Tracing and Debugging Lazy Functional Computations

Jan Sparud

Department of Computing Science
Chalmers University of Technology
Göteborg, Sweden
E-mail: sparud@cs.chalmers.se

October 20, 1999

A dissertation for the Ph.D. Degree in Computing Science
at Chalmers University of Technology

Department of Computing Science
Chalmers University of Technology and
University of Göteborg
S-412 96 Göteborg

ISBN 91-7197-750-3
ISSN 0346-718X
CTH, Göteborg, 1999

Abstract

Functional programming languages have many advantages over the conventional alternative of procedural languages. For example, program construction is more rapid, more modular and less error-prone. Programs themselves are more concise. Moreover, lazy functional languages are declarative and allow the programmer to write programs where operational issues such as the evaluation order are left implicit.

However, programs do not always work first time. When they fail, programmers cannot always see the reason immediately, and they need tools to help them find it.

In traditional, procedural programming languages, fault-tracing or *debugging* can be based on studying sequences of events. But in a lazy functional language, the sequence of events is not so easily presented, nor is it what the programmer really needs.

This thesis presents several approaches to tracing and debugging lazy functional programs. In the first part, a method based on *algorithmic debugging* is presented. An algorithmic debugger records function invocations and results for a program execution. If the program demonstrates a faulty behavior, an algorithm-guided, interactive search is conducted in the execution record for the part of the program that caused the externally visible symptom of the problem. Provided that the user can correctly answer whether results of certain function applications are correct or not, the erroneous function will be located.

Also in the first part, a tailor-made debugger for stream-based, lazy functional programs is presented.

In the second part, another approach to tracing is presented which is based on *Redex trails.* Redex trails are histories of functional computations. The basis of the tracing method is a program transformation carried out by the compiler: transformed programs compute the same values as the original, but embedded in functional data structures that also include redex trails showing how the values were obtained. A special-purpose display program enables detailed but selective exploration of the redex trails, with cross-links to the source program.

Acknowledgements

My interest in the subject of this thesis started at a debugging workshop in Linköping 1993, which I attended together with my supervisor Thomas Johnsson, on his suggestion. On the train back to Göteborg, I was thinking about what I had heard on the subject of debugging functional programs, and thought that something better should be possible. And that it should be simple. How wrong I was, at least on the latter point...

First, I would like to thank Thomas Johnsson, for bringing my interest to the subject, and for helping me with comments along the way.

Colin Runciman convinced me to come to York to take a position as a (pre-doc) post-doc in 1996. I had an wonderful time there, met many new friends, and it was a privilege to work together with Colin, who always seems to quickly grasp and clarify my ill-structured and often strange ideas.

Urban Boquist is partly responsible for me spending much too much time on things I shouldn't do, but that are nevertheless fun and interesting. Somehow, we always seem to head in the same direction, and it has been like that ever since working with our Master's project, many moons ago. It was during that project that our supervisor Staffan Truvé convinced us that we should start our doctoral studies (thanks, Staffan!).

Thanks to the Fudget brothers, Magnus Carlsson and Thomas Hallgren, for many fruitful discussions and for good company at Guldhedstornet and when having the always delicious Panäng kai at Thai Garden.

Thanks also to Lennart Augustsson and Niklas Röjemo for providing me with compilers to play around with to do my research, and also for always gladly answering questions about just about everything.

Henrik Nilsson co-authored the first paper in this thesis. Thanks Henrik, for being an inspiring colleague, and a friendly "competitor" in the quest for our common goal.

Amr Sabry presented the interesting problem that lead to the paper about debugging stream programs in this thesis, which he also co-authored.

I am very grateful to my colleagues at the Department of Computing Science at Chalmers. Thanks to all of you, and in particular to Lena, Christina, Cina, Andy, Dave, Bror, Lars, Christer, Görgen and Hasse.

Finally, I would like to thank my parents Carl and Marianne, my brother Lennart, and sister Carina, for all encouragement and moral support.

List of Included Papers

This thesis is based on the work contained in the following papers:

A 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.
The paper is an extended version of:
Jan Sparud and Henrik Nilsson. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Journal of Automated Software Engineering*, 4(2):121–150, April 1997.

B Jan Sparud and Amr Sabry. Debugging Reactive Systems in Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

C Jan Sparud and Colin Runciman. Tracing Lazy Functional Computations Using Redex Trails. In H. Glaser and P. Hartel, editors, *Proceedings of Ninth International Symposium on Programming Languages: Implementations, Logics, and Programs, Lecture Notes in Computer Science*, volume 1292, pages 291–308. Springer-Verlag, September 1997.

D Jan Sparud and Colin Runciman. Complete and Partial Redex Trails of Functional Computations. In C. Clack, K. Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop (IFL'97)*, volume 1467 of *Lecture Notes in Computer Science*, pages 160–177. Springer-Verlag, 1998.
The paper is concluded with:
Paper **D** Addendum: Jan Sparud. Implementing Adjustable Pruning. Chalmers University of Technology, 1999.

E Jan Sparud. Redex Trail Representation and Access Protocol. Technical Report. Chalmers University of Technology, Sweden, November 1998.

F Jan Sparud. Using the Redex Trail System. Technical Report. Chalmers University of Technology, Sweden, October 1998.

Contents

Part I

Introduction

1 Background and motivation to our work

One of the more serious show-stoppers when it comes to bringing lazy functional languages to the masses, is the appalling lack of good lazy functional programming environments. A good program tracer or debugger, that aids a programmer in understanding what went wrong when executing a program, is an important part of a programming environment. It has been argued that there is no need for a debugger for a language with strict static type-checking [OH88, TR86]. This is true to some extent; many errors that would otherwise be hard to find with run-time type-checking (or no type-checking at all) are caught by the type-checker. However, not all errors can be statically detected.

To locate logical errors and runtime faults, a tracer that is convenient to use and that does not impose too much overhead is needed, and the goal of the work presented in this thesis is to design and implement such a tool.

Our aim is to use a declarative approach to tracing, and to let the user focus on the intended semantics of functions, rather than operational issues such as evaluation order.

2 Lazy functional languages

Functional programming languages have many pleasing properties. They are at a much higher level than traditional, imperative languages. Imperative languages are designed to be easily translated into the language that computer processors understand: a sequence of very basic instructions. This has the effect that it is rather straight-forward to write a translator (or *compiler*) from an imperative program into machine instructions. Unfortunately, this also has the effect that in order to write a program in such a language, a programmer has to adopt his way of thinking to the way the computer works. When writing a program to solve a problem, the programmer cannot concentrate solely on the problem, but must also consider details such as managing the memory needed to solve the problem, and defining the precise sequence of steps the program should take.

Functional programming languages, on the other hand, are designed to let the programmer describe his intentions in a declarative way, not as a sequence of tasks that need be performed. The basic concept in functional programming is that functions are first-class citizens. Functions can be dynamically created, passed to other functions and returned from functions, just like any other value in the language such as integers, strings etc. All functional languages have automatic memory management systems, which relieves the programmer from explicitly having to deal with memory issues. The notation used is often similar to that of mathematics. The unfortunate consequence of making things easier for the programmer is that it is hard to compile a functional program into efficient machine code.

Lazy functional languages take this a step further, in that the system decides in which order separate parts of the program should be executed, based on the principle of doing as little as possible (or to be *lazy*). The force that drives the execution forward is the demand for the result of the top level expression in the program.

In this thesis, we will use the lazy functional language Haskell [HPJW+92] as the host language for examples and discussions and it is also the target language for our tracing and debugging efforts. We will therefore start with a short introduction to Haskell.

3 An introduction to Haskell

This short introduction is not intended as a course in Haskell, but more as a presentation of some of its distinguishing features.

Haskell is a *purely functional language*. A functional program is a single expression, and it is executed by evaluating that expression. The basic building blocks in a functional program are, not surprising, functions. Functional programs are built by defining functions and a top-level expression that uses the defined functions to perform certain tasks.

To define a function in Haskell, you normally specify a series of equations, not much different from mathematical equations, that reflect your intentions with the function. As an example, here is the Haskell definition of a function that takes a number as an argument, and returns that number doubled:

```
double n = 2 * n
```

The single equation consists of three parts: the function name (`double`); the arguments (in this case only one, namely `n`); and the right-hand side of the equation (also called the *body* of the function), `2*n`.

When you have defined a function, you can use it in expressions. If you for example want to know how much `7` doubled is, you can evaluate the expression `double 7`. It will be evaluated as follows:

```
double 7 => 2 * 7 => 14
```

We use the notation $e_1 \Rightarrow e_2$ to indicate that an expression e_1 evaluates (or *reduces*) to e_2.

3.1 Values and types

Evaluation of an expression leads finally (perhaps after many steps, or reductions) to a *value*. Every value has an associated *type*. Examples of values are the integer `14`, which has the Haskell type `Int` and the character 'p' which has the type `Char`. One of the most distinguishing characteristics of functional languages is to treat functions just like ordinary values. Our function `double` is also a value, and has the type `Int -> Int`[1]. All functional values have at least one `->` in their type, and a type *a* `->` *b* should be interpreted as "the type of a function that takes a value of type *a* as argument and returns a value of type *b*".

Haskell is a *strongly typed language*, which means that before any evaluation takes place, the system checks that all values in a program are combined in a type-correct way. For example, if we try to evaluate the expression `double 'p'` the system will complain about us trying to call a function that expects an integer as argument with an argument of type `Char`. When declaring a function, you can also declare a type for it explicitly, e.g.:

```
double :: Int -> Int
double n = 2 * n
```

The `::` can be read as "has type". You don't have to provide type declarations for functions, the system can automatically infer the correct type, but it is a good practice to do so. Type declarations can be seen as a way of documenting functions; the type usually gives a clue to what a function does.

3.2 Other types: boolean, tuples and lists

There are a number of predefined types in Haskell. We have already seen types for integers and characters. The type `Bool` has only two values, `True` and `False`. It is used to represent truth values.

A *tuple* is a value that can contain a fixed number of other values, separated by commas. The whole tuple is enclosed by parantheses. An example of a tuple is `(14, 'p', True)`. The type for a tuple has the same structure

[1] To simplify this introduction, we ignore type classes here.

as tuple values, but with the sub-values replaced by their respective types. Our example tuple thus has the type `(Int, Char, Bool)`.

A *list* is a value that can contain arbitrary many values, which all must have the same type. An empty list is denoted by `[]`, usually pronounced *nil*. A non-empty list having a first element x and a rest xs is denoted by $x : xs$. The : is usually pronounced *cons*. An example of a list is `1:(2:(3:[]))`. Since cons is right associative (binds harder to the right) the parantheses can be omitted, so `1:2:3:[]` denotes the same value. The type of a list is the type of the list elements enclosed by square brackets (`[]`). The type for our example list is thus `[Int]`. For convenience, you can also write lists as a comma-separated list of elements, enclosed by square brackets. Hence, the list `1:2:3:[]` can also be written as `[1, 2, 3]`.

Users can define new datatypes, but we will not describe how to do that here.

3.3 Patterns

A very powerful feature in Haskell is *pattern matching*. When defining a function, you can have it behave in different ways, depending on the patterns the actual arguments take. A function calculating the nth value of the Fibonacci series could be defined as:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Here we have a function definition that consists of three equations. The first equation means: if the argument is `0`, then the result is `1`. The second equation means: if the argument is `1`, then the result is `1`. The third equation means: for any n, the result is the sum of `fib` n`-1` and `fib` n`-2`. The `0` and `1` on the left hand side of the first two equations are patterns, which when matched with an actual argument, decide which equation should be used. The patterns are matched in the order the equations are listed. If no pattern matches, there will be a runtime error.

In the third equation, the `fib` function calls itself, but with "smaller" arguments. A function using itself to evaluate a result is called *recursive*.

Pattern is of most use with more complicated types than integers. Here is a function that selects the second component of a tuple of size two (usually called a *pair*):

```
snd (x, y) = y
```

The pattern `(x, y)` matches any pair. The right-hand side of the equation says that the second component should be returned, and the first component ignored. Haskell has a special notation for a pattern that matches everything, but that we are not interested in using on the right-hand side of the equation. It is called a *wild card* and is written as an underscore, (_). Using a wild card, we can define `snd` as:

```
snd (_, y) = y
```

It is common to use patterns when defining functions over lists. A function that sums all the elements in a list could be defined as:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

When using patterns, function definitions often are very concise and similar to mathematical equations. The definition of `sum` can be read as: the sum of the empty list of integers is `0`, and the sum of a list with a first element x and a rest xs is x plus the sum of the list xs.

3.4 Polymorphism

Using what we have learnt sofar, we can define a function that calculates the length of a list. The length of an empty list is `0`, and the length of a list with a first element and a rest xs is one plus the length of xs. Thus, we can define:

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

What type does `length` have? It takes a list as argument and returns an integer, but what is the type of the elements in the list? Since we don't do anything with the list elements, except counting them, they can be *of any type*. The type of length is `[a] -> Int`, where `a` is called a type variable, which can be instantiated to any type (or, take any form). We can therefore use our `length` function to calculate the length of e.g. character lists, integer lists, or even lists containing functions. A function is called *polymorphic* if it can take arguments of arbitrary type. In section 3.3 we defined the function `snd`, that selected the second component of a pair. `snd` is also polymorphic, it must have a pair as argument, but the components can be of any type. The type of `snd` is thus `(a, b) -> b`.

3.5 List comprehensions and sequences

Haskell has a number of convenient features that make programming with lists easier. One of them is *arithmetic sequences*, with which you can create a sequence of e.g. numbers. As an example, `[1 .. 8]` is a sequence generator that generates the list of numbers from `1` to `8` (`[1, 2, 3, 4, 5, 6, 7, 8]`).

List comprehensions is a powerful notation with which you elegantly and concisely can express algorithms. Here is a simple list comprehension:

```
[f x | x <- xs]
```

It can be read as "the list of all results of $f\ x$, where x is drawn from the list xs". A more concrete example is `[2*x | x <- [1..4]]`, which result is `[2, 4, 6, 8]`.

The x `<-` xs part of the list comprehension is called a *generator*, and you can have more than one in the same list comprehension. They will then be nested from left to right, and in a generator you can refer to variables bound earlier (to the left). For example, the list comprehension

```
[(x, y) | x <- [1 .. 3], y <- [x .. 3]]
```

has the result `[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]`.

You can also have *guards* in list comprehensions, which are boolean expressions that restrict which elements should be generated. Using guards, it is very easy to define a sorting function:

```
quicksort []     = []
quicksort (x:xs) = quicksort [a | a <- xs, a < x]
                   ++ [x] ++
                   quicksort [a | a <- xs, a >= x]
```

The algorithm can be read as follows: The result of sorting an empty list is the empty list. To sort a list with a first element x and a rest called xs, sort all elements in xs which are less then x, concatenate (using the concatenation operator `++`) the result with the singleton list containing x and the sorted list of elements in xs that are greater than or equal to x.

Note how the guards `a < x` and `a >= x` restrict which elements are drawn from the list `xs`.

3.6 Higher order and anonymous functions

We have said that in a functional language, functions are just like any other value. This means that functions can be used as arguments to other functions, and even be returned as results.

Let's say we want to define a function `map` that applies some function on all elements in a list, and returns the list of results. A definition of `map` can look like:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

We also included a type signature to map, and the function can be explained in terms of its type: it takes as argument a function f and a list xs. If f takes arguments of type a and returns values of type b, then xs must be a list of as, i.e. have the type `[`a`]`, and the return type of `map` is then `[`b`]`.

Here are some examples on how to use the map function (the results are also shown):

```
map double [1 .. 4] ⇒ [2, 4, 6, 8]
map fib [1 .. 5] ⇒ [1, 2, 3, 5, 8]
```

Consider this function that multiplies two numbers:

```
times :: Int -> Int -> Int
times x y = x * y
```

The result of the expression `times 2 3` is obviously `6`, but what is the result if you omit the second argument, i.e. the expression `times 2`? In many languages this is illegal, but in Haskell it makes perfect sense, and the result is *a new function*, having the type `Int -> Int`. It expects one argument, and the result is that argument multiplied by two. Using `times`, we can now redefine `double` in the following way:

```
double = times 2
```

Note that we don't explicitly say that `double` is a function taking one argument. We don't need to do that since it is bound to the result of the expression `times 2`, which itself is a function taking one argument.

It is also possible to create functions on the fly, by using *anonymous functions*. The syntax of an anonymous function is

patterns `->` *expression*

An anonymous function (obviously) has no name, but the patterns are defined in the same way as for normal function definitions. The right-hand sides (to the right of `->`) are just ordinary expressions. Here is an example use of an anonymous function:

`map (\x -> 2*x) [1 .. 4]` ⇒ `[2, 4, 6, 8]`

Anonymous functions are also called *lambda abstractions*, and the backslash is used since it is the ascii character that is mostly similar to the Greek character lambda (λ).

3.7 Laziness

We have not said much about the order in which expressions and subexpressions are evaluated. This is intentional. One of the more important aspects of programming in a lazy functional language, is that programmers need not worry about the evaluation order. Evaluation is driven by the demand of the result of the top-level expression, and the system evaluates nothing more than is needed to evaluate that expression. That is why it is called *lazy* evaluation.

This opens up the possibility for (conceptually) inifinite data structures. It is of course impossible to store an infinite amount of data in a computer, but being able to use the concept of infinite data structures often ease programming considerably.

To create an infinite increasing list of numbers starting from `1`, is as easy as writing a sequence generator starting at `1`, with no upper limit, i.e. `[1 .. ]`. The result is `[1, 2, 3, ...]` upto infinity, but since we use lazy evaluation, only as many numbers as we request will be computed. For example:

`take 5 [1 .. ]` ⇒ `[1, 2, 3, 4, 5]`

where `take` n is a function that takes the n first element of a list. Strict languages (such as e.g., SML), as opposed to lazy languages, would first try to fully evaluate the sequence, before picking the five first elements, but since it takes infinitely long time to evaluate the sequence, the evaluation will never terminate.

"Infinite" lists are very useful when you want to enumerate elements in a list. Here is an example:

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []

elements = [3, 5, 4]

enumerated = zip [1..] elements
```

`zip` is a (predefined) function that takes two lists as arguments, and returns a new list which is the pairwise interleaving of the two lists. The result of `enumerated` is `[(1, 3), (2, 5), (3, 4)]`. Note that we didn't need to calculate the length of `elements` first, the `zip` function calculated exactly as many elements from the infinite sequence `[1 .. ]` as were needed.

As a final, rather more complicated, example of laziness, we show an efficient definition of the list of all Fibonacci numbers:

```
fibs = 1 : 1 : [ a+b | (a, b) <- zip fibs (tail fibs) ]
```

The predefined function `tail` returns all but the first element in a list. The definition of `fibs` is *circular*, since it refers to itself in the definition. We leave it as an exercise for the reader to fully understand the definition of `fibs`.

3.8 Combining functions

A common way of building programs in Haskell is to define complex functions by *composing* simpler functions. A predefined function, written as a dot (`.`), is used to compose two functions (the dot is written between the composed functions, or *infix*). To compose two functions f and g, you thus write f `.` g. The composed function takes one argument, applies g to it, and then applies f to the result. An example:

```
(double . snd) (1,2) ⇒ double (snd (1,2)) ⇒ double 2 ⇒ 4
```

You can build remarkably complex functions, just by composing very simple functions.

3.9 More Haskell information

We have only described a small part of Haskell, to give you a taste of the features that the language offer. The full language comes with a large set of standard libraries that contain many functions and datatypes. To learn more about Haskell, look at the official Haskell web site, `www.haskell.org`. There are links to books and texts about Haskell, language specification and standard library documentation, as well as links to the different Haskell compilers, which are all free of charge.

4 The (un)popularity of functional languages

Part of the motivation for this work is to make functional languages more popular and more widely used. Compilers for functional languages are very advanced today. Programs written in e.g. Haskell or SML are often not much slower than if they were written a low-level language such as C. A typical figure is two to three times slowdown for the most advanced Haskell compilers. For some applications the factor is even smaller, and for others it's significantly worse. See [HFA$^+$94] for an extensive benchmark that compares quite a few compilers for functional languages with a modern C compiler. Computers are now fast enough to allow most applications to be written in a very high-level language, and still meet performance criteria. The success of the Java programming language is a proof of this; typical Java implementations are slower than state-of-the-art functional languages implementations, and yet Java is used by *many* more people than those using functional languages.

So, is Java a "better" language than e.g. Haskell? Not necessarily. We think that there are a number of reasons that Java has been successful in attracting programmers, while functional languages have not, and the reasons can be summarised as follows:

Marketing and exposure Java was created by Sun Microsystems [GJS96], a company which has invested enormous sums in the development of the language, and also provided freely available compilers for common architectures. Java has been marketed as "the programming language for the Internet", a point that has been well received since web browsers typically have support for running Java applets within them. The only functional language that has been marketed in a way similar to Java (but on a much smaller scale) is Erlang, and it seems that it has gained a fair amount of interest.

Rich and standardised libraries Java has an extensive set of standardised libraries for network operations, database access, graphical applications, common datatypes, etc. Using these libraries, it is almost trivial to write networked applications that, e.g., access databases remotely and present the results using a graphical user interface. In Haskell, the Standard Prelude includes a set of commonly used data types and functions, but it comes nowhere close to the richness of the Java libraries. Some Haskell implementations have their own set of extended libraries, which helps if one stick to a particular compiler, but it gets quickly very messy when changing to another one.

Portability Java is normally compiled to bytecode instructions, which are interpreted by a runtime system written in software. As long as you have an implementation of the runtime system for an achitecture, you will be able to run Java bytecode compiled on any architecture. The slogan used to get this point through is "write once, run anywhere", which sounds very desirable, even if, as yet, it works better in theory than in practice. There is nothing stopping a Haskell compiler to produce byte code in order to be portable, and in fact, some do, e.g. the nhc [Röj94] compiler. The Hugs [Jon96a, Jon96b] interactive Haskell system is written in C and is easily ported to new architectures. The Glasgow Haskell compiler (GHC) [JHH+93] and the Chalmers Haskell compiler (HBC) [Aug93] both compile to machine code, in order to gain efficiency.

Strong links to other successful languages Java's syntax borrows heavily from C and C++, which traditionally have a large user base throughout the programming world. This means that the step going from C or C++ to Java seems small, and tempts programmers to try it. In contrast, functional programming languages have notations that are closer to mathematics than to traditional programming languages, which seems to scare people away.

Inclusion of features from other languages Two of the more important features of (most) functional languages are that they have strong type systems and automatic memory management. Java has adopted both these features. A strong type system saves programmers from many runtime faults, by detecting faults at compile time. When writing large C or C++ programs, programmers have to spend a lot of effort in correctly allocating/using/deallocating memory, a task that a Java (or a functional) programmer need not be aware of.

Good programming environments Programmers using traditional languages are used to having programming environments which provide means of tracing, debugging and profiling (both in time and space) programs, and also easy access to library components and documentation. Java also has all these features. Functional languages, on the other hand, in general have very poor programming environments. Most language implementations are developed as part of research projects, where the majority of the effort has been put into areas which traditionally count as "academic", e.g., new language constructs based on theoretic results, and performance enhancements. Research on programming environments has not been seen as academically interesting, or scientifically challenging, but more as product development.

Another important issue is the need for the language to support inter-language operations. It has been increasingly common to write programs by combining modules written in different languages. Some program parts may have to be written in another language because of e.g. proprietary drivers or performance issues. Several Java-implementations define foreign language interfaces, but it seems that the standard set out by Sun has not been followed by other implementors. Recently, a lot of work has been put into general interfaces between Haskell and other languages [JNR97, FLMJ98].

Of course, Haskell has some features that makes it attractive even for large scale programming. Haskell programs are often very concise, clear, and easily maintainable. Factors of two to ten times shorter programs than for C or Java is not uncommon. This leads to higher productivity. Ericsson measured an improvement factor of between 9 and 25 in one set of experiments on telephony software using the functional language Erlang[Arm96].

Another advantage is that the semantic gap between the programmer and the language is small in a functional language.

In summary, it seems that Haskell is on par with Java in most aspects. But some improvements are needed. Haskell needs to be better marketed, it needs a much more complete set of standard libraries, and it needs serious work on better programming environments, to get more popular and more widely used. The focus of this thesis is to help with the programming environments part, by providing a tracer.

5 Tracing and debugging

5.1 Classification of program errors

The most common reason for wanting to trace a program is that some kind of problem has been detected. The kind of problems that can occur depends on the language the program is written in, since some programming languages protect the user from introducing certain kinds of errors. A simple classification of errors could be:

Type errors An element is used in a way it is not supposed to be used, e.g., an integer is multiplied with a string. If no type-checking is performed, there will probably be a run-time error when a program containing such an expression is run. Most languages will at least give a compile time warning in these cases. In Haskell, strict type checking is performed at compile time, which means that no run-time type error can occur. Thus, a debugger for Haskell need not be concerned with this kind of error.

Memory referencing errors For example, a pointer is used to write to or read from memory that has not been allocated. Haskell uses automatic memory management (no explicit pointers, automatic space allocation and reclamation), so this kind of error cannot occur in a running program.

Memory exhaustion This means that a program uses more memory than is available. There can be several reasons for this. The problem the program is supposed to solve might actually require more memory than is available, but it could also be the case that the program has a *space leak*, i.e., it holds on to memory that it does not use anymore. This can happen both with manual and automatic memory management.

Fatal exceptions For example, an attempt has been made to divide by zero or to get the square root of a negative number, or a function has been called with arguments outside its domain.

Logical errors A logical error could be to return `1` instead of `0` in the base case of a function taking the length of a list, or to have a condition that does not exactly reflect the programmer's intention. It could also be that the program unexpectedly loops.

The tracer proposed in this thesis will concentrate on the last two kinds of errors. The reason for this is that the first kinds of errors, i.e., type errors and memory reference errors, simply can not occur at runtime in Haskell.

Unfortunately, there could be other errors, such as *space leaks*, in programs. In short, a space leak is the effect of a program requiring unexpectedly much memory. For a classification of space leaks, see [Spa93]. Space leaks have nothing to do with the semantics of functions and expressions in the language, so we will not address that problem here.

There exist other tools that take care of memory exhaustion errors, notably space profiling [RW93].

5.2 Traditional tracing and debugging methods

One may at first think that in order to design a tracer or debugger for functional languages, it would be a good idea to use the tracing and debugging techniques for traditional, imperative languages, since these techniques have evolved over a long period of time. However, it appears that techniques such as print statements, single-stepping, breakpoints, and variable watching are not very useful, or even possible to use at all, in lazy functional languages [OH88, HHO90, NF94].

In a program written in an imperative language, there is a close, or at least closer, relation between the evaluation order of the running program and the source code if one compares with lazy functional languages. It is therefore useful to inpect the execution by stepping through the program line by line, and watching how the functions are called and how variables change. In a lazy functional language, the evaluation order is much harder to predict. Lazy evaluation suspends evaluation of expressions until the very moment when their values are actually needed. "Single-stepping" a lazy functional program could thus be very confusing.

Function invocations are important when debugging imperative programs. At function invocation time the values of the arguments can be inspected. In a lazy functional program, the arguments to a function are in general unevaluated when the function is invoked. It is not always possible to evaluate those expressions in order to inspect their values, since they may represent non-terminating computations. If the value of such an argument is forced, and that value is not needed by the debugged program, we will get a different result (in this case non-termination) of the program when running it in the debugger than otherwise.

From these problems with using imperative debugging techniques for functional languages, we conclude that new methods, tailored to suit the lazy functional paradigm, are needed.

5.3 Tracing lazy functional programs

Since pure functional programming languages have no side effects, it is not possible to insert print statements in programs (except when doing it by changing the final result, which we have already stated is not desirable). This would otherwise be a simple way of tracing programs, in the absence of a real tracer. Some, in other respects pure, functional languages therefore provide this feature. The hbc Haskell compiler [Aug93], for example, implements this (as an option, by including a non-standard library) by means of a `trace` function.

```
trace :: String -> a -> a
```

The semantics of `trace` is to return the value of its second argument. As a side effect it also prints the value of the first argument. This might seem a useful debugging method. The problem is that the printed output can be very hard to understand, since with many `trace` expressions in a program, they are printed in a seemingly arbitrary order and interleaved with the normal output of the program, because of lazy evaluation. Note that one call to `trace` can trigger another call to it, since it evaluates its argument, which when evaluated may enforce a nested call to `trace`. Note also that the insertion of a call to `trace` in a program might change its behaviour, since it artifically forces the value to be printed, which non-intentionally might lead to non-termination. In spite of these problems, many programmers use this method to trace programs. In figure 1 a small program using the `trace` primitive is shown. Every time `f` or `g` are called a message saying so is printed.

The output of the program is shown in figure 2. The expression `[1..]` denotes the infinite consecutive list of integers starting with 1. The program terminates, despite containing an (at least conceptually) infinite list, because only the first four elements of the list are actually needed by the program.

If we would like to see not only the function invocations, but also their arguments, we could change the program in figure 1 into the program in

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

Figure 1: A program using the `trace` *primitive*

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

Figure 2: The output of the program in figure 1

figure 3.

```
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))
result = take 4 (f [1..])
```

Figure 3: Another program using the `trace` *primitive*

Figure 4 shows the output of the program after it has been interrupted. In this case we get another result. The fact that we want to see the argument to `f` turns the program into a non-terminating program! Our debugging trace messages have changed the meaning of the program.

One way of debugging functional programs is to trace their execution on an abstract machine for that language. The problem with this is that

```
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
```

Figure 4: The output of the program in figure 3

most abstract machines for functional languages are at a very low level. Functional languages offer the possibility to abstract away from low level activities such as evaluation order, and we think that a debugger should not change that. It should be admitted though, that certain problems, or bugs, are due to some specific implementation details of the programming language. For instance, to be able to find the reason for a space leak, some tool to trace the program at a low level might be needed. We think that the fact that programmers have to take such low level details into consideration is really a drawback for lazy functional programming. For a discussion of space leaks and methods to find and eliminate them, see [RW93].

Most modern functional languages have compilers, rather than interpreters. Since many program transformations are needed in order to compile a functional program to efficient code, it is not easy to relate the resulting code to a certain piece of source code. Because of this, it has been proposed to use source level interpreters for debugging [OH88]. However, the speed degradation for using an interpreter instead of compiled code is too large for this method to be practically usable for any programs but the smallest. Partial evaluation of the interpreter with respect to the source program will remove this interpretive overhead to some extent [KHC91].

In a debugging environment, it would be convenient to be able to mix compiled and interpreted code. If only the code from modules that are suspected to contain an error are interpreted, the speed degradation might be tolerable. When debugging, it is also important that recompilation and relinking are quick, to make it easy to correct a problem and rerun the program to see that it works. In modern operating systems it is often possible to load a new version of a compiled function at run time; this is called *dynamic loading*. Dynamic loading of changed modules or functions could be used to rerun a program without relinking it.

One might argue that having an interactive interpreter eliminates the need for a debugger, since it is then possible to incrementally develop programs, testing the parts separately. Indeed, such an interpreter is a useful tool. But testing is not enough to make sure a program contains no errors. As Shapiro [Sha82] notes, the computer must be told, in one way or another, what a program is supposed to solve, and some constraints on how to solve it. It does not matter which language we use to express this piece of information, we are bound to make mistakes. And when such a mistake manifests itself, a tracer or debugger is needed to locate it.

6 Algorithmic debugging

Another idea, originally developed for debugging Prolog, is the concept of algorithmic debugging [Sha82], a method for semi-automatic bug detection. This method has now been extended to handle both imperative [Sha91] and functional [NF94] languages.

The user starts the algorithmic debugger when an externally visible bug symptom has been observed, i.e., when an execution of a program did not produce the expected result. Algorithmic debugging is a two phase process. First, the program is run and an evaluation dependence tree at the function level is built. Then, this tree is interactively traversed. Each node in the tree corresponds to a function call and a result of that call. While traversing the tree (in a top-down manner), the system asks, for each node, if the result of the corresponding function call is correct or not. Based on the answers the programmer gives, the system chooses which function call to show next. A bug has been found when the result of a function call is erroneous and all children nodes are correct. The method requires that the functions have no side effects, which makes it very suitable for pure functional languages.

This method looks very appealing, but it has some problems. It is only possible to debug programs that run for a rather short period of time (the actual size of the programs is of no importance), since the size of the evaluation dependence tree otherwise becomes too large for this method to be usable in practice. This problem has cleverly been overcome by Nilsson[Nil98], by just building a part of the total evaluation dependence tree, small enough to fit in the available memory. If some other part of the evaluation dependence tree is needed when debugging, the program is rerun to build the new part. This procedure is then repeated when needed.

Another problem is that the questions the programmer is asked can be very hard to answer. If, for example, the debugged program is a compiler, the programmer may be asked if the resulting assembler code of a certain program is correct. A third problem is that the number of questions the programmer has to answer might be very large, which makes the method inconvenient to use.

6.1 Example

Consider the program in figure 5. It checks the validity of a particular date, February 31. The expected result is of course `False`, but curiously the program returns `True`.

```
daysIn Jan = 31
daysIn Feb = 28
     ...
daysIn Dec = 31
valid (Date d m) = daysIn m <= d
date = Date 31 Feb
main = valid date
```

Figure 5: A (faulty) program for checking validity of dates.

The full evaluation dependence tree for the execution of the program is presented in figure 6.

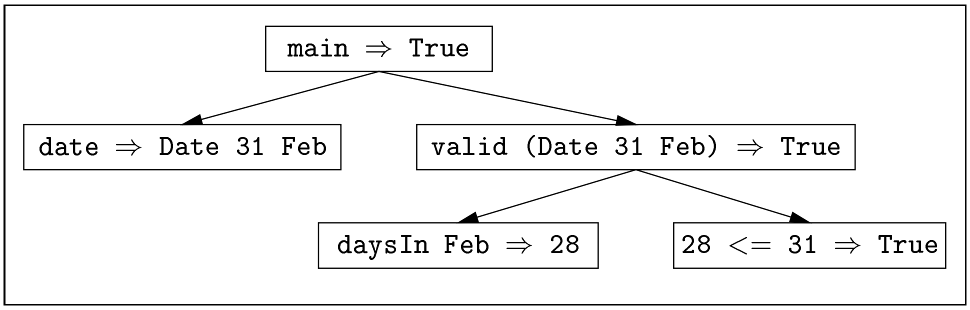


Figure 6: The evaluation dependence tree for the program in figure 5.

A session with an algorithmic debugger for this program might look as in figure 7. The text in *italic* style is user input.

Since the user answers that the result of the call to `valid` is faulty, and since all calls from `valid` (the children nodes to the `valid` call in figure 6) are correct, the system concludes that the definition of `valid` is incorrect.

