Buddha

A Declarative Debugger for Haskell

Bernard Pope
supervisor: Lee Naish

Honours Thesis
June 1998

Department of Computer Science
The University of Melbourne
Parkville Vic. 3052
Australia

Abstract

Due to their reliance on the execution order of programs, traditional debugging techniques are not well suited to locating the source of logical errors in programs written in lazy functional languages. We describe the implementation of a declarative debugger for the programming language Haskell, which assists the location of logical errors based on the declarative semantics of program definitions. The implementation is based on the Hugs interpreter, and both solidifies previous work in the field and extends it to incorporate features typical of many modern lazy functional languages.

Contents

1 Introduction

The theoretical and practical advantages of pure lazy functional programming languages are many [5]. As a instance of the declarative programming paradigm, such languages allow the programmer to focus on the formal definition of the task to be solved, leaving the issue of evaluation strategy to the language implementation. This is a powerful abstraction which permits one to reason about the correctness of a program without the need to consider the underlying computational machinery.

One particular advantage of functional languages is that many run–time errors, or bugs, that are possible in other programming paradigms, are avoided. Referential transparency, implicit memory management and strong typing are three features typical of modern functional languages that prevent a large class of bugs from ever occurring. Unfortunately, no programming paradigm is completely immune from bugs. There are three main sources for bugs in functional languages [20]: program exceptions, such as attempting to divide by zero; logical errors, which occur when the *actual* meaning of the program and the *intended* meaning of the program are not the same; and exhaustion of memory resources. It has been argued that if functional languages are to be widely accepted in the programming community, then effective mechanisms for debugging programs written in them must be developed [19].

Step–wise tracing of program execution is the basis for most current debugging techniques. Such debugging techniques are well suited to the imperative paradigm because the execution order of programs written in imperative languages is explicit in the syntactic structure of the program and also in the user's understanding of how the program works. It is therefore apparent that traditional debugging techniques are not suited to declarative languages, because execution order is not truly reflected in their syntactic structure or their declarative semantics. This observation is particularly true for lazy languages. Expressions in lazy languages are evaluated on demand. This means that an expression will only be evaluated if it is needed in the production of the final result of the program. In general, it is not easy to reconcile the execution order of a program written in a lazy language with the text of the program. Thus debugging strategies based on execution order are of little practical use for lazy languages.

The difficulty of debugging functional languages is well recognised, and several approaches have been suggested to solve this problem. One promising approach is *declarative debugging* in which the sources of bugs are located according to the declarative semantics (or intended meaning) of the program rather than execution order of the program. This technique has its origins in *algorithmic debugging*, which was initially used to debug wrong and missing answers in relational programming languages [18]. The main disadvantages of previous declarative debugging schemes for functional languages are that they prohibit portability and suffer from inefficient memory usage.

In this paper we present Buddha[1], a declarative debugger for the pure lazy functional language Haskell. The debugger is based on previous work by Naish and Barbour, who propose a declarative debugging technique that maximises portability and which is constrained in its memory usage. We extend the technique to incorporate additional language features, typical of modern functional languages, and provide an implementation of a low level primitive function which is critical to the success of the technique.

The paper is structured as follows. In the second section we give a brief overview of the Haskell programming language. We discuss the important features of Haskell that influence the design of the debugger, and briefly provide an argument for implementing the debugger using Hugs, an interactive Haskell interpreter. We also discuss the subset of Haskell that is currently supported by the debugger. In the third section we provide an overview of the declarative debugging technique. We state the formal requirements of the technique, and describe how it can be used for functional languages. We also formalise our own declarative debugging architecture. In the fourth section we describe our implementation of the declarative debugging technique proposed by Naish and Barbour. Several extensions to the technique are discussed, which allow it to incorporate additional (but typical) language features. In the fifth section we survey related work in the field and compare it to the work described in this paper. In the sixth section

[1] Bernie's Ultimate Declarative Debugger for Haskell.

we describe our contribution to the field of debugging pure lazy functional languages. In the seventh section we discuss possible extensions to the current debugger both to increase the class of language features that it supports and to decrease its memory usage.

2 Haskell

In this section we provide a brief overview of the Haskell programming language. We mention the background of the language and discuss the principal features of pure lazy functional languages (the class of languages to which Haskell belongs). We discuss the subset of Haskell that is currently supported by the debugger and provide an argument for the implementation of the debugger in Hugs, an interpreted Haskell system. For a more complete description of Haskell, the reader is directed to the *Haskell 1.4 Language Report* [16], and the Haskell tutorial *A Gentle Introduction to Haskell* [4].

