A Program Transformation for Debugging Haskell 98

Bernard Pope **Lee Naish**

The Department of Computer Science and Software Engineering
The University of Melbourne,
Victoria 3010, Australia,
Email: bjpop@cs.mu.oz.au lee@cs.mu.oz.au

Abstract

We present a source-to-source transformation of Haskell 98 programs for the purpose of debugging. The source code of a program is transformed into a new program which, when executed, computes the value of the original program and a high-level semantics for that computation. The semantics is given by a tree whose nodes represent function applications that were evaluated during execution. This tree is useful in situations where a high-level view of a computation is needed, such as declarative debugging. The main contribution of the paper is the treatment of higher-order functions, which have previously proven difficult to support in declarative debugging schemes.

1 Introduction

The craft of programming is enhanced by the provision of quality tools. Unfortunately, the availability of debugging tools for non-strict functional languages is limited, a factor considered detrimental to their popularity (Wadler 1998). What is the reason for such paucity? Probably the main technical inhibitor is the high level of abstraction such languages present to the programmer, most notably transparent operational semantics, polymorphism and higher-order functions. On the one hand such features are a boon to productivity. On the other hand the great divide between language and machine makes it more difficult to reconcile the execution of a program and its source code.

The most basic function of a debugger is to present the evaluation of a program in small chunks. The user of the debugger compares the actual behavior (as witnessed by the debugger) with the intended behaviour. Where there is a discrepancy there is a potential bug. An effective debugging session occurs when the discrepancy is accurately identified with a narrow section of program text. Before we build a debugger we must ask ourselves: at what level of abstraction should we present the evaluation of a program?

We opt for a high-level presentation which is abstracted from the underlying evaluation order, and closely reflects the syntactic description of the program. It is generally difficult for programmers to adopt a low-level understanding of non-strict functional programs, so the high-level presentation is quite sensible in this context.

We employ an Evaluation Dependence Tree (EDT) to represent the evaluation of Haskell 98[1] programs. Nodes in the tree represent function applications, storing the name (or representation) of the function, its arguments and result. The EDT is the foundation of many declarative debugging tools, we discuss a number of these in section 7.

Our technique for producing an EDT is by source-to-source transformation. The original program text is transformed into a new program which computes the value of the original program and produces an EDT describing that computation. We begin this paper with a definition of the EDT. We outline the transformation and consider the difficult matter of higher-order programming. We then state the complete transformation as a series of equations over an abstract syntax of the language. We relate our work to the rest of the field and then conclude. Familiarity with Haskell is assumed.

[1](*Haskell 98 Language Report* 2002). Hereafter we drop the *98* from the name.

2 The Evaluation Dependence Tree

The EDT represents an instance of the evaluation of a program. Nodes in the tree correspond to function applications that were made during program evaluation, containing function names and references to their arguments and result. The static call graph of the program determines the dependencies between nodes. Consider the toy program below:

```
main = (\(x,_) -> x) (foo True False)
foo x y = (not y, not (xor x y))
not True = False
not False = True
xor True x = not x
xor False x = x
```

Figure 1 illustrates the EDT corresponding to the evaluation of `main`. The node for `main` has two children, one for the inner call to `foo`, and one for the outer call to the lambda abstraction '`\(x, _) -> x`', which extracts the first value from a pair. Lambda abstractions are anonymous when not bound to an identifier, as above. The textual definition of an anonymous lambda abstraction is used to name the function in the EDT. In all other cases we use the identifier to which the function is bound. The function `foo` has three children corresponding to the two calls to `not`, and the one call to `xor`. The lambda abstraction, `not` and `xor` form the terminals of the tree because they do not call any other functions.

The arguments and results of applications are drawn inside the nodes, though in practice we merely store references to those values. When a node in the tree is displayed the references are followed and the values are printed accordingly. Printing can interact badly with non-strict evaluation. In the example program only the first component of the tuple returned by '`foo True False`' is necessary to compute the value of `main`. Unevaluated expressions, such as the second component of the tuple, are shown as '`?`'.

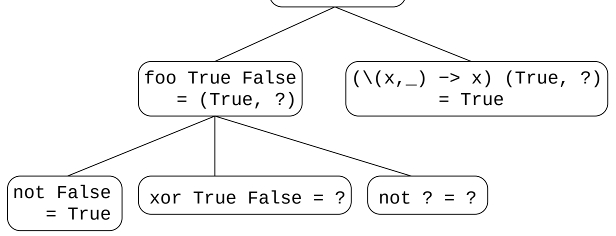

Figure 1: An example EDT.

To accurately represent the computation they must not be evaluated further.

It is impossible to determine whether an expression is evaluated from within Haskell. Therefore, to print values refered to by the EDT we have to momentarily step outside the confines of pure functional programming. Our current implementation calls C code through a foreign function interface. The C code inspects the Haskell heap and generates a printable representation of a desired value from the program, carefully observing which parts of the value are in evaluated form and which are not. To ensure that all values are in their final state of evaluation when viewed, we force the computation of the original program to completion before we traverse the EDT. The inspection of the Haskell heap requires support from the runtime environment.[2]

The EDT is implemented with the following type:

```
data EDT = EDT Exp [Val] Val [EDT]
```

The type has one constructor with four arguments: the name (or representation) of the function involved in an application; a list of references to the arguments of an application; a reference to the result of the application; and a list of children trees. The types `Exp` and `Val`, for representing functions and references, are discussed in section 3. Additional information can be kept in the nodes, such as type annotations, and source code references, however we do not show them here.

We have been slighlty lax in our description of the dependencies between nodes in the tree. In particular we have overlooked higher-order code. Functions are first-class in Haskell, which means that they can be passed as arguments, returned as results and stored inside data structures. To a certain extent this blurs the notion of *syntactic* dependency between function applications, since the application of a function which is passed as an argument to another function may depend on the dynamic behaviour of the program. We address this issue in section 5.

3 Representing Values in the EDT

The type `Val` is a reference to a value from a computation. To implement this type we want encapsulated polymorphism — internally a reference may point to a value of any type, but externally that type should be hidden, so that we can have references to values of different types in the same tree. This can be achieved with existential types, which are not part of the Haskell standard, but are widely supported:

```
data Val = forall a . V a
```

The explicit `forall` occurs in a negative context, which is the same as existentially quantifying the type variable '`a`'.

