The Architecture of a Debugger for Lazy Functional Languages*

Jan Sparud[1] and Henrik Nilsson[2]

[1] Chalmers University of Technology, Department of Computing Science, S-412 96 Göteborg, Sweden
[2] Linköping University, Department of Computer and Information Science, S-581 83 Linköping, Sweden

Abstract. Debugging programs written in lazy functional languages is difficult, and there are currently no realistic, general purpose debugging tools available. The basic problem is that computations in general do not take place in the order one might expect. Furthermore, lazy functional languages are 'declarative'. Hence it is advantageous if debugging takes place at the same, high level of abstraction. Thus, we propose to base debugging on what we call Evaluation Dependence Trees (EDT), which reflect the declarative semantics of the programs rather than operational concerns such as evaluation order. This in turn naturally suggests a two level debugger architecture where the lower level generates the EDT and the upper level allows the user to investigate it. The main advantage of this is flexibility: components realizing the two levels may be chosen independently to suit the debugging problem at hand.

1 Introduction

For various reasons, as will be explained in the following, debugging programs written in lazy functional languages, such as Haskell [HPJW+92], is difficult. There are currently no realistic, general purpose debugging tools available, confining lazy functional programmers to use only the most primitive of debugging techniques, e.g. bottom-up testing of individual functions. Clearly, the situation is rather unsatisfying, and it is imperative that something is done about it, if lazy functional languages are to gain popularity and credibility on a larger scale outside a dedicated scientific community.

The lack of debugging tools has been apparent for quite some time, and, as has been pointed out earlier by others [HO85, OH88, Toy87], it is clear that traditional debugging techniques (e.g. breakpoints, tracing, variable watching etc. in the conventional sense) are not particularly well suited in this context because computations in a lazy program, due to the demand driven execution model, generally do not take place in the order one might expect from reading the source code. Moreover, many programming errors common in imperative

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

languages such as C, e.g. storage related ones such as dangling or uninitialized pointers, just do not occur in a lazy functional language, further motivating specialized tools and techniques. It is equally clear that debugging tools *are* needed, despite functional languages being declarative, since there is always the possibility that a program does not express its author's intentions, even when regarded as a specification.

So how should one go about debugging lazy functional programs and what, to abuse terminology a bit, should a lazy functional debugger look like? In the following we will argue for a declarative approach, by which the user focuses on the *intended semantics* of functions and *what* actually is computed, rather than the details of *how* something gets computed. The main argument for this is that we believe that debugging should take place at the same conceptual level as programming, i.e. declarative programming languages require declarative debuggers. For example, one of the principal advantages of using declarative languages (e.g. lazy functional ones) is that operational issues such as order of evaluation may largely be disregarded. It would thus be very unfortunate if the user had to deal with operational details during debugging.

2 A Basis for Lazy Functional Debugging

In this section, after a short introduction to Haskell syntax, we first explain the problem of debugging lazy functional programs, and then very briefly review some related work in the area (see Nilsson [Nil94] for a fuller account). With this as a starting point, the *Evaluation Dependence Tree* (EDT) is introduced and proposed as a viable basis on which to build debuggers for lazy functional languages.

2.1 Haskell Syntax

The following points on Haskell syntax might be useful for the reader who is not familiar with Haskell or a similar functional language. Function application is denoted by juxtaposition, so `f (1+1) 2` means the function `f` applied to the arguments `(1+1)` and `2`. Function definition follows the juxtaposition pattern, see below. Function application has higher precedence than operator application, which is why the first argument to `f` had to be enclosed in parentheses.

Tuples are written enclosed in parentheses and lists enclosed in brackets. Thus `(1, 'a', 3)` is a three tuple and `[1, 2, 3]` is a list of three elements. The latter is just syntactic sugar for `1:2:3:[]`, where `:` is the (right associative) list construction operator and `[]` is the empty list.