2.1 Background

In 1987 the Haskell programming language was specified by a design committee. The motivation for its inception was to satisfy the need for a standard pure lazy functional language suitable for application in general programming tasks. Haskell incorporates many modern programming language features, and is viewed as "the culmination and solidification of many years of research on lazy functional languages" [16]. Some of the most important features of Haskell are: referential transparency, lazy evaluation, higher–order, strong static polymorphic typing, extensive primitive data–types and monadic I/O. The most recent version of the language is defined in the *Haskell 1.4 Language Report* [16]. There are several implementations available, some incorporating significant extensions to the language definition.

2.2 Haskell is a lazy language

It is possible to classify functional languages according to the order in which the arguments to functions are evaluated. A binary classification exists in which the order of evaluation is described as either strict or non–strict.[2] Arguments to a function in a strict language are evaluated *before* the function is called, whereas the evaluation of arguments to a function in a non–strict language is delayed until the value of those arguments is actually needed. Non–strict evaluation is often called *lazy evaluation*, and languages such as Haskell that use this evaluation order are called lazy languages.[3] The arguments to functions in a strict language are always evaluated, whilst arguments to functions in lazy languages may never be evaluated (if their value was not needed) or partially evaluated (if only part of their value was needed).

The distinction between evaluation orders has important implications for the termination properties of each type of language. Consider the function `main` in figure 1.[4] In a lazy language, the evaluation of the function `main` proceeds in the following manner. The function `fst`, which returns the first element of a tuple, is applied to the expression `foo 1 2`. Since `foo 1 2` is evaluated lazily it simply returns the expression `(fie (1 + 2), inf 2)`, from which `fst` will select the expression `fie (1 + 2)`. The result of applying `fie` to `(1 + 2)` is the expression `2 * (1 + 2)`. The last step in the evaluation causes `+` to be invoked, then `*`, which finally returns the value `6`. An execution tree depicting the lazy evaluation of `main` is given in figure 2. The function `inf` when applied to a numeric argument will cause an infinite computation. Note, however, that the lazy evaluation of `foo 1 2` does not cause `inf 2` to be evaluated, and hence the overall computation is terminating. In a strict language this is not the case as `foo` will completely evaluate its arguments, thus triggering the non–termination of `inf 2`.

[2] It is possible to combine a mixture of both evaluation orders, as is done in many modern functional languages. However such combinations are limited to special cases, which for simplicity, we will not cover.

[3] Haskell has lazy data constructors which, amongst other things, allow the creation and manipulation of potentially infinite data structures. We limit our discussion of laziness to functions for simplicity, however, data constructors can be thought of as a special type of function that map types to types.

[4] This is a slight adaptation of an example given by Sparud [19] to describe lazy evaluation.

```
inf :: Num a => a -> a
inf x = x + (inf x)

foo :: Num a => a -> a -> (a, a)
foo x y = (fie (x + y), inf y)

fie :: Num a => a -> a
fie x = 2 * x

main :: Int
main = fst (foo 1 2)
```

Figure 1: An example program which terminates under lazy evaluation, but not under strict evaluation