```
main ==> True yes/no?  no
date ==> Date 31 Feb yes/no?  yes
valid (Date 31 Feb) ==> True yes/no?  no
daysIn Feb ==> 28 yes/no?  yes
28 <= 31 ==> True yes/no?  yes
Bug located in the function ``valid''.
```

Figure 7: An hypothetical session with an algorithmic debugger for the program in figure 5.

The problem is, in fact, that the `<=` operator in the definition of `valid` should be replaced with `=>`.

7 Tracing using redex trails

Another approach to tracing lazy functional computations is to link values, or failures, to their *parent redexes*, and each part of those redexes in turn to *their* parents, and so on until one reaches input data or initial expressions present in the program. We call such traces *redex trails*.

Consider the example program in figure 5 again. The corresponding redex trail structure can be viewed as in figure 8.

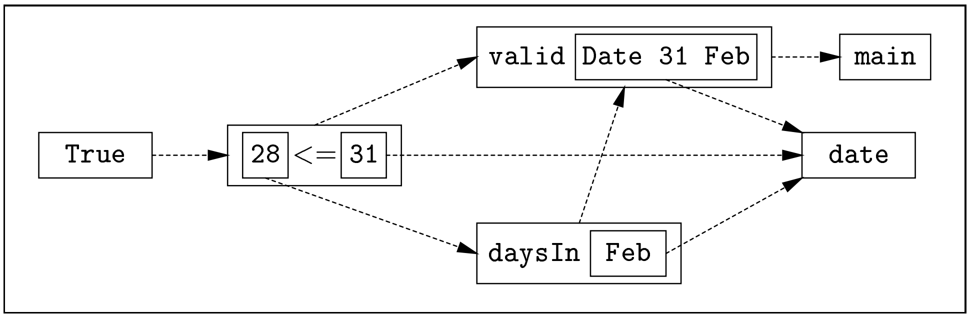


Figure 8: The redex trail structure for the program in figure 5.

The diagram shows the complete redex trail from the value `True` of the main expression. The immediate parent redex (whose reduction caused the creation of this instance of `True`) was `28 <= 31`. The parent redex of this application of `<=` was `valid (Date 31 Feb)`, but the parent redex of the left operand `28` was `daysIn Feb` and that of the right operand `31` was `date`. And so on. There is a link from every part of every expression to its parent redex.

Note that the redex trail can be seen as the inverse to the evaluation dependence tree mentioned in section 6. It is also reflected when traversing the two structures. In the case of the evaluation dependence tree, the traversal starts from the main expression, and the user is guided down to the problematic expression by the system, that itself navigates in the tree based on the user's replies. When using redex trails, the user starts at the very place where the problem symptom was identified (in this case the `True` value) and explores the trace *backwards*.

8 Overview of this thesis

This thesis consists of seven papers about tracing and debugging lazy functional programs. The papers in part II are about algorithmic debugging. Paper A, *The Evaluation Dependence Tree: an Execution Record for Lazy Functional Debugging*, is joint work with Henrik Nilsson. The paper defines an Evaluation Dependence Tree (EDT) as an execution record that captures the declarative aspect of a lazy functional computation. It is shown how the EDT can be used as a basis for debugging lazy functional computations, and two implementations of different ways to create EDTs are presented. Performance problems are discussed along with possible solutions, and some performance figures from experiments in a realistic context are given. The paper is an extended version of of a paper with the same name which was included in *Journal of Automated Software Engineering*, volume 4, number 2, 1997, which was a special issue about automated debugging.

Paper B, *Debugging Reactive Systems in Haskell*, is joint work with Amr Sabry. In the paper, we remark that although Haskell is well-suited for stream-oriented applications, like reactive systems, most implementations lack tools for locating subtle errors like deadlocks. To support the use of Haskell in such applications, we have designed and implemented a debugger with an intuitive graphical interface. The debugger simplifies the task of locating deadlocks by providing users with a mostly declarative view of programs as systems of recursive equations over streams.

Also, we investigate the use of Haskell in the implementation of the debugger. As expected, this effort requires a variant of Haskell, currently used in the intermediate phases of Haskell compilers, that combines call-by-need semantics with computational effects. To support robust programming in this system-oriented variant, we formalize the semantics of handleable exceptions in a call-by-need language. The paper was presented at the *Haskell Workshop* in Amsterdam, 1997.

Part III is about Redex Trails. In paper C, *Tracing Lazy Functional Computations Using Redex Trails*, which is joint work with Colin Runciman, we describe the design and implementation of a system for tracing computations in a lazy functional language. The basis of the tracing method is a program transformation carried out by the compiler: transformed programs compute the same values as the original, but embedded in functional data structures that also include redex trails showing how the values were obtained. The rules for the program transformations are described, and benchmark figures for tracing a few programs are given.

The paper also briefly discusses a special-purpose display program, enabling detailed but selective exploration of the redex trails, with cross-links to the source program. The paper was presented at the *International Symposium on Programming Languages: Implementations, Logics, and Programs* (PLILP), Southampton, 1997.

Paper D, *Complete and Partial Redex Trails of Functional Computations*, is also joint work with Colin Runciman. In paper C we presented redex trails, and also described an implementation of a redex trail tracer, that unfortunately suffered from some severe problems: trails did not record every reduction, only those constructing a new value; and even so, computing trails was very expensive, particularly in terms of the memory space they occupied. In this paper, we address both problems: *complete* redex trails provide a full computational record; *partial* versions of these trails exclude all but selected details, greatly reducing memory costs. Also, another space-saving method is presented: at garbage collection time, all "live" traces longer than a certain, user-settable value can be *pruned*, vastly reducing the total memory need for long-running applications, but possibly risking that some important part of the trace will be removed. We include results of experiments tracing several applications, including a compiler. The paper was presented at the *Workshop on Implementation of Functional Languages*, St Andrews, 1997.

After paper D was written, we have done more work on pruning redex trails. In an addendum to paper D, D.A *Implementing Adjustable Pruning*, we describe how the trail-pruning technique presented in paper D can be modified to automatically adjust the minimum live trail length, to do just as much pruning as is necessary, given the actual amount of memory available. Some examples that show the effects of using the technique are presented.

Paper E, *Redex Trail Representation and Access Protocol*, gives detailed information about the redex trail structure. An interface, the Redex Trail Access Protocol (RTAP), between the runtime system of the traced program and some user interface program accessing and browsing the trace structure is presented. Details about how source code references and identifier information are communicated are also specified. The intent of the paper is to enable compiler implementors to incorporate redex trail tracing in their respective systems.

Paper F, *Using the Redex Trail System*, is divided in two parts. The first part explains how to explore redex trails in our implementation of a trace display program. The display program makes full use of the information given to it, via the RTAP connection to a traced program. The user can expand and follow trails, see sharing between trace components, and easily look up the source code for all components in the trace.

In the second part, we compare our tracing system with Henrik Nilsson's debugger for a Haskell-like language Freja. In his thesis [Nil98], Nilsson gives an example program with a few errors inserted into it, and he shows how to use his debugger in order to locate the faults. To be able to get a valid comparison, we consider the same program, but use our tracer to locate the faults instead.

Part II

Algorithmic Debugging

Paper A

The Evaluation Dependence Tree: an Execution Record for Lazy Functional Debugging

Jan Sparud
Chalmers University of Technology
sparud@cs.chalmers.se

Henrik Nilsson
Linköping University
henni@ida.liu.se

Abstract

Lazy functional languages are declarative and allow the programmer to write programs where operational issues such as the evaluation order are left implicit. This should be reflected in the design of debuggers for such languages to avoid burdening the programmer with operational details, e.g. concerning the actual evaluation order. Conventional debugging techniques tend to focus too much on operational aspects to be suitable in this context. A record of the execution that only captures the declarative aspects of the execution, leaving out operational details, would be a viable basis for debugging lazy functional programs. Various declarative debugging tools could then be developed on top of such records. In this paper we propose a structure which we call the Evaluation Dependence Tree (EDT) for this purpose, and we describe two different construction methods. Performance problems are discussed along with possible solutions, and some performance figures from experiments in a realistic context are given.

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

A shorter version of this report has been submitted to the Journal of Automated Software Engineering

A.1 Introduction

The key advantage of declarative programming languages is that they allow the control component of a program, i.e. the exact evaluation order, to be left unspecified. This often allows a very succinct yet clear program formulation, focusing on the 'what' of the problem at hand rather than on a detailed description of exactly how to solve it. However, this also means that the language implementation must supply the missing control component, which makes it problematic to debug declarative programs using conventional debugging tools. The reason is that conventional debugging tools are based on observing execution events as they occur. This requires the programmer fully understand what is going on operationally, which is exactly what a declarative programmer usually prefers not to be concerned with. In other words, declarative programming calls for declarative debugging techniques which focus on the declarative aspects of the program and allow debugging to take place at the same conceptual level as programming.

One class of declarative languages are the lazy functional ones, such as Haskell [HPJW$^+$92]. They are characterized by demand driven evaluation: no expression is evaluated unless its result is needed to compute part of the output, and then it is evaluated at most once (hence lazy). 'Lazy debugging' is difficult for precisely the reasons discussed above [OH88], and there are currently no realistic, general purpose debuggers available for this type of language. This confines lazy functional programmers to basic debugging techniques, such as bottom-up testing of individual functions, or, if available, tracing at the operational level.

Due to the high-level nature of lazy functional languages, and the fact that most modern lazy languages features static, polymorphic type systems, basic debugging techniques actually go quite a long way. This is simply because fewer mistakes are made when programming in such languages than, say, when programming in an imperative one like C. Indeed, some kinds of very common programming errors, e.g. errors related to explicit memory management, just cannot occur when programming in a lazy language.[1] Nevertheless, sooner or later mistakes are made, and a good debugger is then a valuable and time saving aid.

[1]This is true for any language using automatic memory management, of course.

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 (the program being debugged), and 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 which is too operational, making declarative debugging difficult [NF94].

Our proposed 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. We call this particular kind of execution record *Evaluation Dependence Tree* (EDT), and in the following we will formally define what an EDT is and explain why EDTs constitute a suitable foundation for debugging lazy functional languages. Our EDT is an improvement over previously suggested trace structures in that it is defined for a more complete language than what has been the case earlier, and in that it explicitly includes information that is necessary to display functional values.

We also describe two different ways of constructing EDTs. The first is based on source code transformations and a small library of simple support routines. It should thus be portable without too much effort. The second approach is to change the underlying language implementation. While the implementation effort is substantially larger in this case, the approach offers better performance.

The rest of this report is organized as follows. The basic problems of lazy functional debugging are reviewed in section A.2, and section A.3 then defines the EDT and shows how it addresses these problems. Section A.4 considers the problems of EDT construction. Two different strategies are explained and compared. Some architectural issues are addressed in section A.5 and then related work is reviewed in section A.6. Finally conclusions are given.

A.2 Lazy functional debugging

In this section, we will look at how the demand driven nature (call-by-need semantics) of lazy functional languages makes debugging at an operational level difficult. To fully appreciate the problem, it is useful to first have some intuition regarding what 'declarative' means in a lazy functional context, and in which sense the evaluation order is left unspecified.

	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 A.1: Spreadsheet example. The sheet to the right is the result of evaluating the one to the left. Note how the dependences between the expressions in the cells impose an evaluation order.

Figure A.1 illustrates the evaluation of a simple spreadsheet. The spreadsheet itself is represented as an array `s` of cells, where each cell may contain an expression or be left empty. To compute the result array `r`, the expressions in `s` must be evaluated. But since they contain references to the values of other expressions, this must be done in an order determined by the dependences between the expressions.

In a language with call-by-value semantics, such as an imperative one, 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, could be used.

In a lazy functional language, in this example Haskell, the evaluator can be specified declaratively by simply stating the the relationship between `s` and `r`, as follows:

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

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 corresponding index in `s`. Evaluation of a single expression is done using the auxiliary function `eval` that 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.

This example illustrates that what the user writes can be quite far removed from what happens operationally, with obvious consequences concerning the suitability of operational debugging methods.

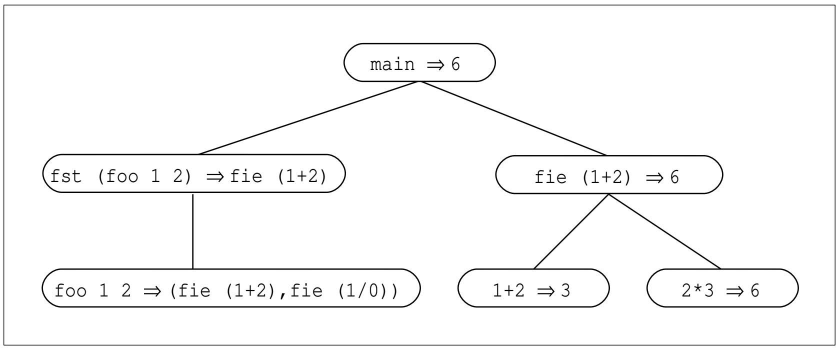


Figure A.2: Execution tree depicting lazy evaluation. The nodes correspond to evaluation of function applications, recording the arguments and the result. The parent–child relationship indicates that the parent caused the evaluation of its children.

Now, let us look at how lazy evaluation works at a more detailed level, and what problems it can cause from a debugging perspective. Consider the following functional program:

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

fie x = 2*x

main = fst (foo 1 2)
```

In a lazy functional language, the computation will proceed as follows. When the value of `main` is demanded in order to print the result of the program, `fst` will be applied to `(foo 1 2)`. Since `fst` extracts the first component of a pair, the result of `(foo 1 2)` is needed. However, `foo` does not know which components of the pair will be needed later, so it simply returns `(fie (1+2), fie (1/0))`, i.e. a pair of two unevaluated elements. Now, `fst` may extract the first component from the returned pair, so the result of `fst (foo 1 2)` is `fie (1+2)`. Since the result still is not a value, `fie` is invoked, in turn causing `+` and `*` to be invoked, yielding the final result `6`. The whole process is depicted as a tree in figure A.2.

There are two things to note here. First, the fact that arguments to functions and results from functions can be expressions, that may or may not be evaluated later. In real programs, such expressions are often very large, and it can be difficult to see what they stand for. Contrast this with the situation in a language with call-by-value semantics, where the arguments and result are always evaluated *before* calling the function and returning from it, respectively. Second, the structure of the source code is not reflected in the structure of the computation in any obvious manner. For example, by looking at the source code, one might arguably expect `foo` to call `fie`, but from the example above it is clear that this does not happen.

These two facts about lazy evaluation combine to make it very difficult to interpret the state of the computation or to make sense out of the order of execution events (such as function invocations), both which are fundamental techniques when debugging programs written in imperative languages.

To give another illustration of the problem, suppose that `main` in the program above had been defined as `snd (foo 1 2)`. Since `snd` is a function that returns the second component of a pair, we would get an execution error because the value of `1/0` is now needed. However, the error will not occur during the invocation of `foo`, the function in which the bug is located, but rather during the invocation of `fie`, which happened to be the first function that needed the value of `1/0`. Thus, in comparison to languages with call-by-value semantics, it is more difficult to relate an execution error to the construct in the source code that causes it.

A.3 The evaluation dependence tree

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 a key 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 section A.6, but it is interesting to note here that most work in the area suggests using tracing in some form to deal with the problem. Kamin [Kam90] even argues that tracing might well be inevitable in the context of debugging lazy functional programs.

Our approach is also based on tracing. We argue that the key to successful debugging in a lazy context is the construction of a declarative, tree structured trace, reflecting the structure of the source code rather than the order in which the various computations really took place, and in which values are seen in their most evaluated form. We will refer to this kind of trace as an *Evaluation Dependence Tree* (EDT) from now on.

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'. In a functional language, x will be a function application (or a constant) and proving equality means rewriting (or reducing) x to y by using the equations in a directed manner as rewrite rules. Debugging is then just a matter of looking for erroneous conclusions and following the chain of reasoning down the tree until the mistake can be identified.

In this section we will define what an EDT is and show how it can be used as a basis for declarative debugging. It should be pointed out that we focus on the key ideas: exactly what a real EDT would look like may depend on the features of the language in question as well as various implementation choices.

A.3.1 A small functional language

In order to present the ideas in a formal as well as concrete manner, we will start by introducing a simple functional language containing only the most essential features. We do this by giving a denotational semantics in direct style 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.) In the next section we introduce the EDT and define it (for this particular language) by modifying the semantics so that the meaning of a program is the EDT corresponding to the execution in addition to the normal result. To avoid introducing yet another notation for the semantics specifications, we use Haskell as the meta-language. Haskell closely resembles the meta-language of traditional denotational semantics.

The language defined below is a small functional language with normal-order semantics. This is actually not quite the same as laziness, since under a lazy evaluation scheme, any particular expression is evaluated at most once, and there is no such guarantee here. However, this is really an operational issue, as laziness is just a way of implementing a language with normal-order semantics efficiently. To keep the semantics simple it focuses on what a program computes rather than how the computation would proceed in a realistic implementation. There is a price to pay for this simplicity, however, as will become clear when the EDT semantics is defined, since a completely formal characterization of the EDT cannot be given unless the lazy aspects are taken into account.

The syntactic domain

First we need a domain for the abstract syntax of programs in the language:

```
data Exp = LitInt Int        -- Literal integer
         | Prim String       -- Primitive
         | Var String        -- Variable
         | App Exp Exp       -- Application
         | Letrec [Def] Exp  -- Recursive definition
data Def = VarDef String Exp          -- x = exp
         | FunDef String [String] Exp -- f x y z = 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, permitting recursive definitions. 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.

If we invent some concrete syntax for this language, the example from section A.2 can be written as follows:

```
letrec
   foo x y = mkpair (fie (x+y)) (fie (x/0))
   fie x = 2 * x
in
   fst (foo 1 2)
```

If this is encoded into the abstract syntax defined above, we get:

```
Letrec
   [ FunDef "foo" ["x", "y"]
        (App (App (Prim "mkpair")
                  (App (Var "fie")
                       (App (App (Prim "+") (Var "x"))
                            (Var "y"))))
             (App (Var "fie")
                  (App (App (Prim "/") (Var "x"))
                       (LitInt 0)))),
     FunDef "fie" ["x"]
        (App (App (Prim "*") (LitInt 2)) (Var "x")) ]
   (App (Prim "fst")
        (App (App (Var "foo") (LitInt 1)) (LitInt 2)))
```

Note that infix operator application such as `(x+y)` has been translated into prefix function application. Furthermore, a function whose *arity*, i.e. number of arguments, is greater than 1, is applied to its arguments one by one. This is known as partial application or *currying*. The idea is simple: an application of a function of arity n to a single argument yields a function of arity $n-1$ that behaves as the old one with the first formal argument fixed to the argument to which it was applied. Thus, in the example above, `(Prim "*")` is a function of arity 2 (arithmetic multiplication) and `(App (Prim "*") (LitInt 2))` is a function of arity 1 that multiplies whatever it is applied to by 2.

The semantic domain

Then a semantic domain is needed, i.e. objects that capture the meaning of syntactic objects in the language:

```
data Val = Int Int
         | Constr String [Val]
         | Fun String Int ([Val] -> Val)
         | PApp Val [Val]
         | Thunk (() -> Val)
```

The `Constr`-constructor is used to represent constructed data objects such as tuples, list-cells, or booleans. As an example, the pair `(1,2)` could be represented as `(Constr "Pair" [Int 1, Int 2])` and the list `[42]` as `(Constr ":" [Int 42, Constr "[]" []])` depending on exactly which names we choose for the constructors.

Built-in and user-defined functions are represented by functions in the meta-language. The semantic value of a function of arity n is a function that expects a list of n semantic values as its only argument and returns a semantic value. It is necessary to explicitly keep track of the arity of the represented function, and thus there is an integer-valued field with this purpose. An unusual feature is that the name of the function (a string) is also present in the semantic representation. This is necessary since, for debugging purposes, it must be possible to provide a sensible textual representation for all semantic values, including functional ones.

Partial applications represent functions that have been applied to some but not all of their arguments. As was explained above, a partial application is also a function. Its arity is given by the arity of the partially applied function less the length of the list of the accumulated arguments. The reason for having a separate, explicit, representation for functions that really are partial applications of built-in or user-defined functions, is again that it must be possible to display such values in a recognizable form during debugging.

A thunk, finally, is a suspended computation, represented by a function that when applied to unit, `()`, performs the suspended computation. Strictly speaking, the introduction of thunks is superfluous since the normal-order semantics of the meta-language carries over to the defined language anyway. However, making the thunks explicit serves to emphasize that some expressions are not evaluated immediately, which is important in the following.

A few utility functions for semantic values will also be needed:

```
arity (Int _)       = 0
arity (Constr _ _) = 0
arity (Fun _ n _)  = n
arity (PApp f as)  = arity f - length as
arity (Thunk _)    = 0

-- First argument is assumed to have arity > 1
mkPApp f@(Fun _ _ _) a = PApp f [a]
mkPApp (PApp f as)   a = PApp f (as ++ [a])

-- First argument is assumed to have arity = 1
apply (Fun _ _ f)           a = f [a]
apply (PApp (Fun _ _ f) as) a = f (as ++ [a])

force (Thunk t) = t ()
force v         = v
```

The function `arity` computes the arity of a semantic value. The function `mkPApp` applies a function or partial application of arity greater than 1 to one argument, which results in a partial application. The function `apply` applies a function or partial application of arity 1 to a single argument which results in the computation of a value. The function `force` forces the evaluation of a thunk.

The valuation functions

Next, environments keeping track of variable bindings are needed. We choose to represent environments as lists of pairs of a variable name (a string) and a semantic value:

```
type Env = [(String,Val)]

emptyEnv = []

updEnv :: Env -> [(String,Val)] -> Env
updEnv env new = new ++ env

lookup :: Env -> String -> Val
lookup [] x = error ("Variable " ++ x ++ " not bound.")
lookup ((x', v) : env) x | x == x'   = v
                         | otherwise = lookup env x
```

Finally we can give the valuation functions, which map syntactic objects to their meaning:

```
eval :: Exp -> Env -> Val
eval (LitInt n)    env = Int n
eval (Prim p)      env = primVal p
eval (Var v)       env = force (lookup env v)


eval (App e1 e2)   env =
    if arity f > 1 then
        mkPApp f a
    else
        apply f a
    where
        f = eval e1 env
        a = Thunk (\() -> eval e2 env)
eval (Letrec ds e) env =
    eval e env'
    where
        env' = updEnv env (elaborate ds env')

elaborate :: [Def] -> Env -> [(String,Val)]
elaborate [] env = []
elaborate (VarDef x e : ds) env =
    (x, Thunk (\() -> eval e env)) : elaborate ds env
elaborate (FunDef f xs e : ds) env =
    (f, Fun f (length xs)
```

```
            (\as -> eval e (updEnv env (zip xs as)))) :
        elaborate ds env
```

The function primVal is left unspecified. It simply maps the names of the built-in constants and functions to the corresponding semantic values.

The result of applying eval to an expression will in general not be a value in normal form (NF), i.e. a value that cannot be further evaluated, but a value in *weak head normal form* (WHNF). WHNF means that no further evaluation, or *reduction*, is possible at the top-level, but it does not preclude the possibility that there are *reducible expressions* (*redexes*), modelled by thunks, in inner positions. For example, if an expression computes a pair, the object returned from eval would be something like `Constr "Pair" [Thunk <f>, Thunk <g>]` where `<f>` and `<g>` represent functions that will compute the first and second component of the pair (again in WHNF) when applied to `()`. Note that partial applications are in WHNF.

Evaluation to WHNF rather than NF is a crucial ingredient in the demand driven evaluation strategy as it postpones evaluation as long as possible. On the other hand this means that demand must be created at the top level, there must be some *printing mechanism* that forces the evaluation of thunks as needed for printing purposes. Since we are going to reason informally about demand in the following anyway, we omit the specification of the printing mechanism here.

Keeping the discussion on WHNF in mind, most of the definition of eval is straightforward. When a variable is evaluated, force must be used to ensure that the returned value is in WHNF. Function application is handled by applying a function to its arguments one by one, accumulating the arguments by creating partial applications, until all arguments are present. Note how the function is always evaluated to WHNF wheras the evaluation of the argument is postponed by creation of a thunk.

The function elaborate elaborates a sequence of definitions in an environment and returns a list of bindings, i.e. pair of a variable name and a semantic value. The returned bindings are used to update the old environment when evaluating a Letrec-expression, yielding a new environment in which the body of the Letrec is evaluated. But note that the elaboration of the definitions *also* occurs in the new environment so as to allow the definitions to be (mutually) recursive.

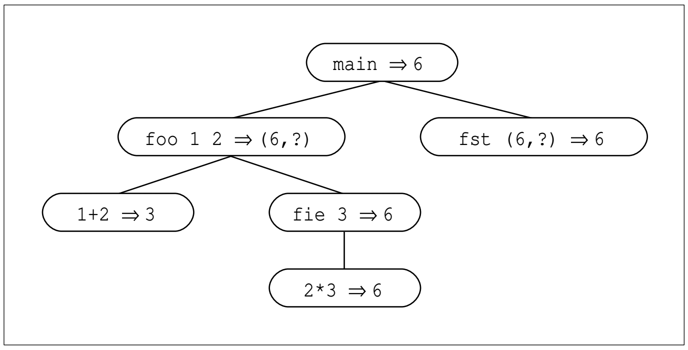


Figure A.3: EDT for the example in section A.2. Note that the structure of the EDT reflects the structure of the source code, but that only nodes corresponding to function applications that actually were evaluated are present. Also note that some values were never evaluated. These are shown as `?`.

A.3.2 Definition of the EDT

An EDT is a tree-structured trace whose structure is given by the structure of the source code, rather than by the structure of the actual execution. A node in the EDT corresponds to a function application, recording arguments and result, and the children of a node are those applications on which the application depends. Declaratively, a node shows that the system was able to prove one term, the application, equal to another, the result. We define a function application to depend on another if the latter is an instance of an application that syntactically occurs in the body of the former, and that *eventually* became evaluated. Here, application is understood to mean application to all arguments. There are no nodes corresponding to partial applications.

As an example, consider figure A.3 which shows the EDT for the example in section A.2. Note that the application `foo 1 2` only depends on one application of `fie`, since only the value of the first component of the pair was needed, even though two applications of `fie` syntactically occur in the body of `foo`.

Furthermore, arguments and results should occur evaluated as far as possible in the EDT. This is both because we think it is easier to understand the behaviour of the target if arguments and results are shown as basic values rather than as expressions representing the same values, and a consequence of the syntactically oriented definition of dependence above. It is important to realize that some values may never become fully evaluated during the execution, and that such values cannot be further evaluated afterwards, during the debugging, without risking causing errors or non-terminating computations that do not normally occur.

Consider the EDT in figure A.3 again. The second component of the pair returned from `foo 1 2`, which also occurs as the argument to `fst`, is shown as `?`, indicating an unevaluated value, i.e. something that was not evaluated during the normal execution. By looking at the source code, we see that this `?` represents the suspended computation `fie (1/0)`, and from the definition of `fie` it is clear that any attempt to evaluate this further would cause an execution error. Should an evaluation attempt cause a non-terminating computation it would in general be impossible for the debugger to regain control, save by means of some heuristic such as a time out.

Now we will define the EDT more formally by modifying the semantics for the small functional language introduced earlier so that the meaning of a program becomes a pair of the normal result and an EDT. The main intention is to show how closely the structure of the EDT resembles the syntactic structure of the program and what kind of information must be obtainable from an implementation in order to construct the EDT. However, as explained above, another important aspect of the EDT is that it only contains nodes corresponding to applications that actually became evaluated. What is evaluated and what is not is dictated by demand, and in order to capture this, as well as the notion of showing values in their most evaluated form, a much more operationally oriented semantics than what was given earlier would be needed.

We will thus content ourselves by stating these aspects informally, noting that in a real setting the necessary information can easily be obtained from the language implementation (see section A.4.1). What we will do is to assume that everything is demanded everywhere, which in general is going to result in an infinite EDT. We then assume the existence of a function, aware of the actual demand structure, that is able to prune the tree into the correct shape. The actions of this hypothetical function are also stated informally in the semantics.

The new semantic domain

First, the semantic domain must be changed slightly. Functions now return a pair of a value and a list of EDTs. Furthermore, to make it possible to display functional values, it is not sufficient to just know the name of the function. Bindings of any free variables must also be recorded. The type of the first field of the `Fun`-constructor is therefore changed to accomodate this.

Thunks are needed to model values that were never needed, but since a debugger is never going to look inside a thunk (it must not cause further evaluation), the `Thunk`-constructor can be made nullary. A value `Bottom` is also introduced to capture run-time errors and non-termination. Note that neither `Thunk` nor `Bottom` are formally introduced by the semantics below. In a real implementation, however, corresponding objects are present once the execution has finished, and `Thunk` and `Bottom` serve to remind us of this fact. [2]

```
data Val = Int Int
         | Constr String [Val]
         | Fun EDTInfo Int ([Val] -> (Val, [EDT]))
         | PApp Val [Val]
         | Thunk
         | Bottom

type EDTInfo = (String, [(String, Val)])
```

To understand why it is necessary to keep track of the free variables, consider the following program fragment:

```
foo x = letrec
           fie y = x * y
        in
           fie 3
```

What is the correct value of `fie 3`? That depends on the value of `x` which is free in the body of `fie`. Thus, to be able to tell whether a function application returned the correct result or not, the user must be informed about any free variable bindings as in

```
fie 3 where x = 10  ⇒  30
```

[2] Clearly, for something corresponding to `Bottom` to be physically present in the case of non-termination, some heuristic for recovering from a non-terminating computation, such as the user explicitly aborting it, is needed.

The EDT domain

Then a domain for the actual EDT is needed. A node in the EDT records which function was applied, its arguments, the result and the EDTs on which the node depends. If the applied function has free variables, it is necessary to keep track of the corresponding bindings. This can be achieved in a straigt-forward way by regarding them as extra arguments to the function. The representation of a function (the type `Function`) is a three tuple consisting of the name of the function, its arity and the names of the free variables. By putting the extra arguments first on the argument list, a pretty printer will be able to distinguish the free variables from the real arguments by referring to the length of the list of free variable names and to present an application to the user in a form that is akin to the source code.

A new value domain `EDTValue` that abstracts away from the run-time representation of values is also introduced. Since there is no longer any point in distinguishing between functions and partially applied functions, these are merged into a single function representation (`EVClosure`) using the same mechanism for keeping track of free variables as was employed for an EDT node.

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

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

type Constructor = String
type Function = (String, Int, [String])
```

The function `vToEV` converts a run-time value to an EDT value and its definition should clarify the relationship between the two kinds of value. The auxiliary function `mkFunction` extracts name, arity and the names of any free variables from a function or partially applied function, whereas `freeVarVals` extracts the values bound to any free variables and converts them to EDT values.

```
vToEV :: Val -> EDTValue
vToEV (Int i)        = EVInt i
```

```
vToEV (Constr cn vs) = EVConstr cn (map vToEV vs)
vToEV f@(Fun _ _ _)  = EVClosure (mkFunction f)
                                 (freeVarVals f)
vToEV f@(PApp _ as)  = EVClosure (mkFunction f)
                                 (freeVarVals f ++
                                  map vToEV as)
vToEV Thunk          = EVUneval
vToEV Bottom         = EVBottom

mkFunction :: Val -> Function
mkFunction (Fun (fn, fvbs) n _) = (fn, n, (map fst fvbs))
mkFunction (PApp f _)           = mkFunction f

freeVarVals (Fun (fn, fvbs) _ _) = map (vToEV.snd) fvbs
freeVarVals (PApp f _)           = freeVarVals f
```

The new valuation functions

The new valuation functions can now be given. Starting with `eval`, the first thing to note is that the return type has been changed: in addition to a value, a list of EDTs, one EDT for each redex in the expression being evaluated, is returned. The order between the EDTs in the list is not crucial, but has been chosen to reflect innermost first, left to right reduction.

```
eval :: Exp -> Env -> (Val, [EDT])
eval (LitInt n)    env = (Int n, [])
eval (Prim p)      env = (primVal p, [])
eval (Var v)       env = ((lookup env v), [])
eval (App e1 e2)   env =
    if arity f > 1 then
        (mkPApp f a, edts1 ++ edts2)
    else
        (r, edts1 ++ edts2 ++ edt)
    where
        (f, edts1) = eval e1 env
        (a, edts2) = eval e2 env -- If e2 will be eval.
     -- (a, edts2) = (Thunk, []) -- Otherwise.
        (r, edt) = apply f a
eval (Letrec ds e) env =
    (r, edts1 ++ edts2)
```

```
        where
            (bs, edts1) = elaborate ds env'
            env' = updEnv env bs
            (r, edts2) = eval e env'
```

The first three cases are straightforward: there are no redexes and the list of EDTs is thus empty. In the case for `App`, the list of EDTs is formed by concatenating the EDT lists obtained by evaluating the function and the argument expressions. If it turns out that this application is really a redex, i.e. not a partial application, the EDT obtained by applying the function is also appended to the list. There are no EDT nodes corresponding to partial applications. In the `Letrec`-case, the EDT list is formed by concatenating the list obtained by elaborating the declarations and the list obtained by evaluating the expression.

Informally it is also shown that if an argument to a function, in a real implementation, was never needed, then that value will remain as a thunk and there will be no corresponding EDT nodes. To put it differently, this is where the hypothetical pruning function would come in and cut away unneeded parts of the tree.

The function `elaborate` is also changed so that it returns a list of EDTs corresponding to redexes in the right-hand sides of variable definitions in addition to the list of bindings. (Redexes in a body of a defined function are not evaluated until the function is applied.)

```
    elaborate :: [Def] -> Env -> ([(String,Val)], [EDT])
    elaborate [] env = ([], [])
    elaborate (VarDef x e : ds) env =
        ((x,r):bs, edts1 ++ edts2) -- If x is ever used.
     -- ((x,Thunk):bs, edts2)      -- Otherwise.
        where
            (r, edts1) = eval e env
            (bs, edts2) = elaborate ds env

    elaborate (fd@(FunDef f xs e) : ds) env =
        ((f, Fun (f, fvbs) arity sf):bs, edts)
        where
            (bs, edts) = elaborate ds env
            arity = length xs
            fvns = freeVars fd
```

```
          fvvs = map (lookup env) fvns
          fvbs = zip fvns fvvs
          sf as = (r, [EDTNode (f, arity, fvns)
                               (map vToEV (fvvs ++ as))
                               (vToEV r)
                               edts])
                where
                    (r, edts) =
                         eval e (updEnv env (zip xs as))
```

In the case of a variable definition, the list of EDTs is formed by appending the list obtained by evaluating the right-hand side of the definition to the list obtained by elaborating the remaining definitions. Informally, we are also reminded that if the defined variable is never used, it will remain unevaluated and there will not be any EDTs from redexes on the right-hand side. [3]

For a function definition, no redexes in the function body are evaluated at this stage. On the other hand, a function that when applied returns a result and an EDT corresponding to the application must be constructed. Furthermore, any free variable bindings must also be made explicit and bundled with the actual function using the `Fun`-constructor.

The names of the free variables, `fvns`, are found by applying the auxiliary function `freeVars` to the function definition `fd`. The definition of `freeVars` is straightforward and omitted. The values of the free variables are then looked up in the environment and collected in the list `fvvs`, and a list of bindings (name–value pairs) formed by 'zipping' `fvns` and `fvvs`.

In practice, some of the free variables are of little interest since it might be clear from the textual context what they stand for. Consider for example two mutually recursive function definitions in a `Letrec`, `f` and `g` say, where both `f` and `g` are free in the two function bodies. From the context is perfectly clear what they are bound to, so it might be a good idea make use of this information and only record free variable bindings that are not known statically.

The function `sf` is the semantic representation of the function being defined. It expects a list of arguments, as before, and returns a pair of the result and a list with a single EDT corresponding to the application. The

[3] An alternative here would be to introduce a single node corresponding to the evaluation of a variable, and make `edts1` the children of this node. Then the user would actually see what the defined variables becomes bound to during debugging, which could be helpful.

EDT is formed by constructing an EDT node where the relevant information about the invoked function, the values of the free variables, the arguments and the result are recorded. Note that all values are represented as EDT values. The children of the node are obtained by evaluating the body in the environment at the point of definition updated by binding the formal arguments to the actual ones.

This pretty much concludes the definition of the EDT for our small language. It remains to note that `eval` in general will return a list of more than one EDT for a program. A single EDT is obtained by adding a root node corresponding to the execution of the entire program. Finally, as was discussed earlier, to obtain the *real* EDT, the above EDT should be pruned so that only nodes and values that were actually needed remain.

An important aspect of the EDT definition above is that it clearly shows what kind of information must be obtainable from the run-time representation of various objects in a real implementation in order to construct EDT nodes and to convert values from their run-time representation to EDT values. In particular, 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. In practice even more information must be available to the debugger through the EDT, for example source code references.

Further, the informal parts of the semantics above makes it evident that to construct an EDT, it is necessary to know what will be evaluated and what will not. The only way of finding out this in general, is to wait until something actually is evaluated or otherwise until the execution has finished. Once the execution has finished anything that was never evaluated will be represented as a thunk. In order to construct the EDT it must thus be possible to inspect the run-time representation of values and recognize any thunks, without causing further evaluation.

A.3.3 EDT based debugging

Algorithmic debugging [Sha82] is one way of guiding the user through the possibly huge amount of data in an EDT. An algorithm-guided, interactive search is conducted in the EDT for the application that caused the externally visible symptom of the bug. Provided that the user can correctly answer whether results of certain function applications are correct or not, the erroneous function will be located. An alternative is to let the user browse the tree freely, looking for function applications and results that somehow seem

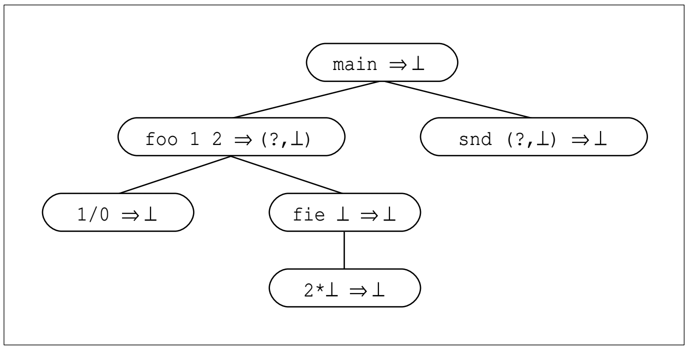


Figure A.4: EDT with a bug. ⊥ represents an undefined value, which in this case caused an execution error. Since the second component of the result from `foo` *was not supposed to be undefined, and since the nodes on which the application of* `foo` *depends show correct behaviour, the bug must be in the definition of* `foo`*.*

wrong.

To illustrate this, consider the example in section A.2 again, but assume that `main` is defined as `snd (foo 1 2)` instead. The resulting EDT is given in figure A.4.

If browsing freely, the user would find that ⊥ comes from the application of `foo`. He would then inspect this node and realize that the problem is a division by 0. Debugging algorithmically, the user would first be asked about the result of `main`, which is wrong, then about the application of `foo`, which is wrong as well, then about `1/0` yielding ⊥, which is correct, and finally about `fie` ⊥, also yielding ⊥, which is correct as well. Given these answers, the debugger would conclude that the bug must be in `foo`.

Algorithmic debugging for lazy functional languages, including handling of more complicated source constructs such as list comprehensions, is considered further in Nilsson [Nil94]. Sparud [Spa96] investigates the the free browsing scheme.

A.4 EDT generation

There are basically two options for generating an EDT from a target program. Either the target program is transformed so that the program itself generates an EDT in addition to its normal output, or the language implementation (a compiler or interpreter) is changed so as to produce an EDT in addition to the normal output when the target is executed. An implementation of each approach is presented in this section, and a comparison of the approaches is also made.

A.4.1 Graph reduction

Implementations of lazy functional languages are in general based on some form of graph reduction. Since the basic ideas are important for the remainder of the report, we will review them here. For details, refer, e.g., to Peyton Jones [PJ87].

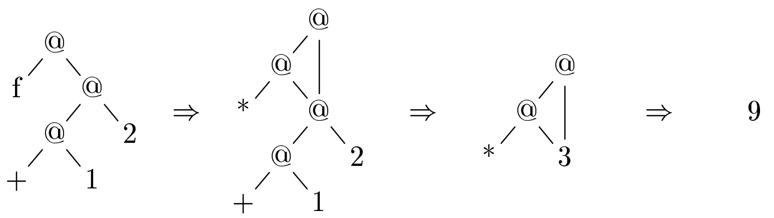
In a graph-reduction based implementation of a lazy language, expressions and data are represented in a uniform manner as pieces of graph (stored in the heap). As a matter of fact, the entire program is represented as one graph which is successively rewritten (or reduced or evaluated), on demand, until a graph representing the final answer is obtained. A key point is that whenever a reducible expression, or *redex*, 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 (to WHNF) 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 an expression otherwise could be evaluated more than once if it happened to be duplicated. For example, suppose that we have `f x = x * x` and consider the following normal-order reduction sequence:

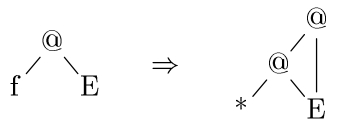
$$\texttt{f (1+2)} \;\Rightarrow\; \texttt{(1+2) * (1+2)} \;\Rightarrow\; \texttt{3 * (1+2)} \;\Rightarrow\; \texttt{3 * 3} \;\Rightarrow\; \texttt{9}$$

Note that the argument `(1+2)` was evaluated twice. However, by representing the expression as a graph the duplication of the argument can be avoided. For this kind of graph, it is conventional to denote an application by `@`, with the graph representing the function down to the left and the graph representing the argument down to the right. Using this convention we get the following reduction sequence. The argument (1+2) is now a shared subexpression, and since it is physically overwritten when evaluated, it is only evaluated once.



For an example of a circular graph, consider the 'infinite' list of ones: `ones = 1:ones`. The graph in this case is a cons-cell where the head is `1` and the tail is (a pointer to) the cons-cell itself.

In an efficient implementation of a lazy language, a function definition is compiled into code that rewrites a graph representing an application of the function to a graph in WHNF. The function `f` above would for instance be compiled into code that performed the following rewriting step:



Here, `E` represents an arbitrary graph. The entire graph above could of course be a subgraph of a larger graph. Thus the compiled code constructs an instance of the function body, substituting actual arguments for formal ones, and overwrites the redex with the constructed instance.

If a function contains free variables, a fixed code sequence constructing an instance of the function body cannot be generated since the values of the free variables in general are not known. One solution is to take out all free variables as extra arguments and lift all function definitions to the top level. This process is known as lambda lifting and results in a program which is just a collection of (mutually recursive) functions without free variables and inner lambda abstractions. Such functions are called *supercombinators* and can be compiled to fixed code sequences.

During the lifting process, all references to lifted functions are replaced by partial applications of the corresponding supercombinator to the values of the free variables. The result is close to the closure representation of functions that was used for `EDTValues` in section A.3.2.

Once the execution has terminated, it is very easy to find out exactly which expressions that were eventually evaluated simply by inspecting the graph. Anything that was not needed will still be a redex, whereas anything that was needed will be in WHNF.

Some implementations also overwrite a redex 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. However, should an execution terminate prematurely, because of a run-time error or the user interrupting the execution, this will not happen, and redexes that were under evaluation when the execution was aborted thus remains zapped. This is useful for debugging purposes since such redexes correspond to the undefined value, $\perp$, which thanks to zapping then has an explicit representation on the heap.

A.4.2 EDT generation by source code transformation

The purpose of the transformation is to make the functions in a Haskell program return not just a result, but also the EDT of that result. The types for the EDTs produced are the same as in section A.3.2, with the exception that `EDTValue` is extended with constructors for all types built into Haskell, not just `Int`.