[2]We have implemented this for the Glasgow Haskell Compiler (GHC), www.haskell.org/ghc.

ues during the exploration of the EDT. A value is printed by passing a `Val` to C code which inspects and traverses the value's heap representation. We provide the following interface to the foreign code:

```
printVal :: Val -> IO ()
```

This function takes a reference to any value and performs a side-effect which prints a representation of the value. The C code is untyped, so it may freely observe the representation of the underlying value, regardless of its type. The impurity of the C code is contained by making it an I/O action, which in Haskell is denoted by the type constructor `IO`. Printing values of primitive type such as `Int` and `Char` is straightforward. Values of composite types such as tuples and algebraic types can be printed providing that constructor names are available, and that there is a mechanism for traversing the components of the value. Ordinarily the compiler would not retain the names of constructors in the executable code of the program, however, in GHC they can be included by special request. The printer must be able to distinguish between values which are computed and those which are not, and it must be able to determine when a value is cyclic. These capabilities depend on the heap representation of values used by the compiler, and what facilities the runtime environment provides for external inspection of the heap. Thankfully GHC is quite generous in this respect, and all these requirements are easily fulfilled. For functional values we make no assumptions about how they are compiled or their runtime representation. Instead we explicitly encode a representation into the program which is derived from the syntax of the function. If ever a functional value is needed to be displayed we interpret the encoded representation which will be stored like any other value on the heap. We discuss this approach thoroughly in section 5.

The name of a function stored in a node is either an identifier or a lambda abstraction. We represent both of these with the data-type `Exp`:

```
data Exp = ExpId Ident
         | ExpAp [Exp]
         | ExpLambda [Pat] Exp
         | ExpLet [Equation] Exp
         | ExpCase Exp [Alt]
         | ExpVal Val
```

This type is an abstract syntax for Haskell expressions, though we have truncated the definition for brevity. Identifiers are formed by '`ExpId` i', where i is a string naming the identifier. Lambda abstractions are formed by '`ExpLambda` ps e', where ps are bound patterns and e is the right-hand-side of the expression. When a node from the EDT is displayed the name of the function is pretty-printed, for example '`ExpId "not"`' would be printed as '*not*'. Applications, let and case expressions are formed by the `ExpAp`, `ExpLet`, and `ExpCase` constructors. We need a full abstract syntax, including patterns, alternatives and equations because we want to be able to show a complete lambda abstraction which may contain all of these syntactic constructs. Free variables in an expression are represented simply as a reference to the value of the variable by the `ExpVal` constructor (notice its argument is of type `Val`). Therefore, pretty printing an `Exp` may involve calls to `printVal`. In the rest of this paper we presume the existence of a function, called $\mathcal{M}$, which produces an appropriate value of type `Exp` for a given piece of syntax, in practice it is trivial to implement, and to save space we do not give its definition. This function will be used extensively when we define the rules of the transformation in section 6. In section 5 we will revisit the `Exp` type when we consider the encoding of functional values.

Each function is transformed to compute its original value paired with an EDT node. As a first approximation, we might transform the function `foo` from our toy program as follows:

```
foo x y = let (v1, t1) = not y
              (v2, t2) = xor x y
              (v3, t3) = not v2
              result = (v1, v3)
              node = EDT (ExpId "foo")
                         [V x, V y]
                         (V result)
                         [t1, t2, t3]
          in (result, node)
```

The new right-hand-side constructs an EDT node (called `node`). Calls to `not` and `xor` are "flattened" (nested calls are eliminated by introducing new variables) and their EDTs become the children of the node for `foo`. The original value of the function is bound to the variable `result` and it is returned in a pair with `node`.

The explicit naming and threading of nodes, as in `t1`, `t2`, and `t3`, is cumbersome, and complicates the transformation. To avoid this clutter we take advantage of a popular functional idiom called *monadic programming*, whereby, the plumbing of the EDT nodes in the transformed program can be hidden. The result is shorter transformation rules, and smaller transformed programs.

Each function is given a new (additional) parameter representing its siblings. When a function is called it constructs a node and prefixes it onto the list. The function returns a pair containing its original value and the new list of nodes (with its own node at the front). To construct its own node, a function must collect its children. The children are formed by function applications that occur on the right-hand-side of the function definition. A list of nodes is threaded through the body of the function. Initially the list is empty. As each child is called a new node is added to the list, so that all children will appear in the list when the evaluation of the body is complete.

To accommodate the threading of nodes, the type of each function is modified to include the new parameter and result. If the result of the original function has type '`t`', the result of transformed function has type '`Comp t`', where `Comp` is the following type synonym:

```
type Comp a = [EDT] -> (a, [EDT])
```

The new functions can be viewed as state transformers, where the state is a list of nodes, and the transformation of the state involves adding new nodes to the list. As Wadler and others have shown, monads offer a neat way to model state transformation in functional languages (Wadler 1993). Haskell has some helpful syntax for monads (called do-notation) that we will use to hide the threading of state through our transformed program.

To enable the do-notation we need a little bit of plumbing:

```
bind :: Comp a -> (a -> Comp b) -> Comp b
bind comp next
  = \s1 -> case comp s1 of
               (val, s2) -> next val s2
```

The role of `bind` is to join two transformed computations together by plumbing the state through each of them. The variable `s1` is the initial state, and it is passed to the first computation resulting in a pair `(val, s2)`, such that `val` is the value of the computation and `s2` is the output state. The value and the

A sequence of computations can be joined together by nested applications of `bind`.

Do-notation has the following syntax:

$$\begin{array}{lcl} m & \in & \texttt{do}\{s_1;\ \ldots;\ s_n\},\ n > 0 \\ s & \in & e \ \mid\ p \leftarrow e \ \mid\ \texttt{let}\ d_1\ \ldots\ d_n,\ n > 0 \end{array}$$

The variables e, p, and d refer to expressions, patterns and declarations, we give their syntax in section 6. Essentially, the do-notation is a sequence of statements (denoted by s). In our context each statement corresponds to a transformed computation. By overloading the syntax with respect to the `bind` function we can write a sequence of computations without explicit mention of the threaded state. The following rules show how do-notation is de-sugared by the Haskell compiler:

$$\begin{array}{rcl} \texttt{do}\{e\} & = & e \\ \texttt{do}\{e;\ \mathit{stmts}\} & = & \texttt{bind}\ e\ (\lambda\ \_\ .\ \texttt{do}\{\mathit{stmts}\}) \\ \texttt{do}\{p\ \leftarrow\ e; \mathit{stmts}\} & = & \texttt{bind}\ e\ (\lambda\ p\ .\ \texttt{do}\{\mathit{stmts}\}) \\ \texttt{do}\{\texttt{let}\ d; \mathit{stmts}\} & = & \texttt{let}\ d\ \texttt{in}\ \texttt{do}\{\mathit{stmts}\} \end{array}$$

In certain circumstances, such as the evaluation of a constant, we will need to thread the state through a computation unmodified. This is done by wrapping the computation inside a call to `return`:

```
return :: a -> Comp a
return x = \s -> (x, s)
```

For example, converting the constant `True` into '`return True`' results in a passive state transformer that computes the constant and passes the state unchanged on to the next computation.

