable: a highly optimizing compiler can always revert to a basic closure representation for debugging purposes.

Closures are ubiquitous and will inevitably turn up during debugging, both in the function position of applications and as arguments or results. As discussed in section 4.1.2, the EDT must therefore record the free variable bindings in order that they are available to the user during debugging.

has any special notational support or not. Section 12.2.6 discusses this a bit further.

7.1.2 Hiding the effects of lambda-lifting

The approach employed to handle local functions in G-machine-based implementations of lazy functional languages is lambda-lifting, where free variables are abstracted out as extra arguments to make it possible to define all functions at the top level in the form of supercombinators. This was described in section 2.4.3. As a result, a closure is represented by a partially applied supercombinator.

There are of course other possibilities for handling local functions and representing closures. What is important from a debugging perspective is that it must be possible to interpret heap-allocated closures. This is easy in principle: the compiler just has to make the name of a function and the names of its variables part of the debug information associated with the function, and make sure that this information can be accessed from a closure.

In an implementation based on lambda-lifting, it is natural and convenient to delegate the task of attributing the generated supercombinators with appropriate debugging information regarding the name of the function and the names of its free variables to the lambda-lifter. The reason is that it is the lambda-lifter that creates the supercombinators and thereby determines the layout of the closures, i.e. the correspondence between the extra arguments and the free variables. However, since lambda-lifting typically is performed late in the compilation process, it may be necessary to preserve the original (source-level) names of various entities as extra attributes in the internal representation.[2]

In our implementation, the lambda-lifter provides the following attributes to the object information record of a supercombinator.

- *Debug name.* The function name as presented to the user during debugging. It is typically the original name of the function, but for supercombinators which do not correspond directly to any named source-level function, a different strategy must be used. See section 7.2 for instance.

- *Qualifiers.* In order to disambiguate between functions having the same name in different modules, or between local functions having the same name, a list of qualifiers is associated with each supercombinator. The qualifiers are simply the name of the module and the names of any enclosing functions.[3]

[2]In the current Freja implementation, local variables are for instance renamed early to make them distinct within a module since this facilitates various transformations.

[3]The syntax of Freja and Haskell is such that this is not quite enough to guarantee globally distinct names. However, it is good enough in most situations. In any case, there is a source code reference associated with each supercombinator which can be used for further disambiguation if necessary.

- *Arity.* The arity of the supercombinator. The tracer must know the arity in order to copy the right number of arguments from the spine stack when it creates an EDT node (see 5.4.4). Furthermore, knowing the arity means that the debugger can tell the difference between a redex and an unsaturated (partial) application. This is useful since an unreduced redex have not affected the computation. Thus its representation cannot be relevant, and the redex can be presented to the user using some special symbol meaning 'unevaluated expression' or 'irrelevant value', thus sparing the user unnecessary details.

- *Number of extra arguments.* The number of extra arguments introduced by transformations such as lambda-lifting. It is assumed that these are the first arguments of the supercombinator, and the debugger uses this information to avoid printing these extra arguments as normal function arguments. Note that there may be other reasons for introducing extra arguments. Haskell's type classes are normally implemented by adding extra dictionary arguments to functions which use methods (overloaded functions), for instance [WB89, Aug93b]. This mechanism is thus a general way of hiding compiler-introduced arguments from the user.

- *Free variables.* This is a list associating the name of each free variable with the position of the corresponding argument in the argument list of the supercombinator. An empty free variable list indicates that there were no free variables.

Consider the code fragment above again, but now assume that `foo` is defined in the module `Main`.

```
module Main where

foo n xs = map fie xs
    where
        fie x = n * x
...
```

Lambda-lifting would transform this into something like the following:

```
module Main where

FOO n xs = MAP fie xs
    where
        fie = FIE n
FIE n xs = n * x

...
```

In the example above and in the following, we follow section 2.4.3 and give the supercombinators names in capitals. These names correspond to internal, system generated names.[4] Notice the partial application of `FIE` in the body of `FOO`. This creates a closure which binds the free variable `n` in the original definition of `fie`. This closure is then passed as an argument to `MAP`, the supercombinator version of `map`.

The attributes in the object information record of the supercombinator `FOO` relevant to this discussion would be as follows (Haskell notation):

- *Debug name*: `"foo"`
- *Qualifiers*: `["Main"]`
- *Arity*: `2`
- *Number of extra arguments*: `0`
- *Free variables*: `[]`

The attributes for `FIE` would be:

- *Debug name*: `"fie"`
- *Qualifiers*: `["Main", "foo"]`
- *Arity*: `2`
- *Number of extra arguments*: `1`
- *Free variables*: `[("n", 1)]`

(Recall that the free variable list associates the name of each free variable with the position of the corresponding argument in the argument list.)

Consider the two reduction steps below. During debugging, each would be recorded in the form of an EDT node.

```
MAP (FIE 2) [1,2,3]  ⇒  [2,4,6]
FIE 2 1  ⇒  2
```

[4] In practice, these names must often be globally unique and their syntax in the end acceptable to a C-compiler or assembler. This may necessitate naming conventions where the internal names bear little resemblance to the original ones, and thus are not suitable for debugging purposes. This is one reason for having the debug name as an attribute in the object information record. Besides, accessing the internal names from the debugger requires knowledge about the object file format. This would make the system more platform dependent.

Given the debug information in the object information records as described above, the debugger can present the questions corresponding to the above reduction steps in a form closely related to the source code. An example is shown below. Here, we are assuming that the debugger prints function names fully qualified. In practice this may not always be necessary. A debugger could opt for printing the qualifiers only when there is another function with the same debug name, for example.

```
Prelude.map (Main.foo.fie where n = 2) [1,2,3]
=>
[2,4,6]
Yes/No?

Main.foo.fie 1 where n = 2
=>
2
Yes/No?
```

7.1.3 Selecting the lambda-lifting strategy

The previous section outlined one way of dealing with free variables, lambda-lifting. It also explained how to exploit the lambda-lifter for providing debugging information which makes it possible to present closures in source-level terms. Implicitly, it was assumed that the free variables the lambda-lifter abstracted out as extra arguments coincide with those the user needs to refer to during debugging. However, as exemplified below, this is not necessarily the case. Since there are several ways to perform lambda-lifting, the question is whether there is some lambda-lifting strategy which perhaps is more suitable than others for debugging purposes. We will look at two strategies from this perspective. One is presented in Peyton Jones [PJ87, pp. 220–238]. The other is due to Johnsson [Joh85].

Up until now, we have argued that it is necessary to tell the user about the bindings of free variables and justified this through several examples. However, this is not quite the whole truth since it is often manifest that a free variable is bound to a particular value. Showing bindings for such variables is just restating the obvious and does not aid debugging. (Indeed, we have already silently ignored free variables which are bound at the global level, e.g. standard functions such as `map`.) A user may even find the unnecessary details distracting. This is particularly true in the case of local, auxiliary functions. Consider the following code fragment:

```
foo xs = fie xs ++ fie xs
    where
        fie xs = map fum xs
        fum x = 2 * x
```

Here, fum is strictly speaking a free variable in the body of fie. However, it is obvious what fum is bound to, and explicitly stating the binding of fum in questions regarding fie, as in the example below, does not add any necessary information. (It is assumed that foo is defined in the module Main.)

```
fie [1,2,3] where fum = Main.foo.fum
=>
[2,4,6]
Yes/No?
```

In practice, the above example may not be problematic since a well-implemented lambda-lifter would avoid taking out fum as an extra argument and instead make the supercombinator version of fie refer directly to the supercombinator version of fum. But it illustrates the problem and shows that the employed lambda-lifting strategy has repercussions on the format of the asked questions. For a more interesting example, consider the following:

```
foo n x y = fie n
    where
        fie 0 = []
        fie n = x : fum (n - 1)

        fum 0 = []
        fum n = y : fie (n - 1)
```

Note that the free variables in fie are x and fum, whereas y and fie are free in fum. Suppose the lambda-lifting strategy described in Peyton Jones is used. Applying the lambda-lifter to the code above would then result in the following supercombinators. Again, the supercombinators are given names in capitals. The debug names of the supercombinators are given as comments.

```
-- Main.foo
FOO n x y = fie n
    where
        fie = FIE fum x
        fum = FUM fie y

-- Main.foo.fie
FIE fum x n = x : fum (n - 1)

-- Main.foo.fum
FUM fie y n = y : fie (n - 1)
```

Notice that the supercombinator FOO has a graphical, recursive body where the closure fie refers to the closure fum and vice versa. If we assume that the expression `foo 5 1 2` has been evaluated, the user may encounter questions like the one below during debugging.

```
Main.foo.fie 5
    where
        fum = Main.foo.fum
            where
                fie = Main.foo.fie
                    where
                        fum = Main.foo.fum
                            where
                                ...
                        x = 1
                y = 2
        x = 1
=>
[1,2,1,2,1]
Yes/No?
```

Since `fum` and `fie` are bound to circular values, the debugger must truncate their textual representation somewhere. This is indicated by the ellipsis above. Since all free variables (except those defined at the top-level) are abstracted out as extra arguments, the net result is quite similar to the EDT generating semantics presented in section 4.4. Compare in particular section 4.4.3.

This looks really complicated, and since the bindings of `fie` and `fum` again are manifest in the source code, it would be desirable to avoid showing these bindings. One approach might be to identify 'uninteresting' free variables through some syntactic criterion, e.g. by virtue of being let-bound rather than lambda-bound. Uninteresting free variables could then be omitted from the list of free variables associated with a supercombinator. This would effectively hide uninteresting bindings from the user. However, a closer inspection reveals that this is not as straightforward as it may first seem, since it is necessary to take free variables of the hidden bindings into account. For example, a question like

```
Main.foo.fie 5
    where
        x = 1
=>
[1,2,1,2,1]
Yes/No?
```

cannot be answered unless it is also stated what `y` is bound to.

Thus we do not pursue this approach further, but instead turn our attention to the lambda-lifting strategy proposed by Johnsson [Joh85]. This approach avoids abstracting out functions as extra arguments, which seems attractive for our purposes. In particular, local, (mutually) recursive functions, as in the example above, are turned into (mutually) recursive supercombinators, thus in many cases avoiding the creation of circular closures for handling recursion.

Using Johnsson's approach, the example above would be translated into the following supercombinators:

```
-- Main.foo
FOO n x y = FIE x y n

-- Main.foo.fie
FIE x y n = x : FUM x y (n - 1)

-- Main.foo.fum
FUM x y n = y : FIE x y (n - 1)
```

Notice that the supercombinators `FIE` and `FUM` now are recursive unlike in the previous translation, and that only known functions are applied in this case. The effect of this translation on our example question is shown below. This version is a bit easier to understand than the version with the circular closures.

```
Main.foo.fie 5
    where
        x = 1
        y = 2
=>
[1,2,1,2,1]
Yes/No?
```

The rationale behind Johnsson's strategy is that it increases the number of applications of known functions. This is good since it is often possible to optimize such function calls. In our case there is an added benefit since the trace class inference (see section 6.4.3) yields more precise results when known functions are applied. In comparison with the other lambda-lifting strategy, the possibility of identifying untraced functions and untraced redexes increases which leads to smaller tracing overhead. Johnsson's lambda-lifting algorithm therefore seems as a suitable choice for debugging.

As always, the implementor could opt for another approach when not compiling for debugging, should that offer better performance or other advantages. For example, one might want to use *fully lazy lambda-lifting* [PJ87, pp. 243–259] in order to compile code which is as lazy as possible. This version of lambda-lifting is likely to be unsuitable for debugging since it can break one function into several supercombinators, most of which do not correspond to any named functions in the source code.

7.1.4 Lambda abstractions

Lambda abstractions (or anonymous functions) constitute a special problem since they do not have a user-supplied name by which they can be referred during debugging. In addition, lambda abstractions feature prominently in monadic and continuation passing programming styles where they are used in a highly idiomatic way, often as a kind of assignment statement. Arguably, such lambda abstractions should be handled in a way reflecting the idiomatic use rather than as plain, ordinary functions. However, we have not investigated debugging support for special programming styles in any depth, so we leave that aspect of the problem for future work (see section 12.2.6). Nevertheless, a debugger needs some way of handling lambda abstractions. Some simple options are outlined in the following.

A rudimentary approach is to give debug names to the lambda abstractions in terms of their source code. This will at least allow debugging of programs containing lambda abstractions. The following example illustrates the idea.

```
foo xs = map (\x -> fie x * fum x) xs

fie x = x + x

fum x = x + 10
```

Performing lambda-lifting on the code fragment above yields the supercombinators below. The debug names are given as comments.

```
-- Main.foo
FOO xs = MAP L42 xs

-- \x -> Main.fie x * Main.fum x
L42 x = FIE x * FUM x

-- Main.fie
FIE x = x + x

-- Main.fum
FUM x = x + 10
```

An excerpt from an imagined debugging scenario is shown below. The user's answers are shown in italics.

```
Prelude.map (\x -> Main.fie x * Main.fum x) [1,2]
=>
[42, 42]
Yes/No? no

(\x -> Main.fie x * Main.fum x) 1 => 42
Yes/No? no

Main.fie 1 => 6
Yes/No? yes

Main.fum 1 => 7
Yes/No? no
```

As an alternative, a lambda abstraction could be assigned an arbitrary name suggestive of its origin (e.g. <lambda42>). This might be suitable if the lambda abstraction is textually large. When such a function is encountered during debugging, the debugger would simply direct the user to the appropriate location in the source code using the source code reference in the object information record associated with the function. This should enable the user to find out what the function is supposed to do.

In some cases, it is possible to side-step the problem through trivial program transformations. For example, consider the two functions below:

```
foo x y = \z -> x + y + z

fie x y = (\z -> z * z) (x + y)
```

The following two definitions are equivalent to the two above, but there are no explicit lambda abstractions.

```
foo x y z = x + y + z

fie x y = let z = x + y in z * z
```

Of course, this is really just a syntactical game, but the transformations are representative of what a compiler does during normal compilation, and if a transformation eliminates a syntactical occurrence of a lambda, it may be a good idea to perform that transformation also when compiling for debugging. Having said that, a user might be mildly surprised by the effects of the transformations. If `foo` is defined as in the first case, it may seem reasonable to expect `foo` to return a lambda abstraction when applied to two arguments, for instance.

Lambda abstractions are sometimes introduced by the compiler as a result of various transformations. It may then be the case that these cannot possibly contain any bugs (if the compiler is assumed to be correct). Such abstractions could be hidden from the user by assigning them to the trace class *invisible* or *untraced*. Inlining of certain known functions is one example. Take function composition `(.)`, for example, which a compiler could inline as follows:

```
f . g  ⇒ \x -> f (g x)
```

Pattern-matching translation offers several other examples. The Freja compiler performs the following translation. `E` stands for some arbitrary expression.

```
(x:_) = E  ⇒ x = (\(x:_) -> x) E
```

In both cases, our compiler hides these lambda abstractions from the user by assigning them to appropriate trace classes since it is evident that their bodies do not contain any bugs.[5]

7.2 List comprehensions

List comprehensions, as any other syntactic sugar, do not add any fundamental power to a functional language and are easily transformed into ordinary (recursive) functions as a part of the compilation process. However, they do increase the expressive power considerably, and help in writing concise and clear programs. Hence list comprehensions are used extensively. As a convenience for the user, it is therefore desirable that a debugger somehow handle list comprehensions so as to allow debugging in terms of the source code rather than some transformed version of it. This section presents one way of doing this. At the time of writing, the proposed scheme has not been integrated into our system, but it has been tried in an earlier prototype implementation.

7.2.1 Introduction to list comprehensions

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

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

where `E` is an arbitrary expression and each *qualifier* $\mathtt{Q}_i$ 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. Appendix C deals with list comprehensions of a more general form.

[5]Note, however, that pattern matching could be an interesting event for debugging. See section 12.2.5 for a proposal for an alternative treatment.

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 used in the expression `E` or in any qualifier to the right of the one that introduced the variable.

Let us illustrate with some examples. 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 the last generator changes most rapidly.

For a final example, here is an implementation of the quicksort algorithm. It uses list comprehensions for partitioning the list to be sorted into two sublists containing the elements which are respectively smaller and larger than a pivot element, in this case the head of the list, bound to the variable `x`.

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

7.2.2 Why list comprehensions ought to be debuggable

As the examples in the previous section show, list comprehensions may, in a sense, be regarded as the functional equivalent of loops, permitting iteration over lists.[6] This actually suggests one way of handling them for debugging purposes. If one is content with locating bugs at the granularity of source-level functions, then 'internal iteration' could be ignored as long as all function applications performed inside the 'loops' are properly traced. Since

[6] In Haskell 1.4 [PHA+97], list comprehensions have been generalized to *monad* comprehensions, which means that the loop analogy is less appropriate. However, it is likely that monad comprehensions will revert to being list comprehensions in future versions of Haskell in the interest of simplicity.

list comprehensions are translated into recursive functions, this is easily accomplished using the techniques described in section 6.4: the compiler simply has to put the generated functions in the trace class *invisible*.

However, a list comprehension may carry out substantial computations. This means that an EDT node recording the call to a function which makes use of list comprehensions may end up having a very large number of children, in particular if the list comprehensions are nested. Finding an erroneous node under such circumstances could be tedious. The problem really is that debugging on the level of function calls is to coarse grained in a case where the body of a function is 'computationally large', and it is thus desirable to make the inner structure of such functions more visible to the user. Compare the discussion on recursive functions and the trace class *invisible recursion*.

Furthermore, a list comprehension, unlike an imperative loop, has a simple declarative meaning: it denotes a list, i.e. an ordinary value. List comprehensions are therefore well suited as debugging units under a declarative debugging scheme.[7]

7.2.3 Translation for debugging

We will now define the meaning of list comprehensions formally, and show how they may be translated into nested function definitions in a way which is suitable for debugging purposes. The presentation is closely based on Peyton Jones [PJ87, pp. 127–138]. The given translation is not the best one possible, but the performance loss is not unacceptably high. There is of course no need to use anything but the best possible translation scheme when a program is not being compiled for debugging.

We begin by defining reduction 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]`	⇒	`[]`
(ii)	`[E	v <- E':L', QS]`	⇒	`[E	QS][v↦E'] ++`
			`[E	v <- L', QS]`
(iii)	`[E	False, QS]`	⇒	`[]`
(iv)	`[E	True, QS]`	⇒	`[E	QS]`
(v)	`[E	]`	⇒	`[E]`

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

[7] In her work on algorithmic debugging for imperative languages, Shahmehri suggests that loops should be units of debugging for basically the same reasons as those outlined here [Sha91]. She also shows how this can be done.

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 transformed 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, the comprehension

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

is translated into the function

```
f1 []     = []
f1 (v:l') = [E | QS] ++ f1 l'
```

where `f1` is a new name, and the list comprehension is then replaced by the expression `f1 L`.

The whole translation scheme must then be applied recursively to eliminate `[E | QS]` from the definition of `f1`. The function that results from this should be defined in a `where`-clause within `f1` 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 the comprehension

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

where `B` is a boolean predicate, is translated into the function

```
f2 False = []
f2 True  = [E | QS]
```

and the whole comprehension is then replaced by `f2 B`. Again, `f2` is a new name. The translation must then 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]`.

While the translation scheme given above is not optimal in terms of the number of list cells created [PJ87, pp. 133–136], it does have the merit that it is very easy to map back the generated functions to the original source constructs. This is important for debugging purposes: questions involving functions introduced by the compiler are unlikely to be meaningful to the user. For instance, looking back at the definition of `f1` above, it should be clear that `f1 L'` denotes the same list as the list comprehension `[E | v <- L', QS]` for an arbitrary list `L'`. But since `f1` is a generated, internal name, unknown to the user, the latter representation is preferable to the former.

Source-level debugging of list comprehensions can be achieved by giving special 'debug names' to the generated functions in terms of the corresponding comprehensions. For example, the debug name for `f1` would be

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

where `E` and `QS` stand for literal copies of the source code for the expression and the rest of the qualifier list, and `@` is a placeholder for `f1`s argument.

Note that all free variables in the source code fragments in a debug name also are free in the definition of the corresponding function. When the generated function are turned into supercombinators by the lambda-lifter, these free variables become extra arguments as usual, and their names are recorded in the object information record as explained in section 7.1. This makes it possible to bind free variables in a debug name by means of a `where`-clause, just like free variables in ordinary functions, ensuring that the questions asked by the debugger are self-contained.

For a larger example, and an illustration of the handling of free variables, consider the following list comprehension:

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

The following function is generated by the translation scheme given above:

```
-- [ x + y | x <- @, y <- fie z, x > y]
f0 [] = []
f0 (x : l) = f1 (fie z) ++ f0 l
    where
        -- [ x + y | y <- @, x > y ]
        f1 [] = []
        f1 (y : l) = f2 ++ f1 l
            where
                -- [ x + y | x > y ]
                f2 = if x > y then [x + y] else []
```

The debug names presented to the user are given as comments, whereas the names `f0`, `f1`, and `f2` are the real, system generated names. Notice the placeholder (`@`) in the debug names which will be replaced by the value of the function's parameter during debugging. To complete the translation, `f0 (foo z)` is substituted for the list comprehension and `f0` is defined in a suitable `let`- or `where`-clause.

The following is an example of a typical question which might be asked during debugging:

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

A moment of thought should convince the reader that the answer to this question ought to be yes.

7.2.4 Avoiding unnecessary questions

As a final touch, the compiler could put the generated recursive functions in the trace class *invisible recursion*. This is a good idea since the actual iteration over the lists is of little interest to the user, assuming that the compiler gets the translation right. For instance, suppose that a function `f` computes the wrong value when applied to 3, and consider the following list comprehension:

```
[ f x | x <- [1, 2, 3] ]
```

This would be translated into the following function:

```
-- [ f x | x <- @ ]
f1 []       = []
f1 (x : xs) = [f x] ++ f1 xs
```

If the function `f1` is traced in the normal way (trace class *traced*), and if it is assumed that calls to `(++)` are not traced, then part of the debugging session would appear as follows (user input in italics). See figure 7.1 for the corresponding tree.

```
[ f x | x <- [1, 2, 3] ] => [2, 4, 42]
Yes/No? no
f 1 => 2
Yes/No? yes
[ f x | x <- [2, 3] ] => [4, 42]
Yes/No? no
f 2 => 4
Yes/No? yes
```

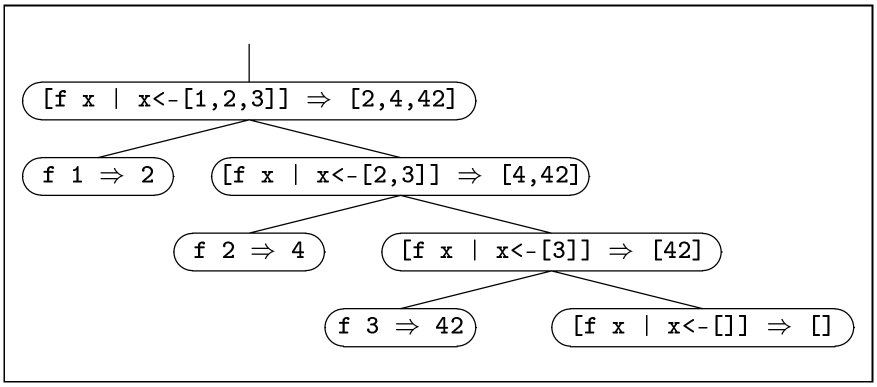


Figure 7.1: The EDT for the code fragment `[f x | x <- [1, 2, 3]]` when the generated function belongs to the trace class *traced*.