How the transformation affects the types of functions

The return type of each function is a pair consisting of a result and a list of evaluation dependences for that result. We define a type abbreviation for this pair:

```
type R a = (a, [EDT])
```

As an example, consider a function that has thr type `Int -> Int -> Int`. The transformed version of the function will have the type `Int -> Int -> R Int`.

When building EDT values, we apply an (overloaded) function `edtVal` to ordinary values. Depending on the type of its argument, `edtVal` produces a corresponding value in the `EDTValue` datatype (see section A.3.2). This is no problem for ground (non-functional) types, but there is a problem with functional types. We have no means of taking a function apart to, e.g., find out its name or, should it be a partial application, which arguments it has already been applied to. This means that we cannot build an `EDTValue` given a functional value (we would like it to be represented by the `EVClosure` constructor).

To alleviate this problem we introduce a new data type:

```
data Fun a = Fun a Function [EDTValue]
```

where the type variable `a` represents a function, `Function` is the name of the function, and the list of `EDTValues` represents the arguments the function has been applied to so far. All closures (partially applied functions) are represented using this datatype, which makes it easy to build `EDTValues` from them.

For example, the type of the standard `foldr` function will change as follows:

$$\texttt{foldr ::}\quad \texttt{(a -> b -> b) -> b -> [a] -> b}$$
$$\Rightarrow$$
$$\texttt{foldr}_D\ \texttt{::}\quad \texttt{Fun (a -> b -> R b) -> b -> [a] -> R b}$$

The type transformation rules below show how the transformation will affect the types of the functions in a program.

$$f :: t \Rightarrow f_D :: \mathcal{T}[\![\ t\ ]\!]$$

$$\mathcal{T}[\![\ t_1 \rightarrow t_2\ ]\!] \Rightarrow \mathcal{F}[\![\ t_1\ ]\!] \rightarrow \mathcal{T}[\![\ t_2\ ]\!]$$
$$\mathcal{T}[\![\ t\ ]\!] \Rightarrow \texttt{R}\ (\mathcal{F}[\![\ t\ ]\!]) \qquad (t \text{ not a function type})$$

$$\mathcal{F}[\![\ t_1 \rightarrow t_2\ ]\!] \Rightarrow \texttt{Fun}\ (\mathcal{F}[\![\ t_1]\!] \rightarrow \mathcal{T}[\![\ t_2\ ]\!])$$
$$\mathcal{F}[\![tcon\ t_1 \ldots t_n\ ]\!] \Rightarrow tcon\ \mathcal{F}[\![\ t_1\ ]\!] \ldots\ \mathcal{F}[\![\ t_n\ ]\!] \qquad (n \geq 0,\ tcon \neq \text{'}\!\rightarrow\!\text{'})$$
$$\mathcal{F}[\![\ \alpha\ ]\!] \Rightarrow \alpha$$

Transforming definitions

The nodes in the EDT are created by the function `mkDep`, which is called from every transformed function.

```
mkDep :: (EDTAble a) => Function -> [EDTValue] -> R a -> R a
mkDep f vs (res, ds) = (res, [EDTNode f vs (edtVal res) ds])
```

The function `mkDep` is only used at the top level in a function or a value definition. The `(EDTAble a)` part of the type for `mkDep` is a type constraint on the type variable `a`, forcing it to be an instance of the type class `EDTAble`. This type class is discussed in section A.4.2. The arguments to `mkDep` are:

- The function name.
- A list of type `[EDTValue]` with the actual arguments to the function.
- A pair of an expression result and a list of evaluation dependences.

With these arguments, `mkDep` creates a dependence node, and returns a pair of the result of the expression and the dependence node. The translation schemes for definitions are shown below:

$$\mathcal{D}[\![\ f :: t\ ]\!] \Rightarrow f_D :: \mathcal{T}[\![\ t\ ]\!]$$
$$\mathcal{D}[\![\ f\ p_1\ \ldots\ p_n = e]\!]$$
$$\Rightarrow$$
$$f_D\ \mathcal{P}[\![\ p_1]\!] \ldots \mathcal{P}[\![\ p_n\ ]\!] = \texttt{mkDep}\ \mathit{fs}\ [\texttt{edtVal}\ a_1, \ldots, \texttt{edtVal}\ a_n]\ (\mathcal{E}[\![\ e\ ]\!])$$
$$\text{where } \mathit{fs} = \text{show } f$$

$\mathcal{P}[\![\ p_i\ ]\!] \Rightarrow a_i@(\texttt{Fun}\ f_i\ \_\ \_)$	if p_i has a functional type
$\mathcal{P}[\![\ p_i\ ]\!] \Rightarrow a_i@p_i$	otherwise

Transforming expressions

We are interested in collecting information about function applications. So what we really want to do is to change the behaviour of function application to automatically collect this information. Consider the definition:

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

The body of `foo` contains three subexpressions, all of which will return values as well as dependence trees. An EDT node corresponding to a call to `foo` will thus have three children (assuming that the result from each subexpression is used). When we transform the expression we must propagate the dependence information. One way is to introduce bindings for the intermediate values and the dependence information for the subexpressions, and then return the final value and the concatenated list of dependence trees. Thus, the following transformation could be performed:

```
                         let  (a1, d1) = g x
                              (a2, d2) = h y
f (g x) (h y)    ⇒            (a3, d3) = f a1 a2
                         in   (a3, d1 ++ d2 ++ d3)
```

To hide the plumbing with the dependence nodes, we will introduce a continuation based application operator `$$`, that transparently propagates dependence information from left to right. It is defined as follows:

```
infixr 6 $$
($$) :: R a -> (a -> R b) -> R b
~(x, ds) $$ f = let (y, ds') = f x in (y, ds ++ ds')
```

The `~` denotes a lazy pattern, i.e., the pattern matching will not be performed until one of the variables bound in the pattern is actually needed. The intention is that `$$` takes as its left argument an expression returning a pair of a value and a list of evaluation dependences. It then passes on the value to the function given as its right argument. That function returns a new value and a list of evaluation dependences. The result of `$$` is finally a pair with the new value and a list with all the evaluation dependences.

An expression is transformed into a chain of `$$` applications, one for each subexpression. Using `$$`, this transformation should be performed on our example expression:

```
                       g x $$ \a1 ->
 f (g x) (h y)    ⇒   h y $$ \a2 ->
                       f a1 a2
```

If a function is built-in, or is one that that we for some reason do not want to debug, we must compose that function with a *return* function, that takes any value and returns a pair with that value and an empty list of evaluation dependences.

```
return :: a -> R a
return x = (x, [ ])
```

For example, the expression `x + f y` should be transformed into

```
f y $$ \a1 ->
return (x + a1)
```

since `+` is a primitive built-in function, which we trust and therefore do not want to debug.

We will now give the transformation scheme $\mathcal{E}$ for transforming expressions. In some of the transformations new variables are introduced. They are always called v_n, where n is unique number, so no name clashes will occur. To simplify the presentation, we will assume that all used functions are transformed.

Constants

Constants, for example the number `1` or the string `"foo"`, need not be transformed, they are just returned, since they are already in weak head normal form.

$$\mathcal{E}[\![\ \mathit{const}\ ]\!] \Rightarrow \texttt{return}\ \mathit{const}$$

Identifiers

Identifiers need not be transformed, except if they have a functional type, in which case they are "lifted" into the `Fun` type as explained in section A.4.2.

$$\begin{array}{lll} \mathcal{E}[\![\ \mathit{id}\ ]\!] & \Rightarrow & \text{if arity } \mathit{id} > 0 \text{ then} \\ & & \qquad \texttt{return}\ (\texttt{Fun}\ \mathit{id}\ \mathit{ids}\ [\,]) \\ & & \text{else} \\ & & \qquad \texttt{return}\ \mathit{id} \\ & & \text{where } \mathit{ids} = \text{show } \mathit{id} \end{array}$$

Data values

Data values need not be transformed, since they are already in WHNF, but their subexpressions are transformed.

$$\begin{array}{lll} \mathcal{E}[\![\ \mathit{conid}\ e_1\ e_2\ \ldots e_n\ ]\!] & \Rightarrow & \mathcal{E}[\![\ e_1\ ]\!]\ \texttt{\$\$}\ \backslash v_1\ \texttt{->} \\ & & \mathcal{E}[\![\ e_2\ ]\!]\ \texttt{\$\$}\ \backslash v_2\ \texttt{->} \\ & & \ldots \\ & & \mathcal{E}[\![\ e_n\ ]\!]\ \texttt{\$\$}\ \backslash v_n\ \texttt{->} \\ & & \texttt{return}\ (\mathit{conid}\ v_1\ v_2\ \ldots\ v_n) \end{array}$$

Note that this general transformation rule may produce unnecessarily inefficient code. For example, the expression `x:xs` will be transformed as follows:

$$\mathcal{E}[\![\ \texttt{x:xs}\ ]\!] \Rightarrow \begin{array}{l} \texttt{return x \$\$}\ \backslash v_1\ \texttt{->} \\ \texttt{return xs \$\$}\ \backslash v_2\ \texttt{->} \\ \texttt{return}\ (v_1 : v_2) \end{array}$$

Of course, this is easily optimized to `return (x:xs)`. The same optimization can be performed in the rules for applications below.

Applications

This is the most complicated transformation. If the function in an application takes k arguments (it has *arity* k), but is given less than k arguments, we have to build a `Fun` value for the application, otherwise we need not. Note that the function part of the application might be a partially applied function itself. The `where`-clause pattern matches on the result of the transformation of that function part, to extract information about it.

$$
\begin{array}{l}
\mathcal{E}[\![\ f\ e_1\ e_2 \ldots e_n\ ]\!] \\
\qquad \Rightarrow \\
\text{if arity } f' \leq m + n \text{ then} \\
\qquad \mathcal{E}[\![\ e_1\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_1\ \texttt{->} \\
\qquad \mathcal{E}[\![\ e_2\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_2\ \texttt{->} \\
\qquad \ldots \\
\qquad \mathcal{E}[\![\ e_n\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_n\ \texttt{->} \\
\qquad f'\ v_1\ v_2\ \ldots\ v_n \\
\text{else} \\
\qquad \mathcal{E}[\![\ e_1\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_1\ \texttt{->} \\
\qquad \mathcal{E}[\![\ e_2\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_2\ \texttt{->} \\
\qquad \ldots \\
\qquad \mathcal{E}[\![\ e_n\ ]\!]\ \texttt{\$\$}\ \texttt{\textbackslash}v_n\ \texttt{->} \\
\\
\qquad \texttt{return}\ (\texttt{Fun}\ (f'\ v_1\ v_2\ \ldots\ v_n) \\
\qquad\qquad\qquad \mathit{fs}\ ([a_1, a_2, \ldots, a_m]\ +\!\!+ \\
\qquad\qquad\qquad\quad [\texttt{edtVal}\ v_1, \texttt{edtVal}\ v_2, \ldots, \texttt{edtVal}\ v_n])) \\
\\
\text{where } \texttt{return}\ (\texttt{Fun}\ f'\ \mathit{fs}\ [a_1, a_2, \ldots, a_m]) = \mathcal{E}[\![\ f\ ]\!]
\end{array}
$$

Lambda abstractions

Lambda abstractions are lifted into the `Fun` type. A lambda abstraction is a function, so a dependence node should be created when it is called.

$\mathcal{E}[\![$ `\`x `->` e $]\!]$
$\Rightarrow$
`return (Fun (\`x `-> mkDep` $slam$ `[edtVal` x`] (`$\mathcal{E}[\![\ e\ ]\!]$`))` $slam$ `[ ])`
where $slam$ = "_lambda_"

Let-expressions

A Haskell `let`-expression consists of a number of local declarations and an expression that can refer to the bindings in the declarations, which may be mutually recursive. The transformation uses the $\mathcal{D}$ transformation scheme defined in section A.4.2.

$\mathcal{E}[\![$ `let` $decls$ `in` e $]\!] \Rightarrow$ `let` $\mathcal{D}[\![\ decls\ ]\!]$ `in` $\mathcal{E}[\![\ e\ ]\!]$

Case-expressions

`case` expressions are straightforward to transform. An alternative is to treat them as functions with the scrutinized expression as the argument, the function name `case`, and the result of the chosen alternative as the function result. The pattern transformation scheme $\mathcal{P}$ was defined in section A.4.2.

$\mathcal{E}[\![$ `case` e `of` $\{alt_1;alt_2;\ldots;alt_n\}$ $]\!]$
$\Rightarrow$
$\mathcal{E}[\![\ e\ ]\!]$ `$$ \`c `->`
`case` c `of` $\{\mathcal{A}[\![\ alt_1\ ]\!];\mathcal{A}[\![\ alt_2\ ]\!];\ldots;\mathcal{A}[\![\ alt_n\ ]\!]\}$

$\mathcal{A}[\![\ p$ `->` $e\ ]\!] \Rightarrow \mathcal{P}[\![\ p\ ]\!]$ `->` $\mathcal{E}[\![\ e\ ]\!]$

An example transformation

We will show how the standard function `map` is transformed. At the top level in the function, an EDT node is built using `mkDep`, containing the name of the function and references to the arguments and to the result. The recursive case of the definition contains two function applications, so the transformed version has two occurrences of `$$` in the body.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

⇒

```
map_D :: (EDTAble a, EDTAble b) =>
        Fun (a->R b) -> [a] -> R [b]
map_D v1@(Fun f _ _) v2@[] =
     mkDep "map" [edtVal v1, edtVal v2] $ return []
map_D v1@(Fun f _ _) v2@(x:xs) =
     mkDep "map" [edtVal v1, edtVal v2] $
     f x $$ \y -> map_D v1 xs $$ \ys -> return (y : ys)
```

Details on EDT generation

As yet, we keep no record of free variables in closures in our implementation. A possible way of doing this would be to add some fields to the `Fun` constructor to hold the free variable information. This change should be rather straightforward, and would make the debugger much more useful.

Arguments to functions are not necessarily fully evaluated, or evaluated at all, when a function is invoked. Because of this, we cannot just copy the values of the arguments when entering a function and building its EDT node. Instead we use the sharing property of a lazy functional language. In the EDT, we save references to the arguments, rather than copies of their values. This has the advantage that after the execution has terminated, we can access the values of the arguments in their most evaluated form. Similarly for function results. So the EDT node of a function call will *share* the result of the function call with whichever function it was that demanded that result.

The tree structure of the EDT is built during the execution of the target. The values in the EDT (of type `EDTValue`), on the other hand, are built *after* the execution has terminated. This is because of laziness: the EDT values are built by calls to `edtVal`, but the results of these calls are not needed until after the execution has terminated and the tree is examined.

However, we have a further problem: how can we detect that a value was unevaluated or undefined (bottom)? It is impossible to 'look' at a value in a lazy functional language to determine whether it has been evaluated or not. In fact, it is by looking at it, thus having a demand for it, that the value gets evaluated! On the other hand, it is easy to determine this on the language implementation level. We have solved this problem by introducing two non-pure functions, implemented as a part of the debugging runtime system:

```
evaluated  :: a -> Bool
evaluating :: a -> Bool
```

The semantics of `evaluated` is to take a value of any type and return `True` if the value is in weak head normal form, or `False` otherwise. If the value is not in WHNF, the value was not used when running the debugged program and we should not try to evaluate it when building an EDT value for it, but instead return `EVUneval`.

The semantics of `evaluating` is to take a value of any type and return `True` if the value is undefined (i.e., the value was under evaluation when the program terminated), or `False` otherwise. When `edtVal` detects, using the `evaluating` predicate, that a value is undefined, it returns `EVBottom` without trying to evaluate the value further. The implementation of `evaluating` is facilitated if the implementation zaps redexes, see section A.4.1.

Definition of `edtVal`

We mentioned earlier that `edtVal` is an overloaded function, defined separately for each type in the program. This is implemented using the Haskell class system. A Haskell *class* is a collection of functions that is parameterized over a certain type. After the class is defined, it is possible to make *instances* of the class for an arbitrary type. A function that uses the functions (or methods) in a class gets a *type constraint* (or type context) on the corresponding type variables. For example, the standard `sum` function takes a list of numbers and returns the sum of them. Its type is `(Num a) => [a] -> a`, which says that it can take a list of any type *that is an instance of the* `Num` *class* as argument, and return the sum of the elements. The type can for example be `Int` or `Double`, which both are instances of the `Num` class, but not `Char`, which is not.

We will introduce a class `EDTAble` that *all* types in a program will be an instance of. In the class we define methods for creating EDT values. The class is defined as:

```
class EDTAble a where
    edtVal :: a -> EDTValue
    edtVal x = if evaluating x then
```

```
                        EVBottom
                    else
                        if evaluated x
                        then safeEdtVal x
                        else EVUneval
        safeEdtVal :: a -> EDTValue
```

The `edtVal` method makes use of the `evaluating` and `evaluated` predicates. When making instances of this class, we only define the `safeEdtVal` method, which is only called by the `edtVal` method when the argument is known to be evaluated. Here are some examples of instances of the `EDTAble` class:

```
instance EDTAble Int where
    safeEdtVal i = EVInt i

instance EDTAble (Fun a) where
    safeEdtVal (Fun _ f as) = EVClosure f as
```

Instances of the `EDTAble` class are automatically generated for user defined types.

Unevaluated dependences

The transformation takes into account all *possible* dependences for an expression. The *actual* dependences may be fewer than the possible ones. To see this, consider the following expression and its transformation:

```
                       f x $$ \a1 ->
fst (f x, g y)    ⇒   g y $$ \a2 ->
                       fst a1 a2
```

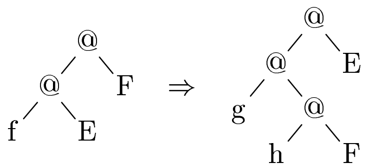
It is clear that even if the result of this expression is needed, the result of `g y` will never be needed, since `fst` returns the first component of a pair. But we have created a dependence for the call to `g`! We solve this problem by checking, once the program has terminated, that the result of a dependence node really is evaluated (using the `evaluated` primitive). Dependence nodes with values that have not been evaluated are filtered out, since they can be of no importance to the result of the program.

A.4.3 EDT generation by modifying the language implementation

We now describe how an EDT can be generated by modifying the graph-reduction machinery of a typical lazy functional language implementation. The case when the amount of memory for storing the EDT is limited, so that the entire EDT will not fit, is also considered, as well as various enhancements of the basic scheme.

The basic scheme

When a redex is being evaluated, the code that has been compiled for the applied function is entered. This code will construct an instance of the function body, i.e. a graph representing the expression that constitute the function body with actual arguments substituted for the formal ones, and then physically overwrite the redex root with the root of the newly constructed graph. Suppose that we have `f x y = g (h y) x`. The code for `f` will then perform the following rewriting step, where `E` and `F` are arbitrary pieces of graph:



The redex root is the top-most application node (@) of the left graph, and it is thus physically overwritten with the root of the right graph. The remaining two application nodes in the right graph are new, constructed by the code of `f` in some newly allocated memory cells, whereas the remaining application node in the original graph becomes garbage, to be collected by the garbage collector later, unless it happens to be shared.

Now, recall that a function application depends on another if the latter is an instance of an application that syntactically occurs in the body of the former and eventually becomes evaluated. This means that an EDT node corresponding to a function application cannot be constructed until that particular application is evaluated. Moreover, when this happens it must somehow be possible to determine where in the EDT this node should be inserted.

Achieving this is easy in principle: whenever a function is invoked, each application in the instantiated function body is annotated with a reference to the EDT node corresponding to the current function invocation. Thus, when a redex is evaluated, it is clear where the resulting EDT node belongs in the tree.

In the example above, the redex root of the left graph would contain a reference to an EDT node. This EDT node becomes the parent of the EDT node corresponding to the reduction of `f E F`, i.e. the former depends on the latter. The application nodes in the right graph are annotated with references to the new EDT node since this will become dependent on any EDT nodes that result when and if any of the three applications in the right graph are reduced.

This process can be optimized by observing that only application nodes that are redexes need to be annotated. If, for instance, `h` above is a function of arity two or more, then `h F` is a partial application. A partial application is not a redex, and there is thus no point in annotating that application node with a reference to the EDT node.

As we have seen earlier, function arguments are not necessarily fully evaluated. This means that there is no point in recording the arguments in the EDT node at the time of a function invocation. Instead, pointers to the arguments should be kept. Once the execution has terminated, these pointers will refer to the arguments in their most evaluated form by the nature of graph reduction. Results from functions are handled in the same way, i.e. a pointer to the result graph is maintained in the EDT node.

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 that the target no longer refers to, but which are part of the EDT, would be lost. Also note that, as the EDT grows, not only will the the EDT nodes themselves occupy memory, but the EDT will hold on to an increasing amount of heap memory. A large portion of this memory would otherwise be reclaimed by the garbage collector. We will return to this point in the next section.

We illustrate by a small example. Suppose that we have `f x = x + x`. Suppose also that we are evaluating `f E`, where `E` is some as yet unevaluated expression, returned as the result from an earlier application. The situation immediately after the reduction of `f E` is shown in figure A.5. 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 EDT (the grey 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

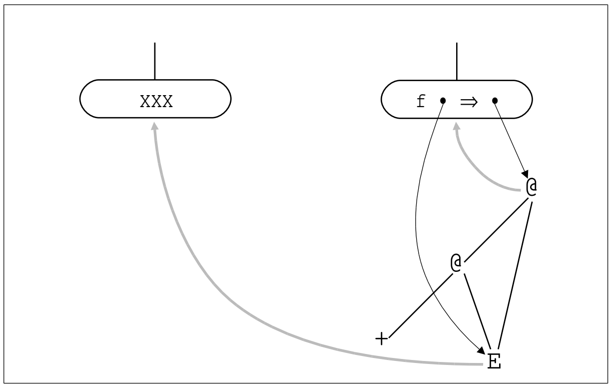


Figure A.5: The situation after reduction of `f E`. *Note that the arguments and results in the EDT nodes are live pieces of graph, referred to via pointers. Also note that each redex refers to the EDT node corresponding to the function invocation that created that redex (the grey arrows).*

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

The new situation is shown in figure A.6. The nodes corresponding to the reductions of the expressions `E` and `E + E` have been inserted as children of the EDT nodes that originally created them, rather than below the node corresponding to the current function invocation. Note how these expressions have been physically overwritten by the result of evaluating them. Thus, once the execution has terminated, values in the EDT will be evaluated as far as possible.

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 A.4.1), the zap nodes may be used as a representation of $\perp$.

If an EDT node has several children, how should they be ordered? In section A.3.2, an inner-most redex first, left to right order was chosen. To achieve this, the redexes must be numbered when created so that they can

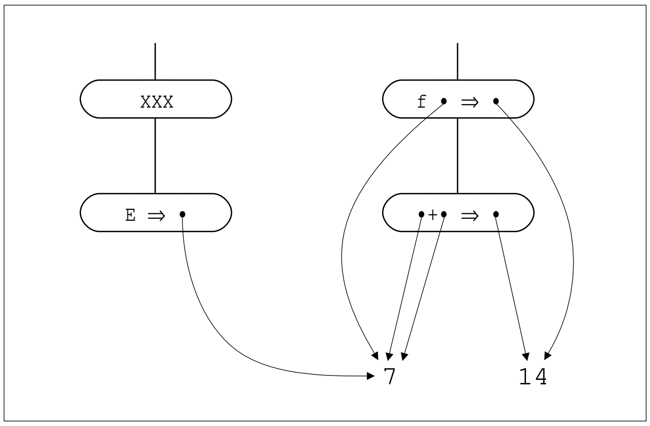


Figure A.6: The situation after reduction of `E + E`.

be sorted into order as and when they are evaluated.

Finally 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.

Constructing an EDT with the proper structure in this case is easy in principle, since the node corresponding to the current function invocation depends on the node corresponding to the function call being made, i.e. the latter node should be a child of the former. Should this for some reason be difficult to implement, it is always possible, at some performance cost, to resort to a more naive evaluation strategy where the graph is constructed explicitly, in which case the EDT construction scheme as described above is directly applicable.

Piecemeal EDT generation

A big problem with the construction of an EDT is of course that there is no upper bound on its size. Even a program that normally runs in constant memory space with very modest memory requirements, may generate a huge EDT if it runs for long enough. Furthermore, as illustrated by the example in the previous section, the EDT grows at many different places simultaneously, and arguments and results are pointers to live pieces of graph which may move around as a result of garbage collection. This makes it difficult to store the EDT on secondary storage with reasonable performance, constraining us to keep everything in primary memory. The problem may be alleviated by means of various filtering techniques (see section A.4.3), but as long as an entire EDT is kept, the problem remains.

Note that the transformational EDT generation scheme suffers from exactly the same deficiency. While actual nodes in the EDT in that case might not be created until the debugger demands them, thanks to working in a lazy language, a large expression representing the entire EDT is certainly going to be constructed as the execution proceeds. This expression contains references to all function arguments and all results throughout the entire execution, and its size is thus of at least the same magnitude as an EDT constructed as described in the previous section.

An interesting alternative is to store only so much of the EDT as there is room for. Debugging is then started on this first piece of the EDT. If this is enough to find the bug, all is well. Otherwise, the target is re-executed, and the next piece of the EDT captured and stored. We refer to this as *piecemeal EDT generation*. Re-executing the program is not a problem in principle, 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 execution time obviously depends on how long it normally would take to execute the program. But it should also be kept in mind that it takes time to construct an EDT node. This means that if only a little memory is set aside for the EDT, relatively few nodes will be constructed and the target program will thus run quicker than if significantly more memory was reserved. If many nodes are stored, it is also likely that only a fraction of these will ever be visited during debugging, so constructing them was really work wasted. The time for garbage collection also enters the equation since a larger EDT means more live data and thus longer garbage collection times.

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 it seems plausible that a suitable compromise between execution time on the one hand, and the frequency of re-execution on the other, could be found by adjusting the amount of memory that the EDT is allowed to occupy. Furthermore, it is not evident that more memory is better: not only will a user be able to start debugging earlier if only a small part of the EDT is stored, and execution time thereby shortened, a piecemeal scheme might also be the cheaper approach overall if the overhead of EDT construction is large and the program is only re-executed a few times.

Also note that it is not absolutely necessary 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.

Reducing the size of the EDT

Up until now, we have assumed that every single function application that takes place during execution should be recorded in the EDT. However, a large number of these are applications of language primitives (e.g. `+`, `if`) which should be correctly implemented. Moreover, large systems are built modularly, so it will often be the case that there are many well tested, trusted functions, and a few, new, 'prime suspects' when a bug manifests itself. At the very least, it is usually reasonable to trust standard library functions.

Thus, it is possible to reduce the size of the EDT by recording only the interesting function applications. This is easy to achieve: just do not create any EDT node when a trusted function is executed, and do not annotate any redexes created in this case with EDT references. The effect will be that no part of the EDT below this point is created.

The above behaviour is what is desired when dealing with correct first-order functions. A function is correct if it computes the expected result for whatever arguments it is applied to. During debugging, this means that a user would not investigate any applications of this function nor, by transitivity, any nodes below them. Thus the entire branch emanating from an application of a trusted first-order function may be deleted.

Higher order functions are more problematic. Since they may invoke their argument functions, the combined behaviour may be wrong even though the higher order function itself is correct. Removing the entire subtree emanating from an application of a trusted higher-order function is thus not a good idea. We are not yet sure how trusted, higher-order functions should be handled in the best way. But just getting rid of trusted first-order functions is a huge win, and until a better scheme is devised, one may simply avoid trusting higher-order functions.

Whether a function is trustworthy or not is something that can be decided both at compile-time and at run-time. In the latter case, a flag, which can be set from the debugger, is associated with the function and the code for the function checks this flag dynamically so as to adapt its behaviour.

Another enhancement that reduces the size of the EDT is to begin EDT construction from selected functions rather than from the root, something akin to break-points. It is straightforward to implement this facility so that the 'break-points' may be set interactively from the debugger. Re-executing the program would then produce a forest of EDTs, where each root node is an application of one of the selected functions.

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 the 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 whatever is deemed suitable. Under a piecemeal scheme, it is possible to get to the desired point by re-executing the program once, as long as the reference to the desired starting node is known. One method is to annotate a redex not only with a reference to its parent in the EDT, but also with references to its grandparent, great grandparent etc. up to some predefined number of levels.

Alternatively, complete 'call chains' containing EDT references could be maintained, similar to what Hazan proposed [HM93] but constructed at a much 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 main memory. Also note that partial chains no longer referenced from any redexes 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. As explained in the next section, one feasible EDT reference is the reduction number. When a reduction takes place, the reference to the parent 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.

Implementation

An implementation of the low-level EDT generation scheme described above consists of two main parts. One part are the routines that perform the actual tree construction. Our current version is implemented in ANSI-C and is thus reasonably portable. The second part is a modified compiler that generates code which interface properly to the above routines. The necessary modifications require a substantial implementation effort each time EDT generation capability is to be added to a new system.

Let us first introduce some basic ideas. 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. Each node in the *complete* tree must therefore be given 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 corresponds to a single reduction, the current reduction count can be used as the *id* of the node corresponding to the latest reduction.

We also need criteria for deciding how much and which part of the tree to keep. The former aspect can be addressed by deciding on an upper limit on the number of tree nodes to store. This is the approach currently used. A better idea, however, would be to let the garbage collector measure how much extra heap space the EDT holds on to, and try to keep this at a reasonable level.

As to which part to store, one particular node in the complete tree is designated the *current root*. 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:

1. The distance of the current root from the current root is 0.

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

3. No other nodes are eligible for being kept, and their distance from the current root is undefined.

Thus each node in the complete tree is conceptually assigned a distance attribute, qd, with respect to the current root. 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 estimation of the distance attribute. This estimation can easily be computed given qd of the EDT node corresponding to the function application that created the redex, and syntactic features of the source code reflecting the innermost-first, left to right 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. As an 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 x)` will be attributed $d+1$, `(f y x)` $d+2$ and `g (h x) (f y x)` $d+3$ if it is known that these are the redexes. 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.

Given the distance attribute of a redex that is being reduced, it is unfortunately not always possible to a priori decide whether or not to create and insert the corresponding EDT node into the stored part of the EDT. This is because there may be other redexes left that are closer to the current root. Should such a redex be evaluated later, the corresponding node should be inserted into the tree at the expense of more distant nodes, which thus are pushed over the edge, as it were, of the stored portion of the tree. Another reason is that arguments and results may grow as the execution proceeds, and if a size based criterion is used for deciding how much of the tree to store, the number of stored nodes may have to be reduced to keep the overall size of the stored part of the EDT within the prescribed limits.

The construction process thus alternates between inserting nodes as redexes are reduced, and pruning of the tree in order to meet the size constraints. The pruning process always work from distant nodes towards closer ones. 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.

Before turning to the tree construction algorithms, let us briefly describe two important data structures: the low-level representation of the EDT nodes and the *traced application nodes*. Below is a somewhat simplified version of the EDT node (C syntax):

```
typedef struct edt_node {
   unsigned        id;
   int             qd;
   fun_info        fun;
   graph           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;
```

The fields `id` and `qd` were described above. The field `fun` is (a pointer to) information about the applied function, such as its name, names of free variables, source code references and any other information a debugger may need. The fields `args` and `result` points to the list of actual 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 (see below) 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. The field `rightsib` points to the sibling to the right of the node (if any), whereas `rightsib_qd` contains *qd* for that sibling *if it has been removed*, in which case the previous field becomes invalid. The *id* of a node and the *qd* of a missing node that one would like to visit during debugging is exactly the information that is needed for initiating a re-execution that captures the next part of the tree. The fields `firstchd` and `firstchd_qd` work in a similar manner, but refers to the first child of the node.

A normal application node in the graph 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*, i.e. the EDT node corresponding to the function application that created the traced application node, and an estimation of *qd* as explained earlier. 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 a pointer (address). This is because the parent might be removed due to pruning, which would result in dangling pointers from the heap into the memory area used for storing EDT nodes had pointers been used. In general it would then be impossible to decide whether the parent was still present. Instead a hash table is maintained that makes it possible to quickly find an EDT node given its *id*, if it is still present.

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 (untrusted) function (really 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 trusted functions) 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 corresponding to applications of instrumented (untrusted) functions.

The following main steps are performed when an instrumented function is invoked:

1. Call *Trace*, the main EDT construction routine, with a pointer to the redex root as argument.

2. *Trace* now determines whether or not to build an EDT node. 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, 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 a static offset, computed at compile time.

As was mentioned in section A.4.3, 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 trusted, there is no point in making that redex traced.

Trace has two modes. Either it is waiting for tree construction to begin, or tree construction has begun. In waiting mode, *Trace* performs the following steps:

1. Increment the reduction counter.

2. Decide whether it is time to begin. 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.

Once tree construction has begun, *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. Check the size constraints. Prune the tree if necessary in order to insert a new node.

6. Create and insert a new node among the children of node *id* at the correct place as indicated by *qd*. References to the arguments are obtained via the redex root. 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` is set to point to the information record for the calling function.

We conclude by a summary of the most important changes that must be made in a compiler and the run-time system to support the above scheme.

- Generation of instrumented code.
- Generation of information for the debugger (function names, source code references, etc.) to be included with the generated code.
- The representation of the graph must be changed so it can be understood by the debugger, e.g. it must be possible to figure out names of functions and identifying free variables. This was discussed in section A.3.2.
- The garbage collector must be made aware of the EDT, since the EDT holds references to the graph.
- Control should be passed to the debugger on run-time errors and user interrupt.
- Support for re-execution, which involves preserving any input to the program and re-issuing a user interrupt at the appropriate moment.

A.4.4 Comparison of the approaches

The main advantage of the transformation based scheme is its portability. While the implementation relies on having access to two impure functions, these should be quite easy to implement in the context of most current Haskell implementations. Also, since only the transformations decide what the resulting EDT should look like, it does not matter what a compiler does to the code. In particular, a highly optimizing compiler does not cause any particular problems. Another advantage is that the code that performs the transformation is itself implemented in Haskell. It is thus not inconceivable that a knowledgeable user could adapt this code to suit his needs better.

The problems with this approach mainly concern performance. As the benchmarks below show, the transformed code runs considerably slower than code using the low-level EDT generation scheme. The worst aspect, however, is the memory consumption. Since the transformed programs always generate a complete EDT, only programs that do not perform a lot of computation can currently be debugged. If a practical debugger is going to be built along these lines, this problem must be addressed, perhaps as suggested by Naish & Barbour [NB95].

By working at the language implementation level it is possible to control the tree construction process in a very precise manner. This makes it possible to keep the resource requirements within reasonable (and user definable) limits. Working at the implementation level also incurs a smaller overhead with respect to execution time than does the current implementation of the transformation based scheme. Various enhancements, such as 'break-points' which can be set and reset interactively during debugging, are also easy to implement at this level. Another advantage is that mixing of untrusted code that generates EDTs, and trusted code which does not, poses no particular problems, save that it may not be desirable to trust higher order functions as discussed in section A.4.3. This makes it possible to call library code without first having to recompile it with debugging support, for example.

The main disadvantage with working at the language implementation level is that it requires a large implementation effort, most of which is specific to a particular implementation.

We conclude this section by giving some performance figures. In the first test, we have measured the time for constructing EDTs using both approaches for three small test cases. The test programs used were:

NFib Computation of 20th Fibonacci number, using a tree recursion.

Sieve Computation of the 200th prime, using the sieve of Erathostene.

Soda A larger program that searches for hidden words in a square of letters.

We used the Chalmers Haskell implementation, HBC, for our tests. The problems were made small enough so that the transformation based approach could cope. A complete EDT is thus built in both cases. To focus the comparison on the difference between our two approaches, time for garbage collection is not included in the figures below. All figures are relative to the normal execution time of the programs (excluding garbage collection), i.e. they show how many times longer the execution took when EDTs were constructed. The normal execution times were 0.11 s, 0.24 s, and 0.19 s for NFib, Sieve and Soda, respectively.

Approach	NFib	Sieve	Soda
Transformation based	14	25	8
Modified implementation	5	4	8

We also tested EDT generation on the language implementation level on larger problems. The same test programs were used, but the computations were made larger (30th Fibonacci number, 1000th prime, a larger array of letters and more words to search for). This time only partial EDTs were constructed, and we measured the times for a varying number of stored nodes. Also, garbage collection times are now included. They are for a generational garbage collector, which performs considerably better than the standard copying one in our case. This is because the stored portion of the EDT after a while does not change much, and thus will become part of the old generation. The old generation is garbage collected less frequently than the new generation which consists of newly created data, and thus we gain in comparison to a copying scheme, where all data is treated the same and moved around during each garbage collection. Again, the times are relative to the normal execution times, which in this case were 3.24 s, 3.29 s and 1.18 s, respectively.

# nodes stored	NFib	Sieve	Soda
1000	6.8	3.7	5.1
2000	7.1	3.8	5.3
4000	7.4	4.0	5.5
8000	7.7	4.5	6.2
16000	8.7	5.2	7.3
28000	11	6.5	9.7

These result are not as good as one could hope for. However, a closer inspection reveals that the actual time to construct the nodes only accounts for at most 10 % of the execution time. The garbage collection time obviously increases with increasing tree size, and accounts for 30–45 % of the execution time when 28000 nodes are stored. The remaining execution time is for the program itself, and it can be seen that it thus runs significantly slower than the original program. This is due to a rather primitive and crude way of fitting our tree construction mechanisms into HBC. It is based on hand instrumentation of the code, and the resulting code has a structure that incurs a significant overhead and gives the HBC compiler less scope for optimization. With better integration, we conjecture that most of this overhead could be removed, and that we then would be looking at execution times 2–4 times longer than normal for trees with 10000–20000 nodes.

A.5 Architectural Issues

So far, we have concentrated on the problem of building EDTs. A practical debugger must also be able to present the EDT to the user. It is not obvious how to give the user an informative view of the EDT: possible solutions range from presenting one node at a time to showing the whole EDT.

The presentation issue is separate from the problem of generating the EDT, so it is quite feasible to have separate solutions for them. We call the part of the debugger that builds an EDT the *EDT generator* and to the part that is responsible for presenting the EDT the *EDT navigator*. Between them there should be a pure, functional interface that just provides an EDT, hiding low-level details regarding re-execution, etc.

There are several advantages with this two level design, both from the user's point of view and from a software engineering perspective. Some of the advantages are:

- *Programmability*: By providing a pure, functional interface to the EDT generator, the EDT navigator may easily be written *in* the language for which the debugger is intended. Since the user is familiar with this language, he may easily extend and adapt the navigator to his needs.

- *Flexibility*: A single EDT generator could serve as a basis for several debuggers, e.g. an algorithmic debugger and one permitting unrestricted EDT-browsing. On the other hand, a single EDT navigator might be able to make use of different EDT generators optimized with

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

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

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

Figure A.7: Typical interface to the EDT generator.

respect to different parameters, such as speed or space requirements. We also gain implementation flexibility, e.g. the generator and navigator may run as separate processes, perhaps on separate machines.

- *Separation of concerns*: The construction of the EDT is rather involved and requires at least some support from the language environment, i.e. it cannot be done in a satisfying manner entirely within a lazy functional language. Thus it is beneficial to factor out the problem of EDT construction and consider it on its own.