Type declarations are introduced by `::`. Function types are written using `->`. Assuming that `f` returns a character, the type of f would be written `Int->Int->Char`. The types for tuples and lists 'look like' the values of that type, `(Int, Char, Int)` and `[Int]` in this case.

2.2 The Problem

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, due to its demand driven nature (i.e. call-by-need), the computation will proceed as follows. When the value of `main` is demanded, `fst` will be applied to `(foo 1 2)`. Since `fst` extracts the first component of a two-tuple, the result of `(foo 1 2)` is needed. However, `foo` does not know anything about which components of the tuple that will be needed later, so it simply returns `(fie (1+2), fie (1/0))`. Now, `fst` may extract the first component from the returned tuple, so the result of `fst (foo 1 2)` is `fie (1+2)`. We are, however, not done yet, since `fie (1+2)` has to be evaluated in order to get a value. Thus, `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 1, where the parent/child-relationship indicates that the parent caused the evaluation of its children.

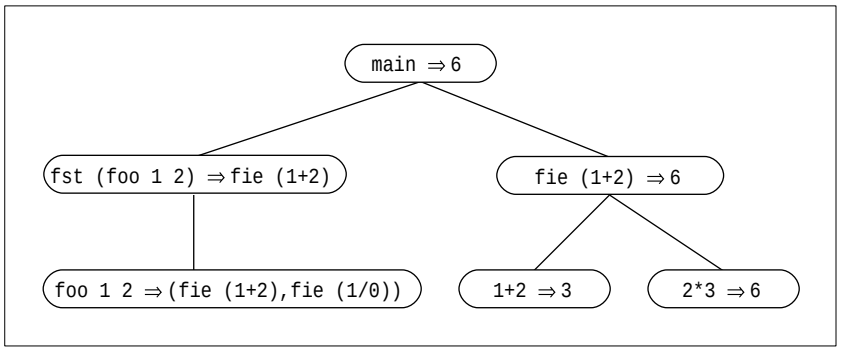

Fig. 1. Execution tree depicting lazy evaluation.

There are two things to note here: First, the fact that arguments to functions and results from functions may be expressions, that may or may not be evaluated later. In real programs, such expressions are often very large indeed, and it is in general very difficult to see what they stand for. Second, that 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 did 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 between execution events such as function invocations, both which are fundamental techniques when debugging programs written in e.g. traditional imperative languages.

To give another illustration of the problem, suppose that `main` in the program above had been defined as `snd (foo 1 2)`. Then we would get an execution error since the value of `1/0` is 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 it becomes a lot more difficult to relate an execution error to the construct in the source code that causes it than what is the case in more traditional languages with call-by-value semantics.

2.3 Related Work

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]. In their articles they focus on implementing debugging tools within an interactive, purely functional environment, the main argument for this being portability.

One approach that they suggest is to transform the source code of the entire program so that, in addition to its normal value, the program produces a trace of its execution whose structure reflects the structure of the source code. The main problem with this, as Hall & O'Donnell also point out, is of course that the very printing of the trace might turn an otherwise terminating program into a non-terminating one when the trace contains references to infinite data structures or to diverging computations, the values of which would not usually be needed. In another approach they rely on working in an interpretative environment and making use of the system function `eval`. However, this is not an option in most modern lazy functional languages since there is usually no `eval` available.

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. Thus, when the computation has terminated, any value will be seen in its most evaluated form. The result of this is very similar to Hall's and O'Donnell's approach, the difference being that Kamin is working outside the language which means that there is no problem with printing of unevaluated expression.

2.4 The Evaluation Dependence Tree

It should be noted that several of the above approaches are based on tracing in some form. Kamin even argues that tracing might well be inevitable in the context of lazy functional languages. We also believe this to be the case, and our

own previous work [NF94, Nil94, Spa94] is based on traces similar to what was discussed above, even though the traces are built by different means and used for slightly different purposes: Nilsson uses a modified language implementation that produces the trace as a side effect, and then does algorithmic debugging [Sha82], whereas Sparud's approach is based on source code transformations and browsing of the resulting trace by means of a graphical user interface, all achieved within a purely functional environment.