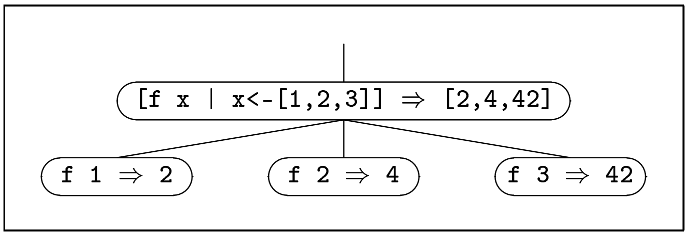


Figure 7.2: The EDT for the code fragment `[f x | x <- [1, 2, 3]]` when the generated function belongs to the trace class *invisible recursion*.

```
[ f x | x <- [3] ] => [42]
Yes/No? no
f 3 => 42
Yes/No? no
```

If the compiler instead assigns the generated function `f1` to the trace class *invisible recursion*, then the corresponding part of the debugging session would be as follows. Note that two unnecessary questions have been eliminated. See figure 7.2 for the corresponding tree.

```
[ f x | x <- [1, 2, 3] ] => [2, 4, 42]
Yes/No? no
f 1 => 2
Yes/No? yes
f 2 => 4
Yes/No? yes
f 3 => 42
Yes/No? no
```

A Haskell program demonstrating a more general list comprehension translation as well as the generation of appropriate debug names may be found in appendix C.

Chapter 8

The User Interface

Chapters 5 to 7 have shown how to construct an EDT efficiently and in a manner which supports debugging at the source-level. To obtain a complete debugger, one last component is needed: an interface which allows the user to explore the EDT in search of bugs. We will refer to this interface as the *EDT navigator*. Figure 8.1 clarifies the relationships between the various components.

An EDT navigator could be designed in many ways; section 8.1.2 briefly discusses some options. However, this chapter focuses on the navigator built into our system which implements an algorithmic debugger. The navigator is presented in section 8.2, and section 8.3 demonstrates, through a substantial example, how to use the resulting debugger. The bugs covered by the example include a black hole (i.e. a detectably self-dependent expression) and non-termination.

One reason for selecting this type of interface, is that algorithmic debugging (at least the version presented in this thesis) causes the EDT to be traversed in an orderly fashion, and, as explained in section 5.3.3, the design of our trace algorithm is to some extent influenced by this property. Another reason is that algorithmic debugging often fits the declarative programming style used in lazy functional programs quite well. Having said that, we will see that our interface can be used in a more browser-like fashion as well, should that be desired.

8.1 Implementation issues

8.1.1 The interface to the EDT generator

Conceptually, the EDT is a static value which provides an abstract view of a computation. In practice, due to limited memory resources, our imple-

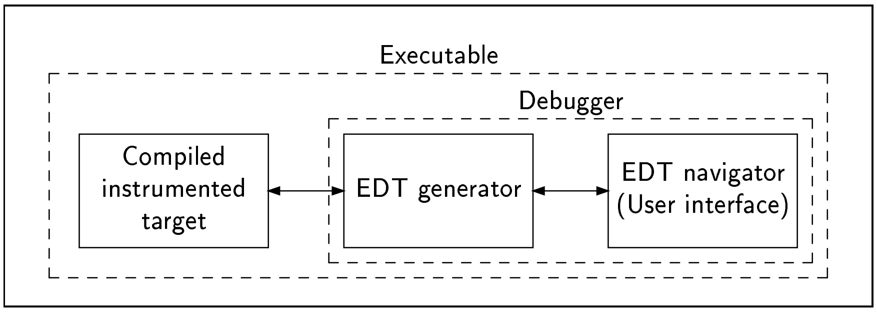


Figure 8.1: The components of a debuggable executable. The target program has been compiled for debugging, resulting in instrumented object code. This is linked with the EDT generator and the EDT navigator thus forming a debuggable executable.

mentation generates the EDT piecemeal, on demand, reusing the memory allocated for storing the EDT by throwing out old parts in favour of new ones. At the implementation level, this means that the EDT is a highly dynamic structure, where traversal is inherently intertwined with re-execution of the target and construction of new parts of the EDT.

Nevertheless, when it comes to implementing the EDT navigator, the interface between the EDT and the user, being able to treat the EDT as a static value and forget about the low-level details greatly simplifies the design and allows the implementor to focus on the main tasks of the interface: navigation and presentation. Thus it is desirable to provide a referentially transparent (stateless) interface between the EDT generator and the EDT navigator. This can be accomplished by treating the EDT as an abstract datatype. Figure 8.2 shows, using Haskell notation, what such an interface might look like.

The abstract datatype `EDTNode` represents nodes in an EDT. The function `root` returns the root of an EDT. It takes two arguments: a list of functions to be used as starting points, and a list of functions which the user dynamically has declared trusted. The various parts of a node are accessed via a number of selector functions such as `function` and `children`. Given the root of an EDT as a handle, it should be clear that the rest of an EDT can be visited through repeated applications of the function `children`.

The type `EDTValue` is basically as described in section 4.4.1. The main differences are that this version provides a representation of $\perp$ (`EVBottom`) and a facility for representing circular values explicitly. How to detect and handle various kinds of $\perp$ was discussed in section 5.2.2 and 6.2.

The explicit representation of circularities is important; otherwise, such values would not have a finite representation, and the interface would be-

```
root      :: [Function] -> [Function] -> EDTNode
function  :: EDTNode -> Function
arguments :: EDTNode -> [EDTValue]
result    :: EDTNode -> EDTValue
children  :: EDTNode -> [EDTNode]

data EDTValue = EVBottom
              | EVUneval
              | EVLabel Int EDTValue
              | EVRef Int
              | EVChar Char
              | EVInt Int
              | ... -- Other primitive types as needed
              | EVConstr Constructor [EDTValue]
              | EVClosure Function [EDTValue]

funName         :: Function -> String
funSourceRef    :: Function -> SourceRef
funFreeVarNames :: Function -> [String]
...             -- Other attributes for Function as needed

ctrName   :: Constructor -> String
ctrLabels :: Constructor -> [String]
...       -- Other attributes for Constructor as needed
```

Figure 8.2: The EDT as an abstract datatype.

come useless unless a lazy language is used (or the type is turned into an abstract datatype). Consider sending `EDTValue`s from one process to another using some textual protocol, for example. It is also useful to have the possibility of displaying circular values as circular graphs in some kind of graphical browser, even if this is an operational representation.

Circularities are encoded by means of special constructors for labelling values and for referring to such labels. For example, consider the circular definition:

```
xs = 1 : xs
```

This is represented as an `EDTValue` as follows:

```
EVLabel n (EVConstr c [EVInt 1, EVRef n])
```

where n is a label unique within the term and c is the `Constructor` representing cons `(:)`.

`Function` and `Constructor` are also abstract datatypes. A number of selector functions such as `funName` and `funSourceRef` allow various attributes from the object information record associated with each function to be accessed. The name of a constructor, field names in case the constructor has labelled fields, and other relevant attributes are accessed in a similar way for the type `Constructor`.

Excerpts from the real C interface are presented in appendix B.6. The basic design is as described above, even if the interface is not completely referentially transparent for practical reasons. See the appendix for details.

8.1.2 User interface design options

There are several approaches to designing a suitable user interface for navigating the EDT. One option, which has already been discussed, is to use algorithmic debugging. This is what the built-in interface, described in the next section, does.

Another option would be to display the EDT graphically in a browser. It might then be possible for the user to quickly spot suspect nodes and thus locate the bug. In practice, however, an EDT is likely to contain many nodes, and individual nodes are also often quite large. Therefore, it would in general only be possible to show a small part of an EDT at once, and it is thus an open question whether there is any real advantage over an approach which focuses on one node at a time (such as an algorithmic debugger). Screen real estate as well as the user's capacity of taking in information are both scarce resources.

Sparud & Runciman [SR97] contains an interesting discussion of this set in a somewhat different context. They present a sophisticated, graphical user interface which permits browsing of their trace structure (called *redex trail*) and gives immediate access to relevant parts of the source code. Their compromise is to display nodes (redexes) on demand in a textual window where each node is confined to a single line by only showing the top structure of values if necessary. Clickable placeholders allow the user to explore hidden parts of the values.

In any event, the EDT generation algorithms in their current incarnation have been optimized for an orderly EDT traversal, so an interface based on the assumption that the entire EDT is readily available may lead to performance problems since frequent re-executions could be necessary to maintain this view. Having said that, an extremely condensed view of the tree, only showing the 'call structure', may be a good way to select starting-points for debugging (see section 12.2.4).

Another important design issue concerns the visualization of values. The built-in interface contains a pretty-printer which simply shows the concrete representation of a value. This is not always what is wanted. Take an abstract syntax tree, for instance. While it is possible to show it as a large term, it may be much more convenient for the user if the tree could be mapped back to something which resembles the concrete syntax. Or take a directed graph represented as an array of adjacency lists. A graphical view with boxes and arrows would almost certainly be easier to understand. However, it is difficult to cater for all possible needs, which suggests that at least the 'pretty-printing part' of an interface ought to be highly customizable.

Thus far, we have not spent a lot of work on user interface design. However, in addition to the built-in textual user interface, there is an interface intended to be used from other processes through a socket or similar arrangement. This interface accepts commands corresponding to the operations of the abstract EDT datatype, and responds with suitably encoded EDT nodes and EDT values, all using a simple text-based protocol. It is thus easy to experiment with various interface designs implemented in various languages.

For instance, in co-operation with Jan Sparud, this author used this inter-process interface to connect an early demonstration prototype of our EDT generator to the graphical user interface provided by the debugger presented in Sparud's licentiate thesis [Spa96]. This interface is written entirely in Haskell, using the Fudget toolkit [CH93, CH98]. The interface has three textual windows showing EDT nodes. Each node occupies one line. One window shows the current node, another its ancestors, and the last one its children. (The two latter are scrollable.) The interface allows the user to either perform ordinary algorithmic debugging or to select any of the displayed nodes as the new current node by clicking on it. Writing the interface in Haskell, and delivering the source code with the system, would thus be an easy way to offer a customizable debugger interface to the user.

8.2 The built-in user interface

When our system compiles a Freja program for debugging, a small EDT navigator is included in the final linking step (see figure 8.1). It implements a simple algorithmic debugger and provides a text-based interface to the user. This interface is activated by invoking the target program with the debug flag (`-d`). The target is then executed on demand, i.e. whenever the debugger needs to access some part of the tree which is not present. The user only has to interact with the target the first time it is executed. Subsequent re-executions are automatic and only noticeable through a (hopefully slight) delay. A summary of the available commands is presented in figure 8.3. The purpose of most of these commands is fairly obvious, so only the central ones are described further below.

The command `debug` starts a debugging session. It accepts optional arguments in the form of function names specifying starting-points as explained in section 6.3. If no arguments are given, a complete EDT is built and debugging starts from the leftmost child of the root. In case starting-points are specified, the result is a forest of sub-EDTs, each rooted in a call to one

General debugging	
`debug` $\{f\}$	Debug program, optionally specifying starting-points.
`stopafter` $[n]$	Stop execution after n trace calls. 0 resets.
`functions`	List debuggable functions.
Answering debug questions	
`yes`	Yes, current reduction is correct.
`no`	No, current reduction is not correct.
`yes?/y?`	Don't know. Proceed as if correct.
`no?/n?`	Don't know. Proceed as if incorrect.
History related commands	
`back` $[n]$	Go back one (or n) question(s).
`forward` $[n]$	Go forward one (or n) question(s).
`goto` $[n]$	Go to question n.
`last`	Go to the last question.
Assertions	
`trust` f	Assert that function f is trusted.
`untrust` f	Revoke earlier assertion regarding f.
`assertions`	List current assertions.
`aclear`	Revoke all earlier assertions.
`aload` *file*	Load assertions from file.
`asave` *file*	Save current assertions to file.
Printing and viewing source code	
`review`	Print the current reduction again.
`print` s $[n]$	Print argument/result/variable (with precision n).
`where`	Print the call stack.
`show` f	Show source code for function f.
`showinfo` f	Show information about function f.
`precision` $[n]$	Show (or set) the default print precision.
`+`	Increase default the print precision.
`-`	Decrease default the print precision.
`printprec` $[n]$	Show (or set) default precision for the print command.
`width` $[n]$	Show (or set) terminal width.
Miscellaneous	
`help/?`	Give help
`quit/exit`	Leave the debugger.

Figure 8.3: Command summary for the built-in user interface.

of the starting-point functions. Debugging then starts from the leftmost of these trees.

A non-terminating target must be explicitly stopped the first time it is executed. The user can do this by pressing `CTRL-C`. This stops the target and causes the debugger to note the current reduction count. This value is then used as an upper limit for the execution time to automatically stop the target on subsequent re-executions. Alternatively, this limit can be specified through the command `stopafter`. The limit can be revised at any time, e.g. to speed up re-executions, but the limit must obviously not be too small, or the target will be stopped before the bug manifests itself.

A debugging session consists of a sequence of questions. The questions are numbered sequentially as they are answered. Each question corresponds to a reduction, and the user has to answer *yes* if the reduction is correct, *no* otherwise. One question is designated as the current question, and the commands `yes` and `no` are used to provide an answer for it. Once an answer has been given, the debugger will move on to the next question according to the algorithmic debugging method. If the user is unsure about the answer, it is possible to answer `yes?` or `no?` instead. The debugger will then proceed as if the question had been answered by yes or no respectively, but unless this leads to the bug being located, the user will later be asked to reconsider the answer to this question.

Notice that a more operational interpretation of the commands `yes` and `no` is possible under the currently employed questioning order. The command `yes` simply means 'go on to the next call', whereas `no` means 'step into the current call'. Thus `yes` roughly corresponds to the command `next` found in imperative debuggers like gdb or dbx, and `no` roughly corresponds to the command `step`. This view is useful if the user finds the questions too difficult. In combination with the history mechanism described below, the commands `yes` and `no` can be used to explore the EDT anyway.

There is also a simple history mechanism which allows the user to move back and forth in the sequence of questions answered to get to the current point in the EDT. The commands `back` and `forward` moves respectively back and forward in the sequence, whereas `goto` allows the user to jump to a question specified by its sequence number and `last` takes the user back to the last (possibly unanswered) question in the sequence. This allows the user to review earlier questions and their answers, but since the currently displayed question also becomes the current question to be answered, the commands `yes` and `no` can be used to revise earlier answers and thereby change the course of debugging. Note that the mechanism is such that only questions on the path from the EDT root to the current last question can be reached directly.

The command `trust` allows the user to dynamically assert that certain functions are to be trusted. This mechanism was described in section 6.4.

Such declarations will affect the tracing the next time the target is re-executed, but meanwhile they are used to skip nodes in the EDT that would not be present had the declarations been in effect during the latest execution. The command `untrust` does the opposite, but the effect may not be evident until the target is next re-executed. Thus it may be a good idea to use the command `debug` to start a new debugging session once `untrust` has been used. Note that `untrust` has no effect for functions which statically have been declared to be trusted.

Quite often a reduction involves large values, sometimes even infinite (circular) ones. To avoid questions becoming textually too large, and to ensure that the printing of a value always terminates, values are only printed to a user-definable precision. In order to examine values more closely, the command `print` is used. This command can print individual arguments, specified by giving the argument position, the value of a free variable, or the result, which has the special name '`=`'. Optionally, a precision may be specified, and subparts of a value can be selected by position. Thus `print foo` means that the value of the free variable `foo` should be printed with the default precision for the command `print`, `print 1 10` that the first argument should be printed with precision 10, and `print =.2.1` that the first part of the second part of the result should be printed.

There is currently no facility for viewing values graphically in the form of trees.[1] However, using tools like VCG [San94] or daVinci [FW94], such a facility would be easy to provide. It is just a matter of sending a textual description of the value to the tool, e.g. via an auxiliary file or a pipe. The tool would then display the tree graphically in a scrollable window, providing facilities for interactive manipulation such as folding and unfolding of parts of the tree.

8.3 Debugging a small program

In this section, we will demonstrate how our debugger, through the built-in interface presented in the previous section, can be used to debug a small but not completely unrealistic lazy functional program. The example is adapted from Johnsson [Joh87a], and makes use of a 'circular' programming style which is typical of many lazy programs.[2] Unfortunately, a number of bugs have crept into the adapted code, leading to a black hole, non-termination,

[1] Or, indeed, in the form of graphs in case of circularities, as long as it is kept in mind that a circular graph is just a finite representation of what semantically is an infinite value.

[2] For instance, this technique is used extensively in our Freja compiler to reduce the number of traversals of the abstract syntax tree. More than once during the development, black holes, which were quite difficult to track down, were encountered in these parts of the code. More than once we wished we had had a debugger.

and attempts to access the head of an empty list, problems with which most lazy functional programmers are all too familiar.

8.3.1 The program

The purpose of the program which we are going to debug is to take a binary tree where the tips contain elements of a type on which a total order is defined (in our case integers), and return a structurally identical tree where the tips have been sorted according to the total order. However, we wish to do so using only one traversal of the tree. In a lazy language, this can be solved through 'circular' programming. Here, the basic idea is that the tree traversal function in addition to the sorted tree returns a list containing the tip values. The list is then sorted and fed back into the tree traversal at the top level. This works fine as long as the traversal is not control dependent on the sorted list.

In general, this technique can be used to combine a sequence of traversals of some structure, where each traversal needs the results of the previous ones, into a single traversal, nominally computing all results in parallel. As shown in Johnsson [Joh87a], from which our example has been adapted, this is perhaps best understood through an attribute grammar formulation. The grammar can then be transliterated into a lazy functional program, where the laziness ensures proper propagation of inherited and synthesized attributes.

An attribute grammar for our problem is given in figure 8.4. S is the start symbol. It has one synthesized attribute, *tree*, which holds the result of the computation, i.e. a sorted version of the tree. T has five attributes, two inherited and three synthesized. The inherited attribute *itips* and the synthesized attribute *stips* are used to collect all the tip values in a list and propagate this to the top-level where it is sorted. The inherited attribute *isorted* and the synthesized attribute *ssorted* are used to propagate the sorted tip values back to the tips of the tree. The synthesized attribute *tree*, finally, holds the resulting tree as before. Note that we have taken some notational liberties and are using x both as a non-terminal in the productions and as a variable in the attribute equations. We are also using `Tip` and `(:^:)` both as terminals and constructors for the returned tree. Figure 8.5 illustrates the attribute propagation for a small tree.

In order to transliterate this grammar into a lazy functional program, one function is introduced for each non-terminal. The functions are defined by pattern-matching over the tree type. There is one case for each of the non-terminal's productions, where the patterns are given by the right-hand sides of the productions in an obvious way. The inherited attributes become additional arguments of the function, and the synthesized attributes are returned as the result, packed into a tuple in case there are two or more. The

Productions	Attribute equations
$S \rightarrow T$	$T{\downarrow}\mathit{itips} = $ `[]`
	$T{\downarrow}\mathit{isorted} = $ `sort` $T{\uparrow}\mathit{stips}$
	$S{\uparrow}\mathit{tree} = T{\uparrow}\mathit{tree}$
$T \rightarrow$ `Tip` x	$T{\uparrow}\mathit{stips} = x$ `:` $T{\downarrow}\mathit{itips}$
	$T{\uparrow}\mathit{ssorted} = $ `tail` $T{\downarrow}\mathit{isorted}$
	$T{\uparrow}\mathit{tree} = $ `Tip (head` $T{\downarrow}\mathit{isorted}$`)`
$T \rightarrow T_{\mathrm{L}}$ `:^:` T_{R}	$T_{\mathrm{R}}{\downarrow}\mathit{itips} = T{\downarrow}\mathit{itips}$
	$T_{\mathrm{L}}{\downarrow}\mathit{itips} = T_{\mathrm{R}}{\uparrow}\mathit{stips}$
	$T{\uparrow}\mathit{stips} = T_{\mathrm{L}}{\uparrow}\mathit{stips}$
	$T_{\mathrm{L}}{\downarrow}\mathit{isorted} = T{\downarrow}\mathit{isorted}$
	$T_{\mathrm{R}}{\downarrow}\mathit{isorted} = T_{\mathrm{L}}{\uparrow}\mathit{ssorted}$
	$T{\uparrow}\mathit{ssorted} = T_{\mathrm{R}}{\uparrow}\mathit{ssorted}$
	$T{\uparrow}\mathit{tree} = T_{\mathrm{L}}{\uparrow}\mathit{tree}$ `:^:` $T_{\mathrm{R}}{\uparrow}\mathit{tree}$

Figure 8.4: Attribute grammar for transforming a binary tree into a binary tree with the same shape where the tip values are sorted according to some order. S is the start symbol. $\downarrow$ indicates an inherited attribute, $\uparrow$ a synthesized one.

result of transliterating into Freja is shown in figure 8.6. The transliteration was performed rather carelessly, however, resulting in a few mistakes. We will use the debugger to find them in the next section.

8.3.2 Eliminating a black hole

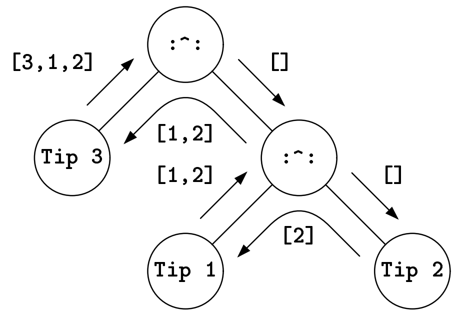
When the program in figure 8.6 is executed, it immediately stops with an error message saying that a black hole has been encountered when evaluating an application of an internal selector function.

```
[Fatal error] Black hole! (Apparently in application of
"Prelude._selTuple_3_2".)
```

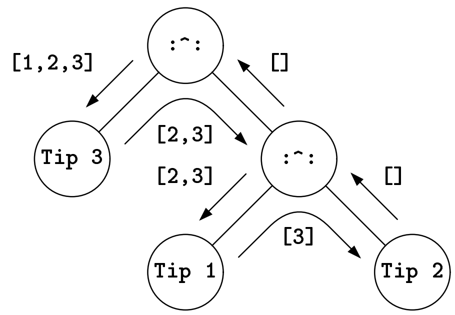
Since this does not offer any particularly good lead as to what the problem may be (it is almost certainly not a bug in the internal selector function `_selTuple_3_2`), we recompile the program with debugging support and start it in debug mode. Below, the user's input is typeset in italics.

```
sen1-102% fc -g sorttree.fr
sen1-103% sorttree -- -d
FREJA DEBUGGER
--------------

(Enter "help" to get help.)
```



(a) Propagation of the attributes *itips* (↓) and *stips* (↑).



(b) Propagation of the attributes *isorted* (↓) and *ssorted* (↑).

