Received September 29, 1995 ; Revised October 11, 1995
Editor:

The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging [*]

HENRIK NILSSON henni@ida.liu.se
Linköping University, Department of Computer and Information Science, S-581 83 Linköping, Sweden

JAN SPARUD sparud@cs.chalmers.se
Chalmers University of Technology, Department of Computing Science, S-412 96 Göteborg, Sweden

Editor: M Ducassé

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.

Keywords: Declarative languages, lazy functional languages, debugging, debugger implementation

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 to fully understand what is going on operationally, which is exactly what a declarative programmer usually prefers not to be concerned with. 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.

---

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

One class of declarative languages are the lazy functional ones, such as Haskell [4]. 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 [11], 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.

Due to the high-level nature of lazy functional languages, and the fact that most modern lazy languages feature static, polymorphic type systems, the lack of general debugging tools is less of a problem than what it otherwise would have been. The type system prevents many mistakes and automatic memory management eliminates another common source of errors. Nevertheless, sooner or later mistakes are made, and a good debugger is then a valuable and time saving aid.

Several declarative debugging techniques which could be used in a lazy context have been suggested, e.g. Shapiro's algorithmic debugging [13], or Ducassé's trace analysis techniques [1]. 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 [9].

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 explicitly includes information necessary to display functional values. Names of functions and free variables are part of the EDT, for instance, and partial applications have an explicit representation. Similar approaches have been used for other languages where the declarative semantics is far removed from what happens operationally, e.g. deductive database systems [15].

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 paper is organized as follows. Section 2 gives a short introduction to Haskell. The basic problems of lazy functional debugging are covered in section 3, and section 4 then defines the EDT and shows how it addresses these problems. Section 5 considers the problems of EDT construction. Two different strategies are explained and compared. Some architectural issues are addressed in section 6 and then related work is reviewed in section 7. Finally conclusions are given.

2. Haskell Syntax and Features

The following points on Haskell syntax and language features might be helpful for readers who are not familiar with Haskell or any similar functional language. It covers most of the constructs that are used in the remainder of this article. A thorough introduction to Haskell may be found in Hudak & Fasel [3] and a definition of the language in Hudak *et al.* [4].

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

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

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

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

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

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

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

```
fst (a,_) = a

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

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

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

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

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

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

Haskell has a *polymorphic* type system. Since the functions `fst` and `length` defined above do not make any assumptions regarding the types of the elements in the pair and list, respectively, they are polymorphic, meaning that `fst` can be applied to pairs of elements of any types, and that `length` can be applied to lists of elements of any type.

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

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

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

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

`Colour` is a simple enumeration type with three values. `Point` is a type synonym, i.e. just a shorthand notation for a tuple of two floating point numbers. `Point` is then used in definition of the data type `Object`. An `Object` is either a `Rectangle` consisting of two points (the coordinates of opposite corners) or a `Circle` consisting of a point and a floating point number (the origin and the radius). The final example illustrates a *recursive* type definition, isomorphic to the built-in list type. Thus, a `NewList` with elements of type `a` is either `Null` (the empty list) or a `Cons`-cell consisting of an element of the type `a` (the head of the list) and a `NewList` with elements of type `a` (the tail of the list).

`Red`, `Green`, `Blue`, `Rectangle`, `Circle`, `Null` and `Cons` are called *constructors* and work as functions (or constants) for constructing values of the corresponding type. Thus we have `Red :: Colour` and `Cons :: a->NewList a->NewList a` for instance. Constructors also work as tags, distinguishing various kinds of objects from one another within a type, and can be used to take objects apart by pattern matching. For example, a function to compute the area of an `Object` and a function to compute the length of a `NewList` may be defined as below:

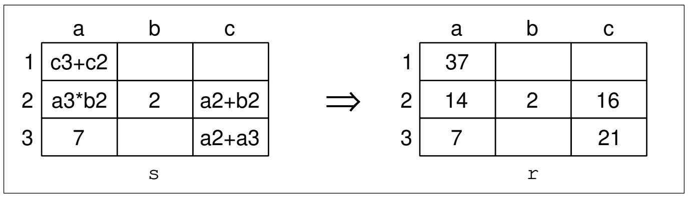
```
area (Rectangle (x1,y1) (x2,y2)) = abs ((x2-x1) * (y2-y1))
area (Circle _ r)                = pi * r * r

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

3. 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.

Figure 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.



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

s

⟹

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

r

Figure 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.

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.