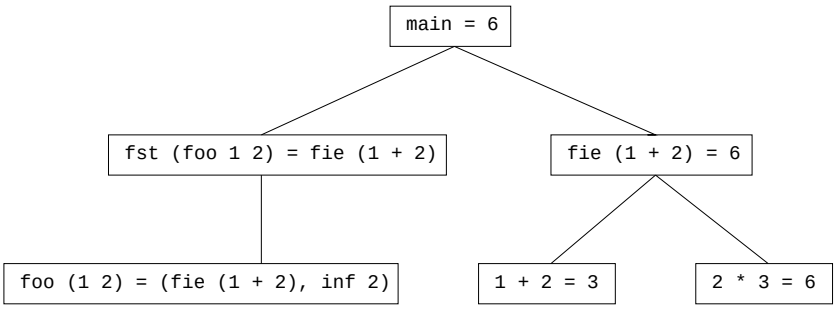

Figure 2: Lazy execution tree for the program defined in figure 1

Nilsson and Sparud [15] make two important observations about the lazy evaluation of functional programs. Firstly, the actual arguments to functions and results of functions during the evaluation of a program are expressions which may or may not be evaluated at a later stage in the computation. In the example above, we expect the argument to `fie` to be a number, but in actuality it is the unevaluated expression `1 + 2`. Secondly, the order of the execution is not obviously reflected in the structure of the program text. In the example above the call to `fie (1 + 2)` is performed outside the evaluation of `foo 1 2`, although the program text appears to suggest otherwise. These two observations have important implications for debugging lazy languages. In the first case, arguments to a function may not be in their most evaluated form upon the invocation of that function, and furthermore, the arguments of a function may only be partially evaluated over the complete execution of the program. If we wish to ask the user of the debugger whether a function application produced the correct result, we must be able to display a representation of the arguments (and result) of the application in a form which reflects the extent to which they were evaluated at the end of the execution of the program. We must also be prepared to encounter potentially infinite arguments, that were only partially evaluated during the execution of the program. In the second case, if the debugger were to follow the actual execution of the program, it may be difficult if not impossible for the user of the debugger to reconcile the progress of the evaluation with the definition of the program. The design of our debugging system, described section 4, is motivated to a large extent by these two observations.

2.3 Haskell is a higher–order language

Functions are *first class citizens* in higher–order functional languages. This means that functions can be treated like any other non–functional object. In effect, functions can be passed as arguments to other functions, they can be returned as the result of functions, and they can occur inside arbitrary data structures. When a function of arity n is applied to m arguments (where $n \in \mathbb{N}\backslash\{0\}$ and $m \in \mathbb{N}$), such that $m < n$, we say that the function is partially applied or curried. Currying can be thought of as a special instance of higher–order programming, because the result of a curried application is a new function of $n - m$ arguments.

Although functions are regarded as first class citizens in functional languages, they differ from the basic types in a number of important ways. For instance, there is no absolute way to define equality over functions in the same way that equality can be defined for many non–functional types. Furthermore, as abstract objects, functions do not have inherent printable representations. In order for the debugger to ask questions about the correctness of a function application it must provide a printable representation of the arguments to the function and its corresponding result. As we will demonstrate in section 4.1.6, this can be difficult when the arguments or result are functional or contain functions within data structures.

2.4 Haskell is purely functional

There is some disagreement in the functional programming community as to what is required of a language for it to be regarded *purely functional*. A stringent definition of purity requires that all computations within the language are performed by function application alone, and side–effects resulting from function applications are not allowed. A side–effect (sometimes called a computational effect) is the performance of an operation external to the evaluation of a function that causes a change in the program's state, for example, incrementing a global variable or writing data to a file. Haskell cannot be regarded pure in respect to this definition due to its allowance of side–effects via various monadic constructs. A broader definition of purity that is often used, allows function application to cause side–effects providing that referential transparency is preserved. A language is considered referentially transparent if the value of any expression depends only upon the values of its well formed sub–expressions. In other words, functions in a referentially transparent language always return the same result given particular arguments no matter when or where they are called in the execution of the program. A further definition of purity is due to Sabry [17], who proposes that a functional language is pure if the value of functions is the same irrespective of the parameter passing mechanism employed by the language. Haskell can be considered purely functional in respect to the second and third definitions of purity outline above.

For the purposes of this paper, the second of the above definitions of purity will suffice. The purity of a language is of great significance for declarative debugging. Within the debugger, the search for the source of a bug is directed by the perceived correctness of functions when applied to certain arguments. The correctness must be judged solely on the value of the arguments to the function and its corresponding result when applied to those arguments. Since the values of functions in an impure language may not rely entirely on the values of their arguments, the determination of the correctness of such functions may become arbitrarily complicated. In other words, the declarative debugging strategy relies on the purity of functions to successfully guide the search for the source of bugs.

2.5 Haskell has a strong static type system

It is a common (although not essential) feature of modern functional languages to have strong static type systems. This means that every value in the language has a type, and all errors due to type mismatch can be detected statically. In effect, this ensures that type errors are impossible during the execution of a program written in such a language. It is not our intent to provide a complete description of the type systems used in modern functional languages, however, the type system of Haskell does impose some constraints on the design of the debugger if it is to be written in Haskell itself. These constraints

will become apparent when we provide a detailed description of our implemented debugger. For a more thorough account of the type system of Haskell, the reader is directed to Section 4.1.3 of the *Haskell 1.4 Language Report* [16].

2.6 The syntactic subset of Haskell that is currently supported by the debugger

Haskell is a syntactically rich language. It allows considerable flexibility in the style of programming adopted by the programmer. Due to time constraints on the completion of the project, some aspects of its syntax are not yet supported by the debugger. It was considered sufficient to support the subset of Haskell syntax that closely resembles that of the Miranda programming language. The main syntactic constructs that are not currently supported are: lambda expressions, let expressions, case expressions, and list comprehensions. Such syntactic constructs do not increase the computational power of Haskell over the supported syntax, however, they do increase its flexibility, and are intended to be supported in future versions of Buddha.