Based on the problems of lazy debugging discussed above and the seeming inevitability of tracing, we argue that the key to successful debugging in a lazy context is the construction of a 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, not as huge suspended expressions. We will refer to such a trace as *Evaluation Dependence Tree* (EDT) from now on. The nodes in the EDT correspond to function applications, and the children of a node are those applications on which the node depends, which we define to be the applications in the instantiated function body corresponding to the node that *eventually* became evaluated.

Thus there is a close correspondence between the structure of the source code and the structure of the EDT. Note, though, that only applications that really were evaluated during the program execution will be present; the EDT is not some kind of static call graph.

To illustrate this, consider the example in section 2.2 again. The EDT for this program is shown in figure 2. Note how arguments and results are shown as values when possible, i.e. whenever they were computed during the program execution. Anything that is left unevaluated at the end of the execution is shown as `?`, the rationale behind this being that that expression has not contributed anything to the result of the program or influenced the execution in any way, and is thus of no importance for debugging purposes.

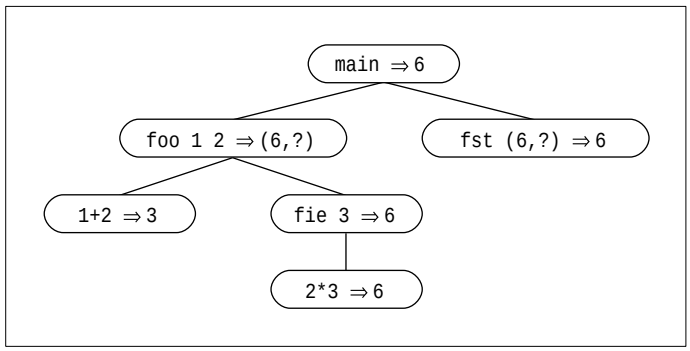

Fig. 2. Evaluation Dependence Tree.

3 A Two Level Debugger Architecture

We will now give some substance to our claim in the previous section by proposing a two level debugger architecture, where the lower level is concerned with the construction of the EDT, and the upper level focuses on presenting the EDT in a convenient way and navigation through it in search for bugs. In the following we will refer to the lower level as the *EDT generator* and to the upper level as the *EDT navigator*.

There are many advantages of a 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, she may easily extend and adapt the navigator to her 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 respect to different parameters, such as speed or space requirements.
- *Separation of concerns*: As will become clear later, the construction of the EDT is somewhat tricky and require 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 considering it on its own.

In spirit, the proposed two level architecture is in many ways similar to the approach taken in Ducassé's Prolog debugger Opium, which seems to have generated a lot of interest in the Prolog community [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.

4 The EDT Generator

In this section, the EDT and two approaches for generating it are presented in greater detail. The first is based on source transformations: the program to be debugged is transformed so that it produces an EDT in addition to its normal value. The main advantage of this is that it requires very little special support from the underlying language implementation, and therefore could be integrated into existing systems with reasonable effort. In the second approach the underlying language implementation is changed so that it produces an EDT as a side effect of execution. The principal advantage of this method is fine control over the EDT construction, e.g. it opens up possibilities of building parts of the EDT on demand, potentially reducing the memory requirements by orders of magnitude.

4.1 Some EDT Generator Requirements

To support debugging in a realistic setting, the generated EDT must in our opinion fulfil the following requirements:

- *Safety*: Inspecting and traversing the EDT must be safe, i.e. it should not cause any previously unevaluated expressions to be evaluated since these may represent diverging computations.
- *Finiteness*: While it would not be a fundamental problem to have conceptually infinite values in the EDT as long as they have a finite (i.e. circular) physical representation, being able to rely on values in the EDT being finite, by making any circularities explicit, certainly helps when it comes to displaying these values.
- *Explicit* $\perp$: Execution errors and diverging computations, i.e. what is semantically thought of as $\perp$ (bottom), must have an explicit representation in the EDT so that it is possible to debug in the normal way even in such cases.