The following function constructs an EDT node:

```
edt :: Exp -> [Val] -> Comp a -> Comp a
edt f args comp
   = do let (v, s) = comp []
        add (EDT f args (V v) s)
        return v
   where add n = \s -> ((), n:s)
```

It is a state transformer with three arguments: the name of the function to store in the node (`f`), a list of references to the function's arguments (`args`), and the value of the transformed right-hand-side of the function (`comp`). An empty list of children is passed to the transformed right-hand-side, which evaluates to a pair containing the original value of the computation (`v`) and the final list of children (`s`). A node for the application is constructed and added to the parent's list of siblings by the state transformer `add`. The original value of the right-hand-side is then returned.

Armed with do-notation, `return` and `edt` the transformation of `foo` is simplified:

```
foo x y = edt (ExpId "foo")
              [V x, V y]
              (do v1 <- not y
                  v2 <- xor x y
                  v3 <- not v2
                  return (v1, v3))
```

5 Transforming Higher Order Code

Higher-order programming is fundamental in Haskell and functions are first-class. Furthermore, functions in Haskell are *curried* — multi-argument functions may be applied to fewer arguments than their arity. The arity of a function is determined by the number of patterns it binds on the left-hand-side of its definition. For example, though '`xor`' has arity two, it is

True', the result being a function equivalent to not. With currying it is possible to create new functions by partially applying existing ones. Lambda abstractions allow anonymous functions to be defined, such as '\(x,_) -> x'. These features cause several problems for transformation-based debugging. Consider the following code fragment:

```
foo ... = ... (f x y)
```

We might expect that an EDT node for `foo` has a child of the form `f x y = z`. However, this assumes the arity of `f` is two (the expression is *saturated*). If the arity is greater, more arguments are expected (the expression is a partial application or *under-saturated*) and the expression is not reduced, hence there should be no corresponding child of `foo` for this application of `f`. With a smaller arity the expression is *over-saturated* and more than one child should be produced. For example, if `f` has arity one, there should be two children, of the form `f x = g` and `g y = z`. Unfortunately, the arity of `f` may be unknown at transformation time — `f` may be an argument of `foo` and different calls to `foo` may have different arity functions as arguments. An advantage of using state transformers is that the construction of the children of `foo` is not done directly by `foo`. It is done by the transformed version of the expression `f x y`, hence avoiding (or at least delaying) the problem of how many children are constructed.

If `f` is an argument of `foo` another problem occurs. Ultimately we want a printable representation of the function in each EDT node but we can't simply use the string `"f"` — we need a representation of the function `f` is bound to at runtime. It may be bound to a function `g` of large arity applied to several values, and the applications may have been done in disparate parts of the computation. Alternatively, the function may be defined with a lambda expression. We cannot expect to derive a reasonable print representation for functions from the heap as we do for first order data structures. While there is a straightforward relationship between a list of integers in the program (say) and the heap representation, the relationship between a lambda abstraction (which could be almost the entire program) and its heap representation is complex to say the least. Even if we wrote a de-compiler, the result may be unrecognisable to the programmer. Thus, as mentioned in section 3, we *encode* representations of higher-order arguments so that they may be pretty-printed when we explore the EDT. Our encoding of `f` also allows us to construct the right number of children of `foo`.

In this section we first give a simplified description of how under- and over-saturated expressions are transformed, then describe how function representations are created and manipulated. We defer discussion of lambda expressions until section 6. We will use the following two functions as examples:

```
ap :: (a -> b) -> a -> b
ap f x = f x

const :: a -> b -> a
const x y = x
```

Following from our informal treatment of the transformation in the previous section of the paper, we might transform each of these respectively to:

```
ap :: (a -> Comp b) -> a -> Comp b
ap f x = edt (ExpId "ap")
             [V f, V x]
             (do v1 <- f x
                 return v1)
```

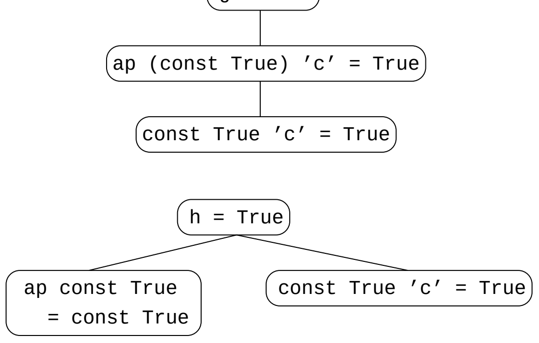

Figure 2: EDTs with higher-order functions

```
const :: a -> b -> Comp a
const x y = edt (ExpId "const")
                [V x, V y]
                (do return x)
```

The first argument of `ap` is a function of type 'a -> b'. As with all functions, the transformed version must be extended to take a list of EDT nodes as an additional argument, and when applied it must generate a value and a list of EDT nodes. Therefore, the type of the first argument must change to reflect the new behaviour, becoming 'a -> Comp b'. This is reflected in the type scheme for `ap`.

5.1 Under-saturated expressions

Consider `g`, which uses `ap` and `const`:

```
g = ap (const True) 'c'
```

The application of `const` on the right-hand-side of `g` is partial — `const` expects two arguments but it is only given one. Partial applications are not reducible expressions, therefore we do not want to record them in the EDT. Applications which are known to be partial are therefore treated specially in the transformation. The type of 'const True' is 'a -> Bool', so in the transformed program it will become 'a -> Comp Bool'. We do not flatten this application, but simply pass it as an argument to `ap`, where it will be bound to `f`:

```
g = edt (ExpId "g") []
        (do v1 <- ap (const True) 'c'
            return v1)
```

The EDT that results from the evaluation of `g` is shown in figure 2. Note that the application of `const` becomes a child of `ap`. The reason is as follows. The variable `f` is bound to 'const True', and the variable `x` is bound to `'c'`. The application 'f x' is therefore equivalent to 'const True 'c''. Thus the expression 'f x' on the right-hand-side of `ap` corresponds (at runtime) to a full application of `const`.

5.2 Over-saturated expressions

The following slightly altered definition offers an interesting comparison:

```
h = ap const True 'c'
```

Both `h` and `g` denote the same value, yet they are transformed differently and result in different EDTs. On the right-hand-side of `h`, `const` is partially applied to zero arguments, where previously it was given one. Furthermore, in `h`, `ap` is given three arguments, when may return a function as its result. We can make this explicit with parentheses: '(ap const True) 'c''. The evaluation of 'ap const True' results in a function, namely 'const True', which is then applied to the character 'c'. The application of const occurs on the right-hand-side of h, therefore const is a child of h rather than ap. The EDT resulting from h is given in figure 2. Transforming h is more challenging. We defer this task until after we have considered encoding function representations.