Figure 8.5: Attribute propagation for a small tree.

```
[no tree]> debug
(((Tip
[Fatal error] Black hole! (Apparently in application of
"Prelude._selTuple_3_2".)
-----------------------------------------------------
aTree
=>
(:^:)
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  ((:^:) (Tip 3) (Tip 1))
1> yes
```

We use the command debug to start a debugging session. The target is then executed, but the execution stops almost immediately with the same error message as before. However, we note that a small part of the result actually

```
module Main where

data Tree a = Tip a | (Tree a) :^: (Tree a) -- deriving Show

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

insert x [] = []
insert x xxs@(x' : xs) | x < x'    = x : xxs
                       | otherwise = x' : insert x xs

sortTree t = t_tree
    where
        (t_stips, t_ssorted, t_tree) = sortTree' t t_itips t_isorted
        t_itips   = []
        t_isorted = sort t_stips

sortTree' (Tip a) t_itips t_isorted = (t_stips, t_ssorted, t_tree)
    where
        t_stips   = a : t_itips
        t_ssorted = tail t_isorted
        t_tree    = Tip (head t_ssorted)
sortTree' (l :^: r) t_itips t_isorted = (t_stips, t_ssorted, t_tree)
    where
        (l_stips, l_ssorted, l_tree) = sortTree' l l_itips l_isorted
        (r_stips, r_ssorted, r_tree) = sortTree' r r_itips r_isorted
        r_itips   = t_itips
        l_itips   = l_stips
        t_stips   = l_stips
        l_isorted = t_ssorted
        r_isorted = l_ssorted
        t_ssorted = r_ssorted
        t_tree    = l_tree :^: r_tree

aTree = ((Tip 7) :^: ((Tip 2) :^: (Tip 5))) :^: ((Tip 3) :^: (Tip 1))

main = print (sortTree aTree)
```

Figure 8.6: A Freja program for solving the tip sorting problem using only one tree traversal. The program is a transliteration of the attribute grammar of figure 8.4, but contains a few bugs.

has been printed (((Tip). The debugger now proceeds to ask the first question. The question concerns the value of the CAF `aTree`: is it correct or not?. This is the first question since the CAF `main` depends on `aTree` (see section 6.1). Since the value of `aTree` looks perfectly fine, we answer yes and move on to the next question.

```
main => "(((Tip :_|_"
2> no
-----------------------------------------------------
sortTree
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
=>
(:^:) ((:^:) (Tip _|_) ?) ?
3> no
```

The next question concerns `main` which evaluated to a string[3] which ends in $\perp$. This is not what we expected, so the answer is no. Now the debugger asks about an application of `sortTree`. The argument is OK, but in the result we find $\perp$ in a tip. So again the answer is no. We also note that two parts of the result were never evaluated, indicated by the two question marks.

```
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
  []
  ?
=>
(?, ?, ((:^:) ((:^:) (Tip _|_) ?) ?))
4> no
```

We are now faced with an application of `sortTree'`. We immediately notice one interesting detail: the third argument (`t_isorted`) was never evaluated (to WHNF). However, this is an operational observation. What does it mean declaratively? Well, we know for sure that an expression which is not evaluated cannot possibly have influenced the computation in any way. In particular, it cannot have caused our black hole. Thus, for the purpose of declarative debugging, we should *assume* that an unevaluated expression represents a correct value!

This is in fact a quite important advantage of debugging lazy functional languages declaratively. Unevaluated values are ubiquitous and tend to be

[3]Freja does currently not support monadic I/O.

represented by very large and complicated expressions. Thanks to the declarative approach, such values can be abstracted to a single symbol meaning 'correct', saving the user a lot of bother. Note that it would *not* be possible to handle unevaluated values in this way in an operational debugger, since it may be (and often is) the case, that a value which is unevaluated at some point during the computation, will be used and thus evaluated later.

Continuing with our example, we see, by the same reasoning, that the result as far as we are concerned is mostly correct. However, $\perp$ does occur in the result which is not intended. This reduction is therefore incorrect.

```
sortTree' ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5))) ? ?
=>
(?, _|_, ((:^:) (Tip _|_) ?))
5> no
```

The next question is again a call to `sortTree'`. Reasoning as above, we see that the arguments are correct, and we would thus expect the answer to be completely defined. But since $\perp$ occurs in the result, this is not the case, and the reduction is again wrong.

```
sortTree' (Tip 7) ? _|_ => ([7:?], _|_, (Tip _|_))
6> yes
```

Once again we encounter a call to `sortTree'`. This time we have to think more carefully about our answer since $\perp$ occurs as one of the arguments. The argument in question is `t_isorted`. Looking at the attribute equations for the tip case, we would then expect the returned tip value to be $\perp$ (since `head` $\perp = \perp$) and `t_ssorted` to be $\perp$ (since `tail` $\perp = \perp$). Moreover, we would expect `t_stips` to be a list whose first element is 7. Thus, given the arguments above, all three components of the result are correct. We therefore conclude that the reduction is correct.

```
sortTree' ((:^:) (Tip 2) (Tip 5)) ? ?
=>
(?, _|_, ((:^:) ? ?))
7> no
```

Question 7 is similar to question 5. The answer is again no.

```
sortTree' (Tip 2) ? _|_ => ([2:?], _|_, (Tip ?))
8> yes
-------------------------------------------------------
sortTree' (Tip 5) ? _|_ => ([5:?], _|_, (Tip ?))
9> yes
```

Questions 8 and question 9 are both similar to question 6, even if the tip values in the results are unevaluated. Both reductions are thus correct.

```
Bug located! Erroneous reduction:
sortTree' ((:^:) (Tip 2) (Tip 5)) ? ?
=>
(?, _|_, ((:^:) ? ?))
[no] 7>
```

The debugger has now collected enough information to locate the erroneous function and exhibit a particular application of it which manifests the bug symptom. The bug evidently occurs in the clause for `(:^:)`. Furthermore, we find that the second and third arguments are unevaluated. From an operational point of view, this is strange. Why are these arguments not used? A quick inspection of the source code reveals that `t_isorted` actually does not occur in the body of the second claues of the function. This must be wrong. Looking back at the attribute equations, we spot the mistake and correct the equation for `l_isorted`:

```
l_isorted = t_isorted
```

An alternative approach would have been to inspect the equations in order to find the cause of the black hole, here shown as $\perp$. The black hole appears as the second component of the returned tuple, i.e. `t_ssorted` is bound to $\perp$. `t_ssorted` is equal to `r_ssorted` which depends on `r_isorted` via the definition of `(sortTree' (Tip 5))`. In turn, `r_isorted` is equal to `l_ssorted` which depends on `l_isorted` via the definition of `(sortTree' (Tip 2))`. But `l_isorted` had by mistake been defined as `t_ssorted`. The definition was thus circular in a self-dependent way, hence the black hole.

Note that we have encountered this reduction before. It is question number 7 to which we answered no, as indicated by the prompt '`[no] 7>`'. We are thus now in a situation where we can revise the answers to earlier questions and continue debugging, should we so desire. This may be necessary if some of our answers were incorrect and this caused the debugger to point out a function which turned out to be correct. The command `where` is useful in this context. It shows the call stack which led to the current reduction, i.e. all questions which have been answered by no. Should we find that the answer to one of these questions in fact should have been yes, then it is just a matter of jumping to that question (using `goto`, for instance), answer yes, and continue debugging. The call stack for our example is as follows.

```
[no] 7> where
*** Question ID = 2 ***
main => "(((Tip :_|_"
```

```
*** Question ID = 3 ***
sortTree
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
=>
(:^:) ((:^:) (Tip _|_) ?) ?

*** Question ID = 4 ***
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
  []
  ?
=>
(?, ?, ((:^:) ((:^:) (Tip _|_) ?) ?))

*** Question ID = 5 ***
sortTree' ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5))) ? ?
=>
(?, _|_, ((:^:) (Tip _|_) ?))

*** Question ID = 7 ***
sortTree' ((:^:) (Tip 2) (Tip 5)) ? ?
=>
(?, _|_, ((:^:) ? ?))

[no] 7>
```

8.3.3 Improving the termination properties

Having corrected the mistake found in the previous section, we re-compile our program and try again. To our disappointment, we soon realize that it still does not work since no output whatsoever appears. Evidently, the program is stuck in an infinite loop for some reason. We thus start the target in debug mode again, start a debugging session and press CTRL-C after a short while to abort the non-terminating computation.

```
FREJA DEBUGGER
--------------

(Enter "help" to get help.)
[no tree]> debug
```

```
^C(((Tip
[Fatal error] User interrupt.
-----------------------------------------------------
aTree
=>
(:^:)
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  ((:^:) (Tip 3) (Tip 1))
1> yes
-----------------------------------------------------
main => "(((Tip :_|_"
2> no
sortTree
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
=>
(:^:) ((:^:) (Tip _|_) ?) ?
3> no
```

The first three questions are the same as before. We thus answer them quickly and move on to question 4.[4]

```
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
  []
  _|_
=>
([7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7:...],
 ?,
 ((:^:) ((:^:) (Tip _|_) ?) ?))
4> print =.1 50
[7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7:...]
4> no
```

[4] In principle, the debugger could maintain a table of asked questions and their answers so as to avoid asking the same question twice.

Now we have to think again. $\perp$ appears as the third argument (`t_isorted`), so we would expect $\perp$ to appear in the tips of the result. Since unevaluated values should be assumed to be correct, we conclude that the returned tree is what it should be. The second component of the result (`t_ssorted`) is also unevaluated and therefore semantically correct. However, given an empty list and a tree with the tip values 7, 2, 5, 3, and 1, the synthesized attribute `t_stips` (the first component of the returned tuple) should be the list `[7,2,5,3,1]`, not a list containing a lot of sevens. As a matter of fact, printing this part of the result with greater precision convinces us, beyond reasonable doubt, that the list is infinite. So the answer must be no.

```
sortTree'
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  [7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
   7,7,7:...]
  _|_
=>
([7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
  7,7,7:...],
 ?,
 ((:^:) (Tip _|_) ?))
5> no
```

Question 5 is similar to question 4. To be sure, `t_itips` seems to be an infinite list of sevens in this case rather than the empty list of question 4, but given a tree with the tip values 7, 2, 5, and 3, we would expect these values first in the list bound to `t_stips` (the first component of the result tuple). Thus this reduction cannot be correct.

```
sortTree'
  (Tip 7)
  [7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
   7,7,7:...]
  _|_
=>
([7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
  7,7,7:...],
 _|_,
 (Tip _|_))
6> yes
```

Reasoning as for questions 4 and 5, we this time conclude that this reduction must be correct. There is only one tip value (7) in the argument tree, and as far as we can tell, it may well have been consed onto the infinite list of sevens.

```
Bug located! Erroneous reduction:
sortTree'
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  [7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
   7,7,7:...]
  _|_
=>
([7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
  7,7,7:...],
 ?,
 ((:^:) (Tip _|_) ?))
[no] 5>
```

The debugger now concludes that it has found the offending function. Again, it is `sortTree'`, and again it is the case for the constructor `(:^:)`. Inspecting the erroneous reduction, we notice that the tips from the right part of the tree have not been consed onto the list `t_stips`. `t_stips` is equal to `l_stips` which depends on `l_itips` via the call `(sortTree' (Tip 7))`. Looking at the equation for `l_itips`, we immediately spot the mistake: it is defined as `l_stips`, i.e. we have a circular definition again. Referring to the attribute equations, we correct the mistake:

```
l_itips = r_stips
```

8.3.4 Head of empty list

Once more we compile our program. By now, we are a bit disillusioned, so we are not too surprised to discover that it still does not work. This time, it seems as if we are attempting to access the head of an empty list. We invoke the debugger again and start debugging.

```
FREJA DEBUGGER
--------------

(Enter "help" to get help.)
[no tree]> debug
PreludeList.tail: empty list(((Tip
[Fatal error] error called; execution aborted.
...
-------------------------------------------------------
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
```

```
  []
  []
=>
([7,2,5,3,1], ?, ((:^:) ((:^:) (Tip _|_) ?) ?))
4> yes
```

The first three questions are exactly as before, so we answer them briskly and arrive at question 4. Given `t_isorted = []`, we would expect $\perp$ to occur in the tips of the returned tree (since `head []` $= \perp$, see the attribute equations). So the returned tree is OK. Moreover, the first part of the result (`t_stips`) is a list of all the tip values in the correct order, exactly what we would expect, and the second part of the result is also correct since it is unevaluated. Thus we answer yes.

```
sort [7,2,5,3,1] => []
5> no
```

We now realize that the sorting routine is wrong. The result of sorting a non-empty list is certainly not an empty list.

```
sort [2,5,3,1] => []
6> no
-------------------------------------------------------
sort [5,3,1] => []
7> no
-------------------------------------------------------
sort [3,1] => []
8> no
-------------------------------------------------------
sort [1] => []
9> no
-------------------------------------------------------
sort [] => []
10> yes
```

The following five questions are quickly answered using the same reasoning.

```
insert 1 [] => []
11> no
-------------------------------------------------------
Bug located! Erroneous reduction:
insert 1 [] => []
[no] 11>
```

We finally arrive at a call to `insert`, and the mistake is now obvious. Thus we correct the definition of `insert` for the empty list case:

```
insert x [] = [x]
```

8.3.5 Not quite right

After eliminating the latest bug, we routinely re-compile the program and try again. To our dismay, it still does not work. Though at least we get more output this time than just a tip of the tree.

```
FREJA DEBUGGER
--------------

(Enter "help" to get help.)
[no tree]> debug
PreludeList.head: empty list(((Tip 2) :^: ((Tip 3) :^:
(Tip 5))) :^: ((Tip 7) :^: (Tip
[Fatal error] error called; execution aborted.
-------------------------------------------------------
aTree
=>
(:^:)
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  ((:^:) (Tip 3) (Tip 1))
1> yes
-------------------------------------------------------
main
=>
"(((Tip 2) :^: ((Tip 3) :^: (Tip 5))) :^: ((Tip 7) \
\:^: (Tip :_|_"
2> no
-------------------------------------------------------
sortTree
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
=>
(:^:)
  ((:^:) (Tip 2) ((:^:) (Tip 3) (Tip 5)))
  ((:^:) (Tip 7) (Tip _|_))
3> no
```

The first three questions are easy. We answer them speedily, and arrive at question 4, which at first sight looks somewhat daunting.

```
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
```

```
    []
    [1,2,3,5,7]
  =>
  ([7,2,5,3,1],
   ?,
   ((:^:)
     ((:^:) (Tip 2) ((:^:) (Tip 3) (Tip 5)))
     ((:^:) (Tip 7) (Tip _|_)))
  4> no
```

Inspecting the question a little closer, we find that the call looks just fine. In particular, the third argument is the sorted list of the tip values of the tree. Yet $\perp$ appears in the result tree. This cannot be right, so the answer must be no.

```
  sortTree'
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    [3,1]
    [1,2,3,5,7]
  =>
  ([7,2,5,3,1],
   [5,7],
   ((:^:) (Tip 2) ((:^:) (Tip 3) (Tip 5))))
  5> no
```

Again, the call for the left subtree looks as we would expect. The second argument is a list of the tip values of the right subtree, and we have a sorted list of all the tips as the third argument. The first two parts of the result are also correct: a list of all the tip values in the tree and a sorted list of the two tips which are to be inserted in the right subtree. The returned subtree, however, is wrong; the tips should be `1`, `2` and `3`, not `2`, `3` and `5`. We thus conclude that this reduction is incorrect.

```
  sortTree' (Tip 7) [2,5,3,1] [1,2,3,5,7]
  =>
  ([7,2,5,3,1], [2,3,5,7], (Tip 2))
  6> no
```

Reasoning as above, we would expect the returned tip value to be `1`. Since that is not the case, the result is wrong and the answer no.

```
  Bug located! Erroneous reduction:
  sortTree' (Tip 7) [2,5,3,1] [1,2,3,5,7]
  =>
  ([7,2,5,3,1], [2,3,5,7], (Tip 2))
  [no] 6>
```

The debugger now concludes that the bug is in `sortTree'`. This time, the erroneous reduction indicates that we should look for the bug in the tip case of the function definition, and since the returned tip value is wrong, we might as well start with the definition of `t_tree`. Doing this, we quickly spot the mistake. We correct the equation

```
t_tree = Tip (head t_isorted)
```

compile the program once more, and run it. And finally it works!

```
sen1-127% fc -g sorttree.fr
sen1-128% sorttree
(((Tip 1) :^: ((Tip 2) :^: (Tip 3))) :^: ((Tip 5) :^:
(Tip 7)))
```

Chapter 9

System Implementation

This chapter gives an overview of our compiler, Freja, and explains how it supports the debugger.

9.1 Overview

A Freja program consists of a set of modules in one or more files. Each file may contain one or more modules. The compiler compiles one or more files containing modules simultaneously, and for each module file it receives as input, it generates three output files: the code for all modules in the input file (assembler), the interfaces for all modules in the input file, and information (run-time system dependences) to the run-time system generator (see below) for all modules in the input file.

The compiler expects to be passed a closed set of modules and module interfaces, i.e. each module mentioned in the `import`-declarations of the modules in the set must be a member of the set in the form of a module or an interface for the module. (The Prelude is implicitly available, however.) The compiler analyses the dependences between the modules and compiles them in a suitable order. The distribution of the modules over the input files is irrelevant. The intention is that mutually recursive modules should be handled automatically by being compiled together, but the current implementation is not quite there.

The compiler is driven from a compiler driver which either compiles modules to object code by invoking the assembler on the generated assembler files, or which goes all the way to an executable by also running the run-time system generator on the generated run-time system dependence files, assembling the result, and finally linking all modules, the generated run-time system, the fixed parts of the run-time system, and the Prelude. If the program is being compiled for debugging, the linking step also includes the debugger, and a traced version of the Prelude is used instead of the standard version.

Compiling a multi-module program is thus very simple: just run the compiler driver on all source files. The compiler will check the dependences and compile the involved modules in a suitable order. Of course, this means re-compiling all modules each time, so in practice one might want to use `make`. There is currently no Freja-specific make facility.

9.2 The compiler

The compiler itself is written in entirely in Haskell (HBC). It uses a hand-written scanner and a combinator parser.[1] Then follows a two-stage transformation from abstract syntax to a relatively simple, sugared, lambda calculus representation (e.g. removing of modules by qualifying all names, pattern-matching compilation, translation of list comprehensions), type checking (Rémy-style `let`-polymorphism [Rém92] and graph unification using HBC's state threads), compile-time simplifications, lambda-lifting, generation of G-code, peep-hole optimization of the G-code, and generation of SPARC assembler. When compiling for debugging, special translation strategies are sometimes used (e.g. for list comprehensions), and fewer optimizations are performed (e.g. no inlining).

For debugging purposes, the intermediate forms carry debugging attributes which are gradually filled in during the compilation process. In particular, source code references are maintained throughout the compilation process (in the form of source code *regions* to make it possible for the debugger to show the relevant source code fragments to the user). As a side effect of this, the compiler is able to generate quite good error messages. However, properly maintaining the source code references through all transformations was not always easy. In addition to source code references, the debugging attributes include such things as function and module name, names of any free variables, arity, and attributes related to the trace class inference (see section 6.4). All of these attributes are eventually embedded in the generated assembler code in the form of an object information record in C format. Thus it is easy to access this information from the debugger. Furthermore, the relevant debugging-related attributes (e.g. for trace class inference) are also propagated to the interface files in order to be available in subsequent compilations, and to the run-time system generator through the run-time system dependence files.

The trace class analysis is performed just prior to G-code generation in case the code is being compiled for debugging. The G-code generation is

[1] The used combinator library is a version of Swierstra & Duponcheel's deterministic, error-correcting parsing combinators [SD96] and was kindly provided by Magnus Carlsson, Chalmers.

performed in the same way regardless of whether the code is being compiled for debugging or not. As explained in section 5.4.5, the special G-code instruction which creates traced application nodes (`MKTAP`) is instead inserted during the following peep-hole optmization pass. The peep-hole optimizer bases its decision regarding whether to transform `MKAP` to `MKTAP` or not on the tracing attributes. If the applied function is unknown or traced, `MKTAP` is used instead of `MKAP`. (If the function which is being compiled is untraced, then `MKAP` is either kept or, if possible, replaced by an instruction which creates vector applications.) During the final code generation, the translation of `FUNSTART` depends on whether the function is traced or not. In the former case, a call to *Trace* is inserted and debug variables are reserved on the local stack for keeping track of the parent identity and the query distance estimate (see section 5.4).

9.3 The run-time system generator

The run-time system generator (RSG) generates a large part of the run-time system. Exactly what is generated depends on what support the compiled modules need from the run-time system. This is communicated via special run-time system dependence files which are generated by the compiler as explained above. Since it is often the case that many modules have similar needs, the run-time system generator arranges for 'resource' sharing whenever possible. This avoids a lot of code-duplication, and may also have a positive impact on the cache behaviour.

For instance, tuple types constitute one such resource. Conceptually, all tuple types from 2-tuples and upwards are predefined in Freja (and Haskell). Freja implements this by implicitly creating a tuple type when it is mentioned. However, if this was done on the basis of compilation units, the result might well be a lot of unnecessary code duplication. So instead the compiler informs the RSG that this tuple type must be created, and then it continues with the compilation assuming that the type exists. Among other tasks, the RSG will then collect all requests for tuple types and generate exactly one instance of each required type.

Other resources are handled in a similar way. For instance:

- Garbage collection routines are shared extensively: the compiler just tells the RSG what kind of routines a particular module needs.
- Integer, double and string constants are shared.
- General comparison routines for data objects are generated (one routine for each object size) and shared.

Another task for the RSG is garbage collection of CAFs. CAF collection is problematic since references to the CAFs are embedded in the code of the supercombinators [PJ87, p. 312]. As long as some particular supercombinator is referred from the heap, the CAFs that it refers to, either directly or indirectly via other supercombinators, are not garbage. Thus, based on dependence information from the compiler, the RSG performs a global dependence analysis involving *all* supercombinators (including the CAFs) in the program being compiled, and for each supercombinator the transitive closure of all referred CAFs is computed. Then a specialized garbage collection routine is generated for each supercombinator which is used during garbage collection to mark the set of CAFs it refers to. (There is a global bit-vector for this purpose, and since it is processed one word at a time, the generated routines usually consists of just a few memory accesses and simple logical operations.) Since it is common that a group of supercombinators refer to the same set of CAFs, it is also possible to share the generated code to a large extent.

For debugging purposes, the RSG generates an index which gives the debugger access to all information records for the traced supercombinators. Furthermore, it computes the globally unique function group numbers (see section 6.4.4) and stores them in the information records. Actually, the function group number is obtained more or less as a side-effect of the global dependence analysis performed to handle garbage collection of CAFs.

Generally speaking, we have found the RSG to be a most useful device since it allows a number of tasks to be postponed to 'link time' when information from all modules are available. Section 12.2.7 hints at another possible use of the RSG for debugging purposes: checking that all instances of a trusted class indeed are trusted.

9.4 The run-time system

The run-time system is fairly conventional in most respects, apart from being partly generated on demand. The following are the important points from a debugging perspective:

- The representation of the graph is such that it can be understood by the debugger. All objects are tagged, and tags are distinct to make it possible to tell isomorphic objects of different types apart (the empty list and `False` have distinct tags; so have a list cell and a pair). The tag is just a pointer to a descriptor table.

- It is also possible to find function names and the names and values of free variables. Again this is thanks to all objects being tagged; in this case, the object information records generated by the compiler can be found via the tags.

- The garbage collector is made aware of the EDT, since the EDT holds references to the graph.
- Control is passed to the debugger on run-time errors and user interrupts since it must be possible to debug non-terminating programs as well as programs which abort.
- Support for re-execution. There is an interface which allows the debugger to execute and then re-execute the generated code. This involves preserving any input to the program and automatically aborting re-executions in case of non-termination (see section 6.2).

The garbage collector which is currently used is a simple two-space copying garbage collector [Che70] with special provisions for handling CAFs as outlined in section 9.3. As seems to be the case with most collectors for lazy languages, the implementation is 'object oriented': the tags of the objects on the heap refer to descriptor tables which among other things point to appropriate garbage collection code. We chose the current garbage collection scheme because of its simplicity and since we already had a working implementation. From a performance point of view, however, it seems very likely that a generational collector would be preferable (see section 10.4). Seward [Sew92], Sansom & Peyton Jones [SPJ93], and Röjemo [Röj95] have all considered generational garbage collection in a lazy context. (See Wilson [Wil92] for a survey of different garbage collection techniques.)

Chapter 10

Performance Evaluation

This chapter evaluates the performance of the Freja debugger. Five different benchmark programs are tested for various settings of the two key parameters: the maximal number of nodes in the stored portion of the EDT and the maximal total size of the pieces of graph retained by the EDT. The effects of declaring the standard Prelude (the library of standard functions and datatypes) trusted are also measured for these five benchmarks.

10.1 Benchmarks and symbols

All measurements have been performed on a 167 MHz Sun UltraSparc 1 equipped with 128 Mbyte of primary memory. Five different benchmark programs have been used. They are all quite small in terms of lines of source code (ranging from 7 lines to 175), but all of them result in substantial computations (the execution times for non-debugged code range from 7 to 34 seconds and between 6 and 16 million *traced* reductions are performed during tracing). The tracing characteristics of the programs are varying. In some cases, the resulting EDTs retain almost no extra pieces of graph, whereas in others they hold on to very large graphs which would otherwise have been disposed of by the garbage collector.