We note that such a two level architecture is not uncommon for debuggers. One example is Ducassé's Prolog debugger Opium [Duc92]. There is also a lot in common between our reasons and Ducassé's for suggesting such an approach; in particular we share her view on the importance of providing an extensible and customizable debugger.

Part of a typical interface between the EDT generator and the EDT navigator, as it might look in Haskell, is shown in figure A.7. The abstract type `EDTNode` represents nodes in the EDT. The root of the EDT is bound to `root`. The various parts of a node may be accessed by means of a number of selector functions such as `function` or `children`.

`Function` is also an abstract type, and various attributes may be accessed by means of functions such as `funName` and `funSourceRef`. `SourceRef` is intended to be a reference to e.g. a function definition in the source code. This makes it possible to access relevant source code and present it to the user, which is an important aid in helping the user understanding the target program.

The `EDTValue` is more or less as described in section A.3.2. However, it is important to have an explicit representation of circular values. Consider sending `EDTValues` from one process to another using some textual protocol, for example. We can achieve this by means of special constructors for labelling values and for referring to such labels. For example, consider the circular definition:

```
xs = 1 : xs
```

The value denotes the infinite list of ones, but requires only a single cons-cell, because of sharing. The corresponding `EDTValue` would be:

```
EVLabel n (EVConstr c [EVInt 1, EVRef n])
```

where n is a unique label and c is the `Constructor` for cons `(:)`. Note that the transformational approach presented in section A.4.1 cannot detect circularities in values, so further low-level support would be needed.

It might also be desirable to pass information from the navigator to the generator, such as a list of trusted functions or a list of 'break-points'. This can be implemented by making `root` a function which takes such lists as arguments and returns the root of the specified tree.

A.6 Related Work

Hall & O'Donnell [HO85, OH88] were among the first who investigated the particular problems of lazy functional debugging. The focus is on on implementing debugging tools within an interactive, purely functional environment, implemented in the language itself, Daisy, a lazy descendant of Lisp.

One approach they suggest 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 structure of the trace reflects the structure of the source code. In spirit this is quite similar to our transformational EDT generation scheme, but there are also differences, one being that we work in a strongly typed language and that our transformations thus have to respect the type discipline. Another difference is the extra effort required in our case for converting values to printable form, in particular functional ones. This is presumably much less of a problem in Daisy, being Lisp-like, since all values have a uniform and well known representation.

One problem, which Hall & O'Donnell 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. We address this problem by introducing the impure function `evaluated` for supporting our transformational approach (see section A.4.2).

In another approach Hall & O'Donnell 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.

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 the computation history of the various pieces of graph on the heap is always maintained as annotations in the graph. Thus it is possible, at any time instant, to take a 'snapshot' of the state of the computation which reveals how it came to be. A snapshot would typically be taken when the program aborts due to a runtime error, or when the user interrupts it because it has entered an infinite loop. It is less obvious how one should get a good, revealing snapshot on a computation that terminates but produces the wrong result. In his thesis, Toyn [Toy87] proposes traditional bottom-up testing to deal with this case.

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 values as possible. The rule simply states that values should be shared, i.e. they should be represented by pointers to unique heap-allocated objects, 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. 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 he 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 that 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.

Kishon & Hudak [KH95] take a similar but more general and systematic approach. 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. As an 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. For getting acceptable performance, Kishon & Hudak use partial evaluation to get rid of as much interpretative overhead as possible.

Since monitoring semantics are 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.

The work by Naish & Barbour [NB95] is also closely related to ours. They use a source-to-source transformation, similar to ours, 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 distinction between their transformation and ours 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 readily be printed, and thus combines the functionality of our functions `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 the two approaches suggested in this article.

Thanks to `dirt` Naish's and Barbour's transformation is simplified with respect to ours since there is no need to handle functional values in a special way (see A.4.2). On the other hand, requiring a function like `dirt` makes their approach significantly less portable. For example, in the Chalmers Haskell implementation HBC, which is one of the major implementations, values in 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 and Barbour suggest building nodes down to a certain, predetermined, depth. (Then the normal, untransformed, versions of the functions can be called for increased performance.) 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. In this respect it is advantageous to work at a very low level, since it then is possible to take corrective measures if the memory consumption is getting out of hand, e.g. due to a reference from the tree to some 'infinite' data structure that grows too big as the execution proceeds.

Finally we would also like to mention the debugger for Standard ML of New Jersey (SML/NJ) [TA95], which to our knowledge is the only general purpose debugger that is available for a widespread functional (but *not* lazy) language. 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 types of functions remain unchanged. The debugger also features an advanced re-execution mechanism (called *time travel*), which for its implementation relies on the first-class continuations of SML/NJ.

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. Displaying functional values still seems to be a problem, however.

A.7 Conclusions

This report has proposed the *Evaluation Dependence Tree* (EDT) as a suitable basis on which to build debuggers for lazy functional languages. The reason for this is that the EDT shows how the performed computations depend on each other, while abstracting away operational concerns such as evaluation order. The EDT may thus be used to systematically search for the bug in a declarative way since it allows the user to focus on what was computed, rather than how things actually were computed.

We formally defined the EDT, presented two different ways of constructing EDTs, and compared their relative merits. The problem of large memory consumption was considered and a solution given. Some performance figures were also given which indicate that the approach is realistic.

We also briefly considered some architectural issues and proposed a two level debugger design, where the bottom level *EDT generator* takes care of the EDT construction and the top level *EDT navigator* is responsible for helping the user navigating through the EDT in search for the bug. Making a clear distinction between these, both conceptually and physically, results in a flexible and extensible debugger architecture.

Acknowledgements

The authors would like to thank Mireille Ducassé and the anonymous referees for many useful comments that substantially improved this report. We would also like to thank Rickard Westman and Graeme Moss who proof-read the report.

Paper B

Debugging Reactive Systems in Haskell

Amr Sabry
Department of Computer Science
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Jan Sparud
Department of Computer Science
University of York
Heslington, York Y01 5DD, UK
sparud@cs.york.ac.uk

Abstract

We support the use of Haskell in two classes of applications.

First, we note that although Haskell is well-suited for stream-oriented applications, like reactive systems, most implementations lack tools for locating subtle errors like deadlocks. To support the use of Haskell in such applications, we have designed and implemented a debugger with an intuitive graphical interface. The debugger simplifies the task of locating deadlocks by providing users with a mostly declarative view of programs as systems of recursive equations over streams.

Second, we investigate the use of Haskell in the implementation of the debugger. As expected, this effort requires a variant of Haskell, currently used in the intermediate phases of Haskell compilers, that combines call-by-need semantics with computational effects. To support robust programming in this system-oriented variant, we formalize the semantics of handleable exceptions in a call-by-need language.

B.1 Streams and deadlocks

As Kahn showed in 1974 [Kah74], any deterministic network of processes can be modeled using recursive equations over streams. Such networks occur in telephones, missiles, and many other embedded *reactive* programs. The stream model corresponds in spirit to the electrical engineer's concept of a signal-processing system [AS85] and thus provides a natural formalism for the design, specification, and verification of these networks [B^+92]. Additionally, stream-based specifications are directly executable in a variety of high-level languages. Examples of languages that elegantly support computations over streams include modern lazy functional languages such as Haskell [$HPJW^+92$], hardware-description languages such as MHDL [BD93] (an extension of Haskell), data-flow programming languages such as Lucid [AW77, AW85], and synchronous programming languages such as Esterel [BC85, BG92], Lustre [CHPP87], and Signal [BLJ91].

These languages have been remarkably successful in supporting the rapid development of reliable systems. First, streams (lazy lists) are the fundamental and most widely supported data structure in most of these languages. Second synchronization is dealt with automatically: programmers are only required to specify sets of recursive functions over streams without having to worry about many tedious and error-prone details. Indeed Caspi and Pouzet [CP96] report that some of the programs running on the Airbus A320 have been obtained in this way. Also Broy *et al.* [B^+92] use this methodology for the development of various distributed systems.

Despite its appeal, programming with streams is quite subtle. A common mistake consists of writing a system of recursive equations over streams whose fixed point is semantically undefined [Sij89]. Operationally this error may manifest itself in terms of an obscure error message from the compiler or an equally obscure error message from the run-time system. For example, when suspecting a deadlock, our liveness analysis [HPS96] returns a large (and incomprehensible) list of unsatisfied constraints. As another example, typical Haskell implementations, *e.g.*, the 'hbc' compiler from Chalmers University, detect and report deadlocks with the message "`Runtime error: Black hole detected in eval zap`," which gives about as much information as the infamous C error message "`Segmentation fault`."

In fact, one can usually locate the sources of segmentation faults using a C debugger but this is not even the case for black holes: users get little debugging support from typical Haskell implementations. Furthermore, conventional debugging techniques for imperative languages such as setting breakpoints, single-stepping, tracing, and watching variables are not particularly suited for locating deadlocks occurring in lazy functional languages [HHO90, OH88, NF94, Spa96] or synchronous languages [Edw]. Indeed our own personal experience in specifying reactive systems in Haskell quickly convinced us of the need for a special-purpose debugging technology specifically tailored towards computations over streams.

This paper reports on the design and implementation of such a debugger in the context of the Haskell compiler 'hbc.' The debugger has an intuitive graphical user interface, and can be used to quickly pinpoint the sources of many deadlocks. Experience with the implementation indicates that the system works remarkably well for small but realistic programs. The debugger nicely complements our formal liveness analysis [HPS96] providing users with a useful set of tools for developing reactive programs.

Like many system-level Haskell programs, the implementation of the debugger relies on expressions with side-effects. Since Haskell bans such expressions, programmers that need this functionality either write low-level routines in C or in an *ad hoc* variant of Haskell that integrates computational effects with the call-by-need semantics. Recently Launchbury and Peyton Jones [LP95] convincingly argued for the second approach:

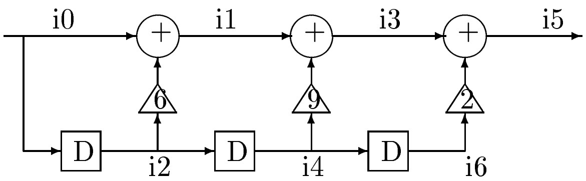
> "Should we outlaw `interleaveST` on the grounds that it is insufficiently well behaved? Not necessarily. Outlawing `interleaveST` would simply drive its functionality underground rather than prevent it happening. For example, we want to have lazy file reading. If it cannot be implemented in Haskell then it will have to be implemented "underground" as a primitive operation written in C or machine code. The same goes for unique-supply trees and lazy arrays.
>
> *Implementing such operations in C does not make them more hygienic or easy to reason about.* On the contrary, it is much easier to understand, modify and construct variants of them if they are implemented in a Haskell library module than if they are embedded in the runtime system"[LP95, p.45].

We therefore study a variant of Haskell with call-by-need semantics and handleable exceptions. This variant of Haskell is reminiscent of the informal Haskell dialect used by some programmers, at their own risk, to implement generation of new names [ARS94], memoization [Hug85], non-deterministic choice [HM95] and other paradigms that are *not* expressible in a modular way in a purely functional language. It is also reminiscent of the intermediate language of Haskell compilers like the Glasgow Haskell compiler, that implement monadic state by updates on a global shared store.

The next section motivates the use of streams in programming reactive systems. Section 3 discusses conventional debugging techniques and explains why they are unsuitable for our purpose. Section 4 presents a realistic example that uses Haskell to write the controller for a small industrial production cell. The example deadlocks due to a subtle bug that we unintentionally introduced in our first attempt at writing the code. We illustrate the user's interface of the debugger and show how a simple debugging session fixes the problem. Section 5 provides the details of the implementation of the debugger, and reveals that the implementation requires handleable exceptions. Section 6 presents the semantics of such exceptions in a call-by-need language. Finally Section B.7 puts our work in perspective by explaining its weaknesses and charting ideas for future development.

B.2 Programming reactive systems

Reactive systems are programs that must *continuously* react to external stimuli from their surrounding environment [Har87]. For example, a digital filter is usually specified as a weighted sum of the n last samples of the input signal. The following diagram depicts such a filter:



The block D stands for a delay element. The triangles represent elements that scale every value in the stream by the corresponding number.

It is relatively straightforward to transliterate the filter to working Haskell code. The first step consists in implementing the intended semantics of each block:

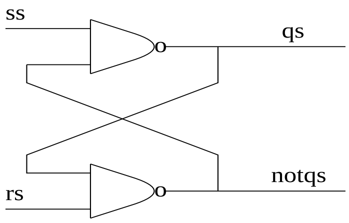
```
sum (x : xs) (y : ys) = (x + y) : sum xs ys
delay x = tail x
weight w = map (w*)
```

Having specified the blocks, it remains to write a set of mutually recursive equations over streams that reflects the interconnections of the network:

```
filter i0 = i5
where i1 = sum i0 (weight 6 i2)
      i2 = delay i0
      i3 = sum i1 (weight 9 i4)
      i4 = delay i2
      i5 = sum i3 (weight 2 i6)
      i6 = delay i4
```

Note that the generation of this last piece of code can be easily automated given the diagram: it only depends on the interconnections between the components, not on their semantics.

The above development process is simple, natural, and elegant. But one must not get carried away; there are some limitations! Consider the following diagram specifying a basic flip-flop circuit with `nand` gates and the program produced by naïvely following our development process:




```
nand (1 : xs) (1 : ys) = 0 : nand xs ys
nand (_ : xs) (_ : ys) = 1 : nand xs ys

ff rs ss = (qs, notqs)
 where qs = nand ss notqs
       notqs = nand rs qs
```

Although the physical flip-flop device works, our program doesn't. And in fact pure Haskell appears ill-suited for such asynchronous circuits.

However, despite its limitations the stream model is suitable for many applications and is at the core of more expressive formalisms that accommodate asynchronous events, real-time, non-determinism, etc.

B.3 Related work

To understand the contribution of our work, it is necessary to review the available debugging tools for Haskell in general and stream computations in particular. But first it is worth noting that most implementations of synchronous languages include debuggers and simulation environments in which users can experiment with the code. The widespread use of such tools reinforces our claim that they are essential in a programming environment that supports computations over streams.

The lazy (demand-driven) evaluation of functional languages schedules computations depending on the demands for their results. This evaluation mechanism is hard to predict and to relate to the source code of the program, and hence is not well suited for conventional debuggers that operate at a low level of abstraction that is closer to the machine code than to the source program. This led researchers to propose *declarative debugging* as a viable strategy for debugging lazy functional programs [NB95, Nil94, NF94].

One promising declarative debugging technique is *algorithmic debugging* [SN97, Spa96, NS95]. Algorithmic debugging is a two phase process. First, the program is run and an execution dependence tree is built. Then, this tree is interactively traversed. Each node in the tree corresponds to a function call and its result. While traversing the tree (in a top-down manner), the system asks, for each node, if the result of the corresponding function call is correct or not. Based on the answers the programmer gives, the system chooses which function call to show next. A bug has been found when the result of a function call is erroneous and all children nodes are correct. The method requires that the functions have no side effects, which makes it suitable for pure functional languages.

The ability to view the value of any expression in a program execution, and the expressions on which that value depends, is generally useful for debugging many lazy functional programs. Unfortunately, it is not of much help when debugging stream computations. To understand a computation that deadlocks while manipulating several streams, it is crucial to:

- understand the sequence of *events* that led to the deadlock, and
- have a view of *all* the relevant streams at the point in which the deadlock occurred.

Algorithmic debugging abstracts away from time and does not faithfully report the evaluation steps in the order in which they actually happened. Furthermore, users of an algorithmic debugging tool never have a global view of the computation in which they can keep track of several streams at once. These properties make algorithmic debugging unsuitable for locating deadlocks in stream computations.

B.4 Controlling an elevating table

We present a simplified version of the actual program fragment that spurred the idea of the new debugger.

B.4.1 The program

Figure B.1 shows a small fragment of a case-study production cell [LL95]. This fragment is concerned with the implementation of the controller of a simple elevating table and testing it with a simulated hardware environment. The table is initially in a low position. When the controller detects an input blank, it issues an `Up` command to the motor interface. This abstract command is implemented by the interface using a low-level protocol that reads sensors and emits control signals (actuators) to a motor. The block `Env` simulates the actual environment. Assuming the environment operates correctly, the controller eventually receives an acknowledgment, delivers the blank to the next device (a robot arm), and issues a `Down` command to lower the table to its initial position. The `Down` command is implemented by the interface using a protocol similar to the one implementing `Up`; when the `Down` command is acknowledged the controller is ready to accept more input blanks.

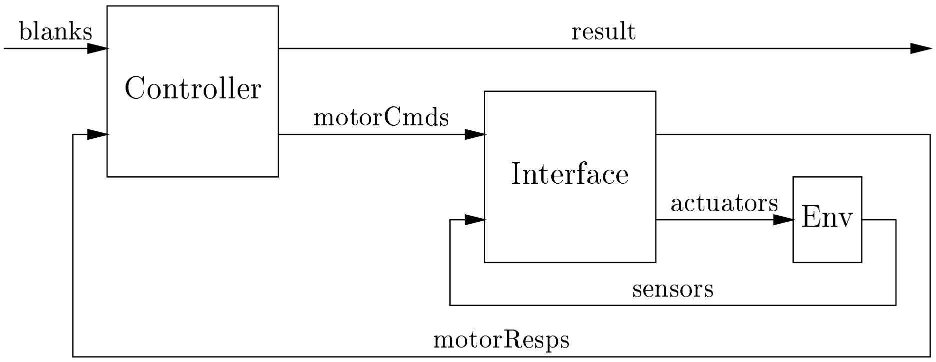


Figure B.1: Elevating Table

We will give the precise implementation of the protocols implementing the commands `Up` and `Down`. First we formally describe the sets of messages that are required by the specification of the table [LL95]:

```
data MotorCmd  = Up | Down
data Sensor = Low Bool | High Bool
type SensorVector = (Sensor,Sensor)
data Actuator = MotorUp | MotorDown
              | MotorStop | GetStatus
```

The specification of the system requires the existence of two sensors that identify the current location of the table. For example, when the table is in the low position, the sensors are `(Low True, High False)`. The controller can only access the current values of the sensors by issuing a `GetStatus` control signal; changes to the sensors between `GetStatus` signals are undetected by the controller. Finally, the motor that moves the table can be controlled using three self-explanatory control signals: `MotorUp`, `MotorDown`, and `MotorStop`.

The above description yields the following code for an idealized environment:

```
env acs = loop acs (Low True, High False) where
 loop (ac:acs) currentSensors =
   case ac of
    GetStatus  -> currentSensors :
                  loop acs currentSensors
    MotorUp    -> loop acs (Low False, High True)
    MotorDown  -> loop acs (Low True, High False)
    MotorStop  -> loop acs currentSensors
```

The environment code assumes that the table is initially in the low position and repeatedly looks at the current control signal in the actuators stream. If the signal is `GetStatus`, then the current values of the sensors are returned. Otherwise the appropriate action is performed, possibly modifying the current values of the sensors.

It remains to perform the calculation of the `actuators` stream in the module `Interface`:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
```

```
    MotorUp : GetStatus : goU mcs ss

  goD mcs ((Low True, High _) : ss) =
    MotorStop : emit mcs ss
  goD mcs ((Low _, High _) : ss) =
    MotorDown : GetStatus : goD mcs ss
```

First the code issues a `GetStatus` signal to get the initial values of the sensors. Then we start a loop that examines the current abstract command and the current values of the sensors and emits the appropriate actuators. In a realistic environment, the actuators would not be immediately acknowledged and the controller might have to loop several times until the sensors confirm that the actuator signal has been acknowledged.

The main program reflects the interconnections in Figure B.1. To test the program fragment independently, let's assume that the stream of `motorCmds` consists of an infinite sequence of alternating `Up` and `Down` commands:

```
motorCmds = Up : Down : motorCmds
actuators = interface motorCmds sensors
sensors = env actuators
```

Note how the definitions of `sensors` and `actuators` are mutually recursive.

The current code is overly simplified because it ignores the remainder of the production cell, and even ignores some aspects of the elevating table like the fact that it should rotate left and right while moving up and down. Nevertheless, the code captures the essence of our initial implementation of the elevating table including the fact that it contains a subtle error causing the system to deadlock! Because of the simplifications, the bug may be obvious at this time, but it certainly wasn't obvious in the complete program (around 600 lines of Haskell).

B.4.2 Finding the deadlock

Our debugger operates at a high-level of abstraction that reflects the source program itself. Instead of getting lost in details about lazy evaluation, our debugging technique maintains the illusion of a mostly declarative computation expressed as a fixed point of recursive equations over streams.

We start by illustrating the user's point of view by going through a possible debugging session for the elevating table code. To find the source of the deadlock the user must identify the relevant streams of interest. This

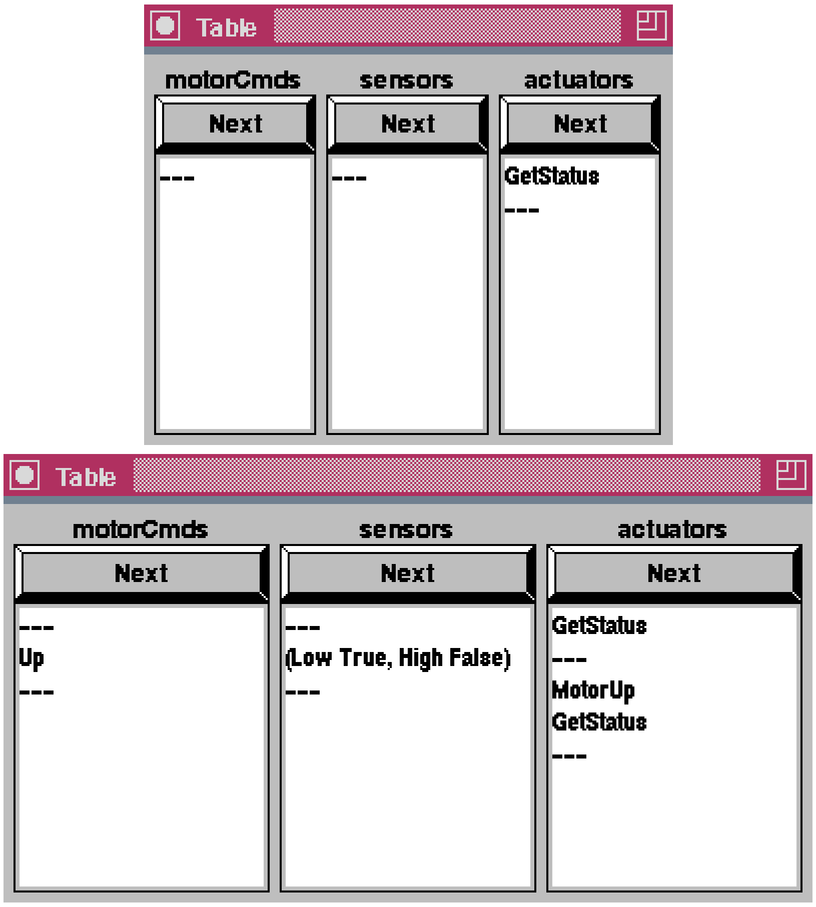


Figure B.2: A Sample Debugging Session

information is used by the debugger to keep track of, and display the sequences of values that belong to these streams. In our case, we only have three streams and we state that we are interested in all of them.

When running the program under the control of the debugger, the window in Figure B.2 appears (initially empty). The user then can interact with the computation by clicking on the buttons `Next`. Each click returns the next value on the corresponding stream as well as all values on the other streams on which it depends. In our example, the first click on the stream `actuators` returned the value `GetStatus` without demanding any

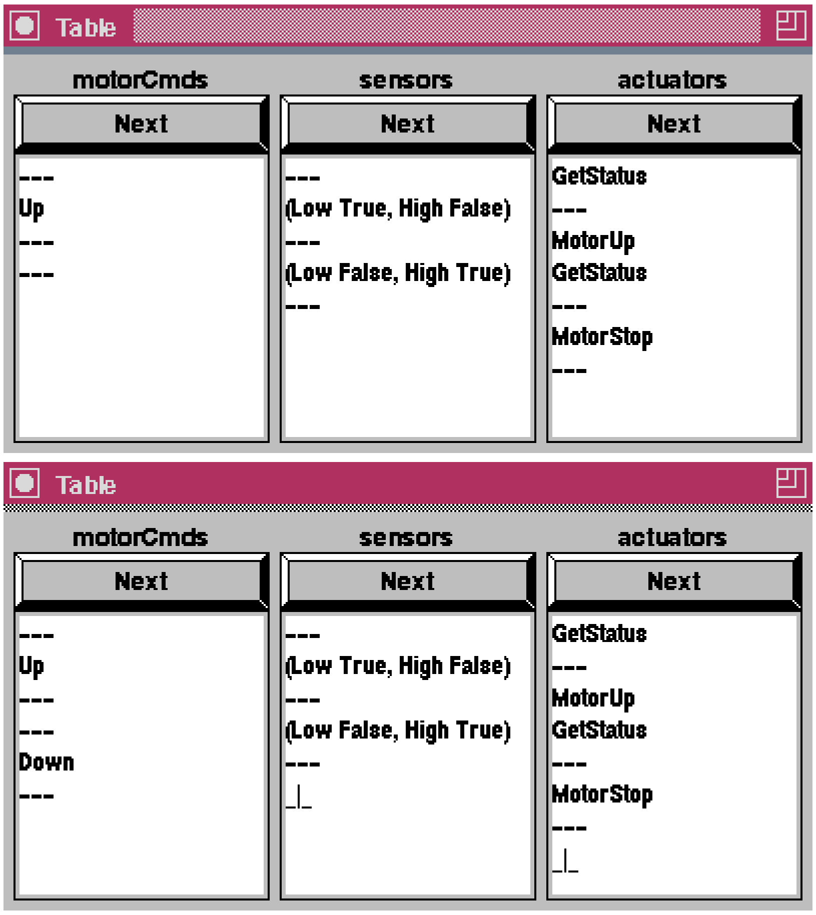


Figure B.3: A Sample Debugging Session, continued.

other values on the other streams. A second click (see Figure B.3) on the same stream returned the values `MotorUp` and `GetStatus`. These two values can only be produced if a command `Up` is issued on the stream `motorCmds` and the sensors indicate that the table is in the low position. After the next click the computation has proceeded to the point where the table is in the high position and the motor is ordered to stop. When attempting to process the `Down` command, the next click displays the symbol $\perp$ in the columns for the streams of sensors and actuators indicating a cyclic dependency between the next two values, *i.e.*, a deadlock.

Even before examining the code for bugs, we can immediately note one important fact: each `GetStatus` signal on the stream of actuators produces the current values of the sensors on the next click. The first two `GetStatus` signals have been processed and acknowledged and no values can appear on the stream of sensors unless a new `GetStatus` signal is issued. It is also clear that the deadlock occurs at a point in the neighborhood of the code that issues the `MotorStop` signal and the error manifests itself immediately after consuming the command `Down`. Looking back at the relevant piece of code we find:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

(*)     goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
          MotorUp : GetStatus : goU mcs ss

        goD mcs ((Low True, High _) : ss) =
          MotorStop : emit mcs ss
        goD mcs ((Low _, High _) : ss) =
          MotorDown : GetStatus : goD mcs ss
```

In the line marked `(*)` we see that after issuing the signal `MotorStop`, we proceed to the `emit` loop which consumes a `Down` command, and then attempts to access the stream of sensors without any intervening `GetStatus` signals. Thus one way to fix the code would be to simply add a `GetStatus` signal after stopping the motor (in both lines that stop the motor):

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _,High True):ss) =
          MotorStop:GetStatus:emit mcs ss
        goU mcs ((Low _,High _):ss) =
          MotorUp:GetStatus:goU mcs ss

        goD mcs ((Low True,High _):ss) =
```

```
    MotorStop:GetStatus:emit mcs ss
  goD mcs ((Low _,High _):ss) =
    MotorDown:GetStatus:goD mcs ss
```

And indeed the code now works just fine. A proof that the new code does not produce deadlocks is outside the scope of this paper but could be achieved either by compiling the code to a finite state machine or subjecting it to our liveness analysis [HPS96].

B.5 Implementing the debugger

Having argued that the debugger would be useful to include in a programming environment for Haskell, we turn our attention to the details of our current implementation in the context of the Chalmers Haskell compiler 'hbc.'

B.5.1 Background

The fundamental philosophy behind lazy evaluation [HM76] is that expressions are not evaluated unless their values are demanded. The initial demand usually starts from the top level routine that seeks to print the answer of the program. During debugging, the initial demand starts when the user clicks on one of the streams. In both cases the demands are propagated until they reach an expression that is already evaluated. Such an expression is said to be in *weak head normal form* (whnf). Examples of expressions in whnf include integers, λ-expressions, and pairs. Generally speaking an expression that consists of a constructor applied to arbitrary (possibly unevaluated) argument expressions is in whnf. For example, the expression `(1+2 : f xs)` is in whnf, since it consists of a top level constructor `(:)`, an unevaluated head `1+2`, and an unevaluated tail `f xs`.

Lazy evaluation is viable because once an expression has been evaluated, there is no need to re-evaluate it again. Since the language is purely functional, all subsequent evaluations of an expression are guaranteed to return the same result as the first, and hence are useless. This property immediately implies that it is impossible to write a Haskell expression that evaluates to `True` the first time, and to `False` the second time.

A deadlock occurs in a lazy functional program when the evaluation of an expression needs its own value to proceed. An obvious way of detecting deadlocks is thus to mark an expression as being under evaluation when its evaluation starts. If the value of such a marked expression is needed, we have a deadlock. Most compilers for lazy functional languages mark expressions in this way and can indeed detect deadlocks at run-time albeit with an obscure error message about black holes.

B.5.2 Global view

The debugger consists of a number of stream agents and a top level coordinator. A stream agent is responsible for evaluating and displaying the elements of one stream. The graphical representation of each stream agent is as shown in Figures B.2 and B.2.

The top level coordinator and the stream agents interact as follows. When the button `Next` of a stream agent is clicked, the agent tries to evaluate the next element of the stream. It then sends a notification message to the coordinator. Upon receipt of such a notification, the coordinator sends a broadcast message to all stream agents, to which each agent responds by checking for, and displaying, the evaluated elements on its stream.

B.5.3 Peeking at streams

To perform their jobs, the stream agents need to peek at streams to see how many elements are already evaluated to whnf. Unfortunately, in a lazy functional language it is not possible to look at a stream to see how many elements have been produced. In fact, it is by looking at them that they will be produced, since there is demand for them! So when inspecting a stream, the agent would want to see the stream as a read-only data structure.

For this reason we have implemented a new primitive function `peek` that takes an argument and returns a value of type EvalStatus:

```
data EvalStatus = Evaluating | Closure | Evaluated
peek :: a -> EvalStatus
```

Clearly the above function is not pure and hence is not expressible in Haskell. Indeed, it may return different answers when called with the same argument twice. This function is not available to programmers.

Using `peek` we define a function `safeInitialList` that takes a stream as argument, and returns the initial list of already evaluated elements in the stream, a flag saying whether the stream has become deadlocked or not, and the stream itself (except for the initial evaluated segment):

```
safeInitialList :: [a] -> ([a], Bool, [a])
```

```
safeInitialList xs =
 case peek xs of
   Evaluating -> ([], True, xs)
   Closure    -> ([], False, xs)
   Evaluated  ->
    case xs of
     y:ys -> let (zs, b, ws) = safeInitialList ys
             in (y:zs, b, ws)
     []   -> ([], False, [])
```

B.5.4 Recovering from fatal errors

A fatal error is one which normally terminates a program with an error message, *e.g.*, a division by zero or a deadlock. Since our debugger is part of the same program as the application, we certainly don't want such errors to abort the debugger. To be able to recover from such faults and get some clues on which part of the source code is responsible for the fault, we introduce a pair of functions `catch` and `throw`:

```
throw :: String -> a
catch :: a -> (String -> a) -> a
```

that provide a simple form of exception handling. The semantics of these extensions is presented in the next section.

Intuitively, they are used as follows. The compiler is modified so that all fatal errors call `throw` with an appropriate error message. These errors are caught (handled) by the debugger. For example, the evaluation of `result` below produces 42:

```
result = catch (f []) handler
f (x:xs) = x + f xs
handler errMsg = 42
```

Otherwise, when not debugging, there will be no active `catch` and the program would terminate with the usual error message: `"No match in f"`.

B.6 Call-by-need & exceptions

Consider a small functional *call-by-need* language extended with exceptions.

Definition B.6.1 (Syntax) *The set of terms includes:*

$$\begin{array}{rcl} M,N & ::= & V \mid M+N \mid x \mid MN \\ & & \mid \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\ V & ::= & n \mid \lambda x.M \end{array}$$

In other words, the language extends the pure λ-calculus with a representative set of constants including numbers and addition as well as two expressions for throwing and handling exceptions. The intended semantics of `throw` and `catch` is the usual one [Ste84] adapted to a call-by-need language. Given an expression M, the evaluation of ($\texttt{throw}\ M$) aborts the computation and returns M. The evaluation of ($\texttt{catch}\ M\ N$) where N is a handler proceeds as follows. First evaluate M to whnf; if this evaluation terminates normally then its result is the value of the entire `catch` expression. Otherwise if the evaluation of M throws an exception, the handler is called with the thrown expression as an argument.

Definition B.6.2 (Syntax of Types) *The set of types is inductively defined as follows:*

$$\tau \quad ::= \quad \texttt{Int} \mid \tau \rightarrow \tau'$$

The typing rules for our language are in Figure B.4. We let Γ scope over type environments (partial mappings from variables to types). A type judgment $\Gamma \vdash e : \tau$ means that under the assumptions in the type environment Γ, expression e has type τ.

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x : \tau} \qquad \frac{\Gamma \cup \{x:\tau\} \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \rightarrow \tau'} \qquad \frac{\Gamma \vdash M : \tau' \rightarrow \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash M\ N : \tau}$$

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \qquad \frac{\Gamma \vdash M : \texttt{Int} \qquad \Gamma \vdash N : \texttt{Int}}{\Gamma \vdash M + N : \texttt{Int}}$$

$$\frac{\Gamma \vdash M : \texttt{Int}}{\Gamma \vdash \texttt{throw}\ M : \tau} \qquad \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \texttt{Int} \rightarrow \tau}{\Gamma \vdash \texttt{catch}\ M\ N : \tau}$$

Figure B.4: Typing Rules

The typing rules for the functional core are standard. The typing rules for `throw` and `catch` are similar to the ones found in SML [WF91] but, for simplicity, we have restricted all thrown expressions to be of type `Int`.

If typechecking guarantees anything, it is that some obviously faulty terms are not typable.

Definition B.6.3 (Faulty Terms) *An expression is* faulty *if it is contains one of the following subterms:*

- $n\ M$, *or*
- $(\lambda x.M) + V$, *or*
- $V + (\lambda x.M)$.

It is easy to confirm that faulty terms are not typable.

B.6.1 Evaluation contexts

Because of the *lazy* semantics of the language, the definition of the axioms is intertwined with the definition of evaluation contexts [AF97, AFM+95]. Intuitively, "needed" variables within subterms correspond to those variables that occur in evaluation context positions. Therefore, we define evaluation contexts first before defining the semantics of the language.

Definition B.6.4 (Evaluation Contexts E) *The set is inductively defined as:*

$$\begin{array}{rcl} E & ::= & [\,] \mid EM \mid (\lambda x.E)\ M \mid (\lambda x.E[x])\ E \\ & & \mid E + M \mid n + E \mid \mathtt{catch}\ E\ M \end{array}$$

The first four clauses are identical to the ones for the pure call-by-need calculus. They have the following intuitive explanation. The context EM reflects the fact that the function position of an application is needed. The next two contexts make the call-by-need nature of the language evident: arguments to functions are not evaluated when the function is called. Indeed, we first attempt to evaluate the body of the function without touching the argument. Only if the body of the function "needs" the value of the parameter, do we start processing the argument.

The contexts $E + M$ and $n + E$ state that both arguments to an addition are needed: first the left one, then the right one. In the presence of exceptions and handlers, this order is clearly significant. Finally the context `catch` $E\ M$ reflect our informal understanding of the semantics of a `catch` expression which first attempts to evaluate the first argument. The second argument (the handler) is only needed if the evaluation of the first argument throws an exception.

B.6.2 Axioms

We are almost in a position to define the semantics of our language. We begin by identifying a syntactic category of terms that are *answers* and how to observe them.

Definition B.6.5 (Answers) *An answer A is one of the following terms:*

$$A \ ::= \ V \mid \texttt{throw}\ M \mid (\lambda x.A)\ M$$

When observed, answers yield the following output to users:

$$\begin{array}{rcl} obs(n) & = & n \\ obs(\lambda x.M) & = & \text{"Procedure"} \\ obs(\texttt{throw}\ M) & = & \text{"Uncaught Exception"} \\ obs((\lambda x.A)\ M) & = & obs(A) \end{array}$$

Our definition of answers extends the usual definition for the pure call-by-need calculus by adding an additional kind of answers for uncaught exceptions.

Abbreviations for one-step contexts:

$$\begin{array}{rcl} G & ::= & H \mid \texttt{catch}\ [\,]\ M \\ H & ::= & ([\,]\ M) \mid [\,] + M \mid n + [\,] \mid (\lambda x.E[x])\ [\,] \end{array}$$

Axioms:

$$\begin{array}{rcll} (\lambda x.E[x])\ V & = & (\lambda x.E[V])\ V & \\ G[(\lambda x.A)\ M] & = & (\lambda x.G[A])\ M & \\ n_1 + n_2 & = & n & \text{where } n = n_1 + n_2 \\ \texttt{catch}\ V\ M & = & V & \\ H[\texttt{throw}\ M] & = & \texttt{throw}\ M & \\ \texttt{catch}\ (\texttt{throw}\ M)\ N & = & NM & \end{array}$$

Figure B.5: Axioms

We can now specify the semantics of the language via the set of axioms in Figure B.5. The first three axioms are essentially the axioms of the pure call-by-need calculus. The next three axioms specify the semantics of exceptions. A `throw` expression aborts any context other than a `catch`. The value of a `catch` expression is either the value of the first argument or the result of applying the handler to the value thrown by the first argument.

The following proposition guarantees that the definitions of evaluation contexts, faulty expressions, and axioms are consistent and complete.

Proposition B.6.1 *Every closed term is either an answer, faulty, or can be uniquely decomposed into an evaluation context and a redex.*

B.6.3 Evaluator

The above axioms can be used in any context to transform programs (perhaps for optimization purposes). They can also be used to specify an evaluator by orienting them from left to right and always choosing the redex that occurs in the hole of the evaluation context.

Definition B.6.6 *The call-by-need evaluator is defined as follows:* $eval(M) = B$ *if* $M \longmapsto^* A$ *and* $obs(A) = B$*, where* $M \longmapsto N$ *if and only if:*

- $M = E[R]$,
- $R = R'$ *is an axiom of Figure B.5 (*R *must be the left hand side), and*
- $N = E[R']$.

B.6.4 Connection to call-by-name

Clearly the semantics of a call-by-need with exceptions is *not* the same as the semantics of the call-by-name variant. This is in contrast to the pure functional subset in which call-by-name reasoning principles like β are sound. However this is no worse than the current intermediate languages of Haskell compilers, like the Glasgow Haskell compiler, that use a call-by-need language with assignments to implement monadic state efficiently. Indeed β is also unsound in the intermediate language of the compiler for precisely the same reason [LS97].

B.7 Current limitations and future work

We have presented a novel debugger for reasoning about the recursive computations over streams usually found in reactive systems. The debugger appears to work remarkably well for small programs, but there is a need for further work to handle larger programs and to formalize the theoretical foundations of the current implementation or to investigate different implementation strategies. We expand on these points in the remainder of the section.

There are three main limitations of the current system. First, the number of streams that can be simultaneously debugged is limited by the size of the screen. In practice we have not found this to be a problem. In large examples, it is relatively easy to use the debugger to narrow down the deadlock to a handful of streams.

Second, our implementation relies on impure primitives that, although confined to the internals of the compiler, could pollute the source language and break some obvious program optimizations. For example, the following two definitions of `ones` are equivalent in pure Haskell but not equivalent during debugging sessions:

```
ones = map id (1 : ones)
ones = 1 : ones
```

To see the difference consider the call `(safeInitialList ones)`. Using the first definition of `ones`, the result is a triple of the form `([], False, ones)`. Using the second definition of `ones`, the call (and hence the debugger itself) diverges! To avoid such situations, a fully evaluated stream cannot be used during debugging sessions. Indeed in the elevating table example, the definition that we used for `motorCmds` is actually:

```
motorCmds = map id (Up : Down : motorCmds)
```

This trick of inserting `map id` around trivial stream definitions works fine with the current version of the 'hbc' compiler but is unsatisfactory. A smart compiler that is unaware of the debugger can certainly optimize `(map id e)` to `e` since they are equivalent in pure Haskell. A better solution would be to integrate the debugger and the compiler in a programming environment which would require them to agree on a common representation for streams such as `motorCmds` above.

As another example, addition is commutative in pure Haskell, *i.e.*, $M + N = N + M$. This equivalence no longer holds in the presence of `catch` and `throw`. The evaluation of `result` below produces 5 or 6 depending on the order of evaluation of the arguments to `+`:

```
result = catch (f ()) handler
f () = throw "L" + throw "R"
handler errMsg = if errMsg=="L" then 5 else 6
```

General programmers do not have access to `catch` and `throw` and this situation does not seem to occur in the current debugger. Nevertheless a proof of safety would be interesting. This situation is reminiscent of the use of assignments in the implementation of monadic state without affecting the purity of the source language [LP95, LS97].

Third, the debugger can only cope with streams defined at the top level of the program. Thus if we have a large program in which one function uses locally defined streams, we cannot debug the program until these streams are globally defined at the top level. Such a change may require a major restructuring of the program, and may even be impossible in certain circumstances. This problem is a general Haskell problem: it is impossible to perform any computational effect (*e.g.*, printing) as a side-effect of evaluating an expression. Instead computational effects must be propagated to the top level either explicitly or by using a monad [Wad90]. This is the major limitation of the debugger; we are actively seeking a solution to this problem.

Finally, we have started an investigation of the semantics of computational effects in call-by-needs languages since we believe that this will lead to more robust programs, and will formalize some of the *ad hoc*, informal, and potentially unsafe programming practices in system-level Haskell programs.

Part III

Redex Trails

Paper C

Complete and Partial Redex Trails of Functional Computations

Jan Sparud and Colin Runciman
Department of Computer Science, University of York
Heslington, York, YO1 5DD, UK
email: {sparud,colin}@cs.york.ac.uk

Abstract

We describe the design and implementation of a system for tracing computations in a lazy functional language. The basis of our tracing method is a program transformation carried out by the compiler: transformed programs compute the same values as the original, but embedded in functional data structures that also include *redex trails* showing how the values were obtained. A special-purpose display program enables detailed but selective exploration of the redex trails, with cross-links to the source program.

Keywords: debugging, graph reduction, Haskell, program transformation.

C.1 Introduction

C.1.1 Why trace functional computations?

Functional programming languages have many advantages over the conventional alternative of procedural languages. For example, program construction is more rapid, more modular and less error-prone. Programs themselves are more concise.

Yet functional programming systems are not very widely used. There are various reasons for this, but one that crops up time and again is the *lack of tracing facilities*. Yes, there is less scope for making mistakes in a functional language; but programmers do still make them! And when their programs go wrong they need to trace the cause. Unfortunately, implementors of functional languages are hard-pressed to provide equivalents of the 'debugging tools' routinely used to investigate faults in procedural programs. Tracing evaluation by normal order graph reduction is more subtle than following a sequence of commands already explicit at source level.

There have been various attempts to tackle the problem of tracing lazy functional programs. We discuss some of them in §C.6. But so far as we know there is as yet no really effective solution — a state of affairs we'd like to change.

C.1.2 How? Some design goals and assumptions

Functional language

We concentrate on the tracing problem for purely functional languages such as Haskell. Despite the absence of side-effects, lazy evaluation and higher-order functions in languages like Haskell make the problem difficult: there is a big gap between high-level declarative programs and the low-level sequences of events in their computations.

Graph reduction

We assume an implementation based on graph-reduction. In essence, the objective of computation is to evaluate an expression represented by a graph. This is achieved by repeatedly replacing one subgraph by another, where the reduction rules used to define replacements are derived from the equations given in the program. At each reduction step, a *redex* matching the left hand side of an equation is replaced by a the corresponding instance of the right hand side. Computation by graph reduction is made efficient by compilation to code for a G-machine, or similar.

Backward traces

We need to provide backward traces from results or from run-time errors, because the most pressing need for traces arises in the context of an unexpected output or failure.

Redex trails

We use the idea of a *redex trail* to provide the overall framework for answering the question 'How has this value/failure come about?'. At each reduction, parts of the redex no longer attached to the main graph are normally discarded. If we instead make a link from each newly created node of the graph to its *parent redex*, the computation builds its own trail as reduction proceeds.

Non-invasive traces

The transformation to introduce redex trails should not change the course of the underlying computation in any way. For example, unevaluated expressions should remain unevaluated.

Complete traces

Until we have a very strong reason to discard parts of the information in redex trails, and a clear argument which parts should go, we want to construct traces in full. There must be a representation of every reduction step for definitions and expressions of every kind.

Selective display

A full trace of even a modest computation contains a great deal of information — too much for the programmer to absorb in its entirety. Programmers need fine control over what trace information is actually displayed to them, down to the level of interactive link-by-link examination of the trails leading from a run-time fault or selected fragment of output.

Traces linked to source

However good the tracing system, source programs are likely to remain the primary reference for programmers. Not only should expressions in traces be displayed just as they might be written in a source program; trace text should be also be linked directly to source text.

A portable implementation

Although a prototype tracer must have some specific host implementation (ours will be the Haskell compiler `nhc` [Röj95]), we aim to produce a portable tracing scheme that could be adopted in other implementations of a functional language. For this reason, we shall prefer to use source-to-source transformation rather than to modify the run-time system.

An efficient implementation

Finally, the implementation must be efficient — or at least, not so inefficient that using the tracer is infeasible or unattractive. If execution slows by no more than a factor of ten, we can hope that the value of tracing information will make the speed tolerable; but factors of a hundred or a thousand are unlikely to be acceptable. Also, memory requirements must be such that the tracer can be used on an ordinary workstation.

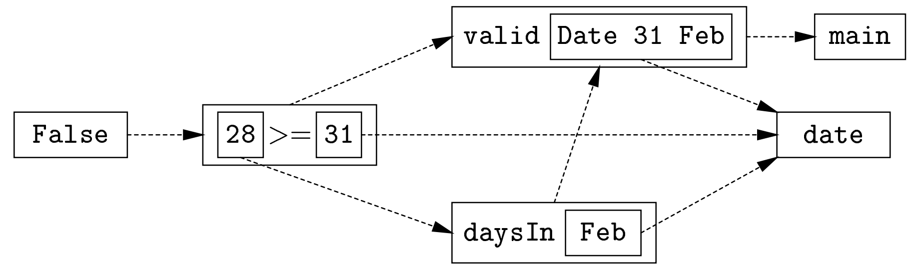
C.1.3 Details that follow

§C.2 explains what we mean by a redex trail, and shows how trails can be represented as functional data structures. §C.3 gives rules for transforming a program so that all values it computes are wrapped in a construction including a redex trail; it also explains briefly how we implement these rules in the nhc compiler. §C.4 explains the design of the interactive display program we use to examine redex trails. §C.5 evaluates aspects of the trace system we have built, including some performance figures. §C.6 discusses related work on other tracing systems for functional programs. §C.7 concludes and also gives some of our plans for future work.

C.2 The design of a trace structure

To appreciate what we mean by a redex trail consider the following example of a simple program testing the validity of a date.

```
daysIn Jan = 31
daysIn Feb = 28
     ...
daysIn Dec = 31
valid (Date d m) = daysIn m >= d
date = Date 31 Feb
main = valid date
```



The diagram shows the complete redex trail from the value `False` of the main expression. The immediate parent redex (whose reduction caused the creation of this instance of `False`) was `28 >= 31`. The parent redex of this application of `>=` was `valid (Date 31 Feb)`, but the parent redex of the left operand `28` was `daysIn Feb` and that of the right operand `31` was `date`. And so on. There is a link from every part of every expression to its parent redex.

The idea is to construct such a trace giving comprehensive information about a functional computation. The trace should be built as the computation proceeds. Once the computation is over the trace is just a static data structure.

Such redex trails can be represented as functional data structures of type:

```
data Trace = Ap Trace [Trace] | Nm Trace Name | Root
```

An application is represented by `Ap` t_{ap} `[`t_f`,` t_{x_1}`,` $\ldots$`,` t_{x_n}`]`, where t_{ap} is the trace for the application itself, t_f is the trace of the function part of the application, and $t_{x_1} \ldots t_{x_n}$ are the traces of the arguments. A name (a function, variable, constructor or literal) is represented by `Nm` t_{nm} *name* where t_{nm} is the trace of the name and *name* is a textual representation of the name. The application and name nodes should also have *source code references*, but since we are only interested in the structure of the trace at this stage, we have omitted them here. `Root` is the null trace attached to top-level names.

C.3 Creating traces

In this section we define transformations to derive a self-tracing version of any given program. We will introduce a number of combinators that actually perform the trace creation, but first we will describe how the *types* of a program will change under transformation.

C.3.1 Type transformation rules

Every value in the program will be wrapped in the `R` datatype, defined as:

```
data R a = R a Trace
```

The intuition is that every value in the original program should have a wrapped value in the transformed program containing the original value and a trace for that value.

Figure C.1 gives the rules determining how types are transformed. Motivation and examples follow.

$$\mathcal{R}[\![t]\!] \Rightarrow \mathtt{R}\ (\mathcal{T}[\![t]\!])$$

$$\mathcal{T}[\![tcon\ t_1\ \ldots t_n]\!] \Rightarrow tcon\ \mathcal{T}[\![t_1]\!]\ \ldots\ \mathcal{T}[\![t_n]\!] \quad (n \geq 0,\ tcon \neq \text{'}{\to}\text{'})$$
$$\mathcal{T}[\![t_1 \to t_2]\!] \Rightarrow \mathtt{Trace} \to \mathcal{R}[\![t_1]\!] \to \mathcal{R}[\![t_2]\!]$$
$$\mathcal{T}[\![\alpha]\!] \Rightarrow \alpha$$

Figure C.1: Type rules for transformed expressions.

Traces for structured data values

For a structured data value, we want a trace not only for the outermost construction but also for all components. The $\mathcal{D}$ scheme is responsible for transforming datatypes into this form. We don't give the definition of the $\mathcal{D}$ scheme, but here is an example of the transformation of a simple tree datatype:

$\mathcal{D}[\![$`data Tree a = Node (Tree a) Int (Tree a) | Leaf a`$]\!]$
$\Rightarrow$
`data Tree a = Node` $\mathcal{R}[\![$`Tree a`$]\!]$ $\mathcal{R}[\![$`Int`$]\!]$ $\mathcal{R}[\![$`Tree a`$]\!]$ `| Leaf` $\mathcal{R}[\![$`a`$]\!]$
$\Rightarrow$
`data Tree a = Node (R(Tree a)) (R Int) (R(Tree a)) | Leaf (R a)`

Traces for values of function type

Perhaps a value of function type $\alpha \rightarrow \beta$ can be treated as a structured value with constructor $\rightarrow$ and components α and β? Then the type $\alpha \rightarrow \beta$ would be transformed into `R` $(\mathcal{R}[\![\alpha]\!] \rightarrow \mathcal{R}[\![\beta]\!])$. Is this adequate? No! A function of type $\alpha \rightarrow \beta$ has a guarantee to fulfill: given an argument of type α it must return a value of type β. But transformed functions have a further obligation; they must also return a *trace* for that value of type β. A function of type $\mathcal{R}[\![\alpha]\!] \rightarrow \mathcal{R}[\![\beta]\!]$ *cannot* fulfill that obligation, since when the function is applied it does not know the current trace (or evaluation context) so it cannot build a full trace for the return value. Transformed functions need the trace of the application site as an extra argument. As an example, here is how the type of the standard *map* function is transformed:

$\mathcal{R}[\![$ (`a` $\rightarrow$ `b`) $\rightarrow$ [`a`] $\rightarrow$ [`b`] $]\!]$
$\Rightarrow$
`R` (`Trace` $\rightarrow$ $\mathcal{R}[\![$`a` $\rightarrow$ `b`$]\!] \rightarrow \mathcal{R}[\![$ [`a`] $\rightarrow$ [`b`] $]\!]$)
$\Rightarrow$
`R` (`Trace` $\rightarrow$ `R` (`Trace` $\rightarrow$ `R a` $\rightarrow$ `R b`) $\rightarrow$ `R` (`Trace` $\rightarrow$ `R` [`a`] $\rightarrow$ `R` [`b`]))

Tracing *partial applications* will only be possible if a transformed function returns an `R`-value for each argument it is applied to. Note that this requirement is indeed fulfilled.

C.3.2 Creating traces for expressions

The trace for an expression depends on the *evaluation context* in which it is computed. The evaluation context of an expression is simply the trace of the redex in which the expression occurs. Given an evaluation context t, we will now define the transformation scheme for expressions, $\mathcal{E}[\![\mathrm{e}]\!]_t$.

Tracing identifiers

Identifiers can be either `let`-bound or λ-bound. Identifiers in patterns are λ-bound, and they already have traces, so the transformation of such identifiers is the identity transformation. Definitions of `let`-bound identifiers are transformed to expect a trace as argument; given a trace they produce a `Nm` node for this particular instance of the identifier.

$$\mathcal{E}[\![ident_\lambda]\!]_t \Rightarrow ident$$
$$\mathcal{E}[\![ident_{\texttt{let}}]\!]_t \Rightarrow ident\ t$$

Figure C.2: Transforming identifiers.

Tracing constructed values

Figure C.3 shows the transformation rules for constructed values. The trace for a constructed value with no components is just the `Nm` node of the constructor. If the constructed value has components, its trace is an `Ap` node containing traces for the constructor and for each component.

$$\mathcal{E}[\![conid\ e_1\ e_2\ \ldots\ e_n]\!]_t \Rightarrow \texttt{con}_n\ t\ \mathcal{N}[\![conid]\!]\ conid\ \mathcal{E}[\![e_1]\!]_t\ \mathcal{E}[\![e_2]\!]_t\ \ldots\ \mathcal{E}[\![e_n]\!]_t$$

```
con0 t nm conid = R conid (Nm t nm)
conn t nm conid e1@(R _ te1) e2@(R _ te2) ...en@(R _ ten) =
     R (conid e1 e2 ... en) (Ap t [Nm t nm, te1, te2, ..., ten])
```

Figure C.3: Transforming constructed values.

Tracing function applications

Figure C.4 shows the rule for transforming function applications to make them create traces as well as results. The auxiliary function vap_n builds the trace node and then applies the function to one of the arguments, leaving it to another auxiliary function ap_n to apply it to the rest of the arguments. (Functions need to take arguments one at a time, so that partial applications have traces).

Case expressions are transformed in much the same way as function applications, with `case` as the function name and the scrutinised expression as the argument.

$\mathcal{E}[\![f\ e_1\ e_2\ \ldots\ e_n]\!]_t \Rightarrow \mathtt{vap}_n\ t\ \mathcal{E}[\![f]\!]_t\ \mathcal{E}[\![e_1]\!]_t\ \mathcal{E}[\![e_2]\!]_t \ldots \mathcal{E}[\![e_n]\!]_t$

$\mathtt{vap}_n$ `t (R f t`$_f$`) e`$_1$`@(R _ t`$_{e_1}$`) e`$_2$`@(R _ t`$_{e_2}$`) ... e`$_n$`@(R _ t`$_{e_n}$`) =`
    $\mathtt{ap}_{n-1}$ `t`$_1$ `(f t`$_1$ `e`$_1$`) e`$_2$ `... e`$_n$
    `where t`$_1$ `= Ap t [t`$_f$`, t`$_{e_1}$`, t`$_{e_2}$`, ..., t`$_{e_n}$`]`

$\mathtt{ap}_n$ `t (R f t`$_f$`) e`$_1$`@(R _ t`$_{e_1}$`) e`$_2$`@(R _ t`$_{e_2}$`) ... e`$_n$`@(R _ t`$_{e_n}$`) =`
    $\mathtt{ap}_{n-1}$ `t (f t e`$_1$`) e`$_2$ `... e`$_n$
$\mathtt{ap}_0$ `t e = e`

Figure C.4: Transforming function applications.

As a simple example application of these rules, consider the transformation of the expression f `True` $(g\ x)$ in the evaluation context t. Assume that f and g are *let*-bound functions and x is a λ-bound variable.

$\mathcal{E}[\![\mathtt{f\ True\ (g\ x)}]\!]_t$
$\Rightarrow$
$\mathtt{vap}_2\ \mathtt{t}\ \mathcal{E}[\![\mathtt{f}\ ]\!]_t\ \mathcal{E}[\![\mathtt{True}]\!]_t\ \mathcal{E}[\![\mathrm{g\ x}]\!]_t$
$\Rightarrow$
$\mathtt{vap}_2$ `t (f t) (con`$_0$ `t "True" True) (vap`$_1$ `t (g t) x)`

Tracing `let`-expressions

The transformation rule for `let`-expressions is shown in Figure C.5. It uses the $\mathcal{F}$ scheme (defined in §C.3.3) to transform the local definitions.

$\mathcal{E}[\![\mathtt{let}\ \{d_1;\ d_2;\ \ldots;\ d_n\}\ \mathtt{in}\ e]\!]_t \Rightarrow \mathtt{let}\ \{\mathcal{F}[\![d_1]\!];\ \mathcal{F}[\![d_2]\!];\ \ldots;\ \mathcal{F}[\![d_n]\!]\}\ \mathtt{in}\ \mathcal{E}[\![e]\!]_t$

Figure C.5: Transforming `let`-expressions.

Tracing other types of expressions

Case expressions are transformed in much the same way as function applications, with `case` as the function name and the scrutinised expression as the argument. This is useful when browsing the trace: one can see in the source code which branch was used in a case expression.

Lambda expressions are treated as functions with the function name `\`. This works surprisingly well, since source links from the trace make it easy to examine the full lambda abstraction, if necessary.

High level syntactic constructs such as list comprehensions and sequence generators are currently traced as if replaced by their standard translations using list-processing functions.

C.3.3 Transforming function definitions to create traces

Figure C.6 shows the transformation scheme $\mathcal{F}$ for function definitions. A traced function accepts a trace as its argument (see Figure C.2) and returns an `R`-value containing a translated version of the function and a trace for the function identifier. The translated function takes one argument (and application-site trace) at a time, and for each one produces a new `R`-value, representing the partially applied function. We use an auxiliary function fun_n to build the trace node for the identifier and to accept arguments one at a time. The translated function f itself is not called until enough arguments are available for a full (saturated) application.

The $\mathcal{P}$ transformation scheme transforms patterns. Currently we only have limited support for using guards. The computation of guards of a function are traced, but as soon as one of them is true and the corresponding function body is evaluated, the traces of the guards are discarded.

We illustrate the $\mathcal{F}$ scheme with an example, in which two simple functions are transformed.

$\mathcal{F}[\![f\ p_1\ \ldots\ p_n = e]\!] \Rightarrow f = \texttt{fun}_n\ f'\ \mathcal{N}[\![f\,]\!]$
$\qquad\qquad\qquad\qquad\quad \text{where } f'\ t\ \mathcal{P}[\![p_1]\!]\ \ldots\ \mathcal{P}[\![p_n]\!] = \mathcal{E}[\![e]\!]_t$

$\mathcal{P}[\![\_]\!] \Rightarrow \_$
$\mathcal{P}[\![ident]\!] \Rightarrow ident$
$\mathcal{P}[\![conid\ p_1\ \ldots\ p_n]\!] \Rightarrow \texttt{R}\ (conid\ \mathcal{P}[\![p_1]\!]\ \ldots\ \mathcal{P}[\![p_n]\!])\ \_$

```
funn f fs t =
   R (λt1 e1→
      R (λt2 e2→
         ...
           R (λtn en→ f tn e1 ... en)
             tn−1)
         ...
        t1)
     [Nm t fs]
```

Figure C.6: Transformation scheme for function definitions.

```
f x = g x x True + x
g x y True = x*y
g x y False = 1
     ⇒
f = fun1 f’ "f"
    where f’ t x = vap2 t ((+) t)
                       (vap3 t (g t) x x
                             (con0 t "True" True)) x
g = fun3 g’ "g"
    where g’ t x y (R True _) = vap2 t ((*) t) x y
          g’ t x y (R False _) = con0 t "1" 1
```

C.3.4 Two problems and their remedies

Over-saturated applications

Trace-construction as we have described it so far works well for both partially and fully saturated applications. But it fails to build correct traces for function applications that *over-saturate* functions (i.e. the function in the application is given more arguments than are shown in the function's definition). In the following example, `f 2 3` is an over-saturated application, since `f` is given two arguments, although by definition `f` has arity 1.

```
let f x = g x
    g x y = x+y
in  f 2 3
```

Unfortunately, traces for calls made from within an over-saturated function are lost. The reason can be seen in the definition of ap_n. If an application of arity m over-saturates a function of arity n, the functional argument f to ap_{m-n} will represent the result of a saturated call complete with a trace t_f, *which ap ignores!*

Including the trace for the function part in *every* call to an *ap* would be excessive. Suppose we are tracing the function call *f 1 2 3*: the result would be a trace with separate nodes for each of *f 1 2 3*, *(f 1) 2 3*, and *((f 1) 2) 3*.

We only want to include the trace of the function part *if* it is the result of a call to a fully saturated function. But how do we know that? Examining the definition of fun_n we see that an unsaturated function is applied to its own trace. When ap_n applies a function, it can check if the trace in the result *is the same* as the one passed to the function as first argument. If the traces are the same, the application is partial; otherwise it is saturated.

It is not enough to test if the trace objects denote equal values, we must check that the trace objects really are one and the same. This sounds simple, but it is not possible in a pure functional language to test equality of *objects*, one can only test equality of *values*. We overcome the problem by introducing an impure primitive for *pointer equality* which we call `sameAs` — it is the *only* impure function we need at run-time. The modified version of ap_n is presented in Figure C.7.

Unevaluated applications

When the computation has finished, exploration of the final trace can start, either from a faulting expression or some part of the output. But what if we are interested in some saturated but unevaluated function application?

There is a conflict here between needing extra evaluation to obtain a trace, and wishing to avoid the extra evaluation to preserve the usual behaviour. If we evaluate the application to obtain the trace, the computation may diverge, giving us different behaviour for traced computated than for non-traced computations, which is clearly not acceptable.

```
apₙ t (R f t_f) e₁@(R _ t_e₁) e₂@(R _ t_e₂) ... eₙ@(R _ t_eₙ) =
    if t `sameAs` t_f then
      apₙ₋₁ t (f t e₁) e₂ ... eₙ
    else
      let t′ = Ap t [t_f, t_e₁, t_e₂, ..., t_eₙ]
      in apₙ₋₁ t′ (f t′ e₁) e₂ ... eₙ
ap₀ t e = e
```

Figure C.7: The modified version of ap_n.

We had a similar problem with partially applied functions, which we solved by forcing functions to return a new function and a trace for each argument. But when the function is fully saturated, the result may or may not be evaluated.

We solve this problem by introducing a new constructor `Sat` in the `Trace` datatype. A `Sat` node is introduced when a function application becomes fully saturated, and contains two parts: the trace of the fully saturated function call, and the trace of the result of the function call, which may be unevaluated. When the display program encounters a `Sat` node in the trace, it checks whether the result is evaluated. If so, the trace of that result is used; if not, the saturated function call and its trace is used instead. Note that `Sat` can be seen as an exceptional *forward* pointer in the trace, and that it is only useful as long as the result part is unevaluated. The revised definition of $\mathtt{fun}_n$ is given in Figure C.8.

C.4 Displaying trace information

In this section we describe the interactive display component of our tracer. Information is presented to the user in three panels: (1) redex trails; (2) program sources; (3) program output[1] (see Figure C.9 for an example). Though the user is free to browse any of these panels by scrolling, links to and from the redex trails provide the most important form of access. The programmer has access by mouse-click from any identifier in the trace display to both the corresponding definition and the relevant applied occurrence of

[1]The current tracer restricts I/O in the traced program to a single textual input and a single textual output.

```
fun_n f fs t =
   R (λt₁ e₁→
      R (λt₂ e₂→
         ...
            R (λt_n e_n→ let R r t' = ft_n e₁ ... e_n
                         in R r (Sat t_n t'))
              t_{n-1})
         ...
        t₁)
     [Nm t fs]
```

Figure C.8: The new definition of fun_n *used in the function definition transformation.*

the identifier in the source program.

The number of nodes in a full trace exceeds the number of reductions in the traced computation, so typical traces are large structures containing a great deal of information. How much of this information can sensibly be presented is limited by both the *capacity of the user* and the *capacity of the screen* used for display.

C.4.1 How the user controls the display

At the start of a tracing session the redex-trail panel may contain an undefined function application detected and reported as a run-time error. If there was no such error the panel is initially empty; by selecting some fragment of text from the output panel, the user requests an initial display of the parent redex for that text.

The user now acts as a source of demand, controlling the extent of 'display by need'. Clicking over an already-displayed subexpression in a trace node requests the display of its parent redex. Moving the mouse to a different part of the display not only changes the currently selected expression; it also causes highlighting of three classes of *related expressions*. If E is the currently selected expression:

- all *shared occurrences* of E currently on display are highlighted in the same colour as E itself;

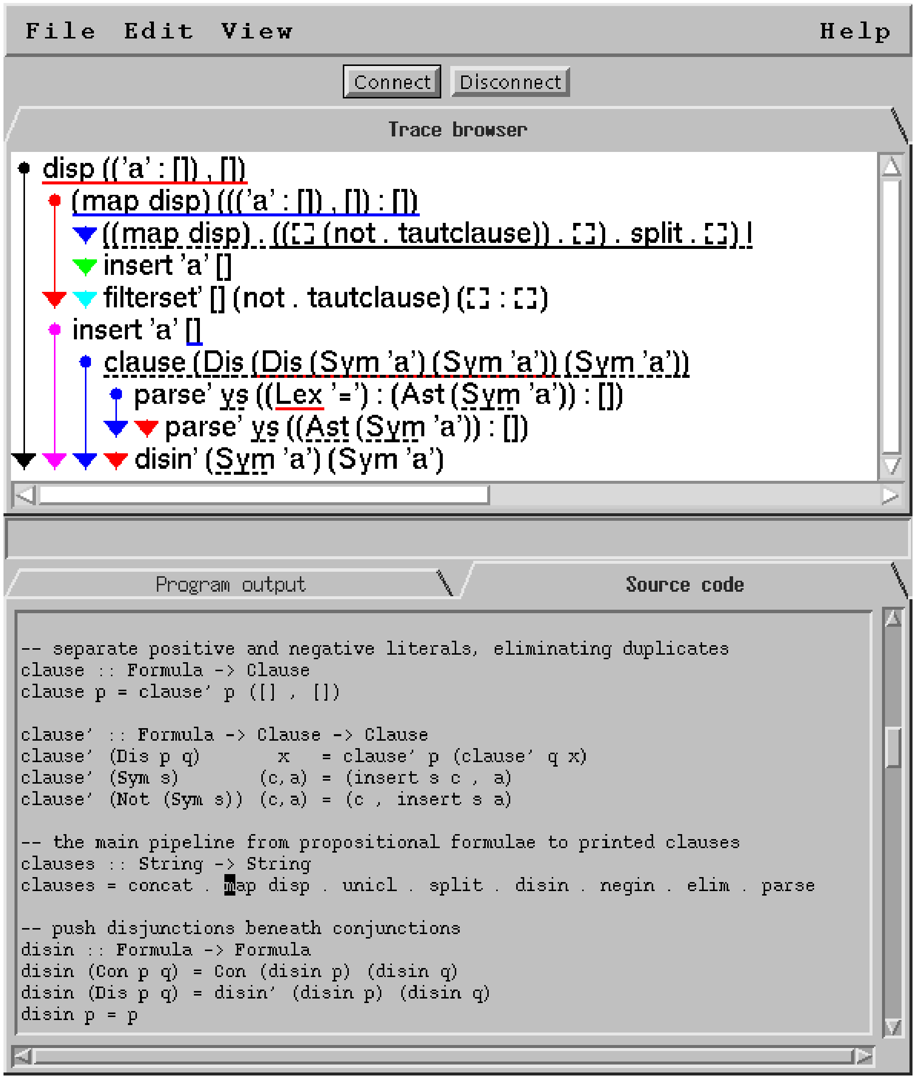


Figure C.9: A session snapshot of the interactive trace display program.

- all *expressions with the same parent redex* as E are highlighted in a different colour;

- if the *parent redex* of E is already on display, it is highlighted using a third colour.

C.4.2 Making the most of screen space

Screen-space is an all-too-scarce resource when displaying a complex structure. We have used a combination of techniques to reduce the amount of display space needed to show part of a trace.

Displaying expressions on one line

Even a single expression may be large, yet the user must be able to view the relationships between several of them. We therefore confine each expression to a single line of the display. To make this possible:

- for all expressions, details below a specified depth in the parse-tree are suppressed by default – clicking on a place-holder for any unprinted subexpression requests more detail if needed;
- when tracing to discover the cause of a fault, subexpressions that are *never evaluated* can be suppressed with a distinguished marker – their details *cannot be relevant*;
- to make space for extra detail in one part of an expression the user can click on other parts to collapse them to place-holders;
- when all else fails, horizontal scrolling is available!

Display by need revisited

We have already described how parent redexes are displayed only as and when the user demands them. Continuing the analogy with a lazy evaluator:

- it is important to have shared references to a common parent redex rather than displaying it several times — hence the colour-coded information about shared, or already-displayed, parent redexes as the mouse-cursor traverses the display;
- also, re-clicking on a subexpression with a currently displayed parent redex removes the parent (and any displayed parents of its subexpressions, recursively) from the display.
- when all else fails, vertical scrolling is available!

C.5 Evaluation

Although our tracing system is still being refined, in this section we offer some observations and measurements by way of a preliminary evaluation. We address the following questions:

- What class of functional programs can the tracing system deal with?
- How much extra time and space does its use require?
- To what extent is the tracing scheme portable to other implementations, or even to other functional languages?
- How well does the tracer achieve its purpose as a useful tool for understanding (possibly faulty) functional computations?

C.5.1 Class of traceable programs

Our current implementation of the tracer is by extension of the nhc13 compiler. Programs to be traced can make use of a very large subset of Haskell 1.3, including type and constructor classes. We have traced a variety of programs, not all of them written by ourselves. The largest programs extend to several modules and some 2000 lines, yet few changes were needed to the original sources.

The most significant of the necessary changes restrict the I/O components of programs to a single output only. We are unsure as yet how best to trace I/O more comprehensively.

Some other things are not yet handled as we eventually intend. For example, comprehensions are currently traced as if replaced by their standard translations into compound applications of list-processing functions; we'd prefer something closer to the source program. Guard expressions are faithfully evaluated, but unless a fault occurs during a guard computation the redex trail for the Boolean value is not currently incorporated in the final trace; we'd like to make this information available to the programmer. We see no fundamental difficulties here; it's just a matter of a bit more work attending to details.

C.5.2 Time and space costs

Using the tracer incurs extra costs both at compile-time and at run-time. We give here some figures for the construction of *complete* redex trails. We take

Table C.1: The compile-time costs of generating redex trails. Only the `cichelli` *program has more than one module.*

	compilation time (s)		size of object code (kb)	
	normal	tracing	normal	tracing
`cichelli`	18.79	50.58	51.21	170.25
`clausify`	8.32	20.06	39.26	104.17
`primes`	6.51	14.39	13.15	37.33

Table C.2: The run-time costs of generating redex-trails. Garbage-collection time is not included because it varies with heap size.

	reduction time (s)		max live heap (b)		number of
	normal	tracing	normal	tracing	reductions
`cichelli`	1.05	21.47	34 k	13 M	354 636
`clausify`	2.05	73.32	7 k	28 M	1 405 539
`primes`	5.09	33.50	60 k	54 M	520 073

three example programs: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` simplifies a given proposition to clausal form; `primes` generates the first 2,500 primes using a wheel-sieve. All three programs are included in the NoFib benchmark suite [Par93].

Table C.1 shows the added costs apparent at *compile-time*. Much the same results are obtained for all three examples: both compile-time and the size of resulting object-code increase by a factor of two or three when we apply the program transformation to build redex trails.

Table C.2 illustrates the *run-time* costs of generating redex trails; figures for the traced computations include the costs of evaluating both the result and the trace in full. Recording the details of the parent redex for every subexpression in every instance of every function body is not cheap! A trace-constructing computation takes 6–36 times longer than a normal one. The trace structure itself occupies from 20–100 bytes per application: so to construct traces with a million recorded reductions we need plenty of memory on our workstations! The wide variations in the performance figures for the different programs are accounted for by factors such as the arity of functions involved.

A final aspect of performance is less easily measured. The interactive trace-display program must respond rapidly to the user's requests. For all the examples we have tried to date, response-time has never been a problem.

C.5.3 Portability

Our current tracing system is implemented in a version of Röjemo's nhc compiler [Röj95]. The transformation to introduce trace values is performed on an intermediate representation of programs internal to nhc, and nhc's run-time system has been modified to support traced computations.

However, our tracing scheme is not closely tied to this one compiler. The results of the program transformation could be expressed in source form and supplied to a different Haskell compiler. Only one special primitive is needed at run-time: `sameAs` tests whether two traces are the same in a referentially opaque sense as explained in section §C.3.4. There are just two other additions to the run-time system: one retains access to output; the other provides a link to the display program.

We see no reason why our tracing scheme could not be applied to other lazy functional languages.

C.5.4 Will it solve 'The Tracing Problem'?

This is both the most important and the most difficult question to answer satisfactorily. A tracer might cope efficiently with a full range of programs and port easily to new compilers, yet fail the acid test: do programmers use it in practice to solve their problems?

At this stage, apart from our own experience trying out the tracer, we have only the results from an extended student exercise with an earlier prototype. Providing direct links between the trace and the source program turns out to be even more important than we expected; as do the links to textual input and output. But from these early trials we are quietly optimistic.

Currently we cannot handle non-terminating programs, unless a *black hole* [Jon92] is detected. A possible solution to this problem is to let the user interrupt the computation (e.g. by pressing `control-C`), and start browsing from the trace of the interrupted application, which must form a part of the cycle that caused the program to loop.

We would welcome enquiries from readers who teach functional programming and would be interested in 'class-testing' the tracer.

C.6 Related work

Some previous systems for tracing functional programs systems have been based on monitoring the *series of events* in a computation [KHC91], perhaps with the ability to examine events immediately preceding or following a suspected fault [TA90]. This approach is viable for a strict functional language with eager evaluation but breaks down for non-strict languages with a lazy evaluation strategy. In a Haskell computation closures for function applications can lie dormant for many reductions. If no equation matches when a closure comes to be evaluated it may be irrelevant to ask 'What happened just before this?' because preceding events may have nothing to do with the faulty closure. Simple traces of event sequences have been tried for lazy languages, but found wanting.

Our tracing based on *redex trails* has in common with Nilsson and Fritzson's system [NF94] the construction of a computational history tracing the origin of expressions back through ancestral redexes. However, the structure of their computational history is rather different. A node in their trace tree contains a saturated function application, along with its result and a list of histories for the function applications evaluated in the body of the saturated function. They also suggest using the algorithmic debugging technique [Sha82], which is a method of navigating in the trace by answering questions about the equivalence of (possibly complex) expressions. Naish and Barbour [NB95] use a source-to-source transformation to produce a computational history (which they call an evaluation tree) similar to Nilsson's and Fritzson's for a simple lazy functional language. Nilsson and Sparud [Spa96, SN97] further explores the creation and browsing of such evaluation trees, and address the problem of high memory consumption by constructing the history incrementally.

Hazan and Morgan [HM93] suggests a tracing technique that could be seen as a simplified abstraction of ours. Their *distinguished paths* are like our redex trails but contain *only function names from fully saturated applications*. For example, if `main` needs the value of `f 1`, and `f` needs the value of `head []` the consequent run-time fault will be assigned the distinguished path `main` $\rightarrow$ `f` $\rightarrow$ `head`.

C.7 Conclusions and future work

So what have we achieved? We have developed a new scheme for constructing a complete trace of a lazy functional computation, based on redex trails.

We have expressed this scheme in rules for program transformation, and we have implemented it in a Haskell compiler: the trace is itself represented as a functional data structure and the compiler transforms programs so that they build traces of their own execution. We have also developed a display program that allows the interactive exploration of redex trails, fully linked to the source program.

Previous attempts at similar schemes have often put severe limits on the language used to express traced programs, or on the amount of detail recorded in traces. We aim to build fully-detailed traces for full Haskell computations. Though our present tracer handles most of Haskell, we are keen to lift some of the current restrictions on I/O; special forms such as comprehensions and guards also need more attention as explained in §C.5.

The usefulness of such a tracer for functional programmers at large critically depends on its performance and capacity. Our ultimate goal is a tracer that can be applied to large computations – such as a functional language compiler. Though applications to date have been modest, as illustrated in §C.5 they do include fully-traced computations of over a million reductions, derived from multi-module programs several pages in length.

However, the speed penalty of tracing is still too high. We hope to reduce it so that in the *worst case* slow-down is no more than a factor of ten. There is certainly scope for speeding up the trace-constructing machinery — for example, by moving key auxiliary functions into the run-time system.

Space costs are even more critical. The important observation here is that even the most enquiring programmer cannot really want to follow a trail through hundreds of thousands of expressions! Rather they will be interested in just a few selected paths, though there is no way of knowing in advance just which paths these will be. We have just begun work on *trace-pruning* techniques that should substantially reduce the cost of building and storing traces, by confining them to *partial* redex trails. For example, we can bound the length of ancestral paths, truncating longer trails at each garbage collection: even bounding the length to zero there is often a useful trail from an error, since trails are still constructed between collections. Another approach is to discard details of computation inside specified modules: some early results here are very promising — for the `primes` computation, the peak amount of memory needed shrinks by a factor of 50 when we eliminate traces inside the prelude. We also have another line of attack on the space problem: we plan to write trace structures to file in a *compressed binary format* [WR97]. Storing traces in files also has the advantage that the trace of a lengthy run can be re-examined many times without re-incurring the cost of the trace-building computation.

Building a fast and space-efficient tracing system is only one side of the problem. Once the above refinements are made, we are keen to put the tracer to the test in the hands of other users. We'd like to run a series of controlled experiments measuring the speed and accuracy with which users of the tracer can find faults in programs. Only by such experiments and the outcomes of wider use can we hope to confirm that constructing and examining redex trails can be an effective part of functional programming.

Acknowledgements

Niklas Röjemo built the `nhc` compiler that we have adapted for our experiments in tracing. Anonymous PLILP referees provided helpful comments, questions and suggestions. Our work is funded by the Engineering and Physical Sciences Research Council.

Paper D

Complete and Partial Redex Trails of Functional Computations

Jan Sparud and Colin Runciman
Department of Computer Science, University of York
Heslington, York, YO1 5DD, UK
email: {sparud,colin}@cs.york.ac.uk

Abstract

Redex trails are histories of functional computations by graph reduction; their main application is fault-tracing. A prototype implementation of a tracer based on redex trails [SR97] demonstrated the promise of the technique, but was limited in two respects: (1) trails did not record every reduction, only those constructing a new value; (2) even so computing trails was very expensive, particularly in terms of the memory space they occupied. In this paper, we address both problems: *complete* redex trails provide a full computational record; *partial* versions of these trails exclude all but selected details, greatly reducing memory costs. We include results of experiments tracing several applications, including a compiler.

D.1 Introduction

Programs do not always work first time. When they fail, programmers cannot always see the reason immediately. Implementors must therefore provide some way of *tracing* computations. By examining traces, programmers can see not only final results or failures, but also how these have come about.

In many programming languages, traces can be based on sequences of events. But in a lazy functional language, the sequence of events is not so easily presented, nor is it what the programmer really needs. A more appropriate form of trace links a final value, or failure, to its *parent redex*, and each part of that redex in turn to *its* parent, and so on until one reaches input data or initial expressions present in the program. We call such traces *redex trails*. In a previous paper [SR97] we described a method for transforming functional programs to self-tracing equivalents, computing not only the result as usual but also a redex trail showing how the result was obtained. In addition, we described a *trace browser* used to examine traces in close connection with programs and their results. The scheme was promising, but its scope of usefulness was limited in two respects.

First, not every reduction was recorded in a redex trail, only those with the construction of a new value as the outcome. For some applications we found this was enough to provide highly informative traces. But in others, for example those with a strong search component, large parts of the computation are concerned with conditions for selecting between given values; if details of these parts are not recorded in trails, the programmer examining traces lacks important information.

Secondly, and despite the restricted class of reductions recorded, it was very expensive to compute redex trails. For a range of small applications we found that a trace-constructing computation could take 30 times longer than a normal one. The demands for memory space were even more prohibitive, with each recorded reduction taking up to 100 bytes. Tracing was just not feasible for computations beyond about a million reductions — sufficient for most textbook exercises, but not for serious applications.

In this paper we describe how we have tackled both problems. Redex trails have been extended to include a representation for every reduction that occurs, providing complete information about all parts of a computation. Desirable as this may be, it makes trails even larger, and more expensive. We have implemented two techniques for controlling the construction and retention of redex trails, so that stored trace structures are simpler and smaller: it is indeed useful to confine trails to a limited class of reductions, but now this class is *selected* by the programmer in terms of the program components they wish to be observed at work; if demands on memory are still too high, we *prune* redex trails using harsh garbage-collection policies.

We give results for a range of applications, including two much larger than those studied in the previous paper.

Section D.2 briefly explains the main elements of redex trails, as implemented in our prototype tracing system. Section D.3 identifies the class of reductions not recorded by that system, and describes the extension to *complete* redex trails. Section D.4 is concerned with *partial trails*; it deals first with *selective* trails and then with the complementary technique of trail *pruning*. Section D.5 gives results, concentrating on the space requirements of the enhanced tracing system. Section D.6 discusses related work. Section D.7 draws conclusions and suggests future work.

D.2 A redex-trail tracing system

Suppose the function `length` is defined by the equations

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

and we compute `3` as the result of `length [1,2,3]`. Figure D.1 shows a trace of that computation in the form of a *redex trail*. The *parent redex*

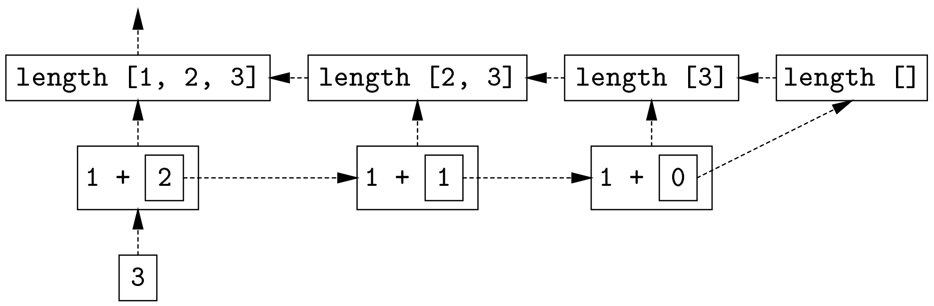


Figure D.1: The full redex trail for the computation of `3` *as the result of* `length [1,2,3]`.

of the final `3` is `1+2`. Considering subexpressions in that redex, everything but the `2` was constructed as an immediate result of reducing the original expression `length [1,2,3]`, but the `2` came about by reduction of `1+1`. And so on.

As described in a previous paper [SR97], we have modified the Haskell compiler `nhc` [Röj95, RR96] to provide traces based on redex trails. On request, our modified compiler applies a program transformation causing each value in the compiled program to be computed along with a redex trail recording the value's origin. A special browser, illustrated in Figure D.2, allows the user to examine redex trails, starting from program output, or a run-time fault, or some point at which evaluation was interrupted.

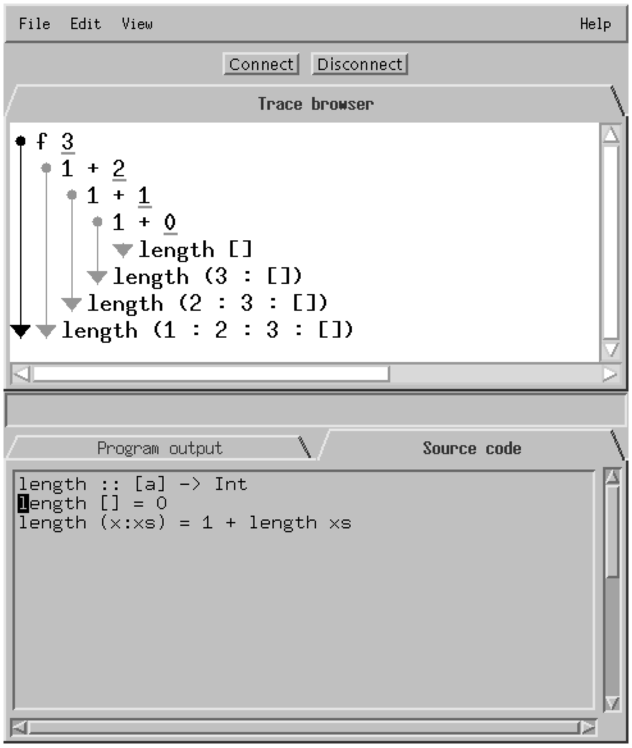


Figure D.2: How redex trails are displayed by the trace browser, shown here being used to examine the `length` *trail of Figure D.1. Each redex on display has been requested by the user in a point-and-click style. Available links to the source program include the points of definition and use for each variable occurring in the trail, and also the site of each function application.*

Redex trails are themselves implemented as Haskell data structures of type `Trace`. A simplified definition of the `Trace` datatype is:

```
data Trace = Appl Trace [Trace] SourceRef
           | Name Trace NameInfo SourceRef
           | Root
```

Each construction `Appl` t_c `[`t_f`,` t_{a_1}`, ... ,`t_{a_n}`]` sr records a function application. The trail recording the origin of the function is t_f, and the trails t_{a_i} are for its arguments. The trail t_c, which we term the *application con-*

text records how the application of this function to these arguments came about. (In a first-order language t_c and t_f would always be the same, but in a higher-order language it is important to distinguish between them.) The source reference *sr* allows immediate access to the corresponding function application in the source program.

Each construction `Name` t i sr represents an identifier, constant or literal. Its redex trail is t. The information field i records a textual representation of the name, and the module and position of its *definition*. The source reference *sr* is linked to the relevant applied occurrence or *use* of the name.

The `Root` constructor is used as a trail terminator: for example, as the trail of each name with a top-level definition.

As all values in a traced computation, including functions, have trails attached to them, traced function applications are translated by the compiler to applications of special combinators that extend the redex trails appropriately.

D.3 Complete trails

In the previous section we had a good reason for choosing the `length` function as an example: our prototype system builds *complete* redex trails for `length` computations, including a record of every reduction. But now consider another simple function over lists:

```
last [x] = x
last (x:xs) = last xs
```

Suppose we again compute `3`, but this time as the result of `last [1,2,3]`. In contrast to Figure D.1, the trail constructed for this computation by our prototype tracer is not very informative (see Figure D.3).

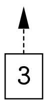


Figure D.3: A rather uninformative redex trail for the result of `last [1,2,3]`.

A programmer examining this trail to answer the question 'Where does the result `3` come from?' finds only the answer that `3` was given in the original expression; there is no mention of the `last` function or its workings. Why such a dramatic difference between the trails for `length` and `last` computations? Because a trail attached to a value by our prototype system records only how that value itself came to be computed. Shared values have identical trails. If the result of a function is (a component of) one of its arguments the trail for the result is exactly the trail already recorded in the argument.

Results that are not fresh values also pose a problem for graph-reducing interpreters or compilers: the application subgraph must somehow be updated to become, in effect, a piece of graph already constructed elsewhere. A commonly adopted solution is to introduce an indirection. We also use indirections to record such reductions in extended redex trails, giving `Trace` a new construction:.

```
data Trace = ... | Indi Trace Trace
```

A trace of the form `Indi` t_{res} t_{app} is constructed for the result of an indirecting application trace t_{app}, the result itself having trace t_{res}. The revised trail for the `last [1,2,3]` example is shown in Figure D.4.

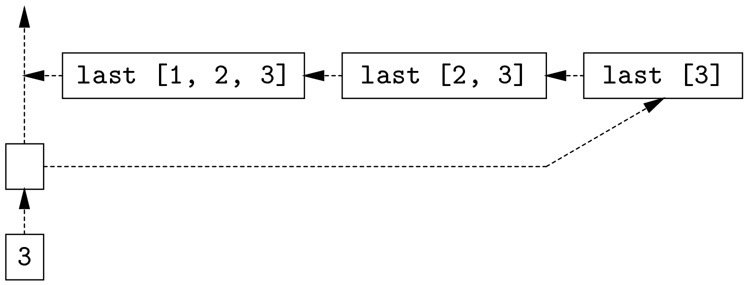


Figure D.4: A complete redex trail for `last [1,2,3]`*. The empty box represents an indirection giving access to the chain of recursive calls selecting* `3` *as the final list element as well as to the trace of* `3` *itself.*

With the addition of these indirection structures, redex trails become complete. Every reduction occurring in the computation can be observed in the trail.

D.4 Partial trails

Our prototype tracer needed tens of megabytes to store the trails of computations involving a few hundred thousand reductions. Plenty of functional programs take many millions of reductions to compute their results. And with redex trails extended to include all reductions, not just the value-producing ones, the demand for memory will be even greater. There is a clear need for techniques to obtain simplified, smaller variants of a full redex trail.

D.4.1 Selective trails

Not even the most inquiring programmer will be able (or willing!) to look at every single reduction in a large computation. So why record all of them? The problem is how to know which reductions to record and which to ignore. One solution is to let the programmer select the parts to be traced when they run a program.

Suppose each function defined at the top level of a program is either *suspected* or *trusted* by the programmer, and somehow marked as such. (Even in a freshly written program, obvious candidates for trusted functions include any from the `Prelude`.) Broadly speaking, an application of a suspected function should always be recorded in the redex trail, but for trusted functions this may be unnecessary.

For example, when taking the length of a list, we are rarely interested in how `Prelude.length` works, so recording a full redex trail for the internal computation of `length` would be excessive, both from the programmer's point of view and in terms of memory space. If we trust `length` (and primitive addition), then instead of the full redex trail in Figure D.1 we might prefer the simpler *partial trail* shown in Figure D.5. Note that the

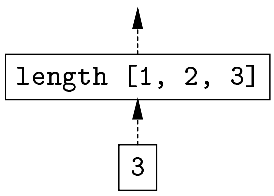


Figure D.5: A selective trail for `length [1, 2, 3]` *when* `length` *is trusted.*

top-level call to `length` is still in the trace. Though we trust the internal workings of `length`, we still want to see the *original application* that reduced to `3`.

The tracing machinery has to decide for each function application whether or not to record it in the redex trail. A simple rule would be: *Do not record applications in a trusted context.* This would produce the desired result for the `length` function, as all the intermediate applications of `length` and `+` occur in the context of `length` itself.

Higher-order functions

However, matters get more complicated when we decide to trust a higher-order function. Figure D.6 shows a full redex trail from the result of `map dbl [1, 2, 3]`, where `dbl` is a function that doubles its argument. Suppose we trust `map` but not `dbl`. Should we then trust the result of `map dbl [1, 2, 3]`? Clearly not. The problem is that the trusted function `map` builds applications of the suspected function `dbl`. If we follow our simple rule, we will not record applications of the suspected function `dbl`, as figure D.7 shows.

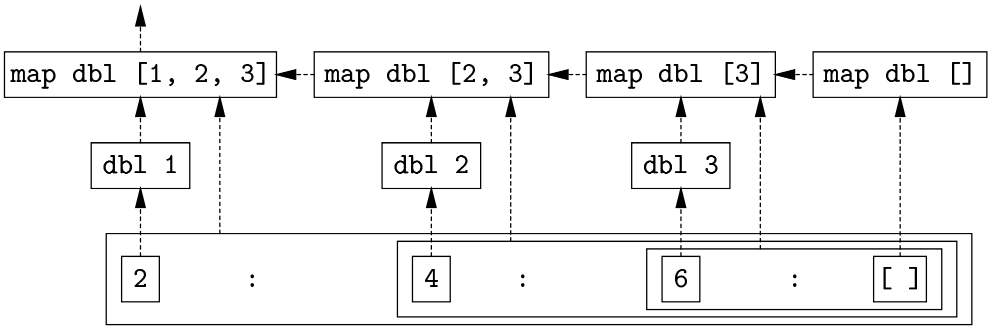


Figure D.6: The complete trail for `map dbl [1, 2, 3]`.

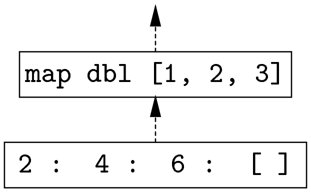


Figure D.7: A selective trail for `map dbl [1, 2, 3]` *when* `map` *is trusted but* `dbl` *is suspected, if context alone determines which reductions are recorded.*

To solve this problem we revise the rule: *Do not record applications of trusted functions in trusted contexts.* Figure D.8 shows the effect of this revised rule for the `map dbl` example: there is no record of applications within `map`, except for the calls to `dbl`.

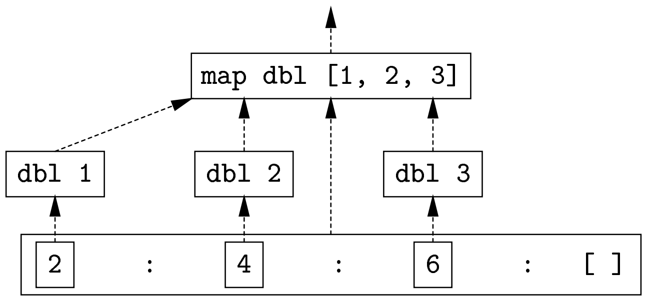


Figure D.8: The trace of `map dbl [1, 2, 3]` *when* `map` *is trusted and* `dbl` *is suspected, using both the context and the applied function to determine recorded reductions.*

Implementing selective trails

To implement selective trails, we need to be able to find out if the context and the applied function are trusted when a function is applied. We can find out the name of the function in the current context and the name of the function to be applied by inspecting their accompanying trace structures. But as this operation will be performed every time a function is applied, it is important to make it cheap. The combinators for traced function application are written in Haskell, but to use a Haskell data structure to record whether a function is trusted or not is inappropriate for two reasons: (1) it would be hard to create the structure in the context of separate compilation; (2) it would be expensive to lookup the information in it.

Recall that wherever a name occurs in a redex trail, there is a `Name` construction in the `Trace`, including a record of all needed information about the name. The record for each name is unique, shared across all `Name` occurrences. For each module, the compiler constructs a table of records for all the names the module defines. Also, for each module there is a table of imported modules; so from the main module it is possible to reach the name table for all modules in the program.

So we simply add a 'trusted' flag to every name record. Whenever a function is applied, the application operator checks if the trusted flag is set for *both* the context of the application *and* the function being applied. The redex trail is only extended if this test fails.

The programmer can decide at compile-time whether or not to trust all the functions in a module by default. But run-time flags allow the default to be over-ridden. The implementation is a straightforward search through import tables, starting from the main module, to find the name table of the specified module. There the trusted flags are set or reset as needed. The search is performed once only, before starting the functional computation.

D.4.2 Trail pruning

Selective trails should go some way to solving the problem of excessive memory demands. But even with reductions within the *Prelude* and other low-level modules excluded, a redex trail for something like a Haskell compilation may be so large as to be impractical. If tracing is to be feasible even for large computations, we need something more.

Another way of limiting the size of redex trails is to apply some kind of *pruning* when memory starts to run out. This idea prompts two questions: (1) when to prune? and (2) what to prune? A reasonable answer to the first question is to prune redex trails during each garbage collection. The garbage collector is called when available heap memory is exhausted; it already scans the heap including both normal program data and trails. Pruning can be implemented as a harsh garbage collection policy. The second question is harder. Our approach is simple: limit the *length* of all trails accessible from live program data or output. Trails beyond a certain length k are pruned, by overwriting them with a new `Trace` constructor `Pruned`.

When exploring a trace that has been pruned to length k, the user is guaranteed to be able to follow each trail at least k steps. When reaching a pruned part of the trace, the display program indicates that the information is no longer available. For the common error that no clause in a function definition matches the actual argument (eg. `head []`), setting $k = 1$ is enough to be able to find out which function called the non-matching function (and the source position of the call). With $k = 2$, the top level of the argument will still be accessible.

D.5 Experimental results

What are the practical consequences of extending the redex-trail tracer in the ways described in previous sections? To find out we first looked again at the three benchmark computations studied in our earlier paper: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` transforms a proposition to an equivalent in clausal form; `primes` computes the first 2,500 primes using a wheel sieve.

Table D.1: A comparison of memory costs for incomplete redex trails without indirecting applications and complete trails with them.

	Incomplete	Complete
`cichelli`	13 Mb	14 Mb
`clausify`	28 Mb	28 Mb
`primes`	54 Mb	85 Mb

D.5.1 Extra cost of complete trails

How much larger are the complete trails, including indirecting applications, than trails without them? The comparative memory costs are shown in Table D.1. Recall our experience with the prototype: we felt a lack of information about some applications but not others. For `primes`, completing the trail means increasing its size by over 50%, but for `cichelli` and `clausify` the trail sizes hardly change.

D.5.2 Sizes of selective trails

Our first remark about the practical consequences of selective trails is a subjective one. We immediately appreciated the benefit of suppressing detail within the `Prelude` when examining traces.

More objectively, how much smaller are selective trails than full ones? As Table D.2 shows, when `Prelude` functions are trusted, between 40% and 80% of reductions recorded in the full trails of these programs are omitted in a selective one. There is a corresponding saving in memory, but not necessarily in proportion. Some function applications have far larger representations than others depending, for example, on the number and size of arguments and whether these are shared. For `cichelli` and `clausify` the memory saving is only 20%–30% — rather less than the reduction figures might lead one to expect. For `primes`, on the other hand, the fraction of memory saved far exceeds such expectations: about a third of all reductions are still recorded, yet the amount of memory needed shrinks by almost two orders of magnitude!

However, in all cases the final column of Table D.2 provides a salutary reminder of how large the trails still are compared with the structures in memory for the normal untraced computation. Even selective redex trails can increase heap-memory requirements by a huge factor. Such increases

Table D.2: The effect of excluding trace information within a trusted `Prelude`.

	Full Trail	Selective Trail		No Trail
	#reductions	#reductions	memory	memory
`cichelli`	355 k	213 k (60%)	10.0 Mb	34 kb
`clausify`	1406 k	291 k (21%)	23.2 Mb	7 kb
`primes`	981 k	322 k (33%)	0.8 Mb	60 kb

may be infeasible for larger applications, confirming the need for a more drastic space-saving technique such as pruning.

D.5.3 How effective is pruning?

To test the effectiveness of pruning we again constructed traces for the three benchmark computations `cichelli`, `clausify` and `primes`, but with trails beyond some length k pruned away at each garbage collection. Figures D.9 and D.10 confirm the expected result that pruning with small values of k can save a great deal of space. What the figures do *not* show is that in practice even for very small values of k one finds a good deal of information in traces — after all, the last garbage collection may have been some time ago. Even setting $k = 0$ is a worthwhile option, and the memory overhead in that case is only a small constant multiple of normal memory use.

Another observation from these graphs is that the fraction of memory that is saved for a given pruning length k varies widely between applications. Compare the effect of pruning the `clausify` trail (upper part of Figure D.10) with the effect of pruning the `primes` trail (lower part of Figure D.10). For `clausify`, even 'light pruning' with $k = 16$ is enough to remove about 95% of the trace: we have had to omit the plot for $k = \infty$, to make comparisons between other k values visible! For `primes`, however, the memory saved by pruning with $k = 16$ is only 1%–2%, barely discernible in the graph, and much harsher pruning is needed to make appreciable memory savings.

Any pruning scheme based on a fixed numeric limit has the drawback that the user may be unsure what limiting value is appropriate. An advantage of directly limiting path-length is that the user knows what losses to expect when it comes to browsing the trace. But what the user may *not* know is whether the trace will fit in available heap memory. As an alter-

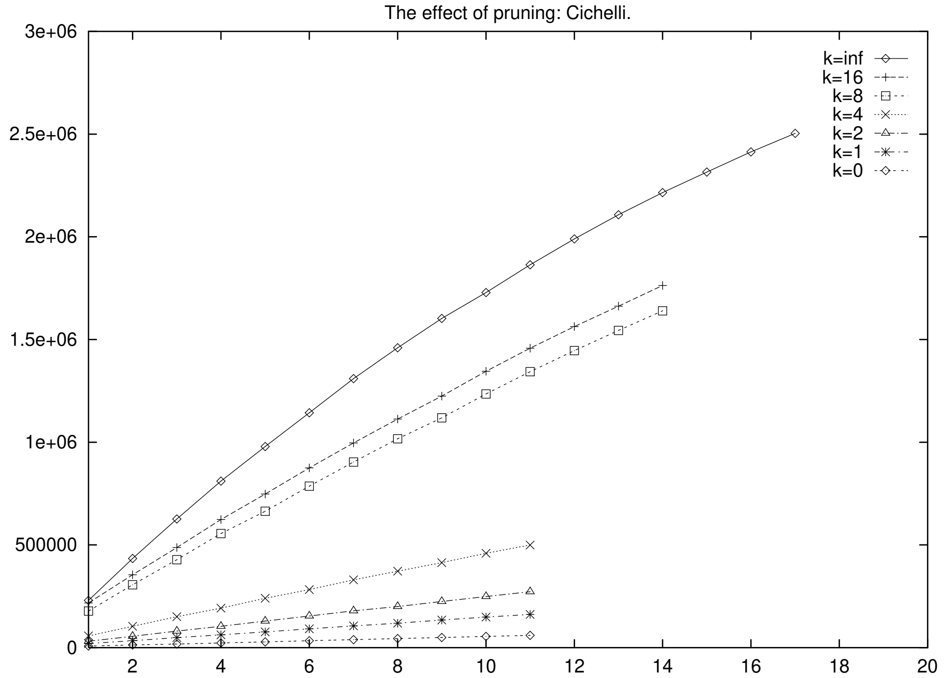


Figure D.9: The effect of pruning trails for the `Cichelli` *computation beyond a fixed length* k *at each garbage collection. The* x *axis shows the number of garbage collections, and the* y *axis the amount of live heap memory in words.*

native to fixing the value of k, programmers could specify only an overall bound on the size of heap memory. The implementation would then vary k automatically to achieve (only) the necessary degree of pruning.

D.5.4 Larger applications

So far we have looked at three quite different but modest benchmark computations, in which programs of up to a few pages in length perform up to a million reductions. Completing redex trails with a record of indirecting applications has not caused a blow-up in memory demands. Partial trails require far less memory than full trails and are still effective as sources of information.

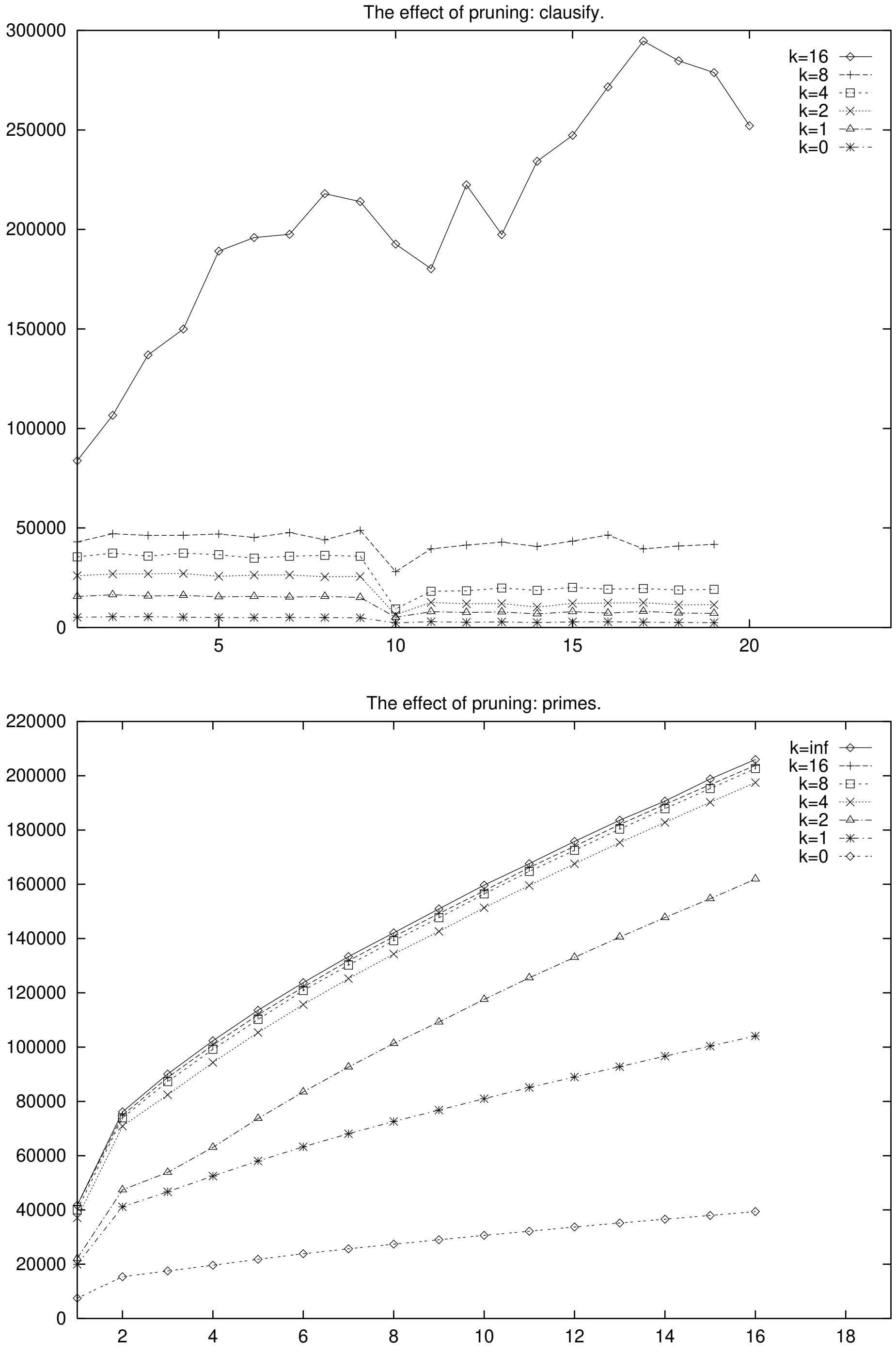


Figure D.10: The effect of pruning trails for the `clausify` *and* `primes` *computations. Again the* x *axis shows the number of garbage collections, and the* y *axis the amount of live heap memory in words.*

```
-- O. Wurzburg, Pittsburgh Gazette-Times, 1914
-- The classic pawn-promotion three-mover.

- - k - - - - -
r - - - - - - -
- - - p K - - -
- - - - P - - -
- - - p p b - -
- - - p - p - -
- - - - - - - -
- - - - - - - -

White to play and mate in 3
```

Figure D.11: The problem used as input to the `mate` *program. "A striking task achievement presented with due regard to artistic canons." [Phi32]. Chess-playing readers might like to verify that solving it requires a significant amount of computation!*

One motivation for partial trails is the wish to handle more demanding applications, beyond the capacity of our prototype system. We now give results for two such applications.

A Chess End-game: Wurzburg-`mate`

Our first larger application involves brute-force search to solve a chess end-game problem of the *White to move and mate in N* variety. The program, called `mate`, is not so very large — about 400 lines excluding `Prelude` definitions. But the problem we shall supply as input (shown in Figure D.11) is sufficiently tough that the computation requires over 20 M reductions.

As one would expect, the `mate` computation is dominated by functions testing and sifting board positions and moves. The results of many of these functions are not fresh constructions but components of arguments. So trails produced *without* indirections are far from complete: constructing a selective trail of that kind with the `Prelude` trusted requires no more than 0.5 Mb memory space. However, some of the most interesting aspects of the computation are not recorded!

So how much space is required to record the Wurzburg-mate computation

Table D.3: The magnitude of redex trails for the Wurzburg-`mate` *computation, with varying degrees of selectivity.*

trusted modules	# reductions in trail	memory needed
none	20,344 k	unknown
`Prelude`	9,827 k	52.7 Mb
`Prelude,Board`	8,371 k	51.2 Mb
`Prelude,Board,Move,Problem`	2,571 k	21.4 Mb

in a redex trail including the so-called indirecting applications? Without some selectivity, building the trail in heap memory is not feasible — by extrapolation, it would require over 100 Mb. Table D.3 shows the extent of various selective trails. Apart from the `Prelude`, the lowest level module in `mate` defines a `Board` type, and basic operations on it such as vacating a square, or occupying one with a given piece. At the next level up, the `Problem` module parses a problem statement into an initial state, and the `Move` module defines legal moves.

Even the most selective trail boosts the memory requirement from less than 10 kb for an untraced (but deliberately space-efficient) search to over 20 Mb — giving some insight into the number of moves and positions the 'suspected' `Solution` module deals with. Also, without tracing the program runs for about two and a half minutes, but with selective trail construction this increases to between 15 and 30 minutes.

Figure D.12 shows the effect for Wurzburg-`mate` of increasingly severe trail-pruning at each garbage collection. Light pruning ($k \geq 4$) is not enough: memory demands still grow steadily to tens of megabytes. But setting $k \leq 2$ curbs memory use very effectively: the self-tracing program runs in constant space below 100 kb.

A Haskell Compilation: nfib-`nhc`

Our second large-scale application is the `nhc` Haskell compiler itself. The compiler is a large program, consisting of about 100 modules with a total of over 14,000 source lines. When compiled for tracing, the compiler binary is 4.2 Mb in size, compared with 1.1 Mb for the non-traced binary. The compiler starts by reading, lexing, and parsing the Haskell input file. It then processes the interfaces for all imported files. The interface for the

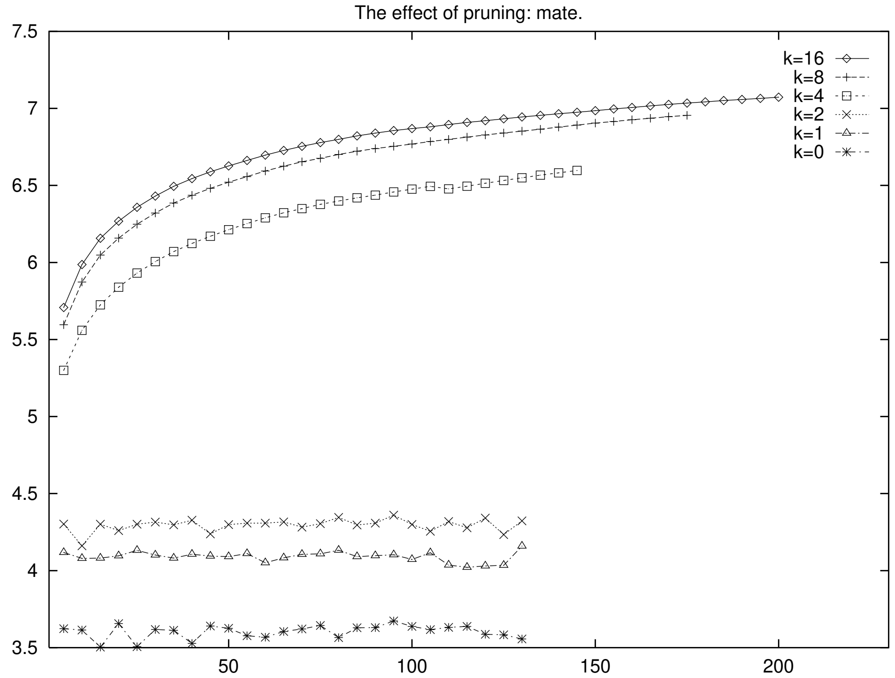


Figure D.12: The effect for the Wurzburg-`mate` *computation of pruning trail length to k at each garbage collection, for various values of k between* 0 *and* 16*. Only the* `Prelude` *module is trusted. The y-axis now uses a* log scale *with base* 10*: each plotted point (x, y) represents 10^y words of live heap memory recorded after x garbage collections.*

standard prelude is always processed. After that the source code (in the form of a syntax tree) is type checked and transformed in around 25 passes before finally the byte-code for the functions in the module is generated.

Traced programs are currently restricted to a limited repertoire of I/O operations. So for the purposes of tracing we modified the `nhc` compiler to take the source as standard input, and to deliver byte-code as standard output. Figure D.13 shows the small source module supplied as input. Even compiling something as modest as the `NFib` module `nhc` takes 5.2 M reductions. But when the `Prelude` is trusted, 4.3 M of these are omitted from the final trace, leaving only about 900 k recorded reductions. We are able to trace the compilation without further pruning using around 70 Mb of memory to store the redex trail. The redex trail for the compilation can be accessed using the display program by clicking over individual instructions or operands in the byte-code output.