5.3 Printable represenations of functions

We modify the transformation such that when a higher-order value is created (by partial application, or by lambda abstraction) a representation of the function is derived from its source code and paired with the function. We use the type Exp, introduced in section 3, to encode the function, and encapsulate the function and representation in the type F:

```
data F a b = F (a -> Comp b) Exp
```

The encapsulated function has type 'a -> Comp b' because it it the transformed version of the function that it refers to. To select the function from the encapsulation we use apply:

```
apply :: F a b -> a -> Comp b
apply (F f _) = f
```

The type of higher-order arguments must be modified to reflect the encoding, and applications of encoded functions must be preceeded by a call to apply. Therefore we modify our transformation of ap:

```
ap :: F a b -> a -> Comp b
ap f x = edt (ExpId "ap")
             [V f, V x]
             (do v1 <- apply f x
                 return v1)
```

Calls to ap will need to provide an encoding of the first argument. For example, the function 'const True', in the transformed version of g will have to be encoded. The representation that we choose is simply the source code expressed in the Exp type. To assist the encoding we introduce the function fun1 which encodes functions of arity one:

```
fun1 :: (a -> Comp b)
        -> Exp -> Comp (F a b)
fun1 f e = return (F f e)
```

The first argument to fun1 is the transformed function, the second is its representation, the result is an encapsulation of the two, returned in the Comp type so that it is convenient to use in the do-notation.

The final transformation of g is as follows:

```
g = edt (ExpId "g") []
        (do v1 <- fun1
                    (const True)
                    (ExpAp
                       [ExpId "const",
                        ExpId, "True"])
            v2 <- ap v1 'c'
            return v2)
```

The expression 'ExpAp [ExpId "const"...]', of type Exp, encodes the function 'const True'. The function and its representation are encapsulated by fun1, resulting in a value of type 'F a Bool' which is bound to v1 and subsequently given as the first argument to ap.

Transformation of h requires an encoding of const, in the expression 'ap const True'. This is

sult of the application is also a function, namely 'const True', which must also be encoded. Clearly the encoding of the output function is dependent on the encoding of the input function. Also the encoding of the output function is dependent on the value of x which is an argument of ap, and can only be known at runtime. What we want to do is build up the representation of a partial application by adding representations of new arguments as they are provided (one at a time). For functions of arity two (such as const), we encapsulate the function with its representation, as before with F, but we trasform the function such that when applied to a value it returns a new encoding of that application. This whole process is performed by fun2:

```
fun2 :: (a -> b -> Comp c)
        -> Exp -> Comp (F a (F b c))
fun2 f e
  = return
      (F (\v -> fun1 (f v) (eAp e v)) e)

eAp :: Exp -> a -> Exp
eAp e v = ExpAp [e, ExpVal (V v)]
```

This is intricate, and best understood by example. The first argument to fun2 is a (transformed) function of arity two, and the second argument is its encoding. For const we would generate the expression: 'fun2 const (ExpId "const")'. Expanding the inner lambda abstraction by substituting for f and e gives:

```
\v -> fun1 (const v)
           (eAp (ExpId "const") v)
```

and, expanding the call to fun1 gives:

```
\v -> return (F (const v)
                (eAp (ExpId "const") v))
```

and, expanding the call to eAp gives:

```
\v -> return (F (const v)
                (ExpAp [ExpId "const",
                        ExpVal (V v)]))
```

The type of this expression is 'a -> Comp (F b a)'. Given *one* argument, this will return an encapsulation of const applied to that argument *and* a representation of that application derived from the representation of const. The derived representation is given by 'ExpAp [ExpId "const", ExpVal (V v)]' which constructs an Exp representing the application of const to whatever the value of v is. In the context of h, v will eventually be bound to True.

Using fun2 we can transform h as follows:

```
h = edt (ExpId "h") []
        (do v1 <- fun2 const
                       (ExpId "const")
            v2 <- ap v1 True
            v3 <- apply v2 'c'
            return v3)
```

The variable v1 will be bound to 'F ... (ExpId "const")', where '...' is the lambda abstraction unfolded above. The type of v1 is 'F a (F b a)', and it is given as the first argument to ap, and thus bound to the variable f. On the right-hand-side of ap, 'apply f x' selects the function from the encapsulation and applies it to x, which is bound to True. The result is:

```
return (F (const True)
          (ExpAp [ExpId "const",
                  ExpVal (V True)]))
```

tion 'const True' and its representation. The type of which is 'Comp (F b Bool)'. The outer call to return simply makes the use of do-notation convenient. Recall that the result of 'ap const True' is a function, the good news is that after encoding we now have a representation of this function that is easy to print: 'ExpAp [ExpId "const", ExpVal (V True)]'. The result of 'ap v1 True' is the encapsulation of 'const True' and the above representation, and it is bound on the right-hand-side of h to v2. The application 'apply v2 'c'' retrieves the function from the encapsulation and applies it to the character 'c'. This is a full application of const and will result in a node being inserted into the EDT under h. The result of the application (True) is bound to v3 and subsequently returned as the result of h.

So far we have shown how to transform partial applications of arity one and two. What about higher aritites? The same principle applies for higher arities, however, we need a family of functions similar to fun2:

```
funₙ :: (v₁ -> ... vₙ -> Comp vₙ₊₁)
        -> Exp
        -> Comp (F v₁ ... (F vₙ vₙ₊₁) ...)
funₙ f e
  = return
      (F (\v -> funₙ₋₁ (f v) (eAp e v)) e)
```

A partial application of arity n is encoded by fun$_n$, however, fun1 remains as before. The maximum necessary value of n is unknown until we have traversed the whole program. At the end of transformation we ensure that the appropriate number of fun$_n$ functions are generated. For most code this number will be small because high arity functions are uncommon in human written programs.

The arity of a partial application is simply the arity of the function being applied minus the number of arguments in the application. When the function is named by a variable, we only know its arity if the variable is bound by the use of =. Such variables are often called *let*-bound. However we do not know the arity of functions that are bound in patterns. Therefore encodings are only made for partial applications of let-bound variables. Of course pattern variables will be bound to encoded functions at runtime, and must be decoded (by apply) when they are applied. In section 6 we will see that the binding occurrence of a variable will determine how applications of that variable are transformed. Partial application of data constructors are treated in a similar way to let-bound functions, since the arity of constructors can be derived from their definition. Function applications are over-saturated when the function is given more arguments than its arity (such as the application 'ap const True 'c'' in h). We break such expressions into a full application of the function ('ap const True') which returns an encoded functional result, and residual applications, one for each of the remaining arguments (in this case just 'c'). The result of all but the last residual applications will be a new encoded function that must be decoded by apply before it may be used.