The five benchmark programs are presented below. The listings may be found in appendix D.

- *Ackermann.* Computes A(3, 4) 50 times, where A is the Ackermann function. The resulting EDT retains almost no extra pieces of graph, so the number of nodes in the stored portion of the EDT is only limited by the number of allocated EDT nodes.

- *Sieve.* Computes the 2500th prime number using the sieve of Erathostene.

- *Isort.* Sorts the integer list `[5000, 4999 .. 1]` into ascending order using insertion sort.
- *Crypt.* Solves cryptarithmetic puzzles (such as SEND + MORE = MONEY) using exhaustive searching. It is written in a typically lazy style with one function generating a list of all possible assignments of digits to letters, and another function picking valid solutions from this list. Thanks to laziness, the program normally runs in constant heap space. However, tracing results in the entire list of generated assignments being referenced from an EDT node close to the root. This causes a very large 'space leak' and hence performance problems. The actual puzzle solved is ONE + ONE + ONE + ONE + ONE + ONE = SIX.
- *Mini-Freja.* This is a Mini-Freja interpreter (see section 4.3) written as a direct style denotational semantics specification. It interprets a program computing a list of the first 500 prime numbers using the sieve of Erathostene (of course!). This is the largest of the benchmarks both in terms of lines of source code (175 lines in 7 modules) and in terms of the number of traced reductions (16 million). It is probably also the most realistic of the benchmarks.

Note that the sizes of the benchmarks (in terms of the size of the resulting computation) from a debugging perspective in most cases are unnecessarily large. One would not compute the 2500th prime when debugging a program like Sieve, for instance, nor would one interpret Sieve computing the 500th prime when debugging the Mini-Freja specification. The exception to this among these benchmarks is Crypt. It performs intrinsically large computations since it is based on naive, exhaustive search.

The reasons for using computationally large benchmarks is that this makes it easier to obtain good measurements and that it demonstrates the debugger's ability or inability to handle large computations. To put the execution times of the benchmarks into context, one can note that high-quality implementations of lazy functional languages generate fairly good code. When this code is executed on a modern, fast computer, quite a lot of work can be carried out within a second or two. The Haskell compiler HBC from Chalmers [Aug97], which is written in LML [Aug84], typically compiles a medium-sized Haskell module in just a few seconds, for instance.

Table 10.1 lists and explains the symbols used to denote the various parameters and measured quantities in this chapter.

10.2 Compiler performance

We first give the results of a small performance comparison between Freja and the HBC compiler from Chalmers [Aug97] for the five benchmark pro-

Symbol	Parameter or measured quantity
T	The number of *traced* reductions, i.e. the number of calls to *Trace*. This is a rough measure of the size of a computation. Also note that it is an upper bound of the number of nodes in the *complete* EDT.
N	The number of nodes in the stored part of the EDT at the end of the execution.
$N_{\max}$	User-definable upper bound on the number of nodes in the stored part of the EDT.
RG	Total size of the pieces of graph retained solely by the EDT. Note that this is the size at the *end* of the execution. This does not necessarily reflect the average size of the retained pieces of graph during the execution, or the effort spent on garbage collecting them (i.e. t_{GC}).
$RG_{\max}$	User-definable upper bound on RG. Note that this is a *soft* limit; there is no guarantee that RG does not exceed $RG_{\max}$ temporarily (see section 5.4.1). Indeed, if a single EDT node close to the root retains a piece of graph whose size exceeds $RG_{\max}$, the current implementation is unable to respect the limit.
t_{tot}	Total execution time; $t_{\mathrm{tot}} = t_{\mathrm{red}} + t_{\mathrm{GC}} + t_{\mathrm{EC}}$.
t_0	Total execution time for the baseline case (no debugging).
t_{red}	Reduction time. Time spent actually performing graph reduction.
t_{GC}	Garbage collection time. Total time spent on garbage collection.
t_{EC}	EDT construction time. Time spent building and pruning the EDT, i.e. executing *Trace* and its subordinate routines.
$QD_{\max}$	The maximal QD estimate of a node in the stored part of the EDT. This is an indication of the number of questions that can be answered before the target program is re-executed.
OCS	Size of the generated object code.

Table 10.1: Parameters and measured quantities.

Benchmark	t_{F} [s]	t_{H} [s]	$t_{\mathrm{F}}/t_{\mathrm{H}}$
Ackermann (10 Mbyte heap)	13.5	6.9	1.95
Sieve (10 Mbyte heap)	7.4	9.6	0.78
Isort (10 Mbyte heap)	14.7	15.8	0.93
Crypt (10 Mbyte heap)	17.2	16.9	1.01
Mini-Freja (32 Mbyte heap)	33.5	37.6	0.89

Table 10.2: Total execution time for the benchmarks when compiled with Freja (t_{F}) and HBC (t_{H}) respectively. Average times over 5 runs.

grams. See table 10.2. The only purpose is to demonstrate that the Freja compiler at least in these five cases generates code which in performance terms is comparable to that generated by one of the main Haskell compilers. This puts the performance measurements made in this chapter into a broader context, and it also indicates that the results probably are representative of what one would find if a debugger similar to ours were integrated into a compiler like HBC.

Having said that, we were pleasantly surprised with the results in most cases given that the Freja compiler currently performs few optimizations (no strictness analysis, function applications are always constructed before being reduced, etc.). One reason may be that the Freja compiler has a code generator written specifically for the SPARC architecture. Differences in garbage collection times also contribute; see tables 10.3 and 10.4.

The measurements in table 10.2 were performed as follows. In the Freja case, the programs were compiled for ordinary execution. In the HBC case, no special flags were passed to the compiler. We took care to run the Freja and HBC versions of the benchmarks with the same amount of memory allocated for the heaps. Furthermore, explicit type signatures were provided to eliminate overloading on numeric types so as to avoid giving Freja an unfair advantage since Freja lacks overloaded numeric constants.[1]

For Freja, the heap sizes shown in table 10.2 are the *initial* heap sizes. These are sufficient for running the benchmarks when no tracing is performed. The same initial heap sizes are used in all performance measurements in the following, but note that the Freja run-time system automatically grows the heap when necessary as long as there is (virtual) memory available. For HBC, the shown heap sizes are the *maximal* heap sizes; the HBC run-time system dynamically adjusts the heap size during execution based on the size of the live data, so the actual heaps could be smaller

[1]In Freja we have `(1::Int)`, for example, with obvious repercussions on type inference and specialization opportunities in the absence of explicit type signatures.

Benchmark	t_0 [s]	t_{red} [s]	t_{GC} [s]	t_{GC}/t_0	*OCS* [kbyte]
Ackermann	13.5	12.5	1.0	0.08	4
Sieve	7.4	7.3	0.1	0.01	4
Isort	14.7	14.6	0.1	0.01	4
Crypt	17.2	17.2	0.0	0.00	23
Mini-Freja	33.5	28.7	4.8	0.14	78

Table 10.3: Breakdown of the execution time and the size of the object code for the benchmark programs when compiled for ordinary execution with Freja. Average times over 5 runs.

Benchmark	t_{tot} [s]	t_{red} [s]	t_{GC} [s]	$t_{\text{GC}}/t_{\text{tot}}$
Ackermann	6.9	5.5	1.4	0.20
Sieve	9.6	8.8	0.8	0.08
Isort	15.8	14.4	1.4	0.09
Crypt	16.9	16.6	0.3	0.02
Mini-Freja	37.6	31.5	6.1	0.16

Table 10.4: Breakdown of the execution time for the benchmark programs when compiled with HBC. Average times over 5 runs.

[Aug93a].

Table 10.3 gives a breakdown of the execution time and, in addition, the size of the object code when the benchmark programs are compiled for ordinary execution with the Freja compiler. To make it easy to see the overhead caused by debugging, the measured times in the following sections are often normalized to the total execution time for each benchmark as given in table 10.3. Hence the column for the total execution time has been labelled t_0. Note that the garbage collection times in most cases are small. This is a result of having a heap which is much larger than the size of the live data.

Table 10.4 gives a similar breakdown of the execution times for the benchmarks when compiled with HBC. Note in particular the garbage collection times which in some cases are significantly larger than what is the case when the programs are compiled with Freja. This is likely a result of the HBC run-time system using smaller heaps than what the Freja run-time system does. However, since the garbage collection times do not dominate the execution times in any case, a comparison between HBC and Freja in terms of reduction time only would roughly give the same results as in table 10.2.

Benchmark	t_{tot} [s]	t_{tot}/t_0	t_{red} [s]	t_{GC} [s]	OCS [kbyte]
Ackermann	15.8	1.17	15.5	0.3	6
Sieve	9.7	1.31	9.6	0.1	6
Isort	21.2	1.44	21.0	0.2	5
Crypt	24.2	1.41	24.2	0.0	35
Mini-Freja	43.6	1.30	37.4	6.2	99

Table 10.5: Instrumentation overhead. The programs have been compiled for debugging, but they have been executed without activating the debugger. Average times over 5 runs.

10.3 Instrumentation overhead

Table 10.5 gives execution times and size of the object code when the benchmarks are compiled for debugging with the Freja compiler. However, the programs have been executed without activating the debugger (i.e. no EDT is built). Thus the table shows the overhead caused by the instrumentation of the generated code and loss of optimization opportunities.

The execution time has increased by 17–44 % which is acceptable. The size of the object code has increased by 30–50 % (compare table 10.3). However, for such small programs as these, this increase is marginal in comparison with the size of the run-time system, the debugger, and the libraries (the Freja version of the standard Prelude), all of which are linked with the object code to form an executable. For instance, the size of the ordinary Mini-Freja executable is 377 kbyte compared with 529 kbyte when compiled for debugging. Most of this increase is due to linking with the debugger.

10.4 Debugging cost

This section evaluates the performance of the system when performing debugging. The five benchmark programs have been compiled with debugging support and the execution time when building the initial part of the EDT has then been measured for various settings of the parameters $N_{\max}$ and $RG_{\max}$.

Note that a program may have to be re-executed several times during a debugging session. Thus, if the debugging cost were to be measured as the total time for all needed re-executions, it would be much larger than indicated here. However, debugging is an interactive activity, so what really matters is response time. The debugging overhead for a single re-execution

Benchmark	T
Ackermann	6 559 705
Sieve	6 366 044
Isort	12 517 505
Crypt	14 602 980
Mini-Freja	16 469 854

Table 10.6: The number of traced reductions for each benchmark.

is therefore more interesting than the total overhead since the former gives an indication of the worst-case response time.

The re-execution frequency should also be taken into account when judging the debugging cost. Provided re-executions do not occur too often, relatively long re-execution times can probably be tolerated since the average response time still would be low. The tables in this section therefore include a column giving the estimated query distance of the nodes furthest from the root of the stored EDT portion ($QD_{\max}$). This gives a rough indication of the number of questions that can be answered before the target program is re-executed.

Table 10.7 shows the performance for different values of $RG_{\max}$, the maximal size of the graph retained by the EDT. In each case, the bound on the number of EDT nodes, $N_{\max}$, has been set to a high value so as to avoid that this limit interferes. This is successful in most cases. The main exception is Ackermann where the EDT hardly retains any extra pieces of graph. $RG_{\max}$ does thus not constitute a useful limit on the EDT size in this case. $N_{\max}$ also interferes with the Mini-Freja benchmark for $RG_{\max} = 8$ Mbyte and $RG_{\max} = 16$ Mbyte, and possibly with Sieve for $RG_{\max} = 16$ Mbyte. In the case of Mini-Freja, lack of primary memory prevented us from using more than 800 000 nodes without causing thrashing (excessive paging).

The results in table 10.7 are more or less as expected. The time spent on garbage collection increases with increasing $RG_{\max}$. In most cases it quickly becomes the dominating part of the total execution time. The time for building the tree is small in comparison, but also tends to grow as the size of the stored part of the tree grows.

Crypt constitutes a troublesome case. The problem is that there are very large nodes (nodes which retain a lot of heap, several megabytes each) close to the root of the EDT. (This can be seen more clearly in table 10.8). When the debugger tries to keep the size of the tree below $RG_{\max}$, the result is that it throws away almost the entire tree (see section 5.4.1). The situation gets even worse when debugging is started and a large node ends up as the current root as the result of a subsequent re-execution. The debugger

$RG_{\max}$ [Mbyte]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Ackermann								
1	99911	0.46	154	34	2.5	1.1	1.1	0.3
2	99911	0.46	154	34	2.5	1.1	1.1	0.3
4	99911	0.46	154	34	2.5	1.1	1.1	0.3
8	99911	0.46	154	34	2.5	1.1	1.1	0.3
16	99911	0.46	154	34	2.5	1.1	1.1	0.3
Sieve								
1	29	0.81	13	16	2.1	1.3	0.4	0.4
2	407	1.57	55	18	2.4	1.3	0.6	0.5
4	6329	3.92	223	29	3.9	1.3	2.0	0.6
8	24978	6.85	446	33	4.5	1.3	2.4	0.8
16	99683	12.4	892	41	5.6	1.3	3.4	0.9
Isort								
1	58	0.96	13	33	2.3	1.4	0.5	0.4
2	328	1.82	28	41	2.8	1.4	1.0	0.4
4	1434	3.28	56	65	4.4	1.4	2.6	0.4
8	5998	6.39	112	79	5.4	1.4	3.6	0.4
16	24534	11.50	224	93	6.4	1.4	4.5	0.5
Crypt								
1	3	0.01	8	33	1.9	1.4	0.3	0.2
2	2	0.00	7	35	2.0	1.4	0.4	0.2
4	6	0.03	10	42	2.5	1.4	0.9	0.2
8	6	0.03	10	52	3.0	1.4	1.3	0.3
16	2	0.00	6	75	4.4	1.4	2.6	0.4
Mini-Freja								
1	31795	0.55	304	74	2.2	1.1	0.8	0.3
2	131862	1.23	609	82	2.4	1.1	1.0	0.3
4	218922	1.84	782	95	2.8	1.1	1.3	0.4
8	799767	5.85	1486	201	6.0	1.1	4.5	0.4
16	799767	5.85	1486	201	6.0	1.1	4.5	0.4

Table 10.7: Performance for different values of $RG_{\max}$. $N_{\max} = 100\ 000$ except for the Mini-Freja benchmark where $N_{\max} = 800\ 000$.

may then be unable to keep the size of the retained graph within the desired limits. The result is excessive garbage collection costs (and thrashing in case the primary memory resources are also exhausted).

One solution is to watch out for the special case of having a large node as the root. Whenever this is the case, the size of the root could be reduced by pruning the graphs retained by it. This may risk throwing away information important for debugging, but as long as a reasonable amount of graph is kept, this is probably not a big problem in practice. An alternative would be to attempt to put a general bound on the node size by pruning retained pieces of graph wherever necessary. (This would also risk throwing away important information, of course.)

Table 10.8 shows the performance for different values of $N_{\max}$, the maximal number of stored EDT nodes. The bound on the size of the retained pieces of graph, $RG_{\max}$, has been set to 64 Mbyte which means that it does not interfere, as can be seen from the column RG. Note that Crypt again is problematic. At most 50 nodes could be stored; after that, too much graph was retained causing excessive paging.

Again, the results are fairly unsurprising. As the number of nodes in the stored portions of the trees grows, the size of the retained graph grows and so do the garbage collection times. The EDT construction times again grow slowly, but as Ackermann and Sieve show, EDT construction sometimes account for a significant part of the total execution time.

Table 10.9 shows the result when $N_{\max}$ and $RG_{\max}$ interact. The bounds have been set to 10 000 nodes and 4 Mbyte respectively. The increase in execution time is below a factor of 3 in all cases, while $QD_{\max}$ indicates that a reasonably large portion of the tree has been stored (except in the case of Crypt).

In conclusion, these benchmarks show that the instrumentation overhead and the cost of building the EDT are low. The costly part of tracing, both in terms of time and space, lies in retaining pieces of graph which otherwise would have been discarded. As the tables show, the time spent on garbage collection can easily account for 75 % or more of the execution time when the retained graph is getting large. This and the fact that memory resources are limited demonstrate the importance of bounding the amount of graph retained by the EDT. Note, however, that it is also important to bound the number of EDT nodes since each EDT node occupies some space (56 bytes in the current implementation) and the number of EDT nodes could become very large if the only size limitation were the amount of retained graph. The Ackermann benchmark is a case in point.

The large overhead for garbage collection is partly due to Freja using a simple two-space copying garbage collector (see chapter 9). A generational garbage collection scheme would almost certainly be beneficial since it is

$N_{\max}$ [nodes]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\mathrm{tot}}}{t_0}$	$\frac{t_{\mathrm{red}}}{t_0}$	$\frac{t_{\mathrm{GC}}}{t_0}$	$\frac{t_{\mathrm{EC}}}{t_0}$
Ackermann								
5000	4697	0.02	37	18	1.3	1.1	0.1	0.1
10000	9575	0.04	49	18	1.3	1.1	0.1	0.1
20000	19742	0.08	67	20	1.5	1.1	0.2	0.2
50000	49592	0.21	103	25	1.9	1.1	0.6	0.2
100000	99911	0.46	154	34	2.5	1.1	1.1	0.3
200000	199703	1.05	273	56	4.2	1.1	2.6	0.5
500000	499703	3.03	714	219	16.2	1.1	14.3	0.8
1000000	999653	6.58	1639	231	17.1	1.1	14.7	1.3
Sieve								
5000	4952	3.58	198	18	2.4	1.3	0.8	0.3
10000	9872	4.60	280	28	3.7	1.3	2.0	0.4
20000	19902	6.27	398	27	3.6	1.3	1.8	0.5
50000	49772	9.25	630	39	5.3	1.3	3.3	0.7
100000	99683	12.37	892	42	5.6	1.3	3.4	0.9
200000	199398	16.54	1262	52	7.0	1.3	4.5	1.2
500000	499502	21.62	1998	77	10.5	1.3	7.3	1.9
Isort								
5000	4953	5.69	102	40	2.7	1.4	1.1	0.2
10000	9873	8.08	143	45	3.1	1.4	1.4	0.3
20000	19903	10.97	202	57	3.9	1.4	2.2	0.3
50000	49773	16.84	318	75	5.1	1.4	3.4	0.3
100000	99684	24.56	449	102	7.0	1.4	5.1	0.5
200000	199399	33.19	634	141	9.6	1.4	7.3	0.9
Crypt								
10	9	0.03	11	29	1.7	1.4	0.0	0.3
20	14	27.41	12	105	6.1	1.4	4.3	0.4
50	50	40.97	15	120	7.0	1.4	5.4	0.2
Mini-Freja								
5000	4930	0.36	126	50	1.5	1.1	0.2	0.2
10000	9952	0.40	174	50	1.5	1.1	0.2	0.2
20000	19978	0.47	243	52	1.6	1.1	0.3	0.2
50000	49841	0.66	378	55	1.6	1.1	0.3	0.2
100000	99710	1.01	531	62	1.8	1.1	0.5	0.2
200000	199565	1.70	747	74	2.2	1.1	0.9	0.2
500000	499217	3.77	1176	124	3.7	1.1	2.3	0.3
1000000	999353	7.23	1660	282	8.4	1.1	6.7	0.6

Table 10.8: Performance for different values of $N_{\max}$. $RG_{\max} = 64$ Mbyte.

Benchmark	N [nodes]	RG [Mb]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Ackermann	9575	0.04	49	18	1.3	1.1	0.1	0.1
Sieve	2487	2.76	140	20	2.8	1.3	1.1	0.4
Isort	531	2.21	35	39	2.6	1.4	1.0	0.2
Crypt	9	0.03	11	34	2.0	1.4	0.3	0.3
Mini-Freja	9952	0.40	174	51	1.5	1.1	0.2	0.2

Table 10.9: Performance for $N_{\max} = 10\ 000$ and $RG_{\max} = 4$ Mbyte.

likely that a large part of the graph retained by the EDT quickly would be moved to an old generation. Earlier experiments carried out in the context of HBC, which has a generational collector [Röj95] (among others), indicate that this indeed is the case (see [NS96]). However, even for a generational collector the garbage collection time increases with the size of the live data, so a generational scheme does not eliminate the need for an upper bound on the size of the graph retained by the EDT (ignoring that the amount of available memory in any case is limited).

Furthermore, the results in table 10.7 and, in particular, table 10.8 hint at an interesting fact: due to the increasing cost of garbage collection as the size of the stored portion of the EDT grows, it may well be cheaper *overall* to execute a target program a few times with a low bound on the size than to execute the same target only once with bounds set sufficiently high to allow the entire tree to be stored. The reason is that only a fraction of the nodes in an EDT typically are visited during debugging, so the re-execution cost is offset by the cost of maintaining irrelevant nodes. If the latter is higher than the former, the piecemeal scheme wins. Had a generational collector been used, the effect might not have been so marked, but it would still be there.

Another interesting fact is that re-execution of the entire target program is not as wasteful as it first may seem whenever garbage collection and construction of the desired portion of the EDT constitute major parts of the execution cost. Naish & Barbour [NB95] propose a partial re-execution scheme based on inferring the demand context from the stored result of the application which is re-evaluated. While such a scheme would be beneficial (as long as the gains are not offset by hidden implementation costs), the overhead of garbage collection and tree construction puts an upper bound on the obtainable speedup. For instance, if the combined overhead of garbage collection and tree construction is roughly equal to the execution time of the target, then the speedup would be at most two.

Benchmark	T_{U}	T_{T}	$\frac{T_{\mathrm{U}}}{T_{\mathrm{T}}}$	$t_{\mathrm{tot,U}}$ [s]	$t_{\mathrm{tot,T}}$ [s]	$\frac{t_{\mathrm{tot,U}}}{t_{\mathrm{tot,T}}}$
Ackermann	13 119 422	6 559 705	2.00	22	18	1.2
Sieve	6 415 741	6 366 044	1.01	18	20	0.9
Isort	12 571 404	12 517 505	1.00	39	39	1.0
Crypt	17 178 085	14 602 980	1.18	34	34	1.0
Mini-Freja	24 973 360	16 469 854	1.52	55	51	1.1

Table 10.10: The effects of trusting the standard Prelude in terms of the number of traced reductions (calls to *Trace*) and total execution time. Symbols with index T refer to a trusted Prelude, symbols with index U refer to an untrusted Prelude. $N_{\max} = 10\ 000$ and $RG_{\max} = 4$ Mbyte.

10.5 Effects of trusting the Prelude

Table 10.10 quantifies the effects of trusting the standard Prelude (see section 6.4). The five benchmarks have been linked both with a trusted version of the Prelude (i.e. the assertion `Trusted` has been inserted into the source code of all Prelude modules) and with an untrusted version of it (no assertions) and then executed. The values of $N_{\max}$ and $RG_{\max}$ are as for table 10.9.

As can be seen from table 10.10, the number of traced reductions drops when the Prelude is trusted, in some cases significantly.[2] This shows that the declarations of trust are effective in avoiding storing irrelevant nodes. This in turn saves the user from having to answer many unnecessary questions.

The effects on the execution times are less marked. Table 10.10 gives a breakdown. The execution times remain relatively unchanged since the amount of graph to be garbage collected did not change much. Note, however, that the tree construction cost for Ackermann and Mini-Freja has increased as a consequence of significantly more *Trace* calls. The somewhat peculiar result for Sieve is due to the EDTs having very different shapes in the two cases. This happened to result in less time spent on garbage collection and EDT construction when the Prelude was untrusted (compare table 10.9). This is not a general effect for Sieve; other combinations of the parameters result in slightly longer execution times for the untrusted Prelude, as one would expect. It should also be kept in mind that the number of re-executions needed to find a hypothetical bug in Sieve could increase as a result of not trusting the Prelude.

[2]Note that Isort hardly makes use of the Prelude at all. Hence its behaviour is virtually unaffected by the declarations of trust.