```
module NFib where

nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) + nfib (n-2)
```

Figure D.13: Input for the nfib-`nhc` *computation.*

D.6 Related work

The problem of tracing functional computations has been tackled previously by numerous researchers. In his recent thesis [Wat96], Watson usefully reviews over twenty different schemes. But many proposed solutions have been unsuitable for full-scale applications. We concentrate here on a selection of previous work specifically addressing the problems of working with large computations.

Five years ago, Andrew Tolmach described in his thesis [Tol92] a comprehensive tracing and debugging system for the *eager* functional language ML. A shorter and more recent account is given in [TA95]. Tolmach's system, like ours, worked by compile-time transformation of the program. He also happened to use machines very similar to our own (MIPS workstations with about 100 Mb of memory). He successfully applied his system to programs of up to 750 source lines, performing computations of over 50 million *events* (such as the binding of a value to a name — events are not directly comparable with reductions, but at a similar level). The overhead in execution time was impressively low: even compared with highly optimised normal compilation, the slow-down was only about a factor of three. However, the basis of tracing in Tolmach's system was navigation through a *sequence of states*, where state information was recorded only at check-point intervals of many thousand events — up to 256,000 for the largest application. This interval, and the size of a *check-point cache* were the main ways of controlling memory use. Because of the sparse recording policy during traced execution, *repeated computation* was needed to provide information about events of interest to the programmer, whereas we record all available information in the trace structure. Though our aims are similar, at a detailed level our techniques are quite different, partly because of differences in evaluation strategy between an eager language such as ML and a lazy one such as Haskell.

In a paper at last year's IFL workshop [JM96] Jarvis and Morgan reported their experience extending the *cost-centre profiler* in Glasgow's GHC compiler [SP97] for use with large programs. Their motivating application was the LOLITA natural language system — at over 50,000 source lines one of the largest Haskell applications to date. Cost-centres are labels for parts of a program, added by the programmer as annotations (or by the compiler for default options such as one centre per top-level definition). Jarvis and Morgan introduced *cost-centre stacks* to gain precision when attributing costs of functions applied in many places, though the cost of maintaining distinct records for each permutation on the stack grows to tens of megabytes for LOLITA. As a side benefit, they found that the current stack of names provides a useful trace of the evaluation context when an error occurs in a large program. Full redex trails in our system carry all the information in a cost-centre stack, and far more besides.

Nilsson and Sparud use a trace structure similar in some respects to redex trails but with *forward* rather than backward links. To cope with larger computations they propose in [SN97] (and more fully in [NS96]) to build only some initial part of the trace when the program is run. When the programmer examines the trace, if a point is reached where the trace goes no further, the program is *re-executed* to obtain a trace extension at the point where it is needed. This process can be repeated many times. It is not clear how to use this incremental technique with backward traces, but the idea is interesting since it allows tracing in full detail for parts of a large computation.

We have already mentioned Watson's thesis [Wat96] for its useful review. Watson's own approach to the tracing problem also involves transforming the program to augment values with traces of their computation, but he chooses to construct low-level traces recording the *sequence of evaluation*. His interpretive prototype incurs very high costs in both space and time, so he cannot demonstrate large applications. However, in a later chapter discussing the problems of scale, Watson discusses selective tracing and proposes a scheme much like our own for excluding trusted applications in a trusted context.

D.7 Summary, conclusions and future work

We had two goals. The first was to extend our redex trail system so that in principle a *complete* record of a computation, including every reduction, can be made. In practice, complete records are often prohibitively expensive and overwhelm the user of the tracing system with too much information. So our second goal was to develop effective ways of specifying and constructing *partial* redex trails, recording only the reduction steps most likely to be of interest to the programmer. An important motivation for partial trails was to make tracing practical even for large computations.

We have attained both goals. Completeness has been achieved by extending our trail representation to incorporate indirections to exactly those applications previously unrecorded. By combining selective construction with controlled pruning we are able to trace much larger applications than before.

We conclude that the approach of compile-time transformation to generate redex trails is viable for a full-scale tracing system — something that implementations of lazy higher-order languages have lacked for too long.

We intend to develop the techniques further. As noted in Section D.5.3, it would sometimes be more convenient to specify a bound on the size of heap memory than on the length of trails; this should be straightforward using a two-pass pruning routine. We also plan an option to archive redex trails to file as the computation proceeds: this will not only save space in main memory, but also allow traces to be re-examined as often as the programmer wishes.

The price we currently pay for these tracing techniques includes greatly increased execution times: programs take about *fifteen times* longer to run when constructing redex trails than when executed normally. This is not acceptable. We have begun work on low-level replacements for the traced application combinators defined in Haskell. Another help towards faster partial tracing would be some way to link modules compiled to build redex trails with others compiled normally. It is not clear how to achieve such linkage as the types of traced and untraced versions of a function differ; we suspect that naive conversion (stripping trails from values, or adding null trails to values) would be too expensive. By whatever means, however, our aim is to bring the speed of traced computation within a factor of two or three of normal speed in most cases.

We have said very little about the way programmers examine traces, though it is an important issue. We have constructed a special-purpose browser for redex trails, as described in [SR97]. This browser is normally applied when a computation is over, with the resulting output or run-time error as the starting point. But trail-building computations can also be interrupted, in which case the browser is attached to the trail for whatever is currently being evaluated; this mode of use allows non-terminating computations to be investigated.

Ultimately any claim to practical success in the field of tracing must be tested by experiments measuring the effectiveness of tracing tools in actual programming tasks. We hope to conduct such experiments in the near future. Meanwhile we have in our sights what many implementors would regard as the ultimate test of a tracer they themselves would wish to use — constructing an informative trace of a boot-strapping compilation.

Acknowledgements

Niklas Röjemo built the `nhc` compiler that we have adapted for our experiments in tracing. The work reported in this paper was funded by the Engineering and Physical Sciences Research Council (ROPA grant GR/K64334). Related implementation work was funded by Canon Research Europe.

Addendum D.A: Implementing Adjustable Pruning

D.A.1 Introduction

Since paper **D** was written, we have done more work on pruning redex trails, and in this addendum we present that work.

In paper **D**, we present some methods of limiting the trace, to reduce the memory demands of the tracer, but also to filter out information from the trace that are not interesting to the user.

One of the methods is to prune trails at a certain distance k from the trail roots, when memory starts to run out. The pruning is performed at garbage collection time. At a slight time penalty, the garbage collector tracks trails and prunes the ones that are longer than k. When exploring a trace that has been pruned to length k, the user is guaranteed to be able to follow each trail at least k steps. When reaching a pruned part of the trace, the display program indicates that the information is no longer available.

We will present a method that automatically adjusts the k-value downwards when needed, so that there is always enough memory in the heap for the computation to make progress.

We will also present benchmarks that show how hard it is to predict a usable k-value. If the k-value is too high, the computation cannot proceed since the heap is full. If, on the other hand, the k-value is too low, then an unnecessarily large part of the trace will be pruned. What makes this problem even more serious, is the fact that using a certain k-value when pruning does not say much about how large the memory consumption will decrease, it very much depends on the program being traced. The benchmarks also show that the automatically adjusting scheme is viable.

D.A.2 Ordinary k-pruning

Trail pruning is very effective and makes it possible to use the tracing system for large computations. With small values of k (say, less than 5), the trail heap usage will be limited, thus allowing long-running computations, but of course the resulting trail will be less informative. For the common error that no clause in a function definition matches the actual argument (eg. `head []`), setting $k = 1$ is enough to be able to find out which function called the non-matching function (and the source position of the call). With $k = 2$, the most shallow part of the arguments will still be accessible. Since the last garbage collection may have been some time ago when an error occurs, even setting $k = 0$ is a worthwhile option, and the memory overhead in that case is only a small constant multiple of normal memory use. Our implementation allows setting the k-value to between 0 and 63 (we only have six bits available for storing the trail length).

In paper **C**, and then again in paper **D**, we considered three benchmark computations: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` transforms a proposition to an equivalent in clausal form; `primes` computes the first 2,500 primes using a wheel sieve. We will use the same benchmark computations in this addendum. Figure D.14 shows the live heap size during the execution of the `cichelli` program with different choices of k. Note that the x axis now shows the number of reductions (it shows in fact the number of function calls that has been started, but that is approximately the same as the number of reductions), instead of the number of garbage collections, as was the case in paper **D**.

Figure D.14 shows clearly that the amount of memory needed can be controlled by changing the k-value. When tracing with k=8, the heap size available to the program must be atleast around 1 million words. If it is less than that, the program will fail because it has run out of memory. We also see that to trace with k=12, the program needs around 3 million words. An interesting point is that for k=16 or k=63 the memory usage is almost exactly the same. Operationally this means that there are few trails with lengths beetween 16 and 63. Using the same reasoning, we can see that there are a lot of trails with lengths between 8 and 12 (using about 2 million words in the heap). As we mentioned earlier, when pruning with k=0 the memory usage will be a constant factor larger than than for the computation of the un-traced program.

Let us now consider the `clausify` program, again with a few choices of k-values, see figure D.15.

We note that the computations differ from the `cichelli` computations, in that this program does not leak (i.e., the memory usage for pruning with k=0 does not increase with time). The memory usage for pruning with k=1 and k=8 are proportional to the memory usage when pruning with k=0, but for larger values of k, it seems as if the memory usage steadily grows with time. We realise that this must be the case, since when tracing with no pruning at all (corresponding to $k=\infty$), the trace will monotonically grow with each reduction.

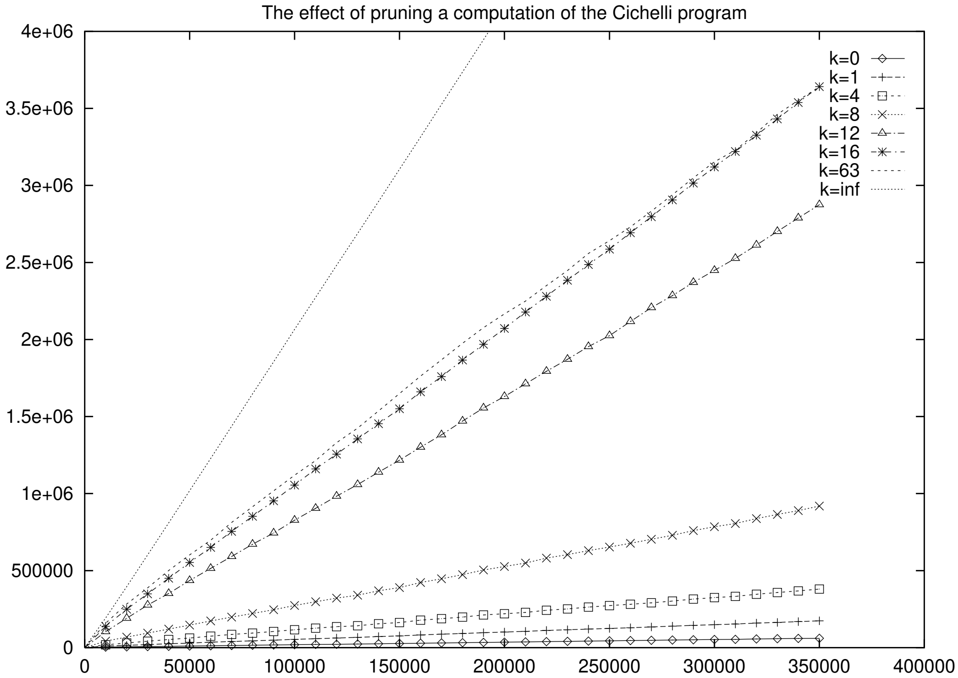


Figure D.14: The effect of pruning trails for the `Cichelli` *computation beyond a fixed length* k *at each garbage collection. The* x *axis shows the number of reductions, and the* y *axis the amount of live heap memory in words.*

D.A.3 Automatically adjusting k-pruning

The drawback with the k-pruning scheme is that the user has to decide a value for it when running a program. If he happens to use too large a value, the program will run out of heap space and he has to try again with a lower value. As we have seen, it is very hard to predict the heap usage of a computation, given a certain k-value.

On the other hand, by directly limiting path-length the user knows what losses to expect when it comes to browsing the trace.

To alleviate these drawbacks we have implemented an automatic or adjustable pruning scheme, which adjust the k-value automatically, based on how much free heap space is available. The algorithm is as follows:

To start with, no pruning is performed. If, after a garbage collection,

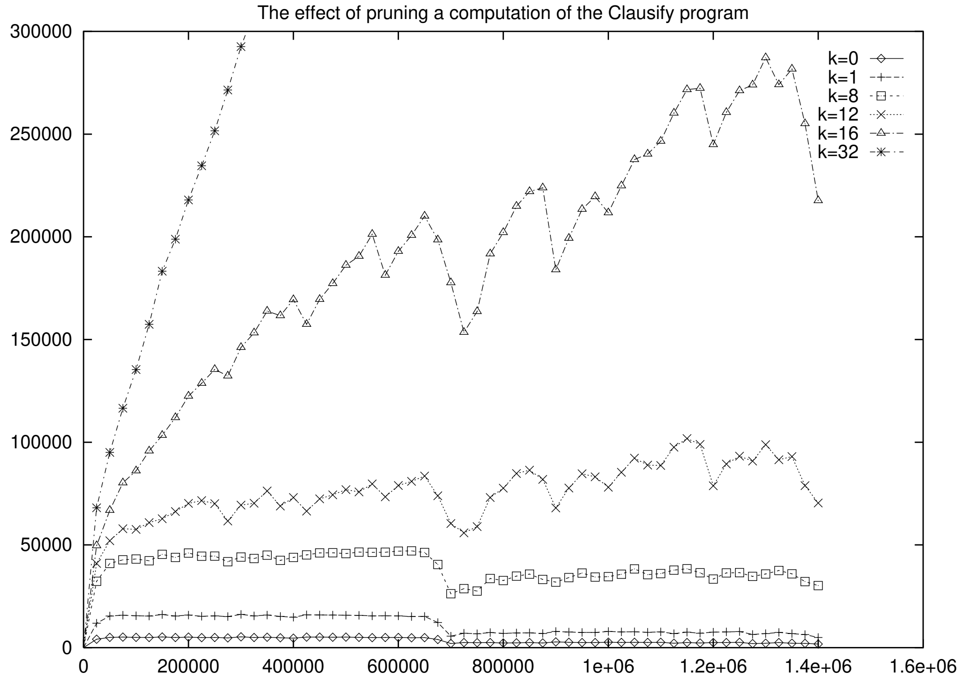


Figure D.15: The effect of pruning trails for the `Clausify` *computation.*

the free heap size is less than *fhs*% of the total heap size, the system sets the pruning distance k to the highest value (63) and prunes the trace using that value. If it still the case that there is less than *fhs*% free heap space, the k-value is decreased by multiplying it with a factor *kf* (less than one) and another pruning is performed. The free heap size test is then performed after each garbage collection, and the k-value is again decreased using *kf* if necessary. If the heap is exhausted when k is zero, the program is aborted with an out of heap error message.

This ensures that the program will be able to complete its task if it is at all possible within its given heap (including the constant overhead introduced by $k{=}0$-pruning).

After some testing, we have found that setting *fhs* to 20% and *kf* to $\frac{3}{4}$ give good results.

D.A.4 Benchmark computations revisited

We will now try the adaptable pruning technique on our benchmark computations.

D.A.5 The `cichelli` program

Figure D.16 shows the results for the `cichelli` program. There are two new things that are plotted: the live heap size when doing adjustable pruning with two different heap sizes.

In the first case the heap size is 0.5 million words. When the computation starts, no pruning at all is performed. If less then 20% of the heap (0.1 million words in this case) is free after a garbage collection, the system starts pruning the trace with the highest k-value. Figure D.16 shows that that happens after around 20 000 reductions. Between reductions 20 000 to 70 000 the k-value is decreased a few times, but then the same k-value is used until about 180 000 reductions. The final k-value is 2.

In the second case the heap size is 1 million words. In this case the final k-value is 6.

D.A.6 The `clausify` program

Figure D.17 shows the results for the `clausify` program. When doing adaptable pruning in this case, the maximum allowed heap sizes are 80% of 125 000 and 250 000 (100 000 and 200 000, respectively). As can be seen, in both computations a reasonable k-value is found rather early (after about 300 000 reductions).

D.A.7 The `primes` program

Figure D.18 shows the results for the `primes` program. When doing adaptable pruning in this case, the maximum allowed heap sizes are 80% of 500 000 and 2 500 000 (400 000 and 2 000 000, respectively). After about 400 000 reductions, the live heap size for the 2 500 000-word sized heap case decreases drastically, and when the program finishes, only about 700 000 words are used. This means that the automatically adjusting scheme decreased the k-value *too much* this time. The solution would be to decrease k slower, but for many other programs that would slow down the computation severely, since many garbage collection might be needed to find the "perfect" k-value for each actual live heap size.

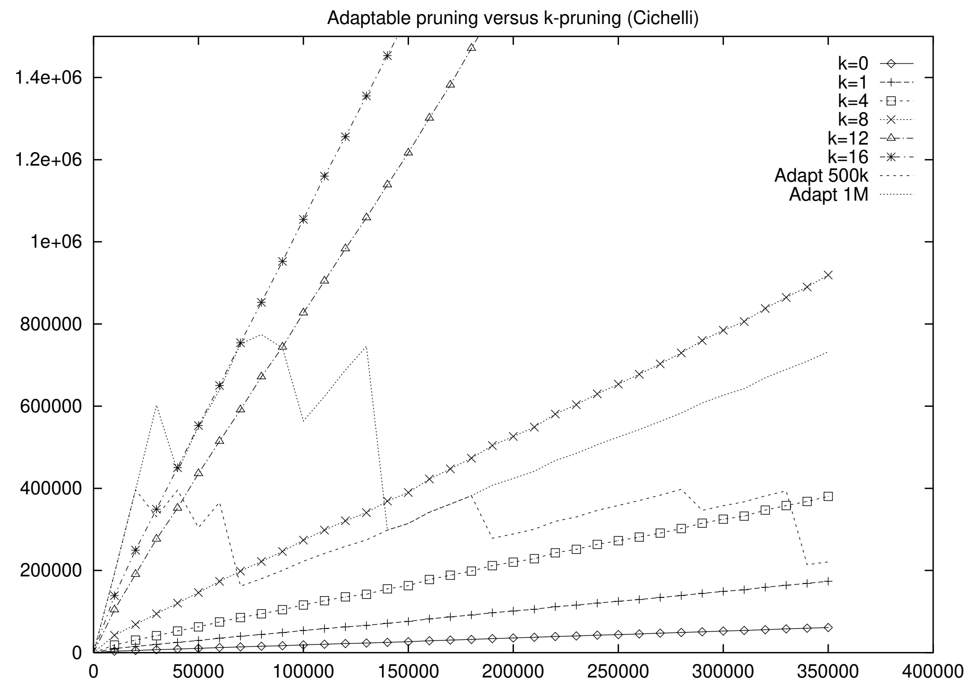


Figure D.16: Adaptable pruning of the `cichelli` *computation. The k-value is decreased a number of times in order to fit the the trace in the available heap space.*

D.A.8 Conclusions

Using the tracing approach described in paper **C**, methods of limiting the trace memory usage is absolutely necessary, for all but the smallest computations. Limiting the trace size by "trusting" modules or functions (also described in paper **C**) is very pleasing, since the user then can control which parts of the program he is interested (or not interested) in. However, for most computations, trusting is not enough. The brute-force method of pruning traces at a certain distance from the trace roots (k-pruning) can dramatically decrease the memory usage, but by doing that, the user risks pruning away trace information that he is interested in. Also, as have been shown in this addendum, it is very hard to come up with a reasonable k-value for a particular program. The adjustable pruning method very effectively takes

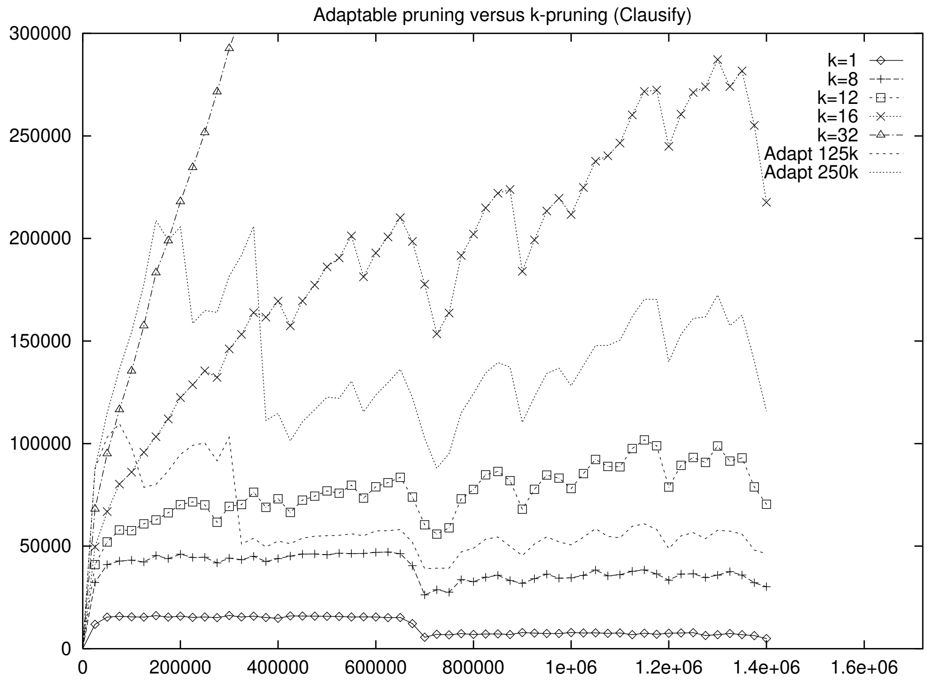


Figure D.17: Adaptable pruning of the `clausify` *computation. The k-value is decreased a number of times in order to fit the the trace in the available heap space.*

care of this problem.

Together, trusting and adjustable pruning give the user a weaponry to get the sought after information in many cases, but it must be admitted that there are also many cases when this is not true.

Instead of automatically adjusting the k-value given a certain heap size, it should be possible to do the other way around, i.e., automatically adjusting the heap size given a particular k-value. We have not further investigated this approach.

Another approach is to do what Nilsson [Nil98] proposes: create just a part of the trace, as much as fits in the available memory, and let the user start browsing it. When reaching the limit of the recorded trace, the program is rerun to produce the requested part. Nilsson reports that this method works very well. It is not straightforward to fit this strategy in our

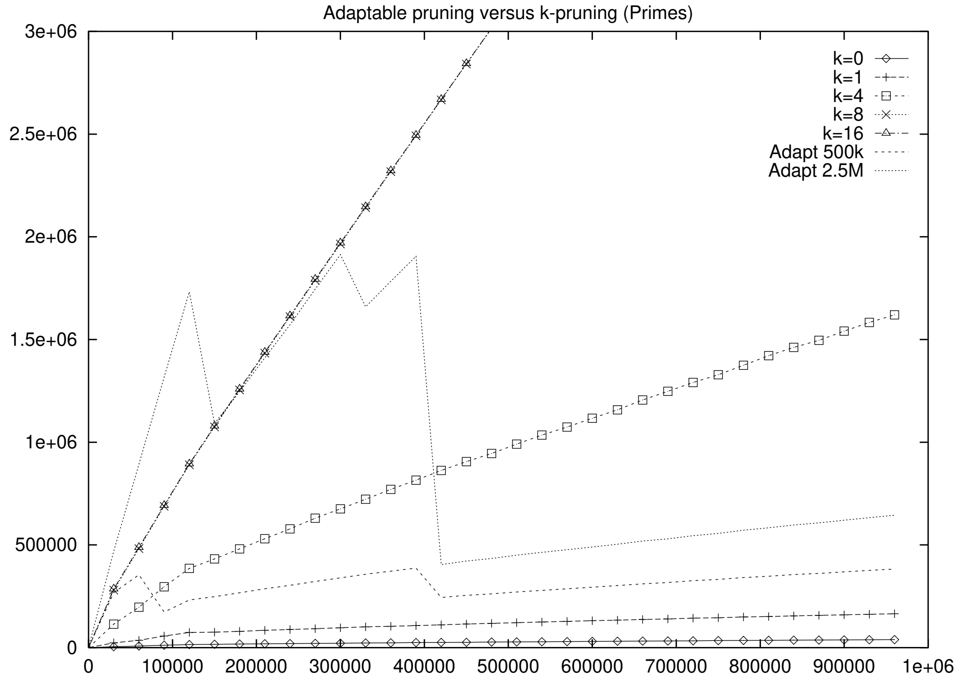


Figure D.18: Adaptable pruning of the `primes` *computation. The k-value is decreased a number of times in order to fit the the trace in the available heap space.*

tracing framework.

Finally, we have started investigating storing the trace on a secondary storage medium instead of in the memory heap, together with the normal program data. This idea is attractive, since it is often the case that more space is available on disk than in memory. Also, when a trace is saved on disk, it can be perused many times, and even sent to other people for investigation. No implementation has yet been made, but we see no serious difficulties with it.

Paper E

Redex Trail Representation and Access Protocol

Jan Sparud
Department of Computing Science
Chalmers University of Technology

E.1 Introduction

The basis of our tracing method is a program transformation carried out by the compiler: transformed programs compute the same values as the original, but embedded in functional data structures that also include redex trails showing how the values were obtained.

In [SR97] we describe an outline of the redex trail representation we are using, and also how the evaluation of Haskell expressions maps to this representation. In this section we will try to give some more details of this process. The intention is to explain the procedures accurately enough to enable other compiler writers to incorporate redex trail tracing in their lazy functional compilers.

In addition we will describe the interface between the runtime system, that has access to the redex trail structure, and the user interface, which is implemented as a separate program.

Some other modifications to the runtime system, e.g., changes to the garbage collector in order to perform trail pruning, will also be presented.

E.2 Redex trail representation

The datatype for redex trails presented in [SR97] was, for presentation purposes, a little simplified. The actual datatype we are using is given in figure E.1. We have tried to keep the representation as concise as possible. It has been changed a large number of times, because of demands from both the producing side (the run-time system) and the consuming side (the browser).

```
data Trace =
      Ap Trace Traces SR
    | Nm Trace NmType SR
    | Root
    | Ind Trace Trace
    | Sat Trace Trace
    | Hidden Trace
    | Pruned

type Traces = [Trace]
```

Figure E.1: The redex trail datatype.

An application (of a function or a constructor) is represented by `Ap` t_{ap} `[`t_f`,` t_{x_1}`,` ... `,`t_{x_n}`]` sr, where t_{ap} is the trace for the application itself, t_f is the trace of the function part of the application, t_{x_1} ... t_{x_n} are the traces of the arguments, and sr is a source reference, holding information about the position of the application in the source program.

A name (a function, variable, constructor or literal) is represented by `Nm` t_{nm} *name* *sr* where t_{nm} is the trace of the name, *name* is a static name information record (containing, amongst other things, a textual representation of the name and a source reference to the definition of the name). A name also has a dynamic source reference, just as the application node.

`Root` is the null trace attached to top-level names.

The `Ind` constructor is used when a function just returns an argument (or a sub-part of an argument) without *constructing* anything. The constructor contains two parts: the trace of the returned value and the current context. The trace browser has a number of possible ways to present this information, but ignoring it would not be truthful to the computation.

The `Sat` constructor is used for saturated function calls. The rules for building traces ensure that an unsaturated function call always has a trace, which starts with its normal form, i.e., the call itself. The trace for a saturated call should start with *trace of the result* of the call, which it also does, *if* the result is evaluated to weak head normal form. If not, the trace is unevaluated and we cannot get *any* information about it. The `Sat` constructor holds two traces; one for the saturated call, and one for the result of the call. So the `Sat` constructor could be seen as an exceptional forward pointer in the trace structure. As soon as the function return is evaluated to weak head normal form, its trace will be evaluated too, and the `Sat` node will no longer be needed. When traversing the trace structure (after the computation has finished), the result part will be presented to the user if it was evaluated, otherwise the call will be presented.

The `Hidden` constructor is just a place holder, saying that the trace it contains is part of a trusted computation. In an invocation of a trusted function in a trusted context, all expressions have the same trace: the top-level call to the trusted function. The trace browser needs information about this, in order to be able to present traces correctly.

Finally, the `Pruned` constructor is used for pruned traces. When pruning is used, the garbage collector prunes (i.e. overwrites with a `Pruned` constructor) traces that are longer than a certain value k (see paper D in this thesis).

E.3 Compiler generated information tables

When tracing a program the runtime system needs access to information about modules (names and dependencies) and identifiers and literals (name, position of occurrence and definition). This section explains how this static information is produced by the compiler and how it is accessed by the runtime system.

E.3.1 Source references

All identifiers and literals in a traced program have *source references*, which means that when browsing, it should always be possible to get to the place in the source code where a certain identifier or literal was used, or (for identifiers) to the point of definition.

Since there may be many source references for a program, it is important that they are stored in a compact way, yet allow easy access. A source reference contains an integer in which a line and column number are encoded, and a string which holds the name of the source file. The compiler generates a static table with all source references in a module, using a shared source file string, which caters for a very compact representation. Since source references are used from within the Haskell part of the runtime machinery, they must be ordinary Haskell data values.

E.3.2 Names and literals

The `Nm` constructor in the `Trace` datatype has a field for static name details of type `NmType`. The information about a name has been split in two parts in this way because some of the needed information about a name is dynamic (the trail of a particular occurrence and its source reference) and some is static (the textual description of the name, the position of its definition *etc*).

Compactness is very important for names and literals too. Names will be referred to by every redex trail node, so it is absolutely necessary that a name's (potentially large) static part is shared across all usages of the name.

Figure E.2 shows the datatype for name details:

```
data NmType =
   -- Identifiers
     NTTrustedId SourceModule SourcePos Name Priority
   | NTSuspectId SourceModule SourcePos Name Priority
   -- Basic types
   | NTInt Int
   | NTInteger Integer
   | NTFloat Float
   | NTDouble Double
   -- Special constructions
   | NTLambda
   | NTCase
   | NTGuard
   | NTIf
```

Figure E.2: The name information datatype.

The compiler generates a static table with all identifiers in a module. The identifiers come in two flavours: trusted or suspected. If a module is compiled to be trusted (using the `-trusted` flag), all identifiers will be trusted, otherwise they will be suspected. This is used for partial trail building, as described in [SR98]. The trustedness of a module or individual functions in a module can be changed at program startup time. This is implemented by traversing each module's identifier table, and, if needed, change the constructor for identifiers to `NTTrustedId` or `NTSuspectId`. The runtime machinery uses this constructor to decide if a full or a partial trail should be built.

The `SourceModule` part of an identifier is a string that contains the name of the module in which the identifier is defined. A single source module string is shared between all identifiers in a module. The `SourcePos` part is an integer in which the line and column number of the point of definition of the identifier are encoded. The `Name` part is a string, which is the textual representation of the identifier. The `Priority` part is an integer, which contains fixity and priority information for the identifier. It is used by the user interface when displaying expressions. It would be easy to add more fields for identifiers, that for example could be used for profiling.

The constructors for the basic types are not put in a separate table, instead the compiler inlines them at the places they occur.

The constructors for the special cases do not contain any information, they are just tags indicating which kind of special node they belong to.

E.3.3 Modules

To be able to change the trustedness of modules at runtime, every module's name table must be reachable from the main module. Because of this, each module contains a module information record with a reference to its identifier table and a table of references to the module records of the imported modules. Starting from the module record in the `Main` module, we can access all module records in the program, and from them we can access the identifier field for all identifiers in the program.

A module record has the following structure (using C syntax):

```
typedef struct _ModInfo {
    char            *srcfile;
    NmType          *idtable;
    struct _ModInfo *modinfo[];
    char            *modname;
} ModInfo;
```

E.4 The redex trail access protocol

When the traced program is stopped (either it has terminated or it has been temporarily interrupted), the runtime system opens a listen socket, waiting for a trace browser to connect. After a connection has been established, the trace browser talks to the runtime system using what we call the *Redex Trail Access Protocol (RTAP)*. Initially, the runtime system transmits information about the status of the traced program. The information consists of the following:

- The reason for the interruption in the program execution (an informative string).
- The program output, and references back into the trace structure from the individual characters in the output.
- If available, a reference to the redex that caused the program failure.

After that, the runtime system awaits commands from the browser, which will, directed by the users requests, ask for more information about the redex trails. The browser requests can be of the following kinds:

- Get information about a node.
- Get indirection information for a node.
- Get more information about a node for which the browser only has partial information about.
- Get a source file.

E.4.1 RTAP node syntax

The syntax of the node information sent from the traced program to the browser is as follows:

Node `::=` `(N` *refnr srcref module name defpos priority trefnr irefnr*`)`
  `|` `(A` *refnr srcref parts trefnr irefnr*`)`
  `|` `(C` *refnr srcref kind val trefnr irefnr*`)`
  `|` `(R` *refnr trefnr*`)`
  `|` `(D` *refnr*`)`
  `|` `(B` *refnr trefnr*`)`
  `|` `(H` *refnr trefnr*`)`
  `|` `P`

srcref `::=` `*` | *module srcpos*
refnr, *trefnr*, *irefnr*, *srcpos*, *defpos*, *priority* `::=` *int*
module, *name* `::=` *string*
kind `::=` `case` | `if` | `guard`
val `::=` *Node*
parts `::=` *Node**

Most node types have a *refnr*, which is a unique reference number identifying the particular node. Most node types also have a *trefnr*, which is the reference number of the redex trail parent of the particular node. Some types have an *irefnr*, which, if greater than zero, means that the node was handed to and returned by the redex pointed out by the *irefnr*.

An `N` node represents a name (a variable name, a constructor name or a literal), and contains information about the locality of the use of the name in *srcref*, which is `*` if no source reference is available, otherwise the module name and file position for the occurrence. It also contains the module name and position for the definition of the name. Finally it contains a priority level, which, if applicable, is used when pretty-printing infix expressions. A `N` node corresponds to the `Nm` constructor in the `Trace` datatype.

An `A` node represents a function (or constructor) application, and it corresponds to the `Ap` constructor in the `Trace` datatype. It contains a source reference and a list of items that are the nodes for the function part and the arguments.

A `C` node represents a superficial trace node for a conditional expression. *kind* says which kind of expression it is. Such nodes are encoded in the `Trace` datatype as applications (`Ap`) with special function names for `if`, `case` and guards, respectively.

An `R` node is a reference node. The traced program's runtime system knows which nodes it already has sent to the browser, and if a previously sent node is requested, instead of sending the node again it just sends a reference node. This node does not correspond to a particular `Trace` constructor.

A `D` node is a placeholder for some other node. A node may contain arbitrarily many sub-nodes. If a node has many sub-nodes, it can take a long time to send over all information about it. Furthermore, the node will take a lot of space when displaying it. Because of this, at a certain recursion depth we stop sending sub-node information, instead we send a placeholder. The placeholder is displayed as a box, □, in the browser, and the user can manually expand it (i.e., get more information about the node). This node does not correspond to a particular `Trace` constructor.

A `B` node is a "black-hole" node. It represents an undefined value, a computation that had started but was not finished when the execution was interrupted. In the browser, this node is displayed as $\perp$. This node does not correspond to a particular `Trace` constructor.

A `H` node is a redex trail from a trusted function application, corresponding to the `Hidden` constructor in the `Trace` datatype. Nodes that have this trail have been short-circuited to the top-level call of the trusted function. This means that nodes of this type cannot be displayed in an argument position of a redex, since the value (the top-level redex) might not even be of the same type. Instead, the browser displays a half-crossed box, ⍁, when the node is occurring in an argument position. The user can click on it, and see the top-level call to the trusted function in a separate trail.

Finally, a `P` node is a pruned node, a node that we have no information about. In the browser, it will be displayed as a crossed box, $\boxtimes$. It corresponds to the `Pruned` constructor in the `Trace` datatype.

Paper F

Using the Redex Trail System

Jan Sparud
Department of Computing Science
Chalmers University of Technology

F.1 The trace display program

To explore redex trails we have developed an interactive trail browser, which runs separately from the traced program. When a traced program terminates, instead of exiting it enters a waiting state, to allow the trail browser to connect to it. After that the user can select parts of the program output, and browse the trail of the expressions responsible for the selected piece of output.

If a runtime error occurs during program execution, the traced program enters the waiting state and when connected the faulting expression will be displayed in the trail browser.

If the program seems to be looping, the user can interrupt it, connect the browser and view the trail of the expression currently under evaluation. When the user disconnects the browser, program execution continues.

An example of the trace browser display can be seen in figure F.1.

The display consists of three major sub-windows:

- The trail display window. Here the user can explore the trail by clicking on interesting parts of it. They will then be expanded to show more information of the trail of the selected item. Colour-coding and boxing is used to show how different expressions relate to each-other (sharing,

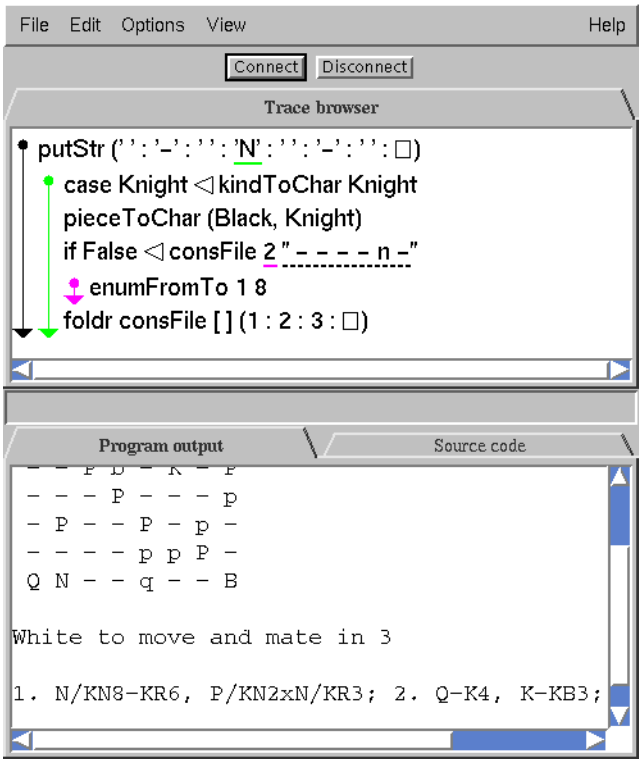


Figure F.1: Trail browser

common trail ancestors etc.) When the user moves the mouse pointer over the trail, the colour-coding and boxing is continuously updated.

- The source code window. Here relevant source code is shown. The user can point at any identifier or literal in the trail display window and immediately see the source code where the identifier or literal occurred. For identifiers, he can also get to the point of definition.

- The program output window. Here the user can explore and select parts of the program output. When some part is selected, the trail of the expression responsible for producing that part will be shown in the trail display window.

Because of screen space economy reasons, the source code window and the program output window share the same space, and only one of them can be active at any time. The user can click on a so called tab to select which one should be active.

At the top there are also two buttons, for connecting and disconnecting to the traced program.

F.1.1 How trails are explored

The trail display window shows trails in a tree structure. To start with, there is just one line with an arrow and a redex in it, see figure F.2.

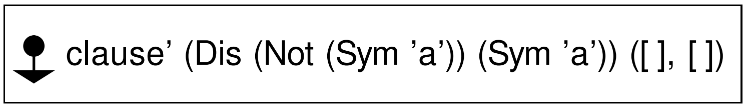


Figure F.2: Initial trace

In the case of a runtime error, the redex shown is the faulting expression. The user can click at the arrow to see the origin (the immediate trail ancestor) of the complete redex. The display will then be extended with a new line containing the ancestral redex, see figure F.3.

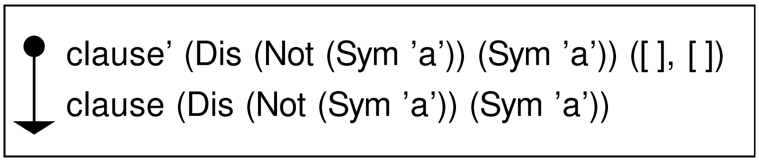


Figure F.3: Initial trace, after a click on the arrow.

The arrow will also be extended, and can be clicked on again to get the ancestor of the newly displayed redex. When the trail has come to an end (back to a top level definition) the arrow will be replaced by a horizontal line, indicating that the trail cannot be further expanded. For an example, see figure F.16 on page 179.

F.1.2 Context-sensitive display

A lot of information can be gained just by moving the mouse pointer over the redexes and their sub-parts. The currently selected item (the item under the mouse pointer) will be marked with a box around it. For example, in figure F.4 the current item is the value `Not (Sym 'a')`.

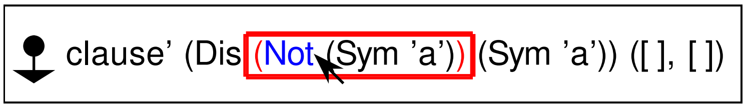


Figure F.4: The currently selected item is boxed.

If a redex or a value currently on display has the same ancestral redex as the currently selected item, it will be displayed in blue colour instead of the normal black color. An example of this is the `Not` constructor in the selected value, which has the same ancestral redex as the complete value `Not (Sym 'a')`.

F.1.3 Trails of redex sub-parts

Now let's say that we are interested in the origin of the selected value. We then just click on it. The expression will be underlined in a certain colour (in this case green), and a new trail will be displayed below the line where the selected item is. The trail will have a new arrow, with the same colour as the underlining of the selected item. Figure F.5 shows the display at this point.

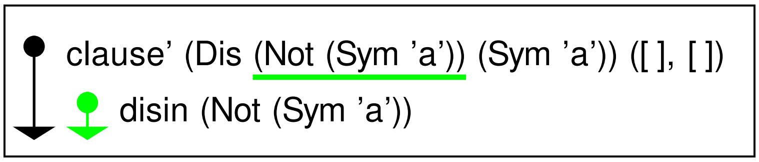


Figure F.5: A new trail has been created for a sub-expression.

If you click on an item a and get a trail redex b it should be interpreted as: a is the full result or part of the result of the call to the redex b. Currently there is no way of knowing if it's the full result or just part of the result, but we have plans of distinguishing the two cases.

F.1.4 Effects of sharing

If we move the mouse pointer to the `Not` constructor, we see something interesting. It is boxed as before, but the `Not` constructor on the next line is *also* boxed. The reason is that the two occurrences are shared, i.e., they are not just equal to each other, they actually share the same space in memory. This is an operational observation, but can often help the user understand the computation. We will in a later example see an example of this (figure F.27 on page 188).

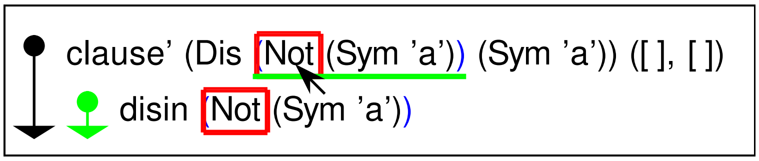


Figure F.6: Shared values.

To see the difference between shared and non-shared occurrences, we move the pointer to the leftmost `Sym` on the top line. We then see that it is shared with the `Sym` on the lower line, but not with the `Sym` further to the right on the top line (see figure F.7).

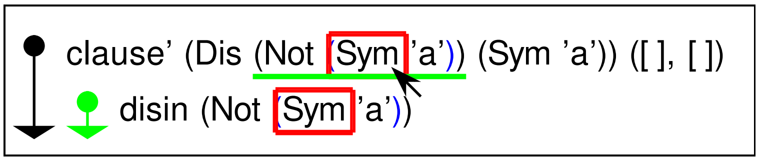


Figure F.7: More shared values.

F.1.5 Coupling between the display and the source code

Often it is not enough just to see the relationships between values and expressions in the program execution, some coupling to the source code is needed. For example, in the topmost redex we have a call to the function `clause’`. Where in the program did we make that call? What does the function do? When you click with the right mouse button on an identifier the source code window will be updated to show the part of source code where the identifier occurred, and the cursor will be placed on the selected identifier. If we right-click on `clause’`, the source code window will look as in figure F.8. To get to the point of definition of an identifier, you right-click as before, but with the shift key pressed.

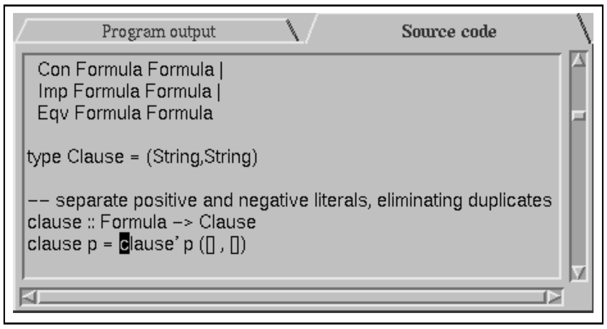


Figure F.8: Coupling to the source code.

F.1.6 Screen space economy

Since the whole trail often is very large, and since we at any time can only view a small part of it, we must try to be economic with display space. For that reason, it would be excessive to show the same trail multiple times in the display. To give the user a chance of not exposing the same trail twice, the immediate ancestor to the currently selected item will be high-lighted with a yellow background if it is already on display. It gives a signal to the user that it is unnecessary to expand the trail of the selected item since it is already available. See figure F.9 for an example.

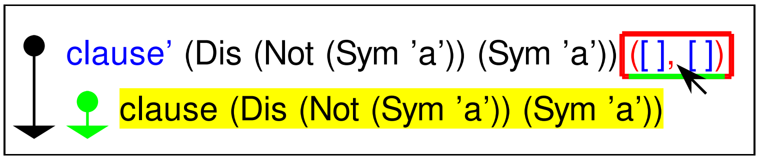


Figure F.9: High-lighting of the immediate ancestor.

F.1.7 Handling structurally large values

The display program must be able to handle structurally large values, e.g. a 100 000 element list. Note that in the case of cyclic structures it is crucial that we interrupt the display at some stage. Our solution to this is to introduce placeholders instead of actual values, whenever the data structure becomes deeper than a certain level. The placeholder looks like an open box, □, and the user can click on it (using the middle button on the mouse) to expand its contents, again just up to a certain depth. Figure F.10 shows an example where the tail of a list has been displayed using a placeholder.

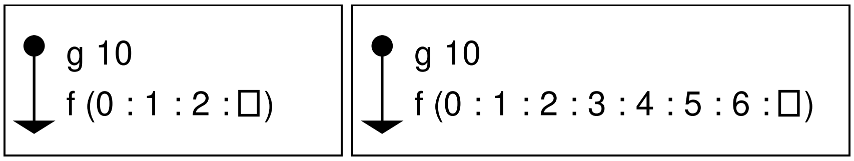


Figure F.10: Using a box as a placeholder for structurally large values. The second figure is the result of clicking the box in the first figure.

The user can also manually contract or expand values, by clicking on them with the middle button. This is useful when the user wants to suppress the display of large uninteresting values.

F.1.8 Side-effects of trusting functions.

If a function is trusted, then calls made from within that function are not traced (untrusted calls from trusted functions are still traced though). For example, the `length` function in the prelude is normally trusted, which means that the trail of the resulting integer (the length of a list) is just the redex `length` l, where l is the list in the top-level call to `length`. Had `length` not been trusted, all internal calls (recursive calls and calls to addition) would still be in the trail. This has the unfortunate effect that closures for values produced by an internal call in a trusted function will have the top-level call to the trusted function as the trail (remember that if a redex is unevaluated we cannot show its value, instead we show the redex). This means that a displayed value doesn't necessarily have the correct type. To come around this, we have chosen to introduce a second placeholder, a box with dash in it, ⧄. It should be interpreted as: this value cannot directly be expanded, but you can see its origin. If you click on the box, a separate trail will be displayed beneath the current line, showing the redex of the trusted function. Figure F.11 shows an example where the dashed box is introduced. The list has 11 elements and an unevaluated tail, which is a recursive call within the trusted function `from`. Instead of showing the redex, we show a dashed box. In the figure the user has clicked on the box, to reveal its origin, which we can see is the top-level call `from 0`. As can be seen, it would not be correct to replace the box with the redex.

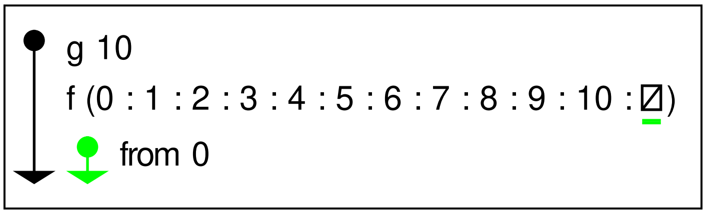


Figure F.11: Using a placeholder for displaying closures with trusted applications.

F.1.9 Effects of pruning

When pruning is used, parts of the trail are discarded, and the display system must somehow deal with this. To denote a pruned trail we use a crossed box, ⊠. The intuition is that it is like a closed window (c.f. to the open box and the dashed box above). The user cannot click on the crossed box, since there is no information about the trail it stands for. Figure F.12 shows an example including pruned trails.

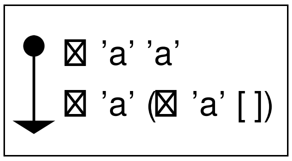


Figure F.12: Pruned traces are denoted by crossed boxes.

In the example, the function part of each redex have been pruned, but the *application* of the functions, as well as the arguments, are still in the trace. Since the application traces are still available, we can source reference them to see where the functions were applied.

F.1.10 Trails of control-flow within functions

The control-flow in a function is determined by conditional expressions, case expressions and guards. Even though they are not part of the redex trail, as we have defined it before, it is often desirable to see why a certain branch was taken. For example, the problem in a function might not be that it returns a strange value, but that a test is erroneous which makes it select the branch that returns the strange value. For that reason, we have chosen to include control-flow information in the redex trail, using a special notation:

control-flow $\lhd$ *containing redex*

It should be read as "the control-flow item *within* the containing redex". We have three different control-flow types: conditional expressions (denoted by `if`), case expressions (denoted by `case`), and guard expressions (denoted by `|`). The type name is accompanied with the expression that decided the outcome of the control-flow test. We will show how this looks in the display by means of an example, see figure F.13.