6 The transformation

In this section we state the program transformation as a series of rules over a core abstract syntax for Haskell, listed in figure 3. To save space we overlook syntactic sugar which can be translated into the core language (such as do-notation, list comprehensions, guarded equations, and where clauses etc.).

written in italics. We presume the following sets of variables for the atoms:

$$\begin{array}{rcl} f & \in & \text{Type constructors (eg } \texttt{Bool}) \\ v & \in & \text{Type variables (eg } \texttt{a}) \\ x, y & \in & \text{Variables (eg } \texttt{const}) \\ c & \in & \text{Data Constructors (eg } \texttt{True}) \end{array}$$

Syntactic variables are sometimes annotated with salient attributes: $x^\star$ denotes a pattern bound variable, x^n denotes a let-bound variable with arity n, c^n denotes a data-constructor with arity n, and $\hat{x}$ denotes a fresh variable, unique in the scope that is is introduced. When more than one variable of the same type is needed we use numeric subscripts to distinguish them.

The transformation rules are equations written in a functional style, which collectively can be understood as a pure functional program. Each rule is named by an uppercase calligraphic letter. Double square brackets '$[\![\ ]\!]$' enclose arguments which denote a syntactic entity, such as an expression, or declaration and so on. Terms appearing in typewriter font are to be interpreted verbatim, for example return refers to the function of that name defined in section 4. Ellipses indicate obvious sequences that do not require full representation.

6.1 Types

A type is either a type variable, a functional type from one type to another, or the application of a type constructor to type arguments:

$$t \ \in \ v \ | \ t_1 \to t_2 \ | \ f\ t_1\ \ldots\ t_n,\ n \geq 0$$

Rules 1 – 3 ($\mathcal{T}$) transform types. All occurrences of the function arrow ($\to$) are replaced by the type constructor F, to accommodate the encapsulation and encoding of higher-order values.

6.2 Data Constructor Declarations

A data constructor declaration names the constructor and lists the zero or more type arguments of the constructor:

$$k \ \in \ c\ t_1\ \ldots\ t_n,\ n \geq 0$$

Rule 4 ($\mathcal{K}$) transforms constructor declarations, by mapping $\mathcal{T}$ over each of the arguments to the constructor.

6.3 Patterns

Patterns are used to bind variables in the arguments of functions and in case alternatives. They are either a variable, an *as*-pattern (a pattern named by a variable), or the application of a data constructor to zero or more patterns:

$$p \ \in \ x \ | \ x@p \ | \ c\ p_1\ \ldots\ p_n,\ n \geq 0$$

There are no rules that deal directly with patterns.

6.4 Declarations

A declaration is either a variable with a type annotation (type signature), a function binding, or a algebraic type declaration:

$$\begin{array}{rcl} d & \in & x :: t \ | \\ & & x\ p_1\ \ldots\ p_n \ = \ e,\ n \geq 0 \ | \\ & & \texttt{data}\ f\ v_1\ \ldots\ v_n \ = \ k_1\ \ldots\ k_m, \\ & & \qquad n \geq 0,\ m > 0 \end{array}$$

$$\mathcal{T}[\![v]\!] \;=\; v \tag{1}$$

$$\mathcal{T}[\![t_1 \;\rightarrow\; t_2]\!] \;=\; \texttt{F}\;\mathcal{T}[\![t_1]\!]\;\mathcal{T}[\![t_2]\!] \tag{2}$$

$$\mathcal{T}[\![f\;t_1\;\ldots\;t_n]\!] \;=\; f\;\mathcal{T}[\![t_1]\!]\;\ldots\mathcal{T}[\![t_n]\!] \tag{3}$$

$$\mathcal{K}[\![c\;t_1\;\ldots\;t_n]\!] \;=\; c\;\mathcal{T}[\![t_1]\!]\;\ldots\;\mathcal{T}[\![t_n]\!] \tag{4}$$

$$\begin{aligned}\mathcal{S}\;0\;[\![t]\!] \;&=\; \texttt{Comp}\;(\mathcal{T}[\![t]\!])\\ \mathcal{S}\;n\;[\![t_1\;\rightarrow\;t_2]\!] \;&=\; \mathcal{T}[\![t_1]\!]\;\rightarrow\;\mathcal{S}\;(n-1)\;[\![t_2]\!]\end{aligned} \tag{5}$$

$$\mathcal{D}[\![x^n\;::\;t]\!] \;=\; x\;::\;\mathcal{S}\;n\;[\![t]\!] \tag{6}$$

$$\mathcal{D}[\![x\;p_1\;\ldots\;p_n\;=\;e]\!] \;=\; x\;\hat{y}_1@p_1\;\ldots\;\hat{y}_n@p_n\;=\;\texttt{edt}\;(\mathcal{M}[\![x]\!])\;[\texttt{V}\;\hat{y}_1,\;\ldots,\;\texttt{V}\;\hat{y}_n]\;(\mathcal{E}[\![e]\!]) \tag{7}$$

$$\mathcal{D}[\![\texttt{data}\;f\;v_1\;\ldots\;v_n\;=\;k_1\;\ldots\;k_m]\!] \;=\; \texttt{data}\;f\;v_1\;\ldots\;v_n\;=\;\mathcal{K}[\![k_1]\!]\;\ldots\;\mathcal{K}[\![k_m]\!] \tag{8}$$

$$\mathcal{A}[\![p\;\rightarrow\;e]\!] \;=\; p\;\rightarrow\;\mathcal{E}[\![e]\!] \tag{9}$$

$$\mathcal{E}[\![x^\star]\!] \;=\; \texttt{return}\;x \tag{10}$$

$$\mathcal{E}[\![x^0]\!] \;=\; x \tag{11}$$

$$\mathcal{E}[\![x^n]\!] \;=\; \texttt{fun}_n\;x\;(\mathcal{M}[\![x]\!]) \tag{12}$$

$$\mathcal{E}[\![c^0]\!] \;=\; \texttt{return}\;c \tag{13}$$

$$\mathcal{E}[\![c^n]\!] \;=\; \texttt{fun}_n\;(\lambda\;\hat{x}_1\;\ldots\;\hat{x}_n\;.\;\texttt{return}\;(c\;\hat{x}_1\;\ldots\;\hat{x}_n))\;(\mathcal{M}[\![c]\!]) \tag{14}$$

$$\mathcal{E}[\![e\;::\;t]\!] \;=\; \mathcal{E}[\![e]\!]\;::\;\texttt{Comp}\;(\mathcal{T}[\![t]\!]) \tag{15}$$