2.7 Hugs

Hugs[5] is an interpreted implementation of Haskell which provides an interactive development environment. It is based significantly on the Gofer interpreter of Mark Jones. There are a number of reasons that Hugs was chosen as a suitable environment in which to develop Buddha. Firstly, the standard Hugs distribution provides useful low–level primitives upon which a key feature of the debugger can be implemented without the need for any modifications to the system. Secondly, Hugs is a small system in comparison to the available compilers for Haskell, making it more suitable for experimentation. Finally, as an interactive environment, users can incrementally debug a program during its development without the need for repetitive re–compilation. Having said this, a significant goal of the Naish and Barbour debugging technique is that it be portable to other programming languages and environments. The debugger described in this paper is an instance of of their technique, however, there is no special reliance on Hugs or Haskell for the success of the technique.

3 Declarative Debugging

The idea of using the declarative semantics to guide a semi–automated search for the source of bugs in programs is due to Shapiro [18]. The original work focussed on locating logical errors in Prolog programs, however the principles behind the method can be extended to other programming languages and paradigms.[6] Much of the original work in the field was performed under the title of *algorithmic debugging*, however, we adopt the term declarative debugging to emphasise the importance of declarative semantics in the technique.

In this section we discuss the principles of declarative debugging when applied to lazy functional programming languages. The intention is to provide the reader with an intuitive understanding of the declarative debugging machinery before our particular implementation is described in section 4. In particular, we provide a high–level description of the main data structure upon which the technique is based, and compare it with the lazy execution tree introduced in section 2.2. We give an overview of two methods commonly used to construct the data structure, and discuss the method adopted in Buddha. We describe the two level architecture of the debugger, and provide an example application of the debugger to a buggy insertion sort program, viewing the execution of the debugger from the user's perspective.

[5] The Haskell User's Gofer System.

[6] Westman and Fritzson [22] report that algorithmic debugging has been applied to imperative languages and parallel logic languages.

Finally we discuss some of the limitations of the declarative debugging technique, and the requirements imposed upon its design to ensure correct behaviour.

3.1 The evaluation dependence tree

In section 2.2 we illustrated the difficulty of reconciling the evaluation order of programs written in lazy functional languages with the text of those programs. The goal of declarative debugging is to allow a programmer to reason about the correctness of the execution of a program in a manner which closely reflects its textual definition. We define an *evaluation dependence tree* (EDT) to represent an instance of the evaluation of a program.[7] The key feature of the EDT is that it reflects the syntactic dependencies of the program definition, rather than the execution dependencies due to lazy evaluation. Each node in an EDT represents a function application that occurred during the execution of a program. In addition to a representation of the value of the application, at each node we record the name of the function that was applied, and a representation of the arguments that it was applied to. For a node representing an application of function f, its children correspond to the syntactic applications in the definition of f that were actually evaluated. Consider the example program in figure 1. We can see that the definition of the function `foo` depends *syntactically* on the calls `fie (x + y)` and `inf y`. The application of `foo` to its arguments represents a two element tuple, however, during the execution of the program (due to the evaluation of `main`) only the first element of the tuple is needed, thus `inf 2` is not evaluated. We can see that the existence of a syntactic dependency between a function definition and a function call does not necessarily lead to an evaluation dependency between the two: the evaluation of `foo 1 2` did not depend on the evaluation of `inf 2` in this instance. An EDT for the evaluation of `main` from figure 1 is given in figure 3.