Benchmark	N [nodes]	RG [Mb]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\mathrm{tot}}}{t_0}$	$\frac{t_{\mathrm{red}}}{t_0}$	$\frac{t_{\mathrm{GC}}}{t_0}$	$\frac{t_{\mathrm{EC}}}{t_0}$
Ackermann	9593	0.04	49	22	1.6	1.1	0.1	0.4
Sieve	9914	3.59	198	18	2.4	1.3	0.8	0.3
Isort	692	2.63	34	39	2.6	1.4	1.0	0.2
Crypt	11	0.03	10	34	2.0	1.4	0.3	0.3
Mini-Freja	9909	0.40	147	55	1.7	1.1	0.3	0.3

Table 10.11: Detailed breakdown of the performance figures for an untrusted Prelude. $N_{\max} = 10\,000$ and $RG_{\max} = 4$ Mbyte. Compare table 10.9.

10.6 Summary

The results presented in this chapter chiefly demonstrate three things. First, given reasonable constraints on the size of the stored portion of the EDT, the cost of debugging is acceptable. For typical values of the size constraints, the execution time increases by a factor of 2 to 4.

Second, as the maximal size of the stored portion of the EDT increases, the time spent on garbage collection also increases and starts to dominate the total execution time. In one case the target ran 17 times slower than usual; almost all time was spent on garbage collection. (Should the primary memory resources not be sufficient, the cost of excessive paging also has to be taken into account.) The time taken to build the tree also increases with the size of the stored part of the EDT, but not to the same extent. Piecemeal EDT construction is thus a vital part of a realistic debugger.

Third, the size of the stored portion of the EDT must be limited both in terms of the number of stored EDT nodes and in terms of the amount of memory it occupies; some target programs result in EDTs where the retained pieces of graph are very small, whereas in other cases even a tiny fraction of the nodes in an EDT can hold on to vast graphs.

The benchmarks also demonstrated that the inability of the current implementation to guarantee that the bound on the size of the retained graph is always respected could cause performance problems. This should be fixed. The solution is to prune the retained graph as a last resort, even though this may loose some information important for debugging.

Finally, declaring the standard Prelude to be trusted was successful in reducing the number of traced application and hence the number of irrelevant questions during debugging. The effects on the execution times were less pronounced.

Chapter 11

Related Work

In this chapter we survey the work which is most closely related to ours or which we find interesting in this context for some other reason. It is not an exhaustive account of the lazy debugging field, nor an overview of all debugging techniques which might be applicable. We present the work roughly in chronological order.

11.1 Hall & O'Donnell

Hall & O'Donnell [HO85, OH88] were among the first who investigated the particular problems of lazy functional debugging. They focused on implementing debugging tools within an interactive, purely functional environment, implemented in the language itself (Daisy, a lazy descendant of Lisp).

In their papers, they suggest several approaches to debugging lazy functional programs. One suggestion is to transform the source code of the entire target program so that, in addition to its normal value, the program produces a trace of its execution. The trace is constructed in a way which reflects the structure of the source code, and it is in many ways similar to our EDT. However, one problem, which O'Donnell & Hall also point out, is that the very printing of the trace might turn an otherwise terminating program into a non-terminating one. This happens when the trace contains references to infinite data structures or diverging computations which normally would not be printed. In this respect their approach is thus very different from ours: we insist that debugging should not affect the semantics of the target. O'Donnell & Hall observe that this problem can be solved by introducing a primitive for determining demand (cf. section 11.7), but refrain from doing so because of the implications for the language semantics and because it violates their debugging philosophy of implementing all tools *in* the language, without resorting to impure extensions.

In another approach O'Donnell & Hall rely on working in an interpretative Lisp-like environment, where the function definitions of the target program can be accessed and manipulated from within the environment, and arbitrary expressions constructed and evaluated by calling the system function `eval`. The fundamental idea is to let the user interactively traverse the program, invoke functions, print the values of local variables etc. Thus an infinite value would only be printed if the user asks for it, and since the user can interrupt an evidently non-terminating printing process, this to some extent overcomes the main problem of the trace based approach. However, modern lazy functional languages like Haskell are not Lisp-like in the above sense, and thus this debugging technique is not directly applicable.

11.2 Toyn & Runciman

Toyn & Runciman [TR86, Toy87] propose a system in the context of a combinator reduction machine, where the graph is annotated in such a way that a limited amount of computational history of the various pieces of graph on the heap is maintained as annotations in the graph. Thus it is possible, at any time instant, to take a 'snapshot' of the state of the computation. This snapshot shows the current graph (in source level terms) annotated with the stored part (the last application) of the computational history for each value.

A snapshot would typically be taken when the program aborts due to a run-time 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 bottom-up testing to deal with this case.

The reason for not storing the the entire computation history is that this would require too much space. By limiting the amount of stored data, the time overhead of using the tool is 50 % and the space overhead below 25 % for a set of small benchmarks.

11.3 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, relying on a 'meta-evaluation rule' to get rid of as many unevaluated expressions as possible. The rule simply states that values should be shared, i.e. they should be represented by pointers to unique heap-allocated objects, meaning that values in the trace will be as evaluated as possible once the computation has terminated.

Thus Kamin gives a formal definition of a trace that reflects the structure of the source code in much the same way as we do in chapter 4. However, in comparison to our specification, Kamin uses a simpler language than we do (no currying and no local let-bindings for instance) which simplifies his specification in many ways. Furthermore, the information needed for properly displaying tree nodes and values to the user, such as names of free variables and user defined functions, is not explicitly included in the trace. Instead Kamin assumes that this information can be obtained from the existing run-time representation of values, which often is not the case for existing language implementations. We think it is an advantage to explicitly include this information in the trace since it makes it clear that the issue must be addressed in the context of existing language implementations. Finally, sharing is explicit in our specification; we do not rely on any 'meta-evaluation' rule.

11.4 Naish

Naish [Nai92] proposed an algorithmic debugger for a lazy functional language which is implemented by translation into Prolog. The translation is achieved through '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 normal result, also return an execution tree with a structure mirroring the structure of the source code. Moreover, closures are represented in such a way that they, when evaluated, are 'updated' both with the result and the execution tree that resulted when the suspended computation was forced. (The implementation makes use of logic variables and a meta predicate `var/1` which can be used to check whether a variable is bound (here indicating something evaluated) or not (indicating something which was not evaluated).) Again, this is similar to our EDT, but the source language appears to be quite simple, e.g. no local bindings and thus no need to worry about free variables.

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. This is a more precise way of saying that the user need not worry about values which were never evaluated (cf. section 4.1.1).

11.5 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 path is then 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 or diverging computations. Though clearly somewhat limited in scope, the tool has been found useful and it was used extensively in the maintenance and development of a large Miranda system for natural language understanding (Lolita, about 12 000 lines of code).

11.6 Kishon & Hudak

Kishon & Hudak [KH95] take an approach similar to Kamin's, but more general and systematic. Starting from a denotational continuation semantics of a language, they derive a monitoring semantics by composing the standard semantics with one or more monitor specifications. For example, by composing a semantics for a lazy language with a monitor specification for a lazy tracer, an instrumented interpreter for the lazy language that generates a trace is obtained. They also derive an (operational) source level debugger with the ability to interactively force evaluation of thunks. To obtain acceptable performance, Kishon & Hudak use partial evaluation to get rid of as much interpretative overhead as possible.

Since the monitoring semantics is given separately from the language semantics, the trace of Kishon & Hudak is specified in a somewhat more indirect way than in our case. Its structure is also different from the structure of our EDT since it is a linear sequence of function calls and returns, reflecting the lazy evaluation order rather than the structure of the source code. Arguments and results are however shown in their most evaluated form. For debugging purposes, we think that a source code related structure is preferable. Another problem is that functional values cannot be shown properly. To solve this, the semantic definition of the language would have to be changed so that the representation of functional values carries additional information, such as the name of the function and the name and values of free variables.

From a practical point of view, neither Kishon & Hudak nor Kamin address debugging in the context of currently available language implementations, and it is an open question whether sufficiently efficient implementations can be automatically derived in the way they suggest. (Nor do they address the problem of large memory consumption.)

11.7 Sparud

In his work, Sparud [Spa94, Spa96] takes a transformational approach to debugging lazy functional programs. The idea is to transform all functions so that they return an execution record in addition to their normal result. Sparud's aim is to provide a debugging tool which is as portable as possible.

The approach is thus in many ways similar to that of O'Donnell & Hall [OH88]. However, Sparud is slightly more pragmatic and makes use of an impure primitive, `evaluated`, to avoid forcing unevaluated expressions. Furthermore, Sparud works in Haskell, a strongly typed language, so his transformations have to respect the type discipline whereas this was not an issue for O'Donnell & Hall who worked in Daisy, a lazy variant of Lisp. To convert values of any type to printable form, Sparud uses Haskell's class system by introducing a class `EDTAble` with a suitable, overloaded, conversion method `edtVal`. Thus there is no need to rely on the language implementation to provide a way of printing an arbitrary value, which O'Donnell & Hall did.

There are also similarities to Naish's work [Nai92]. In particular, the use of the impure function `evaluated` corresponds to the way Naish uses `var/1`. However, Sparud considers source-to-source transformations, whereas Naish changes a transformation based language implementation. Naish also works with untyped languages. Nevertheless, since Naish's target language is NU-Prolog, a declarative language, the the resulting code is in many ways similar.

Sparud and this author have co-authored two papers and a technical report [SN95, NS96, NS97]. In these publications, we developed a common execution record, the EDT, but explored different ways of constructing it: Sparud pursued the transformational approach whereas this author developed the method presented in this thesis.

11.8 Naish & Barbour

The work by Naish & Barbour [NB95] is closely related to Sparud's work [Spa94, Spa96], and there are also similarities to the work presented in this thesis. Naish & Barbour use a source-to-source transformation, similar to Sparud's, which transform the target into a program that generates a tree representing a suitable view of the execution in addition to its normal output. No explicit specification of the generated trace is given, however, and the source language is very simple: a set of top-level function defining equations where the right-hand sides consist solely of function applications. There are no local let-definitions and no currying.

A key difference between their transformation and Sparud's is that they rely on an impure function `dirt` (Display Intermediate Reduced Term), which must be supplied by the underlying language implementation. As its name suggests, it converts *any* value, without evaluating it further, to a suitable term representation that may be printed, and thus combines the functionality of Sparud's impure primitive `evaluated` and the overloaded `edtVal`. It is interesting to note that the use of `dirt` puts Naish's and Barbour's approach somewhere in between Sparud's EDT generation approach and the one presented in this thesis, since the latter also assumes that any piece of graph on the heap can be displayed in source-level terms.

Thanks to `dirt` Naish's and Barbour's transformation is simplified with respect to Sparud's since there is no need to handle functional values in a special way. On the other hand, requiring a function like `dirt` makes their approach significantly less portable. For example, in the Chalmers Haskell implementation HBC [Aug97], which is one of the major implementations, values on the heap do not normally carry precise type information, let alone, in the case of functional values, information about function name and free variables. Implementing `dirt` in HBC would thus be a major undertaking. Implementing `evaluated` is very simple in comparison.

Naish and Barbour also consider the memory consumption problem and suggest generating parts of the tree on demand. Unlike our piecemeal scheme, they do not require the entire program to be re-executed each time a new part of the tree is needed. Instead, once a node at the fringe of the stored portion of the tree is reached, they re-apply the function of that node to its arguments, and then compare this application to the *evaluated* parts of the result of the previous application of the functions, which is also stored in the node. This will drive the computation exactly the right amount for constructing the tree below the node in question. Note that `dirt` again plays a crucial role since comparing against *unevaluated* parts of the result would drive the computation beyond what was originally computed which is unsafe.

From the description given in Naish & Barbour, it seems as if the re-execution scheme so far only has been implemented in an untyped language (a lazy functional subset of NUE-Prolog) where `dirt` does not have to inject the values it inspects into a data type like `EDTValue`. In a strongly typed language `dirt` would have to do so, and the re-execution mechanism would then probably need additional support in the form of a specialized comparison routine.

As to how much of a tree to store, Naish & Barbour suggest building nodes down to a certain, predetermined, depth. (Then the normal, untransformed, versions of the functions can be called to obtain better performance.) As explained in section 5.3.3, this does not give a good handle on how much space the stored portion of the tree really occupies, so in general quite few nodes would probably be stored. This in turn could lead to frequent, partial, re-executions, which are not necessarily much cheaper than a complete re-execution. In this respect it is advantageous to work at the language implementation level, since it then is possible adjust the number of nodes in the tree based on how much space they really occupy. Indeed, based on our experience with the Crypt benchmark (see section 10.4), we would claim that this in necessary since even a single node holding on to some 'infinite' data structure that grows as the execution proceeds could cause the memory consumption to get out of hand. Fixing this requires low-level corrective measures, e.g. co-operation from the garbage collector.

11.9 Sparud & Runciman

Recently, Sparud & Runciman have proposed an alternative debugging method based on maintaining complete computational histories for all values [SR97]. They call these histories *redex trails*. The idea is that it should be possible to single out an erroneous value and follow its history backwards until the bug is found. Note that other erroneous values may be encountered during this process, but since *all* values are associated with a trail, it is then just a matter of following the one of the other trails instead.

Like Sparud's earlier work [Spa94, Spa96], the implementation is based on transformations with some support from the implementation. The transformations currently handle most of Haskell. However, Sparud & Runciman report that the time and space costs are still too high, and work are currently under way to address this. They mention possibilities such as making use of the garbage collector for pruning the trails and avoiding tracing of trusted functions (e.g. from the Prelude). Another idea is to store the trails on secondary storage in a compressed binary format.

Another interesting aspect of their work is a very sophisticated user interface.

11.10 Tolmach & Appel

Finally we would also like to mention the debugger for Standard ML of New Jersey (SML/NJ) developed by Tolmach & Appel [TA95]. The implementation is based on instrumenting the target by source level transformations, which makes the debugger fairly portable. Unlike the transformations discussed above, communication of debug information does not take place via function arguments and results. Instead, calls to the debugger are inserted at strategic places, and all communication with the user is done via imperative side-effects. Thus the types of the functions in the transformed program remain unchanged. The debugger also features an advanced re-execution mechanism (called *time travel*), which relies on the first-class continuations of SML/NJ for its implementation.

An interesting problem concerns displaying values of variables with polymorphic types. Since the representation of values does not carry run-time type information in SML/NJ, they rely on an algorithm for dynamic type reconstruction which is based on inspecting the call chain that was current at the time of *binding*. Such an algorithm may be an alternative to requiring that the run-time representation of values is such that a value always can be printed in the appropriate way. However, we have not investigated whether the algorithm would work in a lazy context.

Chapter 12

Concluding Remarks

We have now almost reached the end of this thesis, and it is time to sum up what has been achieved and point out some possible directions for future work. We do so in the following.

12.1 Summary

The objective of the work presented herein was to demonstrate that declarative debugging is an attractive as well as feasible way to debug lazy functional programs. To this end

- we proposed a particular execution record, the Evaluation Dependence Tree, as a suitable basis for declarative debugging of lazy functional programs (chapter 4),
- we showed how such an execution record could be constructed efficiently and, through piecemeal tracing, how to limit the memory consumption (chapters 5 and 10),
- we showed how the user, statically and dynamically, could convey information regarding where the bug might be located, and how the compiler and debugger could make use of this information to construct the EDT more efficiently and improve the debugging process by reducing the number of asked questions (chapters 6 and 10),
- we described how to support source-level debugging within our framework for a large subset of Haskell, including list comprehensions (chapter 7),
- we demonstrated how our debugger could be used for debugging a program exhibiting a range of common symptoms such as non-termination and a black hole (chapter 8),

- we developed a reference implementation, available to anyone who is interested, and we demonstrated that it has reasonable performance (chapters 9 and 10).

So, has the thesis fulfilled its objective? Well, that is not really for us to say. Not until widely-used debuggers for Haskell (or some other lazy language) emerge will we know to what extent language implementors and users have found our arguments convincing. However, we believe that we have a compelling case.

12.2 Future work

12.2.1 Improved garbage collection

Based on the experience from earlier experiments [NS96], we believe that a generational garbage collection scheme would be very beneficial in our case. The intuition is that pieces of graph retained only from the EDT are likely to become quite old and thus should be ideally suited for being moved to an old generation. Perhaps the EDT should have an old generation of its own.

Another improvement might be to throw away unreduced redexes retained solely by the EDT and replace them with a special node meaning 'unevaluated'. They will obviously never be reduced, and, as explained in section 4.1.1, they will be shown to the user as ? anyway, so there is really no point in keeping these redexes unless it is desired to give the user the possibility to inspect unevaluated expressions.

12.2.2 Pruning large EDT nodes

As indicated by our performance measurements (see section 10.4, the Crypt benchmark), the possibility that very large EDT nodes end up close to the current root of the stored portion of the EDT can be a severe performance problem. This must be addressed. Solutions include pruning of large nodes close to the root as a special case, and a general bound on the node size which might be enforced by the garbage collector. However, note that there in both cases is a risk of throwing away information important for debugging.

12.2.3 Improved trace class inference

In section 6.4 we showed how to avoid tracing of trusted functions. The idea hinges on partitioning the functions into trace classes depending on to what extent they need to co-operate with the tracing mechanisms. In the best case, a function becomes *untraced* which means that it is compiled without any debugging support. Such a function thus execute at full speed.

Furthermore, whenever it is statically known that an untraced function is being applied, there is no need to build a traced redex.

Trusted functions which apply unknown functions in their bodies end up in one of the two *invisible* trace classes. This is because they must co-operate with the trace mechanisms in order to ensure that the unknown functions are properly traced if necessary. Reductions of applications of trusted functions are, on the other hand, not very interesting and can thus be made invisible (i.e. no EDT nodes are created, except possibly summary nodes in the case of recursive functions). Typical examples of functions in this category are `(.)` (function composition), `map`, and `foldl`.

Now, suppose `f` and `g` (of arity 1) are functions which belong to the trace class *untraced*. Consider the following applications:

```
(f . g) 42
map f [1, 2, 3]
map (f . g) [13, 17, 42]
```

A moment of thought should convince the reader that tracing of these redexes is not necessary. If the redexes are traced, `(.)` and `map` will, when called, build traced redexes for the applications of their function-valued argument(s). But since the actual arguments are untraced functions, (`f`, `g`, and the composition of the two) no EDT nodes will ensue (except for a summary node in the case of `map`), and all the work of building traced redexes and calling the tracer has been wasted. This is particularly severe in the case of `map` which wastes time and effort on building traced applications for the function-valued argument and the *recursive* call to `map`. Unfortunately, the current Freja implementation makes the applications above traced simply because `(.)` and `map` are not untraced; it does not know that these functions are traced only to make it possible to trace applications of actual parameters which happen to be traced.

For another example, consider the following function definition:

```
foo x = (f . g) x
```

Here, `foo` becomes traced because its body contains a traced redex. But as we saw above, it is not necessary to trace this redex, so it ought to be possible to make `foo` untraced.

We have not investigated a more precise analysis in any depth, and we do not know to what extent the currently employed scheme really constitutes a problem. It should be noted that the number of EDT nodes in the complete tree would not be decreased drastically given better analysis; we can only hope for more efficient EDT construction. Since the time for EDT construction seems to be relatively small (see chapter 10), the possible gains are probably limited.

Nevertheless, it would be interesting to experiment with an improved analysis, and we speculate that this could be done through a simple type-based analysis where the function arrows are annotated with trace information indicating whether an application needs to be traced or not.

Another idea might be to employ *closure analysis*, i.e. statically determining which functions actually reach which program points (see e.g. [Hen92]). If it is found that all these functions are untraced, then better code can be generated. It may be desirable to perform closure analysis anyway in order to perform other optimizations, so in practice this optimization might not require any extra effort.

12.2.4 Alternative ways of selecting starting-points

When a run-time error occurs, i.e. the result of some application is undefined, it is often the case that the cause of the problem is located in the neighbourhood of the failed application in the EDT. Thus it would be useful to have the possibility of starting debugging close to the node corresponding to the failed application, rather than from the root of the EDT.

If the complete EDT is present this is easy to achieve: it is just a matter of locating the node in question and starting debugging from its parent, grandparent or wherever is deemed suitable. Under a piecemeal scheme, it is possible to get to the desired point by re-executing the program once provided the reference to the desired starting node is known. One method would be to annotate a redex not only with a reference to its parent in the EDT, but also with references to its grandparent, great grandparent and so on up to some predefined number of levels.

Alternatively, complete 'call chains' containing EDT references could be maintained, similar to what has been proposed by Hazan [HM93] but constructed at a lower level. Indeed, the call chains could be the only record built during the first execution. Since a node in a call chain would only hold an EDT reference, it is much smaller than an EDT node and it might be feasible to keep the complete structure in primary memory. Also note that partial chains no longer referenced from any redex can be garbage collected since they are only interesting in the event of a reduction failing.

A third alternative would be to flatten the structure and write it to secondary storage. Recall that a reference to an EDT node just is the corresponding reduction number. When reduction i takes place, the reference to the parent, the *pid* of the traced redex, is appended to a log file. Thus, at position i in the file, the reference to the parent of reduction i will be found. This reference is just a reduction number itself and can be used to index the log file again to get to the grandparent, great grandparent and so on.

The idea above could be extended by also writing some identification of the applied function to the file (one 32 bit word is more than enough). The file would then become a fairly compact encoding of the call tree (i.e. the EDT without any arguments and results), and it could be shown to the user in a browser as a means of selecting interesting starting points for proper EDT construction.

12.2.5 Improved granularity

The units of debugging in the current implementation are mostly whole functions. The exception is list comprehensions where each 'loop' becomes a unit of debugging. By introducing extra lambda abstractions which are invoked as a result of some interesting debugging event, the debugging process would become more fine-grained. For example, it might be possible to follow a complicated pattern-matching process involving guards more closely if pattern matching literally was implemented using explicit representations of matching failure and an operator for catching failures and trying alternatives as described in Peyton Jones [PJ87, pp. 51–77].

12.2.6 Handling monads

Monad structured programs [Wad92] constitute a problem. In principle it is of course possible to debug such programs since they are purely functional. But the purpose of introducing a monad is to raise the level of abstraction by hiding unimportant details, and if this is not taken into account when debugging, the result is that debugging takes place at the wrong conceptual level. This is particularly troublesome since monadic code typically makes use of textually large, anonymous functions (lambda abstractions). Having said this, it is difficult to see how the debugger should be able to figure out what a programmer had in mind when introducing a monad and adapt the debugging strategy in accordance. Perhaps the solution lies in having a flexible debugger which somehow can be customized by the user.

There are, however, important special cases. One concerns monadic I/O. Here we have the added complication that the monad is used to ensure that the 'state of the world' is used in a single threaded manner so as to make an efficient implementation, where the world is updated destructively, possible. Lazy functional state threads [LPJ94] constitute a similar case: a monad ensures that the state component is used in a single threaded way which allows destructive updating.

In both these cases, the conceptual purpose of the monad is clear: permitting stateful computations to be expressed in a way reminiscent of imperative programming. Perhaps the most appropriate debugging technique for such parts of the code is then also imperative.

In the case of lazy functional state threads, declarative debugging is still an option by simply substituting a naive, purely functional implementation of the monad. Of course, the price paid in time and space might be high. A better approach in this case could be to substitute an implementation based on version arrays [AHN88] or some similar device. Since the state component is used in a single threaded manner, the efficiency price should not be too high while allowing a complete history of the state to be maintained with a modest increase in memory consumption. It might even be possible to handle the state of the world in a similar way in the case of the I/O monad.

12.2.7 Handling full Haskell

Finally, our techniques should be extended to cover all of Haskell. With the exception of special monadic support, as discussed above, we do not see any major obstacles. The main thing which is missing from Freja in comparison with Haskell is general type and constructor classes (see appendix A). Adding these means that one name can denote many different functions. For disambiguation, one possibility would be to rely on the argument and result types in the posed questions. Alternatively, the debugger could qualify method names with the name of the instance type. Since type classes are implemented by a transformation which adds extra dictionary arguments to functions, some effort would be required to make it possible for the debugger to identify these arguments so that it can hide the effects of the transformation. But this is not very different from handling the effects of lambda-lifting.