$$\mathcal{E}[\![\texttt{let}\;d_1\;\ldots\;d_n\;\texttt{in}\;e]\!] \;=\; \texttt{let}\;\mathcal{D}[\![d_1]\!]\;\ldots\;\mathcal{D}[\![d_n]\!]\;\texttt{in}\;\mathcal{E}[\![e]\!] \tag{16}$$

$$\begin{aligned}\mathcal{E}[\![\lambda\;p_1\;\ldots\;p_n\;.\;e]\!] \;=\; &\texttt{let}\;\hat{x}\;=\;\mathcal{M}[\![\lambda\;p_1\;\ldots\;p_n\;.\;e]\!]\;\texttt{in}\\ &\texttt{fun}_n\;(\lambda\;\hat{y}_1@p_1\;\ldots\;\hat{y}_n@p_n\;.\;\texttt{edt}\;\hat{x}\;[\texttt{V}\;\hat{y}_1,\;\ldots,\;\texttt{V}\;\hat{y}_n]\;(\mathcal{E}[\![e]\!]))\;\hat{x}\end{aligned} \tag{17}$$

$$\mathcal{E}[\![\texttt{case}\;e\;\texttt{of}\;a_1\;\ldots\;a_n]\!] \;=\; \texttt{do}\;\{\;\hat{x}\;\leftarrow\;\mathcal{E}[\![e]\!];\;\texttt{case}\;\hat{x}\;\texttt{of}\;\mathcal{A}[\![a_1]\!]\;\ldots\;\mathcal{A}[\![a_n]\!]\;\} \tag{18}$$

$$\begin{aligned}&\mathcal{E}[\![x^n\;e_1\;\ldots\;e_m]\!]\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\;x\;\hat{y}_1\;\ldots\;\hat{y}_m\;\}\quad(n=m)\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\;\texttt{fun}_{n-m}\;(x\;\hat{y}_1\;\ldots\;\hat{y}_m)\;\mathcal{M}[\![x\;e_1\;\ldots\;e_m]\!]\;\}\quad(n>m)\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\;\hat{z}_1\;\leftarrow\;x\;\hat{y}_1\;\ldots\;\hat{y}_n;\\ &\qquad\quad\hat{z}_2\;\leftarrow\;\texttt{apply}\;\hat{z}_1\;\hat{y}_{n+1};\;\hat{z}_3\;\leftarrow\texttt{apply}\;\hat{z}_2\;\hat{y}_{n+2};\;\ldots;\;\texttt{apply}\;\hat{z}_{m-n}\;\hat{y}_m\}\quad(n<m)\end{aligned} \tag{19}$$

$$\begin{aligned}&\mathcal{E}[\![x^\star\;e_1\;\ldots\;e_m]\!]\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\;\hat{z}_1\;\leftarrow\;\texttt{apply}\;x\;\hat{y}_1;\\ &\qquad\quad\hat{z}_2\;\leftarrow\;\texttt{apply}\;\hat{z}_1\;\hat{y}_2;\;\ldots;\;\texttt{apply}\;\hat{z}_{m-1}\;\hat{y}_m\;\}\end{aligned} \tag{20}$$

$$\begin{aligned}&\mathcal{E}[\![c^n\;e_1\;\ldots\;e_m]\!]\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\;\texttt{return}\;(c\;\hat{y}_1\;\ldots\;\hat{y}_m)\;\}\quad(n=m)\\ &\quad=\;\texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\ldots;\;\hat{y}_m\;\leftarrow\;\mathcal{E}[\![e_m]\!];\\ &\qquad\quad\texttt{fun}_{n-m}\;(\lambda\;\hat{y}_{m+1}\;\ldots\;\hat{y}_n\;.\;\texttt{return}\;(c\;\hat{y}_1\;\ldots\;\hat{y}_m\;\hat{y}_{m+1}\;\ldots\;\hat{y}_n))\;\mathcal{M}[\![c\;e_1\;\ldots\;e_m]\!]\;\}\quad(n>m)\end{aligned} \tag{21}$$

$$\mathcal{E}[\![e_1\;e_2]\!] \;=\; \texttt{do}\;\{\;\hat{y}_1\;\leftarrow\;\mathcal{E}[\![e_1]\!];\;\hat{y}_2\;\leftarrow\;\mathcal{E}[\![e_2]\!];\;\texttt{apply}\;\hat{y}_1\;\hat{y}_2\;\}\quad(e_1\neq x, e_1\neq c) \tag{22}$$

Figure 3: The transformation rules

from rule 5 ($\mathcal{S}$) for signatures.

The transformation of a type signature is dependent on the arity of the variable in the declaration. Consider: '`const :: a -> b -> a`'. We can define `const` with different numbers of patterns on the left-hand-side:

```
const x y = x
const x = \y -> x
const = \x y -> x
```

The arities of these are two, one and zero respectively. The type of the transformed function depends on the type of the right-hand-side of the function declaration (see rule 7). In these three definitions the type of the right-hand-side's are different (`a` for the first, '`b -> a`' for the second and '`a -> b -> a`' for the third) because of the different number of patterns to the left of `=`. For a function of arity n, the first $n-1$ type arrows in the spine of the type must be preserved to account for the n patterns in the definition. That is the purpose of rule 5 ($\mathcal{S}$). The result of the function is wrapped in the type `Comp` because the transformed function threads a list of EDT nodes, as outlined in section 4. So although each of the definitions of `const` begin with the same type, after transformation their types are different, namely:

```
const :: a -> b -> Comp a
const :: a -> Comp (F b a)
const :: Comp (F a (F b a)
```

It may seem counter-intuitive that definitions which are equivalent in Haskell are transformed differently. However, this is inevitable. Although `h` and `g` defined in section 5 are both equivalent to `True` according to Haskell, these equivalences may not hold in the intended interpretation of the programmer. EDTs, and the transformations which generate them, must respect the programmer's (mis)conceptions about the program. Although the definitions of `const` are very similar, our solution to the difficult problem of curried functions is to treat them differently.

Function declarations are transformed by rule 7. Fresh variables $\hat{y}$ are introduced to name each of the function patterns (using as-pattern syntax). The new variables are used to name the arguments of the function for the purposes of constructing an EDT node. This is strictly necessary only when the patterns are not variables themselves. The new right-hand-side constructs an EDT for the function by calling `edt` on three arguments: the function name (as generated by $\mathcal{M}$), a list of references to the function's arguments (wrapped in the `V` constructor to make them type `Val`), and the transformed original right-hand-side.

Algebraic types are introduced with the keyword `data`. A declaration names the type and its arguments and then lists one or more constructor declarations. Each of the constructor declarations is transformed by $\mathcal{K}$.

6.5 Alternatives

Alternatives are the branches of case expressions. Each alternative consists of a pattern and an expression. The expression is evaluated if the discriminator of the case expression matches with the pattern:

$$a \ \in \quad p \rightarrow e$$

Rule 9 ($\mathcal{A}$) transforms patterns by applying $\mathcal{E}$ to the expression.