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

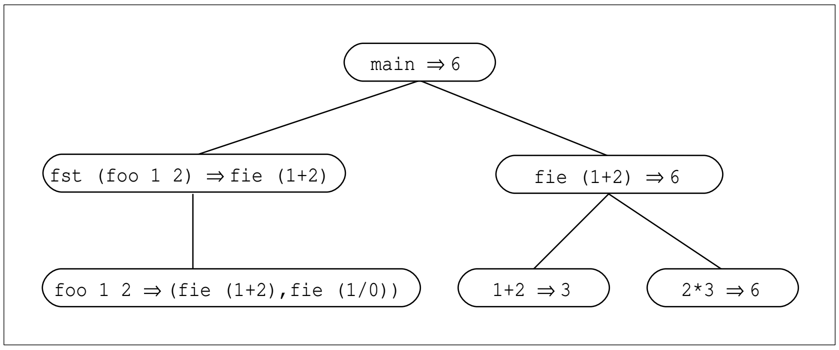


Figure 2. Execution tree depicting lazy evaluation. The nodes correspond to evaluation of function applications, recording the arguments and the result. The parent–child relationship indicates that the parent caused the evaluation of its children.

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 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.

4. 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 [2], [11], 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 7, 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 [5] 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.

4.1. 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.

4.1.1. 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 x1 ... xn = exp
```

The intention is that a program is a single expression whose value is the result of executing the program. The `Letrec`-expression allows the definition of functions and variables that are in scope in the entire `Letrec`-expression, 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 3 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.

4.1.2. 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                           -- Integer
         | Constr String [Val]               -- Constructed value
         | Fun String Int ([Val] -> Val) -- Function
         | PApp Val [Val]                    -- Partial application
         | Thunk (() -> Val)                 -- Suspended evaluation
```

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.

4.1.3. 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 components 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.

4.2. 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 3 which shows the EDT for the example in section 3. 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

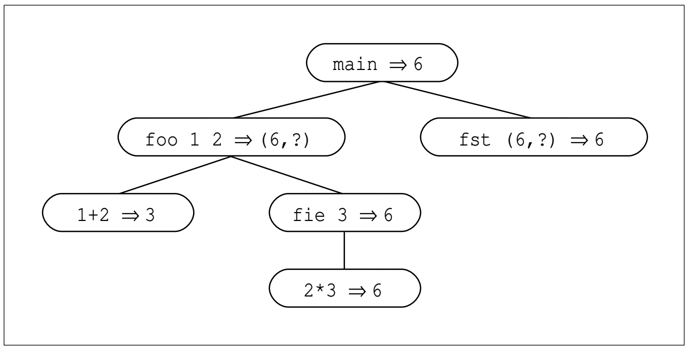


Figure 3. EDT for the example in section 3. Note that the structure of the EDT reflects the structure of the source code, but that only nodes corresponding to function applications that actually were evaluated are present. Also note that some values were never evaluated. These are shown as `?`.

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 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 5.1). 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.

4.2.1. 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. [1]

```
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
```

4.2.2. 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 straigtforward 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
```

4.2.3. 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 evalauated.
     -- (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 variable 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.

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 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 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.

4.3. EDT Based Debugging

Algorithmic debugging [13] 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 wrong.

To illustrate this, consider the example in section 3 again, but assume that `main` is defined as `snd (foo 1 2)` instead. The resulting EDT is given in figure 4.

If browsing freely, the user would find that $\perp$ 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 $\perp$, which is correct, and finally about `fie` $\perp$, also yielding $\perp$,

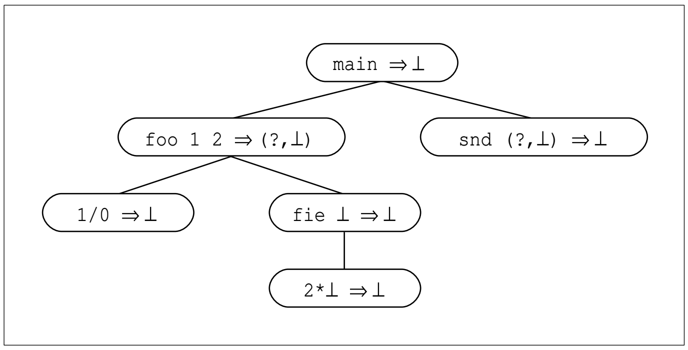


Figure 4. EDT with a bug. ⊥ represents an undefined value, which in this case caused an execution error. Since the second component of the result from `foo` was not supposed to be undefined, and since the nodes on which the application of `foo` depends show correct behaviour, the bug must be in the definition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

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 [8]. Sparud [14] investigates the the free browsing scheme.

5. 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, or the language implementation (a compiler or interpreter) is changed so as to produce an EDT as a side effect of execution. An implementation of each approach is outlined in this section. See Nilsson & Sparud [10] for details.

5.1. Graph Reduction

Implementations of lazy functional languages are usually based on graph reduction. The basic ideas are reviewed here, for details refer, e.g., to Peyton Jones [12].

In a graph-reduction based implementation of a lazy language, expressions and data are represented as pieces of graph (stored in the heap). A key point is that whenever a reducible expression, or *redex*, is evaluated, that redex is overwritten with the result. Thus no expression is evaluated more than once. Consequently it is easy to find out exactly which expressions were eventually evaluated by inspecting the graph once the execution has terminated. Anything that was not needed will still be a redex, whereas anything that was needed will be in WHNF (weak head normal form, see section 4.1.3).

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 x = x * x` would for instance be compiled into code that performs the following rewriting step:

```
   @      ⇒      @
  / \           /|
 f   E         @ |
              / \|
             *   E
```

Here, `@` denotes an application node and `E` represents an arbitrary 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.

5.2. EDT Generation by Source Code Transformation

5.2.1. How the Transformation Affects the Types of Functions

The purpose of the transformation is to make the functions in a Haskell program return an EDT in addition to its normal result. The return type of each function becomes a pair consisting of a result and a list of EDTs for that result. We define a type abbreviation for the return type: `type R a = (a, [EDT])`.

The (overloaded) function `edtVal` converts ordinary values to their representation in the `EDTValue` data type. This is straightforward for ground (non-functional) types. However, for functional types there is a problem since Haskell does not provide a way of finding out the name of a function. We solve this by representing functions with a new data type which explicitly encodes information such as the function name. This enables `edtVal` to handle functions too.

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

The fields are the function in question, information about the function, and the arguments the function has been applied to so far (partial application). As an example, the transformation changes the type of `foldr` as follows:

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

5.2.2. Transforming Function Definitions

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

```
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 definition. The `(EDTAble a)` part of the type for `mkDep` means that the type `a` must be an instance of the class `EDTAble` (see 5.2.4). The arguments to `mkDep` are the function name, the arguments (represented as `EDTValues`), and a pair of an expression result and a list of evaluation dependences.

Transforming a function body is just a matter of collecting the EDTs from all subexpressions in the body and calling `mkDep`. 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. When we transform the expression we must propagate the dependence information. In principle, the transformed version of `foo` is as follows:

```
foo x y = mkDep "foo"
                [edtVal x, edtVal y]
                (let (a1, d1) = g x
                     (a2, d2) = h y
                     (a3, d3) = f a1 a2
                  in (a3, d1 ++ d2 ++ d3))
```

5.2.3. Avoiding Further Evaluation

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 due to 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. Thus values will be seen in their most evaluated form.

However, it is impossible to 'look' at a value in a lazy functional language to determine whether it has been evaluated or not. By looking at something, it gets evaluated. Fortunately, it is easy to determine whether a value is evaluated or not at the language implementation level (see 5.1). It is usually also possible to recognize values that were *being* evaluated when the execution stopped and the debugger took over, e.g. after a run-time error. Semantically, such a value is undefined (bottom). We have thus introduced two non-pure functions, implemented as a part of the debugging runtime system:

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

5.2.4. Definition of `edtVal`

The function `edtVal` is overloaded, i.e. defined separately for each type in the program. This is implemented using the Haskell class system. In Haskell, a class is a collection of functions (or *methods*) that is parameterized over a certain type. These functions are then defined for any type that is an instance of the class. We introduce a class `EDTAble` and make *all* types in a program instances of it.

```
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 generated automatically for user defined types.

5.3. EDT Generation by Modifying the Language Implementation

An alternative approach to EDT generation is to change the implementation of a language, i.e. the compiler or interpreter, so that an EDT is obtained as a side-effect of execution.

5.3.1. 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 and then overwrite the redex root with the root of the newly constructed graph.

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.

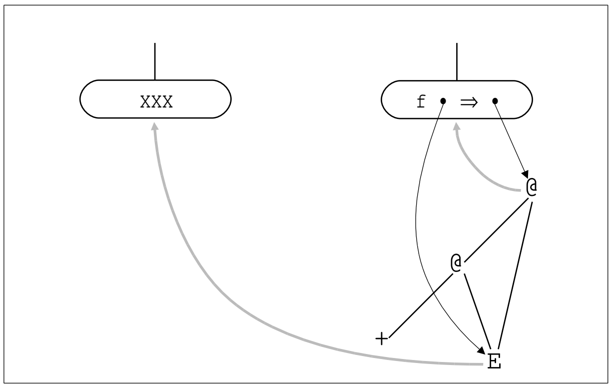


Figure 5. The graph and the EDT after reduction of `f E`. Note that each redex refers to the EDT node corresponding to the function invocation that created that redex (the grey arrows).

Achieving this is simple: 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.

For 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. The situation immediately after the reduction of `f E` is shown in figure 5. Note that the argument and the result are live pieces of graph, referred to via pointers from the newly constructed EDT node. Note also how each allocated redex refers to its parent in the EDT.

The result of `f E` is a new expression, `E + E`, and next this expression is reduced. Now, `+` is *strict* in both its arguments, which causes the evaluation of `E`, yielding `7`, say. The new situation is shown in figure 6. Note that 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.

5.3.2. Piecemeal EDT Generation

A big problem with the construction of an EDT is 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. Note that the transformational EDT generation scheme suffers from exactly the same deficiency. While the EDT nodes are not be created until demanded by the debugger, thanks to working in a lazy language, a large expression representing the entire EDT is going to be constructed as the execution proceeds. Storing this expressions requires about the same amount of memory as storing the EDT.

Furthermore, since the EDT does not grow in a linear way like traces for imperative languages, and since it contains references to live pieces of data on the garbage

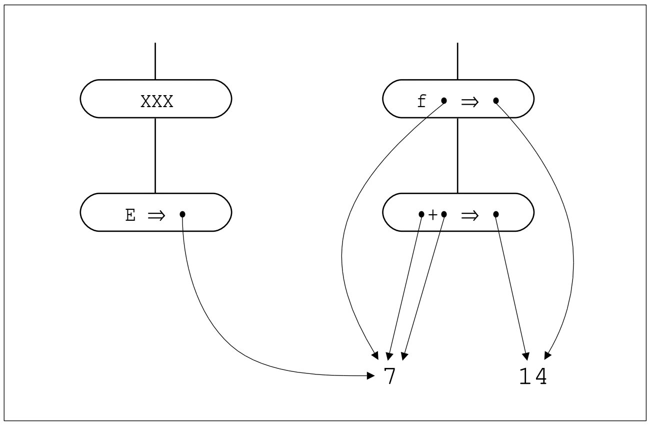


Figure 6. After reduction of `E + E`, the redexes have been overwritten by their results and two new nodes corresponding to the reductions that just took place have been added to the EDT.

collected heap, it would be very expensive to store it on secondary storage. Keeping all references updated would involve too much random access. Moreover, file sizes are typically limited due to sizes of physical disks and arbitrary operating systems restrictions (4 Gbyte under typical UNIX implementations, for instance).

An 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 not enough to find the bug, the target is re-executed, and the next piece of the EDT captured and stored. We refer to this as *piecemeal EDT generation*. Since the construction of a partial EDT is much quicker than constructing an entire EDT, and since it can be expected that only a fraction of the EDT is visited during debugging, the scheme is not as expensive as it might seem at first.

The basic idea of piecemeal generation is quite general, but we have thus far only implemented it in the context of the low-level EDT generation approach. A particular feature, made possible by working at the language implementation level, is that it allows a firm upper limit to be imposed on the memory consumption of the debugger. Thus there is no risk of the debugger running out of memory, which guarantees that any program can be debugged. This might of course require a lot of re-executions if the debugger is given too little memory. Such guarantees are probably more difficult to give in the context of transformational schemes. See the discussion on Naish & Barbour in section 7.

5.4. 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 easy to implement within most current Haskell implementations. Furthermore, since only the transformations decide what the resulting EDT should look like, it does not matter what a compiler does to the code. A highly optimizing compiler does not cause any particular problems. The problems with this approach mainly concern performance. The transformed code runs considerably slower than code using the low-level EDT generation scheme. The worst aspect, however, is the memory consumption. Since a complete EDT is always constructed, only programs that do not perform long computations can currently be debugged. If real programs are to be debugged, this problem must be addressed, perhaps as suggested by Naish & Barbour [7]. See section 7 for a brief description.

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. The main disadvantage is that the implementation effort is substantial.

To get an indication of the cost of our EDT generation schemes, we have compared the normal execution time of a few small programs with the execution time when generating EDTs. For the transformational scheme, execution times increased between 8 and 25 times, whereas it increased between 4 and 10 times for the low-level approach. Tolmach & Appel [16] report a slowdown of a factor 3 for their transformation-based SML debugger, indicating that it might be possible to improve the performance of the transformational scheme. For the transformational approach only small runs could be performed due to memory restrictions. For the low-level approach, piecemeal tree construction was used and up to 30000 nodes (out of between 500000 and 2500000) were kept (execution times increased as the number of kept nodes increased).

A closer inspection of the figures for the low-level approach reveals that the actual tree construction is very quick. Most of the extra execution time is due to garbage collection and crude integration of the mechanisms into the host system (the Chalmers Haskell implementation, HBC). With better integration, the execution times would probably be around 2 to 4 times the normal ones when keeping some 10000 nodes in the EDT. See Nilsson & Sparud [10] for details.

6. 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 the part that is responsible for presenting the EDT the *EDT navigator*. Between them there should be a pure,