As regards trace class analysis in the presence of overloading, it would become more imprecise: a method for which there is not sufficient type information to specialize to a particular instance is similar to a function-valued argument, i.e. an unknown function. However, there is a difference: the functions which actually could be applied are only those which are instances of the method. This is a syntactic property, so no static analysis (like closure analysis) is required to find out which they are, and if it turns out that all instances are untraced, then better code can be generated. Unfortunately, this is also a *global* property, i.e. all modules must be taken into account. This means that separate compilation would have to be sacrificed, or that the analysis would have to be postponed until link-time. But in the latter case there would be less scope for improving the performance since the code has already been compiled.

An alternative would be to have *trusted classes*: the user promises that all instances of the methods in the class are trusted in a particular program. At link-time, it could then be checked that the user has fulfilled his promise. If not a warning or an error could be issued. Note that our system makes it easy to perform things like these at 'link-time' thanks to the run-time system generator (see chapter 9), which collects information from all modules and is executed just prior to linking all object files together into an executable.

Appendix A

Freja

A.1 The main differences between Freja and Haskell

Our compiler is called Freja (for historical reasons) and currently implements what essentially is a large subset of Haskell 1.4 [PHA+97]. In this thesis, the currently implemented Haskell subset is also referred to as 'Freja' when the distinction is important.

The major limitations of Freja w.r.t. Haskell are the following:

- Very limited implementation of the class mechanisms. No overloading on the result type. No constructor classes.
- Predefined classes for handling common cases of overloading (e.g. arithmetic operations, equality) with automatic instance derivation where appropriate.
- No user-defined type classes.
- No special monadic support (no `do`-notation).
- No monadic I/O.
- No implementation of the standard libraries or support for the datatypes they define (in particular, no arrays).

These are of course major functional limitations, but we still think that Freja qualifies as a 'large' Haskell subset. The module system is almost completely implemented, for instance, labelled constructors are supported, and a large part of the standard Prelude is there (subject to the limitations above). Moreover, we do not think that adding the missing features would result in any major problems as far as debugging support is concerned, save for the general problems of debugging monadic code (see section 12.2.6).

A.2 Freja Syntax

A.2.1 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.2.2 Lexical syntax

program → { *lexeme* | *whitespace* }
lexeme → *varid* | *conid* | *quotedid* | *varsym* | *consym* | *literal*
| *special* | *reservedop* | *reservedid*
literal → *integer* | *float* | *char* | *string*
special → `(` | `)` | `,` | `;` | `[` | `]` | `_` | `‘` | `{` | `}`

whitespace → { *whitestuff* }$^+$
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *vertab* | *formfeed* | *space* | *tab* | *nonbrkspc*
newline → a newline (system dependent)
space → a space
tab → a horizontal tab
vertab → a vertical tab
formfeed → a form feed
nonbrkspc → a non-breaking space
comment → `--` { *any* } *newline*
ncomment → `{-` *ANYseq* { *ncomment* *ANYseq* } `-}`
ANYseq → $\{\ ANY\ \}_{\langle \{\ ANY\ \}\ (\ \texttt{\{-} \mid \texttt{-\}}\ )\ \{\ ANY\ \} \rangle}$
ANY → *any* | *newline* | *vertab* | *formfeed*
any → *graphic* | *space* | *tab* | *nonbrkspc*

graphic → Any ISO 8859-1 symbol with graphic representation, including space.
small → *ASCsmall* | *ISOsmall*
ASCsmall → `a` | `b` | ... | `z`
ISOsmall → `à` | `á` | `â` | `ã` | `ä` | `å` | `æ` | `ç` | `è` | `é` | `ê` | `ë`
| `ì` | `í` | `î` | `ï` | `ð` | `ñ` | `ò` | `ó` | `ô` | `õ` | `ö` | `ø`

```
               |  ù | ú | û | ü | ý | þ | ÿ | ß

large          →  ASClarge | ISOlarge
ASClarge       →  A | B | ... | Z
ISOlarge       →  À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë
               |  Ì | Í | Î | Ï | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
               |  Ù | Ú | Û | Ü | Ý | Þ |

symbol         →  ! | # | $ | % | & | * | + | . | / | < | = | > | ? | @
               |  \ | ^ | | | - | ~

digit          →  0 | 1 | ... | 9
octit          →  0 | 1 | ... | 7
hexit          →  digit | A | ... | F | a | ... | a
extdigit       →  digit | large | small

varid          →  ( small { small | large | digit | ' | _ } )⟨reservedid⟩
conid          →  large { small | large | digit | ' | _ }
quotedid       →  ( ' { graphic⟨' | \⟩ | escape } ' )⟨char⟩
reservedid     →  case | data | do | else | if | import
               |  in | infix | infixl | infixr | let | module
               |  newtype | of | then | type | where
specialid      →  as | qualified | hiding

varsym         →  ( symbol { symbol | : } )⟨reservedop⟩
consym         →  ( : { symbol | : } )⟨reservedop⟩
reservedop     →  .. | :: | = | \ | | | <- | -> | @ | ~ | =>
specialop      →  - | !

tyvar          →  varid
tycon          →  conid⟨tycls⟩
tycls          →  Eq | Ord | Enum | Num | Real | Floating
               |  Integral | RealFloat | Show | Eval

modid          →  conid

qvarid         →  [ modid . ] varid
qconid         →  [ modid . ] conid
qquotedid      →  [ modid . ] quotedid
qtycon         →  [ modid . ] tycon
```

qvarsym → [*modid* `.`] *varsym*
qconsym → [*modid* `.`] *consym*

decimal → { *digit* }$^+$
octal → { *octit* }$^+$
hexadecimal → { *hexit* }$^+$

integer → *decimal* | *basespec* { *extdigit* }$^+$
basespec → *decimal* `_` | `0o` | `0O` | `0x` | `0X`
float → *decimal* `.` *decimal* [(`e` | `E`) [`-` | `+`] *decimal*]

char → `'` ($graphic_{\langle \texttt{'} \mid \texttt{\textbackslash} \rangle}$ | $escape_{\langle \texttt{\textbackslash\&} \rangle}$) `'`
string → `"` { $graphic_{\langle \texttt{"} \mid \texttt{\textbackslash} \rangle}$ | *escape* | *gap* } `"`
escape → `\` (*charesc* | *ascii* | *decimal*
| `o` *octal* | `x` *hexadecimal*)
charesc → `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | `'` | `&`
ascii → `^`*cntrl* | `NUL` | `SOH` | `STX` | `ETX` | `EOT` | `ENQ` | `ACK`
| `BEL` | `BS` | `HT` | `LF` | `VT` | `FF` | `CR` | `SO` | `SI` | `DLE`
| `DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
| `EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
cntrl → *ASClarge* | `@` | `[` | `\` | `]` | `^` | `_`
gap → `\` { *whitechar* }$^+$ `\`

A.2.3 Context-free syntax

module → `module` *modid* [*exports*] `where` *body*
body → `{` [*impdecls* [`;`]] [*fixdecls* [`;`]] [*topdecls* [`;`]] `}`

exports → `(` [*export* { `,` *export* }] `)`
export → *qvar*
| *qtycon* [`(..)`]
| `module` *modid*

impdecls → *impdecl* { `;` *impdecl* }
impdecl → `import` [`qualified`] *modid* [`as` *modid*] [*impspec*]
impspec → `(` [*import* { `,` *import* }] `)`
| `hiding` `(` [*import* { `,` *import* }] `)`
import → *var*
| *tycon* [`(..)`]

fixdecls → *fix* { `;` *fix* }
fix → `infixl` [*digit*] *ops*

	$\mid$	**infixr** [*digit*] *ops*
	$\mid$	**infix** [*digit*] *ops*
ops	$\rightarrow$	*op* { , *op* }
topdecls	$\rightarrow$	*topdecl* { ; *topdecl* }
topdecl	$\rightarrow$	**type** *simpletype* = *type*
	$\mid$	**data** [*context* =>] *simpletype* = *constrs*
	$\mid$	**newtype** [*context* =>] *simpletype* = *con atype*
	$\mid$	*decl*
decls	$\rightarrow$	[*decl* { ; *decl* }]
decl	$\rightarrow$	*signdecl*
	$\mid$	*valdef*
decllist	$\rightarrow$	**{** *decls* [;] **}**
signdecl	$\rightarrow$	*vars* :: [*context* =>] *type*
vars	$\rightarrow$	*var* { , *var* }
type	$\rightarrow$	*btype* [-> *type*]
btype	$\rightarrow$	[*btype*] *atype*
atype	$\rightarrow$	*gtycon*
	$\mid$	*tyvar*
	$\mid$	(*type* { , *type* }$^+$)
	$\mid$	**[** *type* **]**
	$\mid$	(*type*)
gtycon	$\rightarrow$	*qtycon*
	$\mid$	()
	$\mid$	**[]**
	$\mid$	(->)
	$\mid$	(, { , })
context	$\rightarrow$	*class*
	$\rightarrow$	(*class* { , *class* })
class	$\rightarrow$	*tycls tyvar*
simpletype	$\rightarrow$	*tycon* { *tyvar* }
constrs	$\rightarrow$	*constr* { | *constr* }
constr	$\rightarrow$	*con* { [**!**] *atype* }
	$\mid$	(*btype* | **!** *atype*) *conop* (*btype* | **!** *atype*)

$$
\begin{array}{lll}
 & | & \mathit{con}\ \texttt{\{}\ \mathit{fielddecl}\ \{\ \texttt{,}\ \mathit{fielddecl}\ \}\ \texttt{\}} \\
\mathit{fielddecl} & \rightarrow & \mathit{vars}\ \texttt{::}\ (\ \mathit{type}\ |\ \texttt{!}\ \mathit{atype}\ ) \\
 & & \\
\mathit{valdef} & \rightarrow & \mathit{lhs}\ \texttt{=}\ \mathit{exp}\ [\ \texttt{where}\ \mathit{decllist}\ ] \\
 & | & \mathit{lhs}\ \mathit{gdrhs}\ [\ \texttt{where}\ \mathit{decllist}\ ] \\
\mathit{lhs} & \rightarrow & \mathit{pat}^{0} \\
 & | & \mathit{funlhs} \\
\mathit{funlhs} & \rightarrow & \mathit{var}\ \{\ \mathit{apat}\ \}^{+} \\
 & | & \mathit{pat}^{i+1}\ \mathit{varop}^{(a,i)}\ \mathit{pat}^{i+1} \\
 & | & \mathit{lpat}^{i}\ \mathit{varop}^{(\mathrm{l},i)}\ \mathit{pat}^{i+1} \\
 & | & \mathit{pat}^{i+1}\ \mathit{varop}^{(\mathrm{r},i)}\ \mathit{rpat}^{i} \\
\mathit{gdrhs} & \rightarrow & \{\ \mathit{gd}\ \texttt{=}\ \mathit{exp}\ \}^{+} \\
\mathit{gd} & \rightarrow & \texttt{|}\ \mathit{exp}^{0} \\
 & & \\
\mathit{exp} & \rightarrow & \mathit{exp}^{0}\ \texttt{::}\ [\ \mathit{context}\ \texttt{=>}\ ]\ \mathit{type} \\
 & | & \mathit{exp}^{0} \\
\mathit{exp}^{i} & \rightarrow & \mathit{exp}^{i+1}\ [\ \mathit{qop}^{(\mathrm{n},i)}\ \mathit{exp}^{i+1}\ ] \\
 & | & \mathit{lexp}^{i} \\
 & | & \mathit{rexp}^{i} \\
\mathit{lexp}^{i} & \rightarrow & (\ \mathit{lexp}^{i}\ |\ \mathit{exp}^{i+1}\ )\ \mathit{qop}^{(\mathrm{l},i)}\ \mathit{exp}^{i+1} \\
\mathit{lexp}^{6} & \rightarrow & \texttt{-}\ \mathit{exp}^{7} \\
\mathit{rexp}^{i} & \rightarrow & \mathit{exp}^{i+1}\ \mathit{qop}^{(\mathrm{r},i)}\ (\ \mathit{rexp}^{i}\ |\ \mathit{exp}^{i+1}\ ) \\
\mathit{exp}^{10} & \rightarrow & \texttt{\textbackslash}\ \{\ \mathit{apat}\ \}^{+}\ \texttt{->}\ \mathit{exp} \\
 & | & \texttt{let}\ \mathit{decllist}\ \texttt{in}\ \mathit{exp} \\
 & | & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
 & | & \texttt{case}\ \mathit{exp}\ \texttt{of}\ \texttt{\{}\ \mathit{alts}\ [\ \texttt{;}\ ]\ \texttt{\}} \\
 & | & \mathit{fexp} \\
 & & \\
\mathit{fexp} & \rightarrow & [\ \mathit{fexp}\ ]\ \mathit{aexp} \\
 & & \\
\mathit{alts} & \rightarrow & \mathit{alt}\ \{\ \texttt{,}\ \mathit{alt}\ \} \\
\mathit{alt} & \rightarrow & \mathit{pat}\ \texttt{->}\ \mathit{exp}\ [\ \texttt{where}\ \mathit{decllist}\ ] \\
 & | & \mathit{pat}\ \mathit{gdpat}\ [\ \texttt{where}\ \mathit{decllist}\ ] \\
\mathit{gdpat} & \rightarrow & \{\ \mathit{gd}\ \texttt{->}\ \mathit{exp}\ \}^{+} \\
 & & \\
\mathit{aexp} & \rightarrow & \mathit{qvar} \\
 & | & \mathit{gcon} \\
 & | & \mathit{literal} \\
 & | & \texttt{(}\ \mathit{exp}\ \texttt{)} \\
 & | & \texttt{(}\ \mathit{exp}\ \{\ \texttt{,}\ \mathit{exp}\ \}^{+}\ \texttt{)}
\end{array}
$$

$$
\begin{array}{lcll}
 & | & \texttt{[}\ exp\ \{\ \texttt{,}\ exp\ \}\ \texttt{]} & \\
 & | & \texttt{[}\ exp\ [\ \texttt{,}\ exp\ ]\ \texttt{..}\ [\ exp\ ]\ \texttt{]} & \\
 & | & \texttt{[}\ exp\ \texttt{|}\ qual\ \{\ \texttt{,}\ qual\ \}\ \texttt{]} & \\
 & | & \texttt{(}\ exp^{i+1}\ qop^{(a,i)}\ \texttt{)} & \\
 & | & \texttt{(}\ qop^{(a,i)}\ exp^{i+1}\ \texttt{)} & \\
 & | & qcon\ \texttt{\{}\ [\ fbind\ \{\ \texttt{,}\ fbind\ \}\ ]\ \texttt{\}} & \\
 & | & aexp_{\langle qcon \rangle}\ \texttt{\{}\ fbind'\ \{\ \texttt{,}\ fbind'\ \}\ \texttt{\}} & \\
 & & & \\
qual & \rightarrow & pat\ \texttt{<-}\ exp & \\
 & | & exp & \\
 & & & \\
fbind & \rightarrow & var\ |\ qvar\ \texttt{=}\ exp & \\
 & & & \\
fbind' & \rightarrow & var\ |\ qvar\ \texttt{=}\ exp\ |\ \texttt{\~{}}\ qvar & \\
 & & & \\
pat & \rightarrow & var\ \texttt{+}\ integer & \\
 & | & pat^0 & \\
pat^i & \rightarrow & pat^{i+1}\ [\ qconop^{(\mathrm{n},i)}\ pat^{i+1}\ ] & \\
 & | & lpat^i & \\
 & | & rpat^i & \\
lpat^i & \rightarrow & (\ lpat^i\ |\ pat^{i+1}\ )\ qconop^{(\mathrm{l},i)}\ pat^{i+1} & \\
lpat^6 & \rightarrow & \texttt{-}\ (\ integer\ |\ float\ ) & \\
rpat^i & \rightarrow & pat^{i+1}\ qconop^{(\mathrm{r},i)}\ (\ rpat^i\ |\ pat^{i+1}\ ) & \\
pat^{10} & \rightarrow & apat & \\
 & | & gcon\ \{\ apat\ \}^+ & (\text{arity } gcon \geq 1) \\
 & & & \\
apat & \rightarrow & var\ [\ \texttt{@}\ apat\ ] & \\
 & | & gcon & (\text{arity } gcon = 0) \\
 & | & qcon\ \texttt{\{}\ [\ fpat\ \{\ \texttt{,}\ fpat\ \}\ ]\ \texttt{\}} & \\
 & | & literal & \\
 & | & \texttt{\_} & \\
 & | & \texttt{(}\ pat\ \texttt{)} & \\
 & | & \texttt{(}\ pat\ \{\ \texttt{,}\ pat\ \}^+\ \texttt{)} & \\
 & | & \texttt{[}\ pat\ \{\ \texttt{,}\ pat\ \}\ \texttt{]} & \\
 & | & \texttt{\~{}}\ apat & \\
 & & & \\
fpat & \rightarrow & qvar\ \texttt{=}\ pat & \\
 & | & var & \\
 & & & \\
gcon & \rightarrow & qcon &
\end{array}
$$

	|	`()`
	|	`[]`
	|	`( , { , } )`
var	→	*varid* | `(` *varsym* `)`
qvar	→	*qvarid* | `(` *qvarsym* `)`
con	→	*conid* | `(` *consym* `)` | *qoutedid*
qcon	→	*qconid* | `(` *qconsym* `)`
	|	*qqoutedid* | *con*
varop	→	*varsym* | ` *varid* `
qvarop	→	*qvarsym* | ` *qvarid* `
	|	` *quotedid* `
conop	→	*consym* | ` *conid* `
qconop	→	*qconsym* | ` *qconid* `
	|	` *qquotedid* ` | *conop*
op	→	*varop* | *conop*
qop	→	*qvarop* | *qconop*

A.3 Predefined Types and Classes

A.3.1 Built-in types

The current Freja implementation supports machine-word-sized integers (`Int`) and double precision floating point numbers (`Double`). As usual, there are special, syntactical, provisions for lists, arbitrarily large tuples (memory permitting), and characters. The other standard types (`Bool`, `Maybe`, `Ordering`, etc.) are ordinary types defined in the Prelude.

A.3.2 The Freja class system

Freja does not have user definable type classes as found in e.g. Haskell. There is, however, a built in class system for dealing with overloading of certain standard functions and operators. Moreover, user defined types automatically become instances of certain of these classes when appropriate. There is currently no way to override the system derived instance methods, e.g. it is not possible to have a user defined `(==)`.

The class hierarchy is basically a simplified version of the standard Haskell hierarchy, and the methods of each class are mostly the same as in the corresponding Haskell class. There are however differences, motivated by the simplified hierarchy as well as by the fact that Freja does not allow overloading resolution solely on the result type. The last point means that there are no methods like `fromInteger :: Num a => Integer -> a` or `toEnum :: Enum a => Int -> a` in Freja. This also means that Freja

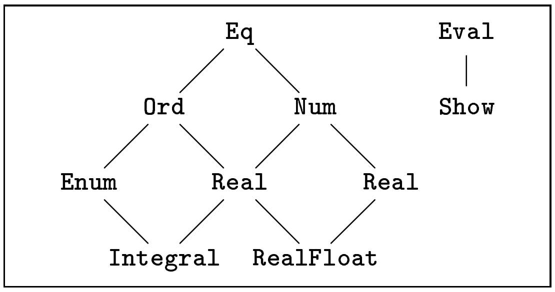


Figure A.1: The Freja class hierarchy.

does not have overloaded numerical literals. The Freja class Hierarchy is shown in figure A.1.

Since the only kinds of number in Freja currently are `Int` and `Double`, the class hierarchy is unnecessarily general. However, as it stands it permits numbers of other kinds, such as arbitrary precision integers and complex numbers, to be easily added.

In the following sections, these classes and their methods are presented. Haskell syntax is used to define the classes.

The Eq class

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x /= y = not (x == y)
```

All basic datatypes except functions are instances of the `Eq` class. An instance is automatically derived for any user-defined datatype whose constituents are instances of `Eq`. (This is defined to include mutually recursive types as well, as long as all constituents which do not belong to the group of mutually recursive types are instances of `Eq`.)

The Ord class

```
class (Eq a) => Ord a where
        compare                 :: a -> a -> Ordering
        (<), (<=), (>=), (>)    :: a -> a -> Bool

        x <= y = compare x y /= GT
        x <  y = compare x y == LT
        x >= y = compare x y /= LT
        x >  y = compare x y == GT
```

The `Ord` class is used for totally ordered datatypes. All basic datatypes except functions are instances of this class. An instance is automatically derived for any user-defined datatype whose constituents are instances of `Ord`.

The following auxiliary functions are defined in the Prelude:

```
max :: Ord a => a -> a -> a
max x y | x >= y    = x
        | otherwise = y

min :: Ord a => a -> a -> a
min x y | x <  y    = x
        | otherwise = y
```

The Enum class

```
class (Ord a) => Enum a where
        enumFrom       :: a -> [a]
        enumFromThen   :: a -> a -> [a]
        enumFromTo     :: Enum a => a -> a -> [a]
        enumFromThenTo :: Enum a => a -> a -> a -> [a]
        succ           :: a -> a
        pred           :: a -> a

        enumFromTo n m = takeWhile (<= m) (enumFrom n)

        enumFromThenTo n n' m =
            takeWhile (if n' >= n then (<=m) else (>=m))
                      (enumFromThen n n')
```

Class `Enum` defines operations on sequentially ordered types. Instances are derived for all enumeration types (a datatype whose constructors are all nullary). The primitive type `Int` is also an instance.

The methods `enumFrom` and `enumFromThem` never return infinite lists: once the next element on the list would be outside the type in question, the enumeration stops. This also holds for `Int`. (In the current implementation, all methods above are defined as auxiliary functions built on a set of more primitive methods.)

Some auxiliary functions and constants are also automatically defined for enumeration types. If `X` is an enumeration type, then:

`minX`	`::`	`X`	The lowest numbered constr. in the type.
`maxX`	`::`	`X`	The highest numbered constr. in the type.
`toX`	`::`	`Int -> X`	Maps `Int` onto `X`.
`fromX`	`::`	`X -> Int`	Maps `X` onto `Int`.

The Eval class

```
class Eval a where
        seq     :: a -> b -> b
```

An `Eval` instance is automatically derived for every datatype. The following auxiliary function is defined in the Prelude:

```
strict :: (Eval a) => (a -> b) -> a -> b
strict f x = x `seq` f x
```

The Show class

```
type ShowS = String -> String

class (Eval a) => Show a where
        showsPrec :: Int -> a -> ShowS
```

The methods in the `Show` class is used to convert arbitrary objects to textual representations. All basic types are instances, and instances are automatically derived for all user-defined types. Some auxiliary functions are defined in the Prelude:

```
shows :: (Show a) => a -> ShowS
shows = showsPrec 0

show :: (Show a) => a -> String
show x = shows x ""

showList :: (Show a) => [a] -> ShowS
showList []     = showString "[]"
showList (x:xs) = showChar '[' . shows x . showl xs
    where
        showl []     = showChar ']'
        showl (x:xs) = showString ", "
                       . shows x
                       . showl xs

showChar :: Char -> ShowS
showChar = (:)

showString :: String -> ShowS
showString = (++)

showParen :: Bool -> ShowS -> ShowS
```

```
showParen b p = if p
                then showChar '(' . p . showChar ')'
                else p
```

The Num class

```
class (Eq a) => Num a where
        (+), (-), (*)   :: a -> a -> a
        (^)             :: a -> Int -> a
        negate          :: a -> a
        abs, signum     :: a -> a
intToNum :: a -> Int -> a
doubleToNum :: a -> Double -> a
```

Every numeric type is an instance of this class, currently `Int` and `Double`. The Prelude defines the following auxiliary function:

```
subtract :: (Num a) => a -> a -> a
subtract = flip (-)
```

The Real class

```
class (Ord a, Num a) => Real a where
        toInt           :: a -> Int
        toDouble        :: a -> Double
```

Every real numeric type is an instance of this class. Currently `Int` and `Double`. The methods of this class are used for coercion between real types.

The Integral class

```
class (Enum a, Real a) => Integral a where
        quot, rem, div, mod     :: a -> a -> a
        quotRem, divMod         :: a -> a -> (a,a)
        even, odd               :: a -> Bool
```

Every integral type is an instance of this class. Currently only `Int`, but e.g. arbitrary precision integers could also be an instance. The Prelude defines the following auxiliary functions:

```
gcd :: Integral a => a -> a -> a
gcd x y | x == zero && y == zero
          = error "Prelude.gcd: gcd 0 0 is undefined"
        | otherwise = gcd' (abs x) (abs y)
    where
```

```
        gcd' x y | y == zero = x
                 | otherwise = gcd' y (x `rem` y)
        zero = intToNum x 0

lcm :: Integral a => a -> a -> a
lcm x y | x == zero || y == zero = zero
        | otherwise = abs ((x `quot` (gcd x y)) * y)
    where
        zero = intToNum x 0
```

The Floating class

```
class (Num a) => Floating a where
        (/)                     :: a -> a -> a
        recip                   :: a -> a
        exp, log, sqrt          :: a -> a
        (**), logBase           :: a -> a -> a
        sin, cos, tan           :: a -> a
        asin, acos, atan        :: a -> a
        sinh, cosh, tanh        :: a -> a
        asinh, acosh, atanh     :: a -> a
```

Every numeric type based on floating point representation is an instance of this class. Currently only `Double`, but e.g. complex numbers could also be an instance of this class. The Prelude defines the following auxiliary function:

```
(^^) :: (Floating a) => a -> Int -> a
x ^^ n = if n >= 0 then x^n else recip (x^(-n))
```

The RealFloat class

```
class (Real a, Floating a) => RealFloat a where
        truncate, round, ceiling, floor :: a -> Int
```

The only instance is `Double`.

Appendix B

The Trace Algorithm

This appendix contains the C code for the central parts of the EDT generation algorithm. A brief explanation of the stateless interface presented to the rest of the debugger is also included, together with listings of the interface routines for illustrating how it works.

B.1 Important data types

The type `tt_node` is the internal representation of EDT nodes. A simplified version was presented in section 5.4.2 where it was called `edt_node`. A number of macros are defined to facilitate access and manipulation. There is also an external representation of EDT nodes. This is used by the rest of the debugger, i.e. the user interface, which sees it as an abstract datatype.