These requirements are what necessitate special support from the language implementation in the transformation-based approach: unevaluated expressions and circularities cannot normally be detected from within the language, nor is it usually possible to recover from execution errors or diverging computations.

4.2 The Interface

Part of a typical interface between the EDT generator and the EDT navigator, as it might look in Haskell, is shown in figure 3.

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 accessor functions such as `function` or `children`.

Values, i.e. function arguments and results, are represented by the type `EDTValue`. Note the explicit representation of $\perp$ and unevaluated expressions. The constructors `Label` and `Reference` are used for representing circularities explicitly. Constructed data objects are encoded by means of the constructor `ConsVal`. `Closure` represents closures where the `EDTValue` list contains the values for free variables and any arguments in partial applications.

`Function` and `Constructor` are also abstract types, and various attributes may be accessed by means of functions such as `funName` and `consName`. `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 we regard as a more or less essential aid in helping the user understanding the program that is being debugged.

4.3 A Transformation Based EDT Generator

Generating the Evaluation Dependence Tree Our aim is to transform the functions in a program so that they return not just a result, but also the evaluation dependence tree of that result.

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

data EDTValue = Bottom
              | Unevaluated
              | Label Int EDTValue
              | Reference Int
              | PrimValChar Char
              | PrimValInt Int
              | ...                 -- Other primitive types
              | ConsVal Constructor [EDTValue]
              | Closure Function [EDTValue]

funName::      Function->String
funSourceRef:: Function->SourceRef
...                                 -- Other attributes for Function
consName::     Constructor->String
...                                 -- Other attributes for Constructor
```

Fig. 3. Typical interface to the EDT generator.

The EDT data types define the interface to the EDT navigator. In the `EDTValue` type, values from all types in a program are encoded. The encoding must be done when the program has terminated, since we do not want to work with encoded values at runtime. Because of this we define a separate dependence tree data type to be used at runtime. After program termination the runtime dependence tree can be turned into an EDT on a per node basis.

A dependence in the tree consists of a node and a list of dependences:

```
data Dep = Dep Node [Dep]
```

A node consists of a function call and the result of that function call. `If`-expressions and `case`-expressions are also seen as function calls with `if` and `case` as their respective functions. They could also have separate constructors in the Node type, and this may change in the future.

```
data Node = FunApp Call Value
```

The call can be represented in many ways. To start with we use a string representation for the function and the arguments are represented as a list of values:

```
data Call = MkCall String [Value]
```

The value type is more problematic. We could parameterize the dependence type with respect to this value type, but then every dependence in the dependence

list must be of the same type. This means that *all* functions in the program must have the same return type.

Here existential types come to our rescue[3]. We define the `Value` type as:

```
data Value = (EDTConvertible a) => V a
```

which means that `V` e is a proper `Value`, if and only if the type of e is an instance of the class `EDTConvertible`. The only operations we can do on elements of type `Value` are those defined in the `EDTConvertible` class.

The `EDTConvertible` class is defined as follows:

```
class EDTConvertible a where
    mkEDTVal :: a -> EDTValue
```

Instances of the `EDTConvertible` class must be defined for all types used in a program. Currently the user must supply these instances, but nothing prevents the compiler from generating these instances automatically. Two example instances for Ints and Lists, are given below:

```
instance EDTConvertible Int where
    mkEDTVal = PrimValInt

instance EDTConvertible a => EDTConvertible [a] where
    mkEDTVal     [] = ConsVal "[]" []
    mkEDTVal (x:xs) = ConsVal ":" [mkEDTVal x, mkEDTVal xs]
```

In these examples we have the `Function` type `String`, but it should really be abstract. The `mkEDTVal` functions are used to 'look' in the graph once the debugged program has terminated. After that we do not want any more reductions to take place, so before calling the appropriate `mkEDTVal` function, the system checks that the value in question is evaluated and defined. If not, it returns `Unevaluated` (or `Bottom`). The checks that a value is evaluated and defined are implemented as system primitives. They can not be defined within the language.

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

```
type R a = (a, Dep)
```

Higher order arguments to functions are a little problematic. When debugging a call to a higher order function, it is desirable to see *which* function that is given as its argument. But functional values cannot be displayed in any reasonable way, they can only be applied to arguments. We avoid this problem by introducing a new data type:

[3] Existential types are not (yet) allowed in standard Haskell, but are implemented in the Chalmers Haskell compiler (hbc).