An expression is either an application of two expressions, an expression with an type annotation, a variable, a data constructor, a lambda abstraction, a let expression, or a case expression:

$$\begin{array}{rl} e \ \in & e_1\, e_2 \ \mid\ e :: t \ \mid\ x \ \mid\ c \ \mid \\ & \lambda\, p_1\ \ldots\ p_n\, .\, e,\ n > 0 \ \mid \\ & \texttt{let}\ d_1\ \ldots\ d_n\ \texttt{in}\ e,\ n > 0 \ \mid \\ & \texttt{case}\ e\ \texttt{of}\ a_1\ \ldots\ a_n,\ n > 0 \end{array}$$

Expressions are transformed by rules 10 – 22 ($\mathcal{E}$). The basic philosophy underlying the transformation is as follows. All transformed expressions are state transformers, that is they thread a list of EDT nodes, and may occur as a statement in the do-notation. If an expression definitely denotes a functional value then it must be encoded. If an expression definitely does not denote a function then it is not encoded, but may be turned into a state transformer by the application of `return`. If a transformed function is applied, it must be decoded first.

Rule 10 transforms pattern bound variables into basic state transformers by the application of `return`. Rules 11 and 12 transform let-bound variables. If the variable has arity zero (rule 11), then there is nothing to be done. Unlike pattern variables, we do not need to apply `return` to nullary let-bound variables because their definitions are transformed, and are thus already state transformers. If the variable has arity greater than zero the variable is partially applied, and thus constitutes a function. As discussed in section 5 partial applications require encoding and encapsulation, this is done by $\texttt{fun}_n$, applied to the variable and its representation (n is the arity of the variable). The representation of the variable is generated by $\mathcal{M}$.

Rule 13 transforms nullary constructors. Since they cannot be functions they do not require encoding and are converted into basic state transformers by `return`. Rule 14 transforms constructors of arity greater than zero, which are functions. Before the constructor can be encoded it must be turned into a state transformer, by the use of a lambda abstraction and a call to `return`. As with non-nullary let-bound variables, encoding and encapsulation is done by $\texttt{fun}_n$, where the representation of the constructor is generated by $\mathcal{M}$.

Rule 15 transforms expressions with an explicit type annotation. We include this rule in the presentation because it offers an important insight into the relationship between transformed expressions and their resulting type. In particular, the rule states that if the original expression has type t, the transformed expression has type '$\texttt{Comp}\ (\mathcal{T}[\![t]\!])$' – which is a state transformer, over t, where all higher-order arguments are encoded. The type-correctness of the resulting program depends on this relationship, and it must satisfied by all the equations for $\mathcal{E}$.

Let expressions provide local variable declarations whose scope is restricted to a particular expression. The transformation of these (rule 16) is straightforward: all declarations are transformed by rule $\mathcal{D}$, and the expression is transformed by $\mathcal{E}$.

Rule 17 transforms lambda abstractions. Lambda abstractions introduce new (anonymous) functions in place – their use and declaration are given by the same syntax, unlike let-bound functions whose use and declaration are distinct. Two things must be done for lambda abstractions: their representation must be encoded, and the function must be transformed to produce an EDT in a similar fashion to let-bound function declarations (see rule 7). The representation in the EDT, and to represent the whole expression when it is encoded (in case it is passed as a higher-order argument). For efficiency we reuse the encoding generated by $\mathcal{M}$ by binding it to the variable $\hat{x}$, allowing the two uses to share the one construction. Note that $\hat{x}$ is the first argument to `edt`, and also the second argument to $\texttt{fun}_n$. The construction of the EDT node is the same as for let-bound variables, except for the encoding of the function name.

Case expressions are the primary branching construct in Haskell, and are transformed by rule 18. They consist of an expression (called the discriminator) and a number of alternatives. The discriminator is evaluated and compared with the head of each alternative in turn. The right-hand-side of the first matching alternative becomes the value of the whole case expression. The original discriminator is transformed by $\mathcal{E}$ and its value is bound to the new variable $\hat{x}$ which becomes the discriminator of a new case expression whose alternatives are transformed by $\mathcal{A}$.

Function applications are transformed by rules 19 – 22. It would be sufficient if we only gave rule 22, however this would result in particularly inefficient programs due to redundant encoding and immediate decoding of functional values. To avoid this we include rules 19 – 21 which are specialisations for the common cases of applications where the function is either a variable or a data constructor. Further specialisations are possible, for situations where the function is a lambda abstraction, a case expression or a let expression. To simplify the presentation of the transformation we do not show them here.

Rule 19 transforms applications of let-bound variables, recall from the discussion in section 5 that such applications may be saturated, under-saturated, or over-saturated. The three alternative equations for rule 19 handle each of these situations respectively. In all cases the argument expressions are transformed by $\mathcal{E}$, the values of which are bound to fresh variables $\hat{y}_1 \ \ldots \ \hat{y}_m$. If the application is saturated the last statement in the do-notation is simply the application of the variable to the values of its transformed arguments. If the application is under-saturated the last statement constructs an encapsulation of the whole expression and its representation. The arity of the partial application is $n - m$ where n is the arity of the variable and m is the number of arguments in the application, hence $\texttt{fun}_{n-m}$ is used to generate the encapsulation. As usual $\mathcal{M}$ generates a representation of the application from its syntax. If the application is over-saturated then the variable is applied to its expected number of (transformed) arguments. The result is an encapsulated function, which is bound to the fresh variable $\hat{z}_1$. Each of the residual applications (including that of $\hat{z}_1$) must be performed one at a time, and the intermediate encapsulated functions must be selected from their encapsulation by `apply`. It is worth noting that rules 11 and 12 for un-applied occurrences of let-bound variables are just special cases of the first two equations for rule 19, which can be derived by making the number or arguments in the application zero.

Rule 20 transforms applications of pattern variables. Each of the arguments in the application is transformed by $\mathcal{E}$ and the values of each are bound to fresh variables using the do-notation. Pattern variables which are applied must be bound to functions, and those functions will be transformed and encoded. Due to the transformation of higher-order arguments, the functions that pattern variables are bound to are encapsulated in the type `F`. Thus when we apply the function we must select it from the encapsulation using `apply`. Where there is more than one argument

a time because after encoding the result of each application is a new encapsulated function.

Rule 21 transforms applications of data-constructors. Unlike let-bound variables, data-constructors cannot be over-saturated, hence there are only two alternative equations for the application of constructors, the first is for saturated applications and the second is for partial applications. In both cases the transformation follows almost directly from the one used for let-bound variables, except that the application must be turned into a state transformer by `return`. Again it is worth noting that rules 13 and 14 for un-applied constructors are just special cases of the two equations for rule 21, and can be derived by making the number of arguments in the application zero.