```
typedef struct tt_node {
   unsigned           ttn_id_f;
   int                ttn_qd_f;
   obj_info           ttn_fun_info_f;
   list               ttn_args_f;          /* List of graph */
   graph              ttn_result_f;
   struct tt_node     *ttn_parent_f;
   struct tt_node     *ttn_leftsib_f;
   struct tt_node     *ttn_rightsib_f;     /* Valid if rightsib_qd = 0 */
   int                ttn_rightsib_qd_f;
   struct tt_node     *ttn_firstchd_f;     /* Valid if firstchd_qd = 0 */
   struct tt_node     *ttn_lastchd_f;      /* Valid if any children. */
   int                ttn_firstchd_qd_f;
   struct tt_node     *ttn_next_f;
   struct tt_node     *ttn_next_same_qd_f;
} trace_tree_node;


#define ttn_id(ttnp)            ((ttnp)->ttn_id_f)
```

```
#define ttn_qd(ttnp)              ((ttnp)->ttn_qd_f)
#define ttn_fun_info(ttnp)        ((ttnp)->ttn_fun_info_f)
#define ttn_args(ttnp)            ((ttnp)->ttn_args_f)
#define ttn_result(ttnp)          ((ttnp)->ttn_result_f)
#define ttn_parent(ttnp)          ((ttnp)->ttn_parent_f)
#define ttn_leftsib(ttnp)         ((ttnp)->ttn_leftsib_f)
#define ttn_rightsib(ttnp)        ((ttnp)->ttn_rightsib_f)
#define ttn_rightsib_qd(ttnp)     ((ttnp)->ttn_rightsib_qd_f)
#define ttn_has_rightsib(ttnp)    ((ttnp)->ttn_rightsib_qd_f == 0 \
                                   && (ttnp)->ttn_rightsib_f != NULL)
#define ttn_has_had_rightsib(ttnp) ((ttnp)->ttn_rightsib_qd_f != 0)
#define ttn_firstchd(ttnp)        ((ttnp)->ttn_firstchd_f)
#define ttn_lastchd(ttnp)         ((ttnp)->ttn_lastchd_f)
#define ttn_firstchd_qd(ttnp)     ((ttnp)->ttn_firstchd_qd_f)
#define ttn_has_children(ttnp)    ((ttnp)->ttn_firstchd_qd_f == 0 \
                                   && (ttnp)->ttn_firstchd_f != NULL)
#define ttn_has_had_children(ttnp) ((ttnp)->ttn_firstchd_qd_f != 0)
#define ttn_next(ttnp)            ((ttnp)->ttn_next_f)
#define ttn_next_same_qd(ttnp)    ((ttnp)->ttn_next_same_qd_f)

/* The rightsib and firstchd fields must ALWAYS be set using the following
 * macros.
 */

#define ttn_set_rightsib(ttnp, rs)      \
do {                                    \
   ttn_rightsib(ttnp) = (rs);           \
   ttn_rightsib_qd(ttnp) = 0;           \
} while(0)

#define ttn_set_rightsib_qd(ttnp, rsqd) \
do {                                    \
   ttn_rightsib_qd(ttnp) = (rsqd);      \
} while(0)

#define ttn_set_firstchd(ttnp, fc)      \
do {                                    \
   ttn_firstchd(ttnp) = (fc);           \
   ttn_firstchd_qd(ttnp) = 0;           \
} while(0)

#define ttn_set_firstchd_qd(ttnp, fcqd) \
do {                                    \
   ttn_firstchd_qd(ttnp) = (fcqd);      \
} while(0)
```

The type `edt_node` is the external representation of an EDT node. It is used by the user interface for accessing various attributes of a node (such as the arguments, the result, the first child, or the sibling to the right) through a few interface routines. See section B.6 for examples. The point is that the external representation of a node is valid regardless of whether that node belongs to the currently stored part of the EDT or not. The external representation is thus crucial for obtaining a stateless interface. The real

EDT node is located by looking it up in a hash table using the field `id`. If the node is not present, the parent identity (`pid`) and the relative query distance (`rqd`) is used to initialize the tracer in order to capture a part of the EDT where the desired node is present.

```
typedef struct {
   unsigned            id;     /* ID */
   unsigned            pid;    /* Parent ID */
   int                 rqd;    /* Relative QD = QD(ID) - QD(PID) */
} edt_node;
```

The type `tap_info` is used to access the attributes of a traced application node. The field `pid` is the identity of the parent of the redex, and `qd` is its estimated query distance.

```
typedef struct {
   unsigned     pid;
   int          qd;
} tap_info;
```

B.2 Important constants and global variables

The definitions of some important constants and global variables are listed below. `ttn_tbl_size` gives the maximum number of EDT nodes. It is initialized at the start of a debugging session. A table of `ttn_tbl_size` nodes is then allocated and assigned to `ttn_tbl`. `edt_heap_limit` gives the desired maximal size (in bytes) of the amount of memory held by the EDT on the heap. `htbl` is the hash table used to find a node given its identity. `qdix` is a table used to find all notes at a certain query distance from the current root. It is mainly used for pruning purposes. `NO_PARENT` is a special node *id* used to signal to the target that no tracing should take place. `EDT_ROOT` is the *id* of the EDT root. `NOT_EDT_ROOT`, for lack of a better name, is the special *id* of the root of the partial EDT which results when user-defined starting-points are being used.

```
private size_t ttn_tbl_size;
#define htbl_size 0x100000      /* Must be power of two. */
#define htbl_mask 0x0FFFFF      /* htbl_size-1 and must be all ones. */
#define qdix_size 10000
#define initial_max_qd (qdix_size - 1)

/* Special node IDs. */
#define NO_PARENT 0
#define NOT_EDT_ROOT 1
#define EDT_ROOT 2
/* first_id_less_one = max(NO_PARENT, NOT_EDT_ROOT, EDT_ROOT) - 1 */
#define first_id_less_one 1

#define hash(i) ((i) & htbl_mask)
```

```
private bool first_time_round = false;
private size_t edt_heap_limit;
private unsigned cur_id, start_id, stop_id = UINT_MAX;
private int initial_qd, max_qd;
private trace_tree_node *trace_root, *free_list;
private trace_tree_node *ttn_tbl;
private trace_tree_node *htbl[htbl_size];
private trace_tree_node *qdix[qdix_size];
```

B.3 Initialization

The routine `reinitialize_tracer` (re-)initializes the tracer's data structures and prepares for new trace starting from child number $(-\mathtt{iqd}+1)$ of node `sid`. Then the routine `execute_target` can be called to create the desired part of the EDT. See section B.6 for an example.

```
private void reinitialize_tracer(int sid, int iqd)
{
   int i;

   cur_id = first_id_less_one;
   start_id = sid;
   initial_qd = iqd;
   max_qd = initial_max_qd;

   /* Clear tables */
   for (i = 0; i < htbl_size; i++)
      htbl[i] = NULL;
   for (i = 0; i < qdix_size; i++)
      qdix[i] = NULL;

   /* Build free list */
   free_list = &(ttn_tbl[0]);
   for (i = 0; i < ttn_tbl_size - 1; i++) {
      ttn_tbl[i].ttn_id_f = 0; /* Indicates free node. */
      ttn_tbl[i].ttn_next_f = &(ttn_tbl[i + 1]);
   }
   ttn_tbl[ttn_tbl_size - 1].ttn_id_f = 0;
   ttn_tbl[ttn_tbl_size - 1].ttn_next_f = NULL;
} /* end reinitialize_tracer */
```

B.4 Trace

The routine `trace` is the main interface to the EDT construction routines for the graph reducer. It is explained in section 5.4.4. The next section lists the most important auxiliary tree construction routines.

```
void trace(tap_info *tapip, unsigned *idp, int *qdp,
           obj_info fi, graph args[], graph redex)
```

```
{
   if (!debugging) {
      *idp = NO_PARENT;                /* Disable tracing. */
      *qdp = 0;
      return;
   }

   /* Attempt to limit the heap consumption. */
   if (edt_heap_consumption > edt_heap_limit && max_qd > 3) {
      int target;

      edt_heap_consumption = 0;        /* No longer a valid measure. */
      while (qdix[max_qd] == NULL)     /* Find highest existing QD. */
         max_qd--;
      target = max_qd / 2;
      while (max_qd > target && max_qd > 2)
         prune_tree();
   }

   cur_id++;                           /* Increment identity counter. */
   if (cur_id >= stop_id) emulate_sigint();
   if (start_id != 0) {
      /* Waiting for tracing to start. */
      if (cur_id < start_id) {
         /* Not yet time to start tracing. */
         *idp = NO_PARENT;             /* Signal no tracing. */
         *qdp = 0;
      } else {
         /* cur_id >= start_id. Start tracing! */
         trace_root = mk_trace_tree_node(start_id, initial_qd,
                                         fi, args, redex, NULL);
         /* The sibling of the (current) root should *never* be accessed! */
         ttn_set_rightsib(trace_root, NULL);
         *idp = start_id;
         *qdp = initial_qd;
         start_id = 0;                 /* Indicate that tracing has begun. */
      }
   } else {
      /* We are tracing. */
      if (tapip != NULL && !is_untraced(fi)) {
         trace_tree_node *parent;

         parent = look_up_tree_node(tapip->pid);
         if (parent == NULL) {
            /* Parent has been removed. */
            *idp = NO_PARENT;          /* Disable tracing. */
            *qdp = 0;
         } else {
            /* Parent is present. */
            if (tapip->pid == NOT_EDT_ROOT && !fi_starting_point(fi) ||
                is_invisible(fi) ||
                is_inv_rec(fi) && (fi_fun_group(fi)
                                   == fi_fun_group(ttn_fun_info(parent)))) {
               /* This node should not be visible. Its children should
```

```
            be adopted by its parent.
            Note that we ignore qd <= 0 here, in case descendants of
            this node turns out to have qd > 0.                    */
         *idp = tapip->pid;
         *qdp = tapip->qd; /* This ensures that large number of equal
                              QDs occur late in the list of children.
                              Thus it is always possible to start
                              debugging.                           */
      } else {
         /* This is a node that is present in the full tree. */
         if (tapip->qd > 0) {
            /* Get rid of nodes wich user has seen. */
            if (tapip->qd <= max_qd) {
               if (free_list != NULL) {
                  /* There are free trace nodes. */
                  tt_insert(parent, cur_id, tapip->qd, fi, args, redex);
                  *idp = cur_id;          /* Enable continued tracing. */
                  *qdp = tapip->qd;       /* Initial qd. */
               } else {
                  /* No free trace nodes. */
                  if (tapip->qd < max_qd) {
                     /* Pruning will make room for the node and we
                        will have qd <= new max_qd afterwards. */
                     prune_tree();
                     tt_insert(parent, cur_id, tapip->qd, fi,args,redex);
                     *idp = cur_id;       /* Enable continued tracing. */
                     *qdp = tapip->qd;    /* Initial qd. */
                  } else {
                     /* qd == max_qd. Prune tree ONLY if this will not
                        remove the last nodes from the tree. Otherwise,
                        "fail safe" by throwing away the node that we
                        are trying to insert.                          */
                     int i;

                     for (i = max_qd - 1; i > 0 && qdix[i] == NULL; i--);
                     if (i > 0) {
                        /* Tree will not become empty. */
                        prune_tree();
                        pseudo_tt_insert(parent, tapip->qd);
                        *idp = NO_PARENT; /* Disable tracing. */
                        *qdp = 0;
                     } else {
                        /* Fail safe. Loose this node. */
                        printf("Failed to insert node! QD = %d\n",
                               tapip->qd);
                        *idp = NO_PARENT; /* Disable tracing. */
                        *qdp = 0;
                     }
                  }
               }
            } else {
               pseudo_tt_insert(parent, tapip->qd);
               *idp = NO_PARENT;  /* Disable tracing. */
               *qdp = 0;
```

```
                    }
                 }
              }
           }
        } else {
           /* Not a traced redex. */
           *idp = NO_PARENT;                      /* Disable tracing. */
           *qdp = 0;
        }
    }
} /* end trace */
```

B.5 Auxiliary tree construction routines

These routines inserts nodes into the EDT and performs pruning. The routines (together with `trace` which calls them under the appropriate circumstances) maintain the following invariants:

1. At any time, it is true that there is no stored tree node with a $qd >$ `max_qd`, and that all nodes seen so far with $qd \leq$ `max_qd` are present in the store.
2. At any time, the stored portion of the execution tree has a shape as if it had been created from the full tree by following the procedure below:
   (a) Identify the node where debugging is to be started. Call it N.
   (b) Remove all siblings to the left of N.
   (c) Remove all nodes that are not descendants of the parent of N.
   (d) Let q be the qd of N in the full tree. Adjust the qds of the nodes remaining in the tree by subtracting $(q-1)$ from them.
   (e) Prune the tree successively, starting from large qds working towards smaller ones, until the remaining nodes are so few that they will fit into the allocated storage
3. Whenever there are no free nodes, it is true that `max_qd` equals the largest qd of any stored node.

The routine `prune_tree` removes nodes with $qd =$ `max_qd` and lowers `max_qd` by one. It assumes that there are nodes with $qd =$ `max_qd` (invariant 3). Note that several children may have the same qd since nodes are not always created for functions in the invisible recursion trace class. These children are freed together.

```
private void prune_tree(void)
{
   trace_tree_node *ttnp, *removal_np, *nnp;
   int qd = max_qd;
   bool done;

   ttnp = qdix[max_qd];
   qdix[max_qd] = NULL;
   max_qd--;
   while (ttnp != NULL) {
      if (ttn_id(ttnp) != 0) {
         /* This node has not yet been freed. */
         if (ttn_qd(ttn_firstchd(ttn_parent(ttnp))) == qd) {
            /* All remaining children are to be removed. */
            removal_np = ttn_firstchd(ttn_parent(ttnp));
            ttn_set_firstchd_qd(ttn_parent(ttnp), qd);
         } else {
            /* There are siblings to the left with lower QD. Find the
               last such sibling and the first with same QD.          */
            trace_tree_node *ls;

            for (ls = ttn_leftsib(ttnp);
                 ttn_qd(ls) == qd;
                 ls = ttn_leftsib(ls));
            ttn_lastchd(ttn_parent(ttnp)) = ls;
            removal_np = ttn_rightsib(ls);
            ttn_set_rightsib_qd(ls, qd);
         }
         done = false;
         while (!done) {
            if (ttn_has_rightsib(removal_np))
               nnp = ttn_rightsib(removal_np);
            else
               done = true;
            remove_tree_node(removal_np); /* Sets ID to 0 for removed nodes. */
            removal_np = nnp;
         }
      }
      /* Note: next_same_qd untouched by remove_tree_node. */
      ttnp = ttn_next_same_qd(ttnp);
   }
} /* end prune_tree */
```

The routine `tt_insert` inserts a node in the tree when $qd \leq$ `max_qd`. The parent is assumed to exist and space is assumed to be available. Children are inserted in qd order. The order between children with the same qd is undefined.

```
private void tt_insert(trace_tree_node *parent, int id, int qd,
                       obj_info fi, graph args[], graph res)
{
   trace_tree_node *ttnp;
```

```
   ttnp = mk_trace_tree_node(id, qd, fi, args, res, parent);
   /* If no more free nodes, adjust max_qd to highest existing QD. */
   if (free_list == NULL) while (qdix[max_qd] == NULL) max_qd--;
   if (!ttn_has_children(parent)) {
      /* Parent have no children (but may have had). */
      ttn_leftsib(ttnp) = NULL;
      if (ttn_has_had_children(parent))
         ttn_set_rightsib_qd(ttnp, ttn_firstchd_qd(parent));
      else
         ttn_set_rightsib(ttnp, NULL);
      ttn_set_firstchd(parent, ttnp);
      ttn_lastchd(parent) = ttnp;
   } else {
      /* Parent have children. Insert new child at inv_rec place (QD order). */
      if (ttn_qd(ttn_firstchd(parent)) >= qd) {
         /* Insert new child first among the children. */
         ttn_leftsib(ttnp) = NULL;
         ttn_set_rightsib(ttnp, ttn_firstchd(parent));
         ttn_leftsib(ttn_rightsib(ttnp)) = ttnp;
         ttn_set_firstchd(parent, ttnp);
      } else if (ttn_qd(ttn_lastchd(parent)) <= qd) {
         /* Insert new child last among the children. */
         ttn_leftsib(ttnp) = ttn_lastchd(parent);
         if (ttn_has_had_rightsib(ttn_lastchd(parent)))
            ttn_set_rightsib_qd(ttnp, ttn_rightsib_qd(ttn_lastchd(parent)));
         else
            ttn_set_rightsib(ttnp, ttn_rightsib(ttn_lastchd(parent)));
         ttn_set_rightsib(ttn_lastchd(parent), ttnp);
         ttn_lastchd(parent) = ttnp;
      } else {
         /* Insert at right place among children.
          * Will NOT be first or last since QD is strictly between the QD
          * of the first and last child.
          */
         trace_tree_node *p, *n;

         /*   for (p = ttn_firstchd(parent), n = ttn_rightsib(p);
          *      ttn_qd(n) < qd;
          *      p = n, n = ttn_rightsib(p));
          */
         for (n = ttn_lastchd(parent), p = ttn_leftsib(n);
              ttn_qd(p) > qd;
              n = p, p = ttn_leftsib(n));
         ttn_leftsib(ttnp) = p;
         ttn_set_rightsib(ttnp, ttn_rightsib(p));
         ttn_set_rightsib(p, ttnp);
         ttn_leftsib(ttn_rightsib(ttnp)) = ttnp;
      }
   }
} /* end tt_insert */
```

The routine `pseudo_tt_insert` is a special version of the insertion routine which is used when inserting a node which is just outside the stored part of the EDT. It makes it look as if the node has been inserted by `tt_insert` and then removed by `prune_tree`. The parent is assumed to exist.

```
private void pseudo_tt_insert(trace_tree_node *parent, int qd)
{
   if (!ttn_has_had_children(parent) && !ttn_has_children(parent))
      /* Parent has no, and has not had any, children. */
      ttn_set_firstchd_qd(parent, qd);
   else {
      if (ttn_has_had_children(parent)) {
         /* Parent has had a child that has been removed. */
         if (qd < ttn_firstchd_qd(parent))
            ttn_set_firstchd_qd(parent, qd);
      } else {
         /* There are children. They must all have QD < qd. */
         if (ttn_has_had_rightsib(ttn_lastchd(parent))) {
            /* Last child has had a sibling to the right. */
            if (qd < ttn_rightsib_qd(ttn_lastchd(parent))) {
               ttn_set_rightsib_qd(ttn_lastchd(parent), qd);
            }
         } else {
            /* Last child has not had any siblings. */
            ttn_set_rightsib_qd(ttn_lastchd(parent), qd);
         }
      }
   }
} /* end pseudo_tt_insert */
```

B.6 The interface to the EDT navigator

The routines below are part of the interface between the EDT generator and the rest of the debugger, i.e. the EDT navigator. The routine **en_right_sibling** is used to get the right sibling of a node. There are similar routines for getting the EDT root (see below) and various attributes of nodes such as the first child or arguments and results. Note that the *external* representation of EDT nodes is used in the interface. Also note how the code arranges for re-execution if the node in question is not found. The EDT navigator thus has a stateless, referentially transparent interface to the EDT. As far as the navigator is concerned, the EDT is just an abstract datatype, and through the interface it 'sees' a complete tree, completely oblivious of low-level details such as piecemeal tracing and re-execution.

```
edt_node en_right_sibling(edt_node en)
{
   trace_tree_node *ttnp, *rsnp;
   edt_node sibling;

   if (en_is_null(en))
      fatal_error("Attempt to get sibling of NULL node (en_right_sibling).");
   if (en.pid == NO_PARENT) {
      /* Attempt to access sibling of the root of the entire tree. */
```

```
      en_mk_null(sibling);
   } else {
      ttnp = look_up_tree_node(en.id);
      if (ttnp == NULL || ttn_parent(ttnp) == NULL) {
         /* Node not present, or is the current root. Re-execute, starting
          * tracing from parent making the sought node its first child.    */
         reinitialize_tracer(en.pid, -(en.rqd) + 1);
         execute_target();
         if (ttn_has_had_children(trace_root))
            fatal_error("Unable to debug: empty tree (should not happen).");
         rsnp = ttn_firstchd(trace_root);
      } else if (ttn_has_had_rightsib(ttnp)) {
         /* Right sibling has been removed. Arrange for tracing to start at
          * parent node and reexecute program.                             */
         reinitialize_tracer(ttn_id(ttn_parent(ttnp)),
                             -(ttn_rightsib_qd(ttnp)
                               - ttn_qd(ttn_parent(ttnp)))+1);
         execute_target();
         if (ttn_has_had_children(trace_root))
            fatal_error("Unable to debug: empty tree (should not happen).");
         rsnp = ttn_firstchd(trace_root);
      } else {
         /* Node and its sibling, if any, present. */
         rsnp = ttn_rightsib(ttnp);
      }
      if (rsnp != NULL) {
         sibling.id = ttn_id(rsnp);
         sibling.pid = ttn_id(ttn_parent(rsnp));
         sibling.rqd = ttn_qd(rsnp) - ttn_qd(ttn_parent(rsnp));
      } else {
         en_mk_null(sibling);
      }
   }
   return sibling;
} /* end en_right_sibling */
```

The EDT which results from an execution is of course a function of user-defined starting-points and user assertions regarding the correctness of certain functions. Thus `edt_root`, the function which returns the root of the EDT, has two arguments: a list of starting-points (an empty list indicates that tracing should start from the root of the complete EDT) and a list of functions asserted to be trusted. See below. Note that these arguments are specified dynamically by the user, during the debugging, and that the assertions complement correctness assumptions made at compile-time.

In view of what was said about the statelessness of the interface above, it should be admitted that we have had to compromise in one respect for ease of implementation: the tracer only maintains *one* EDT at a time. Whenever `edt_root` is called, it is assumed that there is a new *trace case*, i.e. a new combination of starting-points and asserted functions, which results in a new EDT. Since only one EDT is maintained, this means that all references (external EDT nodes) to the previous EDT immediately become invalid and must be thrown away by the debugger. Thus the debugger interface is not stateless in this respect. This problem could be avoided if the external representation of EDT nodes contained a reference to the trace case which was used to build the EDT to which the node belongs. But this would be very expensive (frequent re-executions would be necessary if the debugger refers to many EDTs simultaneously) and of dubious practical utility.