```
data Fun a = Fun a Call
```

where the type variable `a` represents a function and `Call` is a representation of the function call, which makes it possible to present calls to partially applied functions. Functions are transformed so that higher order arguments are wrapped in the `Fun` data type.

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
⇒
foldr_D :: Fun (a -> b -> R b) -> b -> [a] -> R b
```

Transforming Expressions We are interested in collecting information about function applications. So what we really would want to do is to change the behaviour of function application (i.e., juxtaposition) so that it automatically collects this information. One cannot redefine the meaning of juxtaposition in Haskell, so instead we define an operator that takes care of the plumbing.

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

Here we, for convenience purposes, use a type abbreviation `P` to hold a value and a list of evaluation dependences:

```
type P a = (a, [Dep])
```

The intention is that `$$` takes as its left argument an expression returning a pair with a value and an evaluation dependence. It then passes on the value to the function given as its right argument. That function returns a new value and a list of evaluation dependences. The result of `$$` 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, finally ending in a call to the `return` function, that takes a value and returns a pair with the value and an empty evaluation dependence list.

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

As an example, we will transform the expression `fst (foo 1 2)` where `fst` and `foo` are transformed functions (cf. the example in section 2.2 and figure 2). The transformed functions need values as arguments and return value-evaluation dependence pairs. We use the `$$` operator to turn results from transformed functions into values and to pass on evaluation dependences.

```
fst (foo 1 2) ⇒ foo 1 2 $$ \p -> fst p $$ \v -> return v
```

The result of the transformed expression is a pair containing the value of the original expression, and a list of evaluation dependences for the calls to `fst` and `foo`. The evaluation dependences are transparently collected by `$$`.

Transforming Definitions The operators we have seen this far deal with collecting evaluation dependences, but we have not yet said anything about how they are created. The basic objects in the evaluation dependences are the dependence nodes, which are created by a call to `mkDep`.

```
mkDep :: (EDTConvertible a) => String -> [Value] -> P a -> R a
mkDep f vs (res, ds) = (res, Dep (MkCall f vs) (V res) ds)
```

`mkDep` is only used at the top level in a function definition. The arguments are the function name, a list of type `[Value]` with the actual arguments to the function, and a pair of an expression result and a list of evaluation dependences.

With these arguments, `mkDep` creates a dependence node, and returns a pair with the result of the expression and the dependence node.

Examples We will show how the standard function `map` is transformed.

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

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

As can be seen, the plumbing with evaluation dependences are taken care of by `mkDep`, `$$` and `return`, and is not visible in the transformed programs. This is not very important in an implementation but it makes it easier to understand the transformation.

4.4 EDT Generation at the Implementation Level

In this section we outline an alternative way of generating the EDT. It is based on modifying the graph-reduction machinery of a typical lazy functional language implementation (see e.g. Peyton Jones [PJ87]), so that an EDT is constructed as a side effect of execution. We also describe how *piecemeal* EDT generation, i.e. generation of parts of the EDT as needed, may be achieved. The method described here has been tried out in a small experimental implementation and found to work very well, but not yet integrated into any language implementation. See Nilsson [NF94] for further details.

Getting the Tree Structure Right When a function is invoked in a lazy functional language, it typically constructs an expression representing an instance of its body and returns this as the result. (In a real implementation, this

process is often side-stepped in order to gain efficiency, but that is not important for this presentation.) This expression may be further evaluated later, and, as was discussed in section 2.4, the resulting EDT-node should then be a child of the EDT-node corresponding to the function invocation that constructed the expression. Achieving this is easy in principle: just keep references from expressions to their parent EDT-nodes. Thus, whenever an expression is evaluated, it is clear where the new, resulting EDT-node belongs in the tree.