```
f :: Int -> String
f x | x > 0 =
    if x < 0 then
        ""
    else
        case g "Hello" of
            c:cs -> g "World"

g :: String -> String
g ('H':xs) = xs

main = print (f 1)
```

Figure F.13: An example program to demonstrate how control-flow decisions are presented in the display program.

When we run the program, we get an error when calling `g` with the argument `"World"`, since `g` is not defined to accept that argument. When connecting the trail browser, the initial trail display is, as usual, the failing expression, i.e., in this case `g "World"`. If we expand the trail one step, we see that the call was within a case expression where the scrutinized value was `"ello"`, which was within a conditional expression where the test was false, which was within a guarded branch of the function `f`, and the argument to `f` was `1`. Figure F.14 shows how the display looks. In the figure we have also requested the trails for the expressions controlling the flow. If there are several guards in a function definition, all the failed guard expressions will be shown as well as the succeeding one.

The user can select not to see control-flow items in the redex trail, by setting an option in the Option menu in the display program.

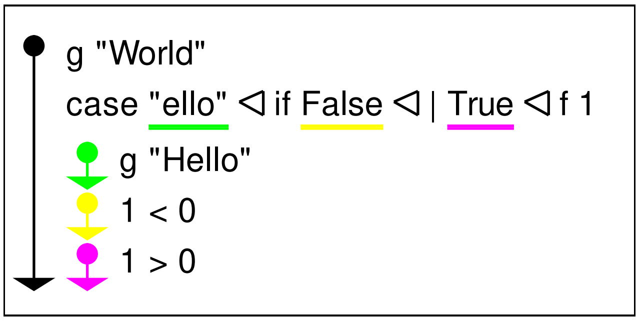


Figure F.14: Showing the control-flow.

F.1.11 Undefined values

In a lazy language, a redex will not be evaluated if its value is not needed. From that follows that all redexes in a program will either be unevaluated or fully evaluated, when the program finishes successfully. In some cases though, the program is terminated because of a runtime fault. It can then happen that for some redexes evaluation has begun, but not yet resulted in a return value. We call the result of such redexes *undefined* or *bottom*, and denote it by $\perp$. A typical case where this happens is when we get a so called *black hole*, i.e., in order to evaluate a redex to weak head normal form, its own weak head normal form is needed. This is normally because of a circular dependence between one or many functions. This kind of non-termination can be detected by most implementations of lazy languages, but the error message is often extremely uninformative and it is often very hard to find the source of the problem. Figure F.15 shows a program that demonstrates the problem with black holes.

```
a = 1 + b
b = c + 2
c = a + 3

main = print a
```

Figure F.15: An example program to demonstrate how undefined values are handled.

In the program, `a` is defined in terms of `b`, which is defined in terms of `c`, which is defined in terms of `a`. When this program is run, it will be terminated with an error message saying that a black hole has been detected. When we connect the browser we see (cf. figure F.16) that the faulting expression is $\perp$ `+ 3`, that occurs in the definition of `c`.

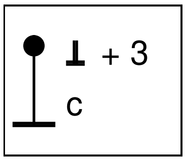


Figure F.16: A black hole has been detected.

If the user clicks on the bottom symbol $\perp$, the redex of the undefined value will be displayed beneath the current line in a separate trail. In this case it is easy to spot the circularity in the program just by expanding the trail twice and by moving the mouse pointer over the three different undefined values, see figure F.17. The information gained by high-lighting of the immediate ancestor reveals the cycle. We can also see that the undefined values correspond to the variables `a`, `b` and `c` respectively.

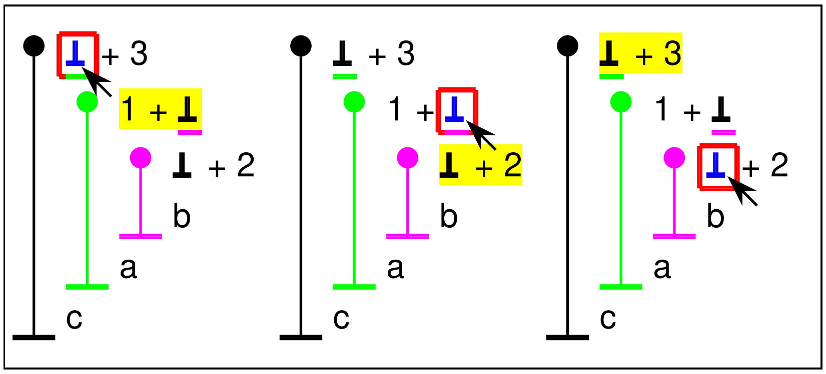


Figure F.17: Exploring the trail of the program reveals information about the circularity.

F.1.12 Tracing from the program output

A program does not have to incur a runtime fault to be erroneous; the problem might be that the output of the program is incorrect. In that case the program output will be displayed in the Program output window in the browser. The user can then select and click on some piece of the output, and the redex trail of the code responsible for that piece of output will be displayed in the trail window.

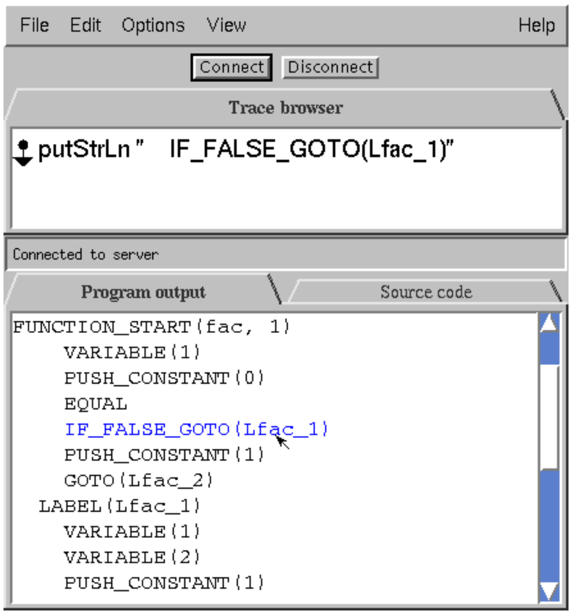


Figure F.18: Tracing from output

Figure F.18 shows an example of tracing from output. The program emits stack code instructions, and the user has selected a piece of the output. When clicking, the trace of the expression responsible for the selected piece will be shown in the trace window.

F.2 Comparing with a related tracer: Sorting a binary tree

Henrik Nilsson presents in his PhD thesis [Nil98] a debugger for a small lazy functional language called Freja. As an example, a program with a few bugs injected is discussed, and the reader is walked through a session with the debugger for Freja, in order to discover and correct the bugs.

In this section we will do the same exercise, but use our tracer instead. The reason for using the same example as Nilsson is twofold. Firstly, the example program is a very good one, in that it contains some of the most common types of bugs. Secondly, by doing the same exercise in our system, we get a good picture of the similarities and differences between the two systems.

F.2.1 The program

The example program is adapted from Johnsson [Joh87], and makes use of a circular programming style. The purpose of the program is to take a binary tree where the tips contain elements of a type on which a total order is defined, and return a structurally identical tree where the tips have been sorted according to the total order. This sorting will be performed using just one traversal of the tree. We can solve that in a lazy language by using circular programming. The basic idea here is that the 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 result of the previous ones, into a single traversal, in effect computing all results in parallel. Johnsson shows in [Joh87] that such algorithms can effectively be formulated through an attribute grammar. The grammar can then easily be transliterated to a lazy functional program, where the laziness ensures proper propagation of inherited and synthesized attributes.

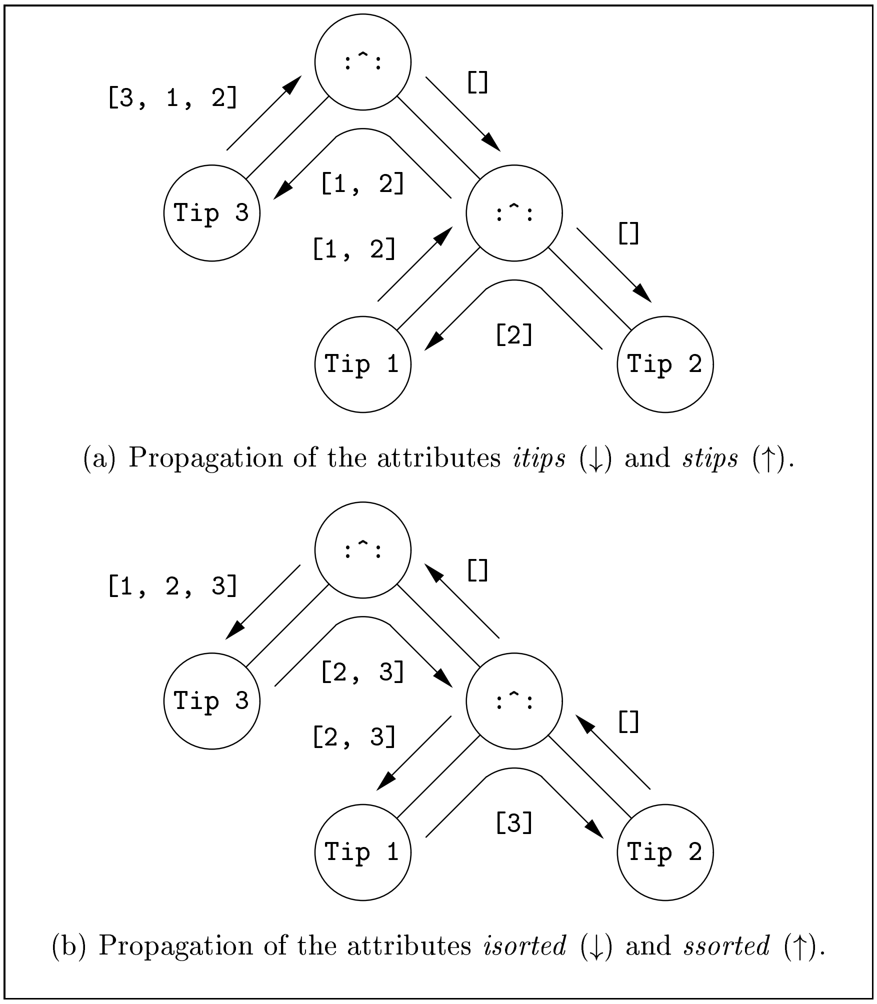
An attribute grammar for our problem is given in figure F.19. S is the start symbol. It has one synthesized attribute, *tree*, which holds the result of the computatation, i.e., a sorted version of the tree. T has five attributes, two inherited and three synthesized. The inherited attribute *itips* and 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. Figure F.20 illustrates the attribute propagation for a small tree.

In order to transliterate the attribute grammar into a Haskell 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. The inherited attributes become additional arguments

Productions	**Attribute equations**		
$S \rightarrow T$	$T{\downarrow}itips$	$=$	`[]`
	$T{\downarrow}isorted$	$=$	`sort` $T{\uparrow}stips$
	$S{\uparrow}tree$	$=$	$T{\uparrow}tree$
$T \rightarrow$ `Tip` x	$T{\uparrow}stips$	$=$	x `:` $T{\downarrow}itips$
	$T{\uparrow}ssorted$	$=$	`tail` $T{\downarrow}isorted$
	$T{\uparrow}tree$	$=$	`Tip (head` $T{\downarrow}isorted$`)`
$T \rightarrow T_L$ `:^:` T_R	$T_R{\downarrow}itips$	$=$	$T{\downarrow}itips$
	$T_L{\downarrow}itips$	$=$	$T_R{\uparrow}stips$
	$T{\uparrow}stips$	$=$	$T_L{\uparrow}stips$
	$T_L{\downarrow}isorted$	$=$	$T{\downarrow}isorted$
	$T_R{\downarrow}isorted$	$=$	$T_L{\uparrow}ssorted$
	$T{\uparrow}ssorted$	$=$	$T_R{\uparrow}ssorted$
	$T{\uparrow}tree$	$=$	$T_L{\uparrow}tree$ `:^:` $T_R{\uparrow}tree$

Figure F.19: 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. ↓ indicates an inherited attribute, ↑ a synthesized one.

to the function, and the synthesized attributes are returned as the result, packed in a tuple when there are two or more. The result of transliterating into Haskell is shown in figure F.21. For the sake of this example, the transliteration was done rather carelessly, resulting in a few mistakes. We will use our tracer to find them in the next section. We will also compare our experiences with the ones presented in Nilsson's thesis.



(a) Propagation of the attributes *itips* (↓) and *stips* (↑).

(b) Propagation of the attributes *isorted* (↓) and *ssorted* (↑).

Figure F.20: Attribute propagation for a small tree.

```
data Tree a = Tip a | Tree a :^: Tree a deriving Show

sort :: (Ord a) => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: (Ord a) => a -> [a] -> [a]
insert x [] = []
insert x xxs@(x':xs) | x < x' = x : xxs
                     | otherwise = x' : insert x xs

sortTree :: (Ord a) => Tree a -> Tree a
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' :: Tree a -> [a] -> [b] -> ([a], [b], Tree b)
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 :: Tree Int
aTree =
  (Tip 7 :^: (Tip 2 :^: Tip 5)) :^: (Tip 3 :^: Tip 1)

main = print (sortTree aTree)
```

Figure F.21: A Haskell program for solving the the tip sorting problem using only one tree traversal. The program is a transliteration of the attribute grammar in figure F.19, but contains a few bugs.

F.2.2 Running the program

When we run the program, it terminates quickly with the error message `Black hole detected`. The runtime system has caught the error, and awaits the user to connect to it with the trace display program. When connected, the trace display shows the redex that was being evaluated when the error occurred. Figure F.22 shows how it looks in the display program.

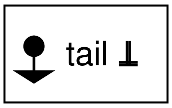


Figure F.22: Initial trace.

It seems that the prelude function `tail` was called with an argument that has become a black hole. There are a number of questions we can ask ourselves now. The most straight-forward one: What's wrong? the tracing system unfortunately cannot answer, but it can give us useful information that aids us in finding the problem.

By using the source reference mouse button, we can see that the failing call to `tail` was done in the local definition of `t_ssorted` in the `sortTree'` function (line 24 in figure F.21), and that the argument (which is undefined) is in fact `t_isorted`. We can also get that information by doing some trail expansions. The result of doing that is shown in figure F.23.

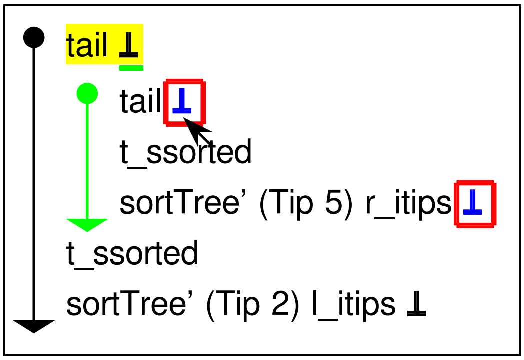


Figure F.23: The trace after a few trail expansions.

The second argument to `sortTree'` is interesting; it is `l_itips` and not a value. This means that `l_itips` was never evaluated to WHNF. Here our display differs from Freja's. If we encounter an unevaluated expression, we show the expression itself, where Freja instead shows a question-mark. Nilsson remarks that since the expression wasn't evaluated, it cannot possibly have influenced the computation in any way, and in particular, it cannot be the source of the problem. This is a very important point, since it decreases the number of suspects when trying to locate an error. On the other hand, the very fact that some expression has not been evaluated could give a vital clue, so we think it is an advantage to be able to see what the expression is instead of just a question-mark. If the user really is uninterested in the expression, and it just consumes display space, it can always be folded manually into a box (see section F.1.7).

Another important observation we can make in figure F.23 is that the undefined value that is the argument of `tail` is *the same as* the third argument to `sortTree'`. We know this since when the mouse pointer is over the first undefined value, the second one is also boxed, which means that they are shared. By source referencing the call to `sortTree'` (on the last line in figure F.23) we see that the call was made in `sortTree'` itself (line 29 in figure F.21), and that the third argument is `l_isorted`. Being able to see which values are shared can be very informative when looking for errors. Freja has no notion of sharing.

We now have enough information to find out what is the root of the problem. `l_isorted` is defined as `t_ssorted`, which is defined as `r_ssorted`, which is the second component of the result of `sortTree' (Tip 5) r_itips r_isorted` (line 30). That component is defined (line 21 and 24) as `tail t_isorted` which is the same as `tail r_isorted`. `r_isorted` is defined as `l_ssorted`, which is the second component of the result of `sortTree' (Tip 2) l_itips l_isorted`. That component is, as was the case in the right branch, defined as `tail t_isorted` which is the same as `tail l_isorted`. So, `l_isorted` is defined as `tail (tail l_isorted)` and we have found the source of the black hole. If we compare the definitions of the involved variables with the grammar, we see that `l_isorted` mistakenly has been defined as `t_ssorted` instead of as `t_isorted`. Figure F.24 shows a by-hand derivation that explains the cycle in the program. It reveals that `l_isorted = tail r_isorted = tail (tail l_isorted)`.

The reasoning around the discovery of where the fault is a bit involved, that must be admitted, but it is often the case with black holes that it is hard to quickly pinpoint where the real problem is, since it could be in any of the definitions that are circularly dependent. On the other hand, the tracer gave us a very good starting point for finding the problem. When locating

```
l_isorted
= t_ssorted (line 34)
= r_ssorted (line 36)
= #2 (sortTree' (Tip 5) r_itips r_isorted) (line 30)
= t_ssorted (line 21)
= tail t_isorted (line 24)
= tail r_isorted (since r_isorted is the third argument to sortTree'
in the call on line 30, which is bound to t_isorted on line 20).

r_isorted
= l_ssorted (line 35)
= #2 (sortTree' (Tip 2) l_itips l_isorted) (line 29)
= t_ssorted (line 21)
= tail t_isorted (line 24)
= tail l_isorted (since l_isorted is the third argument to sortTree'
in the call on line 29, which is bound to t_isorted on line 20).
```

Figure F.24: A derivation that reveals the cycle leading to the black hole.

this problem with Freja, you start from the main expression, and walk down the (strictified) call stack, answering questions whether right hand sides of equations are correct, given the values on the left hand sides. It can take some time to get down to the part of the program where the real problem is. Freja has the advantage though, that it can automatically locate where the fault is, if you can answer all the questions correctly.

F.2.3 Non-termination

We fix the problem, and rerun the program. To our disappointment, it still doesn't work properly, since it produces no output at all. It seems to be stuck in a loop. After waiting for a while, we interrupt it, to see what it is doing. When interrupting a program, it can be connected to by the browser, and when disconnected from it will continue running. There is thus no need to rerun the program, just to see what it is doing at the moment. When connecting the browser, the redex that was currently under evaluation will be displayed as the starting point. Figure F.25 shows the trail at this stage.

We expand the trail a few times to see some context, the result is in

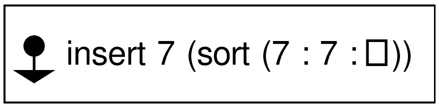


Figure F.25: The expression currently under evaluation when the program was interrupted.

figure F.26.

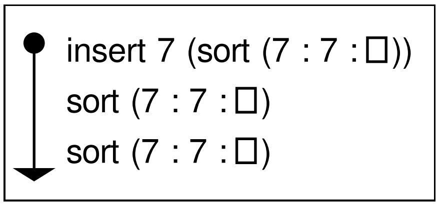


Figure F.26: The expanded trail.

It looks as `sort` calls itself with the same argument, over and over again. But is the argument the same? Because we can detect shared values, we move the mouse pointer to the list and see something very interesting.

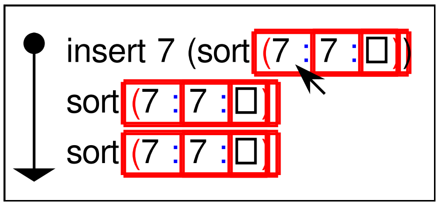


Figure F.27: Sharing in the list.

We can see by the boxing that the lists are indeed the same. Moreover, the tails of the list are shared with the list itself! This means that the list is cyclic, i.e., it is infinite. No wonder it takes a long time to sort it then. In Freja there is no way to tell that the list is cyclic, and thus infinite. The only thing you can do is to print it out deep enough that you are convinced "beyond reasonable doubt" that the list is infinite.

So, how has the infinite list come to be? We select it and click on it to see its trail (see figure F.28). We then see that the list is `t_stips` and by source referencing it we see that it is defined as a local variable in `sortTree'`.

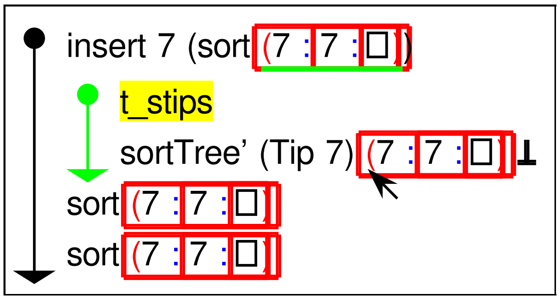


Figure F.28: Expansion of the list trail.

We also see that the second argument to `sortTree'` (`l_itips`) on line 29 in figure F.21) is the same as `t_itips`. But `l_itips` is defined as `l_stips` which is the first component of the `sortTree'` call, which is `t_stips`, thus we have found the cycle. Again we check the definitions of the involved variables against the grammar and find that `l_itips` on line 32 should be defined as `r_stips`, and not as `l_stips`.

F.2.4 Pattern match failure

We correct the error, recompile and run the program again. To our disappointment, it still doesn't work correctly. This time we get a runtime error `Prelude.tail: no match`. Again, we connect the browser and this time the starting point is a call to `tail` with an empty list as argument, see figure F.29.

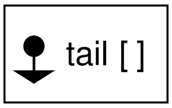


Figure F.29: Pattern match failure.

If we source reference the `tail` call, we see that it is the one on line 24. Apparently `t_isorted` is empty. It should contain the sorted list of tip values, and should not be empty at this point. So why is it empty? We source reference the empty list, and see that it is the empty list returned by `insert` at line 8. There we have the problem! `insert` should insert a value at the correct place in a sorted list, so that the list is still sorted. But here, when inserting a value into the empty list, the value is omitted. Instead it should return the singleton list. This time we found the error very quickly, the only thing we had to do was to follow two source references. In Freja, the debugger asks a few questions from the top down, then starts asking questions whether recursive calls to `sort`, with shorter and shorter lists as argument, really should return the empty list. After that, the debugger concludes that the bug is in `insert`, again requiring the programmer to answer the questions correctly, though it is not very hard to do so in this example.

F.2.5 Almost there

Surely, that must have been the last error. We fix the problem, compile and run the program again. This time we get some output, but then we get a runtime fault again. The output is shown in figure F.30.

```
Tip 2 :^: (Tip 3 :^: (Tip 5)) :^: (Tip 7 :^: (
Error: Prelude.head: no match
```

Figure F.30: The output of the fourth version of the sort program.

For (hopefully) the last time we connect the browser, and, unsurprisingly, get a call to `head` with the empty list as argument as the starting trail, see figure F.31.

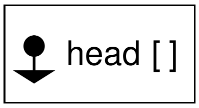


Figure F.31: Another pattern match failure.

We start as usual to source reference the function in the failing redex. This time we get to line 25, where it says `head t_ssorted`. We immediately see that this is not correct. At that line we should be annotating the tips in the sorted tree with the inherited list of sorted values, but we are taking the values from the synthesized list of values! Checking against the grammar again we find that `t_ssorted` should be replaced with `t_isorted`.

F.2.6 Finally, success!

Confident that the program now works we compile and run it, and sure enough, the output (see figure F.32) is what we expect.

```
Tip 1 :^: (Tip 2 :^: (Tip 3)) :^: (Tip 5 :^: (Tip 7))
```

Figure F.32: The output of the final, correct version of the sort program.

F.2.7 Summary

The main difference between our tracer and Freja is that our strategy is bottom-up while Freja is top-down. With our strategy it is most often the case that the very problem is near the starting point of the redex trail, but it might take a bit of detective work and understanding of the program in order to locate the faulty definition. With algorithmic debugging, you have only to answer questions concerning left hand sides and right hand sides of equations, and the system will, based on your answers, automatically guide you to the faulty definition. The problem with this approach is that there might be a lot of questions to answer, some of them clearly irrelevant for the problem at hand. Also, the questions might be hard to answer, because of complicated program logic or because of inability to determine the correctness of structurally large values.

F.3 Evaluation of the tracing system

As yet, no thorough evaluation has been done to determine the usability of the tracing system. In 1996, a student used and evaluated a very early version of the tracer, and the input from that exercise has been used when developing the current version.

There are still a number of user interface details that need to be improved. For example, the underlining of selected expressions tends to clutter the display, but we have yet to find a better solution. Also, the system uses colours to a large extent, and it might be hard for a novice user to interpret the meaning of the different colours. This can be solved by running the display in different modes, depending on the expertise of the user. For example, a novice user might not even know what sharing is, so the colour coding of sharing could then be suppressed.

The system has almost exclusively been tested by the developers, using small example programs, but also larger programs, with, in one case, more than 10 000 lines of Haskell code. In a recent experiment, two developers took one program each that they had written, with around five modules and about 500 lines of code and let the other person introduce a number of errors in the respective programs. The results were very promising, all errors were found rather quickly. In one or two cases, we found short-comings of the user interface that will be addressed in future versions.

In the end of 1998, a student at York University will start evaluating the current tracing system as a 20 week student project. Hopefully we will be able to do a full-scale evaluation with a group of students, to see how useful the system is and where it needs to be improved.

Bibliography

[AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3), 1997.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. Communications of the ACM, 1995.

[Arm96] Joe Armstrong. Erlang - A survey of the language and its industrial applications . In *Proc. The 9'th Exhibitions and Symposium on Industrial Applications of Prolog. (INAP'96)*, October 1996.

[ARS94] Lennart Augustsson, Mikael Rittri, and Dan Synek. Functional pearl: On generating unique names. *Journal of Functional Programming*, 4(1):117–123, Jan 1994. Original version: Splitting Infinite Sets of Unique Names by Hidden State Changes, Tech. Rep. 67, Programming Methodology Group, Chalmers University of Technology, 1992.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[Aug93] L. Augustsson. *HBC User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the HBC compiler.
See `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a non-procedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[AW85] E. A. Ashcroft and W. W. Wadge. *Lucid, the Data-Flow Programming Language*. Academic Press, 1985.

[B+92] M. Broy et al. The design of distributed systems—an introduction to Focus. Technical Report SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, 1992.

[BC85] G. Berry and I. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*. Springer, 1985.

[BD93] David L. Barton and Douglas D. Dunlop. An introduction to MHDL. In *Proceedings of IEEE 1993 MTT-S Symposium*, pages 1487–1490, San Diego, CA, May 1993. (Paper UU-1).

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BLJ91] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1987.

[CP96] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, 1996.

[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.

[Edw] Stephen Edwards. An Esterel compiler for a synchronous/ reactive development system. `ftp://ic.eecs.berkeley.edu/-pub/Esterel/esterelcomp.ps.gz`.

[FLMJ98] Sigbjörn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. "h/direct: A binary foreign language interface for haskell". In *Proc. Intl. Conf. on Functional Programming*. ACM Press, June 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HFA+94] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, P. Lee, X. Leroy, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, P. Weis, and P. Wentworth. Pseudoknot: a Float-Intensive benchmark for functional compilers (DRAFT). In J. R. W. Glauert, editor, *Implementation of Functional Languages*, pages 13.1–13.34. School of Information Systems, Univ. of East Anglia, Norwich NR4 7TJ, UK, Sep 1994.

[HHO90] Cordelia V. Hall, Kevin Hammond, and John T. O'Donnell. An Algorihmic and Semantic Approach to Debugging. In *Proceedings of the Glasgow Workshop on Functional Programming*, 1990.

[HM76] P. Henderson and J. H. Morris, Jr. A lazy evaluator. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[HM93] Jonathan E. Hazan and Richard G. Morgan. The Location of Errors in Functional Programs. In Peter Fritzson, editor, *AADEBUG'93, First International Workshop on Automated and Algorithmic Debugging, pp 166-182*, volume 749 of *Lecture Notes in Computer Science*, pages 135–152, Linköping, Sweden, May 1993.

[HM95] J. Hughes and A. K. Moran. Making Choices Lazily. In *FPCA '95 - Conference on Functional Programming Languages and Computer Architecture*. ACM Press, June 1995.

[HO85] Cordelia V. Hall and John T. O'Donnell. Debugging in a Side Effect Free Programming Environment. In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, pp 60-68*, volume SIGPLAN Notices 20(7), June 1985.

[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.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[Hug85] John Hughes. Lazy memo-functions. Technical Report 21, Programming Methodology Group, Chalmers University of Technology, 1985.

[JHH+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. "glasgow haskell compiler: A technical overview". In *Joint Framework for Information Technology Technical Conference*, 1993.

[JM96] S. A. Jarvis and R. G. Morgan. The results of: profiling large-scale lazy functional programs. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, pages 200–221. Springer LNCS 1268, September 1996.

[JNR97] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green Card: A Foreign Language Interface for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Joh87] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings 1987 Conference on*

Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 274, Portland, Oregon, U.S.A., 1987. Springer Verlag. Available from `http://www.cs.chalmers.se/~johnsson`.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Jon96a] M. P. Jones. Hugs 1.3 user manual. Technical Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, August 1996.

[Jon96b] M. P. Jones. The Hugs distribution. Currently available from `http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`, 1996.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[Kam90] Samuel Kamin. A Debugging Environment for Functional Programming in Centaur. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), July 1990.

[KH95] Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, Oct 1995.

[KHC91] Amir Shai Kishon, Paul Hudak, and Charles Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing and Reasoning About Execution Monitors. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 338–52, June 1991.

[LL95] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science 891. Springer-Verlag, 1995.

[LP95] John Launchbury and Simon L Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:193–341, 1995.

[LS97] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 227–238. Communications of the ACM, 1997.

[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.

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 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, Sep 1994.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, 1998.

[NS95] Henrik Nilsson and Jan Sparud. The Architecture of a Debugger for Lazy Functional Languages. 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.

[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.

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

[Par93] W. Partain. The **nofib** benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer Verlag, Workshops in Computing, 1993.

[Phi32] H. Phillips. *The week-end problems book.* Nonesuch Press, 1932.

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

[Röj94] N. Röjemo. nhc: a space-efficient haskell compiler. In J. R. W. Glauert, editor, *Proceedings of the workshop on Implementation of Functional Languages*, pages 30.1–30.29. School of Information Systems, Univ. of East Anglia, Norwich NR4 7TJ, UK, Sep 1994.

[Röj95] N. Röjemo. Highlights from nhc — a space-efficient Haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292. ACM Press, June 1995.

[RR96] N. Röjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. Intl. Conf. on Functional Programming*, pages 34–41. ACM Press, June 1996.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–246, April 1993.

[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.

[Sij89] B. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[SN97] Jan Sparud and Henrik Nilsson. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

[SP97] P. M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–85, March 1997.

[Spa93] Jan Sparud. Fixing Some Space Leaks without a Garbage Collector. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 117–122. ACM Press, June 1993.

[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.

[SR97] Jan Sparud and Colin Runciman. Tracing Lazy Functional Computations Using Redex Trails. In H. Glaser and P. Hartel, editors, *Proceedings of Ninth International Symposium on Programming Languages: Implementations, Logics, and Programs, Lecture Notes in Computer Science*, volume 1292, pages 291–308. Springer-Verlag, September 1997.

[SR98] Jan Sparud and Colin Runciman. Partial Redex Trails of Large Functional Computations. In C. Clack, K. Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop (IFL'97)*, volume 1467 of *Lecture Notes in Computer Science*, pages 160–177. Springer-Verlag, 1998.

[Ste84] G. L. Steele. *Common Lisp—The Language*. Digital Press, 1984.

[TA90] A.P. Tolmach and A.W. Appel. Debugging standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 1–12, June 1990.

[TA95] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

[Tol92] A. P. Tolmach. *Debugging Standard ML*. PhD thesis, Princeton, USA, October 1992.

[Toy87] Ian Toyn. *Exploratory Environments fo Functional Programming*. PhD thesis, University of York, England, 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.

[Wad90] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[Wat96] R. D. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross, Australia, October 1996.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report 91-160, Rice University, April 1991. Final version in *Information and Computation* **115** (1), 1994, 38–94.

[WR97] M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *Proc. 2nd ACM SIGPLAN Workshop on Haskell*, Amsterdam, June 1997.