```
edt_node edt_root(list start_funs, list asserted_funs)
{
   edt_node root;
   int i;

   /* Reset the first-time-round flag. */
   first_time_round = true;

   for (i = 0; i < fun_info_table_size; i++) {
      if (is_traceable(fun_info_table[i])) {
         fi_starting_point(fun_info_table[i]) = false;
         fi_trace_class(fun_info_table[i]) = fi_initial_tc(fun_info_table[i]);
      }
   }
   if (is_empty_list(start_funs)) {
      reinitialize_tracer(EDT_ROOT, 0);
   } else {
      for_each_list_element(fi, obj_info, start_funs) {
         if (is_traceable(fi)) {
            fi_starting_point(fi) = true;
         }
      } end_for_each_list_element;
      reinitialize_tracer(NOT_EDT_ROOT, 0);
   }
   for_each_list_element(fi, obj_info , asserted_funs) {
      if (is_traceable(fi)) {
         if (!fi_must_trace(fi))
            fi_trace_class(fi) = TC_UNTRACED;
         else if (fi_recursive(fi))
            fi_trace_class(fi) = TC_INV_REC;
         else
            fi_trace_class(fi) = TC_INVISIBLE;
      }
   } end_for_each_list_element;
   execute_target();
   root.id = ttn_id(trace_root);
   root.pid = NO_PARENT;
   root.rqd = 0;
   return root;
} /* end edt_root */
```

There are also a number of routines for accessing various parts of an EDT-node. For instance:

```
edt_value en_result(edt_node en, int prec);
```

We do not provide details regarding the C representation of EDT values here, but the structure is similar to what was outlined in section 8.1.1. In particular, circular values have an explicit representation using labels and references. However, for efficiency reasons, it is also possible to represent truncated values. The reason is that a debugger typically displays values with a certain (user-definable) precision (this is at least what the built-in user interface does, see section 8.2). It may, for instance, display only the first ten elements of a long list. Constructing a complete representation of such a value is then wasteful, and the routines which return EDT values therefore work to a given precision which is supplied as an argument, `prec` in the example above. If it turns out out that the user needs to see more of a value, the value is simply re-converted at a higher precision.

Appendix C

Compiling List Comprehensions for Debugging

This appendix presents a Haskell program which demonstrates how list comprehensions can be translated into functions in a way that supports debugging. Compared with the discussion in section 7.2, the program handles a more general version of list comprehensions where patterns are allowed to the left of the arrow (`<-`) in generators. This is still not quite as general as the list comprehensions in Haskell 1.4 [PHA+97] which also allow variables to be bound by means of `let`-constructs in the qualifier list. However, this can be handled by adding these bindings to the appropriate **where**-clause in the translation. The required generalization of the program below is straightforward.

C.1 Datatypes

Expressions, left-hand sides of function definitions and patterns are simply represented by strings (character lists). This is sufficient for the purpose of this demonstration.

```
type Expr = [Char]
type FunLHS = [Char]
type Pat = [Char]
```

List comprehensions are represented by the type `Comprehension`. A comprehension consists of an expression and a list of qualifiers.

```
data Comprehension = MkCompr Expr [Qualifier]
```

Qualifiers are represented by the type `Qualifier`. A qualifier is either a generator or a predicate. A generator consists of a pattern and an expression. A predicate is a (boolean) expression.

```
data Qualifier = MkGen Pat Expr | MkPred Expr
```

The type `Function` is used to represent functions generated by the list comprehension compiler. Each such function is represented by its debug name (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 left-hand side, the right-hand side and a list of local function definitions (to be defined in a `where`-clause) are given.

```
type FunName = [Char]
data Function = MkFun FunName [(FunLHS,Expr,[Function])]
```

C.2 Translation

The function `tc` translates a list comprehension. It takes an integer (`ns`) acting as a name supply and the comprehension as its arguments. It returns a pair consisting of the source text with which to replace the comprehension and the generated comprehension functions (currently one or none). The generated functions should be defined in a `where`-clause within whose scope the occurrence of the comprehension is situated.

```
tc :: Int -> Comprehension -> (Expr, [Function])
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) =",
              replExpr ++ " ++ " ++ funName ns ++ " l",
      whereDefs),
     (funName ns ++ " (_ : l) =",
              funName ns ++ " l", [])
            ]])
    where
        (replExpr, whereDefs) =
    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 " ++ replExpr ++
                " else []",
         whereDefs)]])
        where
            (replExpr, whereDefs) =
        tc (ns+1) (MkCompr e rest)
tc ns (MkCompr e []) = ("[" ++ e ++ "]", [])

funName ns = "f" ++ show ns
```

The function `debugName` takes a list comprehension and generates a printable 'name' for the function that is generated when the comprehension is translated by `tc`. The 'name' is similar to the original source code of the comprehension, thus permitting debugging at source code level. The character `@` in the generated names is a placeholder for which the value of the function argument is substituted during debugging.

```
debugName :: Comprehension -> FunName
debugName (MkCompr e (MkGen gp ge : rest)) =
    "[ " ++ e ++ " | " ++ gp ++ "<-" ++ "@"
    ++ printQualifiers rest ++ " ]"
debugName (MkCompr e (MkPred pe : rest)) =
    "[ " ++ e ++ " | " ++ pe
    ++ printQualifiers rest ++ " ]"

printQualifiers :: [Qualifier] -> [Char]
printQualifiers [] = []
printQualifiers (MkGen gp ge : rest) =
    ", " ++ gp ++ "<-" ++ ge ++ printQualifiers rest
printQualifiers (MkPred pe : rest) =
    ", " ++ pe ++ printQualifiers rest
```

C.3 Pretty printing

The function `printFunction` prints a generated function as properly indented source code.

```
printFunction :: Int -> Function -> [Char]
printFunction i (MkFun dn eqs) =
    indent i ++ "-- " ++ dn ++ "\n" ++ printEqns eqs
    where
        printEqns [] = ""
```

```
        printEqns ((lhs, rhs, wheredefs) : eqs) =
            indent i ++ lhs ++ " " ++ rhs ++ "\n"
            ++ printWhereDefs (i+4) wheredefs
    ++ printEqns eqs
        printWhereDefs i [] = ""
        printWhereDefs i whereDefs =
            indent i ++ "where\n"
    ++ concat (map (printFunction (i+4)) whereDefs)

indent i = take i (repeat ' ')
```

C.4 Example

The list comprehension

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

is represented as follows:

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

Translating this list comprehension, thus:

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

yields the following result:

```
f0 (foo z)
where
-- [ x + y | (x, 2)<-@, (y, 3)<-fie z, x > y ]
f0 [] = []
    f0 ((x, 2) : l) = f1 (fie z) ++ f0 l
    where
        -- [ x + y | (y, 3)<-@, x > y ]
        f1 [] = []
f1 ((y, 3) : l) = f2 ++ f1 l
            where
```

```
                    -- [ x + y | x > y ]
                    f2 = if x > y then [x + y] else []
            f1 (_ : l) = f1 l
    f0 (_ : l) = f0 l
```

The debug names are given as comments. `f0`, `f1` and `f2` are assumed to be fresh names.

Appendix D

Benchmark Programs

This appendix lists the benchmark programs used for performance evaluation of the Freja debugger. See chapter 10 for details.

D.1 Ackermann

```
module Main where

ackermann 0 y = 1
ackermann 1 0 = 2
ackermann x 0 = x + 2
ackermann x y = ackermann (ackermann (x-1) y) (y-1)

loop :: Int -> Int -> Int
loop 0 a = a
loop i a = loop (i-1) (ackermann 3 4 + a)

main = show (loop 50 0) ++ "\n"
```

D.2 Sieve

```
module Main where

sieve :: [Int] -> [Int]
sieve (x : xs) = x : sieve (filter (not_div x) xs)

not_div :: Int -> Int -> Bool
not_div x y = y `mod` x /= 0
```

```
primes = sieve [2..]

main = show (primes !! 2500) ++ "\n"
```

D.3 Isort

```
module Main where

isort :: [Int] -> [Int]
isort []       = []
isort (x : xs) = insert x (isort xs)

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x xxs@(x' : xs) | x < x'    = x : xxs
                       | otherwise = x' : insert x xs

main = show (isort [5000, 4999 .. 1])
```

D.4 Crypt

```
-- Cryptarithmetic puzzle solver.
-- Brute force method.

module Main where

import Prelude hiding (lookup)

-- List utilities

(\\) :: (Eq a) => [a] -> [a] -> [a]
bs \\ cs =  flt bs cs

[]     `del` _             = []
(x:xs) `del` y | x == y    = xs
               | otherwise = x : xs `del` y

flt z []     =  z
flt z (x:xs) =  flt (del z x) xs
```

```
nub :: (Eq a) => [a] -> [a]
nub l = nub' l []

nub' [] _     = []
nub' (x:xs) l =
    if x `elem` l then nub' xs l else x : nub' xs (x:l)


-- Main program

type VarName = Char
type CompiledPuzzle = [([VarName], VarName)]
type Mapping = [(VarName,Int)]

allChooseN :: (Eq a) => Int -> [a] -> [[a]]
allChooseN 0 _  = [[]]
allChooseN n [] = []
allChooseN n xs =
    foldr (++) [] (map (allChooseN' n xs) xs)

allChooseN' n xs x = (map (x:) (allChooseN (n-1) (xs\\[x])))

mappings :: [VarName] -> [Mapping]
mappings cs = map (zip cs) (allChooseN (length cs) [0..9])

lookup :: Mapping -> VarName -> Int
lookup m c = foldr (find c) (error "not bound") m

find c (c', v) v' = if (c == c') then v else v'

test :: CompiledPuzzle -> Mapping -> Bool
test puzzle m = testHlp m puzzle 0

testHlp m []          cy = cy == 0
testHlp m ((xs, x):r) cy =
    let
       cy'y = divMod (foldl (+) cy (map (lookup m) xs)) 10
       cy' = fst cy'y
       y = snd cy'y
    in
       lookup m x == y && testHlp m r cy'
```

```
compile puzzle = cHlp sum terms
                 where
                     puzzle' = map reverse (reverse puzzle)
                     terms = tail puzzle'
                     sum = head puzzle'

cHlp [] _      = []
cHlp (x:xs) ys = (concat (map (take 1) ys), x) :
                 cHlp xs (map (drop 1) ys)

solutions puzzle = (filter (test puzzle') (mappings vnames))
                   where
                       puzzle' = compile puzzle
                       vnames = nub (foldr1 (++) puzzle)

-- puzzle = ["FOUR", "FIVE", "NINE"]
-- puzzle = ["SEND", "MORE", "MONEY"]
puzzle = ["ONE", "ONE", "ONE", "ONE", "ONE", "ONE", "SIX"]

main = show (solutions puzzle)
```

D.5 Mini-Freja

D.5.1 Main

```
module Main where

import MiniFreja

dtake = FunDef "take" ["n", "xs"]
            (mkIf (App (Prim "null") (Var "xs"))
                  (Prim "Null")
                  (mkIf (mkApp2 (Prim "<=") (Var "n") (LitInt 0))
                        (Prim "Null")
                        (mkApp2 (Prim "Cons")
                                (App (Prim "head") (Var "xs"))
                                (mkApp2 (Var "take")
                                        (mkApp2 (Prim "-")
                                                (Var "n")
                                                (LitInt 1))
                                        (App (Prim "tail") (Var "xs"))))))

dfilter = FunDef "filter" ["p", "a"]
              (mkIf (App (Prim "null") (Var "a"))
                    (Prim "Null")
                    (Letrec
                         [ VarDef "x" (App (Prim "head") (Var "a")),
                           VarDef "xs" (App (Prim "tail") (Var "a")) ]
```

```
                         (mkIf (App (Var "p") (Var "x"))
                               (mkApp2 (Prim "Cons")
                                       (Var "x")
                                       (mkApp2 (Var "filter")
                                               (Var "p")
                                               (Var "xs")))
                               (mkApp2 (Var "filter")
                                       (Var "p")
                                       (Var "xs")))))

dfrom = FunDef "from" ["n"]
            (mkApp2 (Prim "Cons")
                    (Var "n")
                    (App (Var "from")
                         (mkApp2 (Prim "+") (Var "n") (LitInt 1))))

dsieve = FunDef "sieve" ["a"]
             (Letrec
                  [ VarDef "x" (App (Prim "head") (Var "a")),
                    VarDef "xs" (App (Prim "tail") (Var "a")),
                    FunDef "notdivx" ["y"]
                        (mkApp2 (Prim "/=")
                                (mkApp2 (Prim "mod") (Var "y") (Var "x"))
                                (LitInt 0)) ]
                  (mkApp2 (Prim "Cons")
                          (Var "x")
                          (App (Var "sieve")
                               (mkApp2 (Var "filter")
                                       (Var "notdivx")
                                       (Var "xs")))))

dones = VarDef "ones" (mkApp2 (Prim "Cons") (LitInt 1) (Var "ones"))

dprimes = VarDef "primes" (App (Var "sieve") (App (Var "from") (LitInt 2)))

defs = [dtake, dfilter, dfrom, dsieve, dprimes]
prog = Letrec defs (mkApp2 (Var "take") (LitInt 500) (Var "primes"))

main = show (vfE prog emptyEnv)
```

D.5.2 MiniFreja

```
module MiniFreja (module Basic, module Absyn, module Val, module Env,
                  module Prim, vfE) where

import Basic
import Absyn
import Val
import Env
import Prim

vfE :: Exp -> Env -> Val
vfE (LitInt n)  env = Int n
vfE (Prim n)    env = vfP n
```

```
vfE (Var i)     env = env i
vfE (App e1 e2) env = apply (vfE e1 env) (vfE e2 env)
vfE (Letrec ds e) env = vfE e env'
    where
        env' = fix (\env' -> updEnv (elaborate ds env') env)

elaborate []                  env = []
elaborate (VarDef i e    : ds) env = (i, vfE e env) : elaborate ds env
elaborate (FunDef i is e : ds) env =
    (i, Fun (length is) (\as -> vfE e (updEnv (zip is as) env)))
    : elaborate ds env
```

D.5.3 Prim

```
module Prim(vfP) where

import Basic
import Val

vfP :: Name -> Val
vfP n | a == 0    = Ctr n []
      | otherwise = Fun a (primFun n)
    where
        a = primArity n

primArity :: Name -> Int
primArity "True"  = 0
primArity "False" = 0
primArity "Null"  = 0
primArity "Cons"  = 2
primArity "neg"   = 1
primArity "+"     = 2
primArity "-"     = 2
primArity "*"     = 2
primArity "/"     = 2
primArity "mod"   = 2
primArity "not"   = 1
primArity "<"     = 2
primArity "<="    = 2
primArity "=="    = 2
primArity "/="    = 2
primArity ">="    = 2
primArity ">"     = 2
primArity "head"  = 1
primArity "tail"  = 1
primArity "null"  = 1
primArity "if"    = 3
primArity _       = error "Unknown primitive."

primFun "Cons" = \[hd, tl] -> Ctr "Cons" [hd, tl]
primFun "neg"  = \[Int n] -> Int (-n)
primFun "+"    = \[a, b] -> applyIII (+) a b
primFun "-"    = \[a, b] -> applyIII (-) a b
primFun "*"    = \[a, b] -> applyIII (*) a b
```

```
primFun "div"  = \[a, b] -> applyIII div a b
primFun "mod"  = \[a, b] -> applyIII mod a b
primFun "not"  = \[Ctr n _] -> if n == "True" then
                                    Ctr "False" []
                                else
                                    Ctr "True" []
primFun "<"    = \[a, b] -> applyIIB (<)  a b
primFun "<="   = \[a, b] -> applyIIB (<=) a b
primFun "=="   = \[a, b] -> applyIIB (==) a b
primFun "/="   = \[a, b] -> applyIIB (/=) a b
primFun ">="   = \[a, b] -> applyIIB (>=) a b
primFun ">"    = \[a, b] -> applyIIB (>)  a b
primFun "head" = \[Ctr n fs] -> if n == "Null" then
                                     error "Head of empty list"
                                 else
                                     fs !! 0
primFun "tail" = \[Ctr n fs] -> if n == "Null" then
                                     error "Tail of empty list"
                                 else
                                     fs !! 1
primFun "null" = \[Ctr n _] -> if n == "Null" then
                                    Ctr "True" []
                                else
                                    Ctr "False" []
primFun "if"   = \[Ctr n _, a, b] -> if n == "True" then a else b
primFun _      = error "Unknown primitive."

applyIII op (Int n1) (Int n2) = Int (op n1 n2)

applyIIB op (Int n1) (Int n2) =
    Ctr (if (op n1 n2) then "True" else "False") []
```

D.5.4 Absyn

```
module Absyn(Exp(..), Def(..), mkApp2, mkApp3, mkIf) where

import Basic

data Exp = LitInt Int           -- Literal integer
         | Prim Name            -- Primitive
         | Var Id               -- Variable
         | App Exp Exp          -- Application, exp1 exp2
         | Letrec [Def] Exp     -- Recursive definition, letrec defs in exp

data Def = VarDef Id Exp        -- x = exp
         | FunDef Id [Id] Exp   -- f x1 ... xn = exp

mkApp2 f a1 a2 = (App (App f a1) a2)

mkApp3 f a1 a2 a3 = (App (App (App f a1) a2) a3)

mkIf c e1 e2 = mkApp3 (Prim "if") c e1 e2
```

D.5.5 Env

```
module Env(Env(..), emptyEnv, updEnv) where

import Basic
import Val

type Env = Id -> Val

emptyEnv i = error ("Identifier " ++ i ++ " not bound.")

updEnv :: [(Id, Val)] -> Env -> Env
updEnv bs env = \i -> case lookup i bs of
                          Nothing -> env i
                          Just a  -> a
```

D.5.6 Val

```
module Val(Val(..), apply) where

import Basic

data Val = Int Int                         -- Integer
         | Ctr Name [Val]                  -- Constructed value
         | Fun Int ([Val] -> Val)          -- Function
         | PAp Int ([Val] -> Val) [Val]    -- Partial application

apply :: Val -> Val -> Val
apply (Fun n f)     a | n == 1 = f [a]
                      | n > 1  = PAp (n - 1) f [a]
apply (PAp n f as)  a | n == 1 = f (as ++ [a])
                      | n > 1  = PAp (n - 1) f (as ++ [a])
apply _             _          = error "Application of non-functional value."
```

D.5.7 Basic

```
module Basic (Name(..), Id(..), fix) where

type Name = String
type Id = String

fix f = x where x = f x
```

Bibliography

[AHN88] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28:490–503, 1988.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[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 Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, December 1987.

[Aug93a] Lennart Augustsson. *HBC user's manual*. Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, 1993. Distributed with the HBC Haskell compiler.

[Aug93b] Lennart Augustsson. Implementing Haskell overloading. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 65–73, Copenhagen, Denmark, June 1993. ACM Press.

[Aug97] Lennart Augustsson. The HBC compiler. `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`, 1997.

[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.

[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 (FPCA '93)*, pages 321–330, Copenhagen, Denmark, June 1993. ACM Press.

[CH98] Magnus Carlsson and Thomas Hallgren. *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Cop94] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[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.

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

[FW94] Michael Fröhlich and Mattias Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science, University of Bremen, September 1994.

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

[H$^+$96] Pieter H. Hartel et al. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[Hen92] Fritz Henglein. Simple closure analysis. Technical Report D-193, DIKU, University of Copenhagen, March 1992.

[HFP96] Paul Hudak, Joseph H. Fasel, and John Peterson. A gentle introduction to Haskell. Technical Report YALE/DCS/RR-901, Yale University, Department of Computer Science, May 1996.

[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 (POPL '84)*, pages 122–132. ACM Press, January 1984.

[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 Hughes, 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.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to combinators and λ-calculus*. Cambridge University Press, 1986.

[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).

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, 1985. Springer-Verlag.

[Joh87a] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173, Portland, Oregon, September 1987. Springer-Verlag.

[Joh87b] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, 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.

[KH95] Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, October 1995.

[Kow79] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, October 1979.

[Llo94] John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.

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

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

[Mor82] J. H. Morris. Real programming in functional languages. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

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

[NB95] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, Australia, 1995.

[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 (PLILP '92)*, 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 (AADEBUG '93)*, 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):337–370, July 1994.

[Nil94] Henrik Nilsson. A declarative approach to debugging for lazy functional languages. Licentiate Thesis No. 450, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, September 1994.

[NS96] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, August 1996. This is an extended version of [NS97].

[NS97] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

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

[PHA+97] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Technical Report YALE/DCS/RR-1106, Yale University, Department of Computer Science, February 1997.

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

[PJ89] Simon L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Functional Programming and Computer Architecture (FPCA '91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, September 1991.

[Pla84] David A. Plaisted. An efficient bug location algorithm. In *Proceedings of the Second International Logic Programming Conference*, pages 151–157, Uppsala, Sweden, July 1984.

[Rém92] Didier Rémy. Extension of ML type system with a sorted equational theory on types. Research Report 1766, INRIA, October 1992.

[Rep90] John H. Reppy. Asynchronous signals in standard ML. Technical Report TR 90-1144, Department of Computer Science, Cornell University, August 1990.

[Röj95] Niklas Röjemo. Generational garbage collection without temporary space leaks for lazy functional languages. In Henry G. Baker, editor, *International Workshop on Memory Management (IWMM '95)*, volume 986 of *Lecture Notes in Computer*

Science, pages 145–162, Kinross, United Kingdom, September 1995. Springer-Verlag.

[RR96] Colin Runciman and Niklas Röjemo. New dimensions in heap profiling. *Journal of Functional Programming*, 6(4):587–620, July 1996.

[RW93] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, July 1993.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, FB 14 Informatik, Universität des Saarlandes, 1994.

[Sch86] David A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[SD96] S. D. Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.

[Sew92] Julian Seward. Generational garbage collection for lazy graph reduction. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management (IWMM '92)*, volume 637 of *Lecture Notes in Computer Science*, pages 200–217, St. Malo, France, September 1992. Springer-Verlag.

[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.

[SN95] Jan Sparud and Henrik Nilsson. The architecture of a debugger for lazy functional langauges. In Mireille Ducassé, editor, *Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995. IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France.

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

[Spa96] Jan Sparud. A transformational approach to debugging lazy functional programs. Licentiate Thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, February 1996.

[SPJ93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 106–116, Copenhagen, Denmark, June 1993. ACM Press.

[SPJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Principles of Programming Languages (POPL '95)*, pages 355–366, 1995.

[SR97] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '97)*, Southampton, September 1997.

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

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

[TA95] Andrew Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley Publishing Company, 1996.

[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 Colin 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.

[Veg84] Steven R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, C-33(12):1050–1071, December 1984.

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

[Wad98] Philip Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, February 1998.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Annual Symposium on Principles of Programming Languages (POPL '89)*, pages 60–76. ACM Press, January 1989.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management (IWMM '92)*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.