As we have seen earlier, function arguments are in general not fully evaluated when a function is invoked. Thus, there is no point in recording the arguments at this point in time. Rather, references from the arguments to the expressions to which they are bound should be kept. By the nature of graph reduction, these expressions will be physically overwritten with the values to which they evaluate when and if they are evaluated. This means that once the execution has terminated, arguments will be present in their most evaluated form in the resulting EDT. Results from functions are handled in the same way.

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 runs in constant memory space with very modest memory requirements, may generate a huge EDT if it runs for long enough. The problem may be alleviated by means of various filtering techniques (see section 4.5), but as long as an entire EDT is kept, the problem remains.

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 program to be debugged is re-executed, and the next piece of the EDT captured and stored. Such a scheme is what we refer to as *piecemeal EDT generation*. Re-executing the program is not a problem, since pure functional programs are deterministic, but any input to the program must obviously be preserved and reused.

A piecemeal scheme might actually be beneficial from a time as well as a space perspective, since the time overhead of EDT generation should be significantly reduced if only a small portion of the EDT is constructed. The user would therefore be able to start debugging quicker. Furthermore, if the overhead is large enough, and the program is only rerun a few times, the scheme might turn out to be cheaper overall.

4.5 Reducing the Size of the EDT

Up until now, we have more or less assumed that every single function application that takes place during execution should be recorded in the EDT. However, quite a few of these are applications of language primitives (e.g. `+`, `if`) which should be correctly implemented. Moreover, large systems are built modularly, so it will often be the case that there are a large number of well tested, fairly trusted modules and a few, new, 'prime suspects' when a bug manifests itself. The very least, library modules may reasonably be trusted. Thus, it should be possible

to reduce the size of the EDT considerably by filtering out only the interesting function applications.

There is however a problem as to what is meant by a function being 'correct' in this context. For a first-order function it is obvious: a function is correct if it computes the expected result for whatever arguments it is applied to. During debugging, this means that a user would not investigate any applications of this function, thus the entire branch in the EDT emerging from that point may be cut away.

Higher order functions are more problematic since they may invoke their argument functions, meaning that the combined behaviour might be wrong even though the higher order function itself is correct. Thus, cutting the tree as above cannot be justified.

5 The EDT Navigator

The computations that take place when a program is run are saved in an EDT. In order to find a possible bug the user has to traverse this tree in some way. The *EDT navigator* is responsible for interacting with the user in this process. The navigator communicates with the EDT generator via the interface defined in section 4.2. How the interaction between the user and the navigator should work could be defined in different ways, and a few approaches are outlined in this section.

5.1 Searching for the Bug

The algorithmic debugging method [Sha82] is one way of guiding the user through the possibly huge amount of data in an EDT, and can locate a bug accurately, provided that the user can answer whether results of certain function applications are correct or not. An alternative is to let the user browse the tree looking at function calls and their results.

The advantage with the algorithmic debugging method is that the bug can be located 'semi-automatically'. Furthermore, the answers given by the user during a debugging session constitute a partial specification of the behaviour of the program, and could be saved and reused during later debugging sessions, thus further automating the process. A drawback is that the user might have to answer a lot of questions, some of which may be clearly irrelevant, in order to find the bug.

With the free browsing method the user can, if she has rather a good idea of where the bug might be located, quickly move to the interesting part of the EDT. On the other hand, if she has no idea of what might be wrong, the systematic approach of algorithmic debugging might be better.

We think that both of the methods mentioned has their advantages and that a navigator should support both. They can even be used in combination: the user can start the debugging session by browsing to an interesting part of the tree, and then start algorithmic debugging from there.

To illustrate the above approaches, consider the example in section 2.2, but assume that `main` is defined as `snd (foo 1 2)` instead. The resulting EDT is given in figure 4.