```
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
```

Figure 7. Typical interface to the EDT generator.

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 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 complicated and requires at least some support from the language environment, i.e. it cannot be done in a satisfactory 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 [1]. 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 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 understand the target program.

The `EDTValue` is more or less as described in section 4.2.2. 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. This can be achieved by means of special constructors for labelling values and for referring to such labels.

7. Related Work

Hall & O'Donnell [2], [11] 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. This is 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. We also have to work harder to make functional vales displayable.

A problem 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 5.2.3).

Kamin [5] 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. A 'meta-evaluation rule' is used 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. The effect is that values in the trace will be as evaluated as possible once the computation has terminated.

Kamin thus gives a formal definition of a trace that reflects the structure of the source code, much in the same way as we do. However, in comparison to our specification, Kamin uses a simpler language (no currying and no local let-bindings) which simplifies his specification. Furthermore, the information needed for properly displaying tree nodes and values, such as names of free variables and user defined functions, is not explicitly included in the trace. Instead it is assumed that this information can be obtained from the run-time representation of values, which is often impossible. 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.

Kishon & Hudak [6] take an approach similar to Kamin's, but more general and systematic. Starting from a denotational continuation semantics of a language, they derive a monitoring semantics by composing the standard semantics with one or more monitor specifications. 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.

Since monitoring semantics are given separately from the language semantics, the trace of Kishon & Hudak is specified more indirectly 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, not 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. A problem of their work is that functional values cannot be shown properly. To solve this, the semantic definition of the language would have to be changed so that functional values carry additional information for printing purposes.

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 [7] is closely related to ours. They use a source-to-source transformation which transforms the target into a program that generates a trace, similar to our EDT, in addition to its normal output. However, no explicit specification of the generated trace is given. The language is simpler than ours: 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 displayable representation. `Dirt` 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 simpler than ours since there is no need to handle functional values in a special way (see 5.2.1). On the other hand, requiring a function like `dirt` makes their approach significantly less portable. For example, in the Chalmers Haskell implementation HBC values in the heap do not 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 simple.

Naish and Barbour also consider the memory consumption problem and suggest generating parts of the tree on demand. Once a node at the fringe of the stored portion of the tree is reached, the function of that node is re-applied to its arguments. This application is then compared to the *evaluated* parts of the result of the previous application of the function, 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, and the scheme thus avoids re-executing the entire program. 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.

To limit the amount of memory required to store the tree, Naish and Barbour suggest keeping a fixed number of levels of the tree below the current root. This will probably work in most cases, but it should be noted that a single node can refer to data structures of arbitrary size. Thus their scheme is not able to put a firm upper limit on the resource consumption of the debugger. An advantage of our piecemeal scheme is that it allows the number of tree nodes to vary in an effort to keep the overall size of the stored part of the EDT below a prescribed limit.

8. Conclusions

This paper has proposed the *Evaluation Dependence Tree* (EDT) as a basis to build debuggers for lazy functional languages. The reason 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 preliminary performance measurements 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 Ducass and the anonymous referees for many useful comments that substantially improved this article. We would also like to thank Rickard Westman and Graeme Moss who proof-read the article.

Notes

1. In the case of non-termination some heuristic for recovering is needed, such as the user explicitly aborting it.

References

1. 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.
2. 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).
3. Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.
4. 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.
5. 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.
6. Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, October 1995.
7. 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.
8. 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.
9. Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.
10. 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.
11. John T. O'Donnell and Cordelia V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
12. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
13. Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, May 1982.
14. 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.
15. Günter Specht. Generating explanation trees even for negations in deductive database systems. In *Proceedings of the 5th Workshop on Logic Programming Environments*, pages 8–13, Vancouver, Canada, October 1993.
16. Andrew Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.