The final rule (22) transforms applications that do not match the previous three rules, namely applications of let, case and lambda expressions. The function and argument are transformed by $\mathcal{E}$ and their values are bound to fresh variables. Since the left expression is a function, it will be encapsulated after transformation and must be selected from the encapsulation by `apply` before it can be applied.

6.7 Type Classes

Haskell also has type classes which allow the definition of (let-bound) functions to be overloaded with respect to their type. For example, the function '`+`' can be used to add two integers, or two floating point numbers, or any two numerical types for which it has a corresponding definition. Two changes to our core syntax are required to include type classes into the language: qualified types — types are extended with constraints over type variables to indicate overloaded entities, and new function binding rules for defining class interfaces and instances. Handling qualified types is straightforward: leave the constraint unchanged, and transform the type as usual. Class instances and declarations are almost always transformed by the normal rules for declarations, however, in certain circumstances two or more declarations of the one overloaded function may have different numbers of patterns and hence different arities. Our transformation relies on knowing each let-bound variable's arity, so we must ensure that all instances of an overloaded function have the same number of patterns. Where there is a difference in the arity of an overloaded function we use the number of arrows in the spine of the function's type scheme to determine the arity of the function and adjust the number of patterns in each declaration to suit by eta-conversion.

7 Related work

The use of an EDT for declarative debugging is well known: (Naish & Barbour 1996, Pope 1998, Sparud 1999, Nilsson 2001, Caballero & Rodri'guez-Artalejo 2002). The main detractor of the earlier approaches being a lack of support for higher-order programming. We have outlined the difficulty of supporting higher-order programs in section 5.

The first known complete solution to supporting higher-order functions for a declarative debugger based on program transformation is in (Caballero & Rodri'guez-Artalejo 2002). They require multiple intermediate functions to be introduced into the program for every curried function, which we avoid by treating transformed functions as state transformers. They also require the creation of empty nodes in the EDT for partial applications, whereas we only create tion encodes the representation of higher-order values, however, they require the names of functions to be provided by the runtime environment, which is unreasonable for Haskell. Their transformation is described for a very simple functional language, which does not include let, case or lambda expressions.

Sparud gives a program transformation for declarative debugging of Haskell (Sparud 1999). He provides a rich set of combinators to simplify the transformed program, and the state transformers in our work are inspired by this. He does not use the do notation, though the difference is largely a matter of presentation. Unfortunately this work only supports some types of higher-order programming. For printing values, he uses type-classes to provide an overloaded function which produces a representation for values in the program. This still requires support from the runtime environment to determine whether an expression is evaluated, or cyclic. In some cases the overloading will be ambiguous, and it is difficult to resolve the ambiguity without detailed type information during transformation. There is no implementation available for this work.

Nilsson (Nilsson 2001) uses an instrumented runtime environment to construct an EDT as a side-effect of computation. The advantage of this approach is that it allows for greater access to the runtime representation of values. A technique called *piecemeal tracing* is employed to constrain the memory consumed by the EDT, by placing an upper limit on the size of memory occupied by the EDT at any one time. Re-computation of part of the program is required to generate branches of the tree that do not fit into memory. The disadvantage of this approach is the complexity of implementation. A whole new compiler for a large subset of Haskell was created for the purposes of providing the necessary instrumented runtime environment. Our motivation for employing program transformation is to simplify the implementation of the debugger and to take advantage of existing compiler technology.

A general framework for tracing, debugging and observing lazy functional computations based on reduction histories (or *Redex Trails*) has been proposed in (Sparud & Runciman 1997*b*, Sparud & Runciman 1997*a*). The trails record a rich amount of information about a computation and various post-processing tools have been developed to view the information in different ways, including declarative debugging (Wallace, Chitil, Brehm & Runciman 2001). The main cost of recording Redex Trails is the space required to store the trail, the size of which being proportionate to the duration of the computation. To cope with the large space requirement, the trail is serialised and written to file rather than being maintained in main memory.

8 Conclusion

Debugging higher-order code would be near impossible without a means for printing functions in a meaningful way. In this paper we have presented a program transformation over a core Haskell syntax for the purposes of debugging. Our main contribution is the treatment of higher-order code, as presented in section 5, where we solve the difficult issue of curried applications and show how to generate detailed representations of them. We support full Haskell, and only require a small amount of help from the runtime environment of the compiler. The transformation is purely syntax directed, and so can be performed without type information which aids efficiency and simplifies its implementation.

paper suffers from one significant drawback: excessive space usage. Creating a node in the EDT for every function application (even if lazily) maintains a reference to all intermediate values in the computation, prohibiting garbage collection. Mechanisms for reducing the size of the EDT are essential if the debugger is to be useful for large programs. This is a significant focus of our future research. The current implementation of our debugger is available from: `www.cs.mu.oz.au/~bjpop/buddha`

References

Caballero, R. & Rodri'guez-Artalejo, M. (2002), A declarative debugging system for lazy functional logic programs, *in* M. Hanus, ed., 'Electronic Notes in Theoretical Computer Science', Vol. 64, Elsevier Science Publishers.

Haskell 98 Language Report (2002), `http://www.haskell.org/onlinereport`.

Naish, L. & Barbour, T. (1996), 'Towards a portable lazy functional declarative debugger', *Australian Computer Science Communications* **18**(1), 401–408.

Nilsson, H. (2001), 'How to look busy while being as lazy as ever: The implementation of a lazy functional debugger', *Journal of Functional Programming* **11**(6), 629–671.

Pope, B. (1998), Buddha: A declarative debugger for Haskell, Technical Report 98/12, The Department of Computer Science and Software Engineering, The University of Melbourne.

Sparud, J. (1999), Tracing and Debugging Lazy Functional Computations, PhD thesis, Chalmers University of Technology, Sweden.

Sparud, J. & Runciman, C. (1997*a*), Complete and partial redex trails of functional computations, *in* T. D. C. Clack, K. Hammond, ed., 'Selected papers from 9th International Workshop on the Implementation of Functional Languages', Vol. LNCS 1467, pp. 160–177.

Sparud, J. & Runciman, C. (1997*b*), Tracing lazy functional computations using redex trails, *in* 'PLILP', pp. 291–308.

Wadler, P. (1993), Monads for functional programming, *in* M. Broy, ed., 'Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School', Springer-Verlag.

Wadler, P. (1998), 'Why no one uses functional languages', *SIGPLAN Notices* **33**(8), 23–27.

Wallace, M., Chitil, O., Brehm, T. & Runciman, C. (2001), Multiple-view tracing for Haskell: a new hat, *in* 'Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop', pp. 151–170.