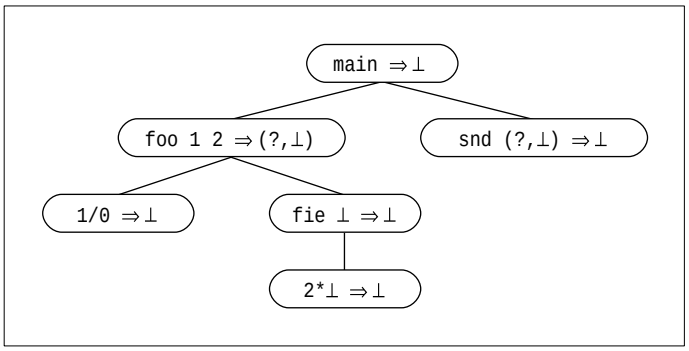

Fig. 4. EDT with a bug.

If browsing freely, the user would immediately see that $\bot$ comes from the application of `foo`. She would then inspect this node and realize that the problem is a division by 0. In algorithmic debugging, the navigator would first ask about the result of `main`, which is wrong, then about the application of `foo`, which is wrong as well, then about `1/0` yielding $\bot$, which is correct, and finally about `fie` $\bot$, also yielding $\bot$, which is correct. Given these answers, it would conclude that the bug must be in `foo`. See Sparud [Spa94] and Nilsson [NF94] for further debugging examples.

5.2 Displaying Values

Another big problem is how to present function calls and values to the user. There are a number of different reasons for this:

- The way the system shows values of a certain type might not agree with the user's intuition.
- Values may be unevaluated or undefined (bottom).
- Values may be structurally infinite, at least conceptually, or very large.

Our approach to these problems is to have value visualization functions at different levels. When displaying a function call, a short textual representation will be displayed for each value (the arguments and the result). If the user wants to look closer at a value she can select it and a more detailed view of the value will be displayed.

For the predefined types the system can define suitable low detail and high detail visualization functions. For example, a string can in low detail be shown

as the first part of the string and in high detail it can be displayed in a text browser with scrollbars and text search functions.

For user defined types the system can define default low detail visualization functions, leaving it as an option for the user to define low and high detail visualization functions for values on a per type basis. For example, a value in a tree data type may be displayed as a graphical picture with the possibility for users to select and look closer at subparts of the tree. A problem when defining such functions is that values can be unevaluated or undefined, but since these cases have explicit representations in the EDT value type, the problem is not very severe.

6 Future Work

Currently we have two different debugger implementations as have been described in this paper. Both of them are incomplete in many ways. The ultimate goal is to combine the good parts of them and develop a debugger for full Haskell. In this process we would like to apply the debugger to real programs, not just small examples.

Many of the ideas in this paper will be pursued, e.g., more efficient construction of the EDT (to reduce memory overhead). Much effort will be put in the EDT navigator, since we believe that it is very important that the debugger is easy to use, intuitive, and user customizable. Methods of filtering out 'uninformative' information from the EDT, to reduce the number of questions the user has to answer will be developed along the lines of trusted functions/modules, assertions etc. We will also look into different strategies for debugging different kinds of programs, e.g., programs written in a monadic style.

7 Conclusions

This paper 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.

Basing debugging on the EDT naturally suggests a two level debugger architecture, 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.

This paper has advocated such an approach, where the interface between the levels is purely functional so that the EDT navigator may be written in the language for which the debugger is intended. This is very useful since it allows the user to easily customize the navigator to fit her needs. Two different ways

methods for EDT generation and two different ways of navigating through the trace were suggested, and their relative merits discussed and compared.

Based on our previous experience of debugging lazy functional programs, we feel that a debugger designed along the lines described in this paper would be a very good starting point from which to develop a realistic debugger for a real language such as Haskell.

References

[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.

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

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

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

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[Nil94] Henrik Nilsson. A declarative approach to debugging for lazy functional languages. Licentiate Thesis No. 450, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, September 1994.

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

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

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

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

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