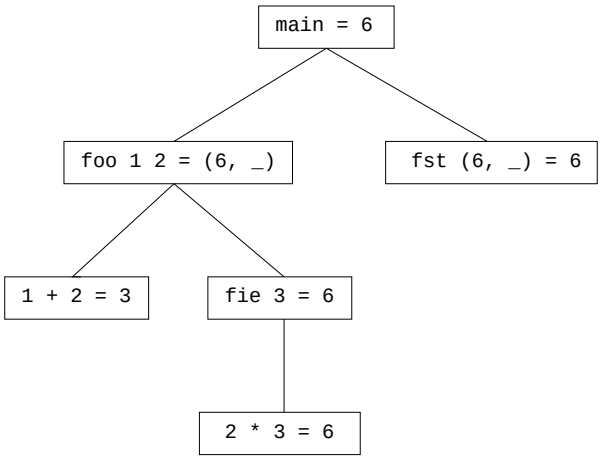

Figure 3: EDT of the example program in figure 1

A comparison of the two trees in figure 2 and figure 3 is useful. The nodes in the lazy execution tree are ordered according to the time at which they were evaluated, whereas the nodes in the EDT are ordered according to their syntactic position in the program definition. Note that each tree is a true reflection of this instance of the execution of the program. In particular, neither tree contains a node for the application `inf 2`, even though the application of `inf` to an argument occurs in the definition of `foo`. Note also the presence of underscores in the second element of the tuple returned by `foo 1 2` in the

[7] The term *evaluation dependence tree* is borrowed from Nilsson and Sparud [15], however, the structure of our tree is different from theirs. The differences between our tree and that of Nilsson and Sparud are presented in section 4.1.1 and section 4.1.2.

EDT. The underscores indicate that this part of the tuple was not evaluated at the end of the program, and hence not involved in producing the value of `main`. Each tree represents an understanding of the evaluation of the program, however the first is operational and the second is declarative.

3.2 Program transformation

Several declarative debugging techniques for lazy functional languages have been based on the creation and traversal of an EDT [14, 11, 15, 3]. In general there have been two approaches taken to constructing the EDT. The first approach, as exemplified in the work of Nilsson *et al* [12, 21, 13][8], requires a modified language implementation that generates the EDT as a side–effect of executing the program. As the proponents of this technique note, a significant effort is required to implement the necessary language modifications. Such a technique allows for several optimisations to be made in the generation of the EDT, however it is not very portable between different language implementations. The second approach uses a source to source transformation on the program text. The definition of each function in a program is transformed to return a tuple containing the original result of the function and a EDT node representing the application of that function during execution. The transformed program is then executed, the result of which is a tuple containing the value of the original program and a tree representing the EDT for that execution. This approach has been independently specified by Nilsson and Sparud [15] and Naish and Barbour [11], however there are significant differences in the EDTs that they define. We adopt the proposal of Naish and Barbour, and in section 4.1 we discuss the transformation of programs in detail.

3.3 A two level debugging architecture

Debugging systems based on the program transformation approach are typically constructed in two levels. The first level consists of the source to source transformation of the original program. The second level executes the transformed program and traverses the generated EDT, usually interactively, to locate the source of bugs. Nilsson and Sparud [21][9] refer to the two levels as the *EDT generator* and the *EDT navigator*, and offer three reasons for the suitability of such a classification. The underlying argument for designing the debugger in two levels is that the issues in the generation of the EDT and the eventual use of the tree are logically distinct, thus justifying a separation of the concerns into two parts. We describe the first level of the debugger in section 4.1 and section 4.2, and the second level of the debugger in section 4.3.

There may be many ways to search for bugs by traversing the EDT, and indeed many other uses of the EDT other than for finding bugs. We describe one particular traversal of the EDT which is very simple, yet suitable for locating logical errors in a large class of programs. The technique requires an oracle that knows the intended value of every application performed in the execution of the debugged program. For simplicity we assume the oracle to be the person who wrote the program, although other oracles are possible, such as executable specifications of the program. The traversal begins at the top of the EDT which contains the topmost function application of the program. The function name, representations of the actual arguments, and a representation of the value of the application at the node are presented to the oracle, upon which the oracle must answer *yes* if the application was correct or *no* otherwise. Nodes which are determined to be incorrect by the oracle are called *erroneous nodes*. If the topmost application (i.e. the function application represented by the top node in the EDT) is correct then the debugger concludes that it cannot find any bugs in this execution of the program. However, if the application is incorrect, the debugger traverses the children of the node from left to right. If the application performed at a particular child is correct, the debugger moves to the next child of the node. Children of correct nodes are not further investigated because their correctness illustrates that they and their descendants

[8] The authors of these papers discuss both EDT generation techniques.

[9] Nilsson and Sparud use this classification for both EDT construction techniques, where we simply refer to the transformation technique.

did not contribute to the logical error of the program. If the child of the node is incorrect the debugger recursively investigates the sub–tree defined by the erroneous child. The traversal of the EDT ends when an erroneous node is located, whose predecessors are all erroneous, and which either has no descendants or all of its descendants are deemed correct by the oracle. We call such a node a *topmost buggy node*. In the general case there may be many such topmost buggy nodes[10] in the EDT, however in this case we return the first that is encountered. The textual definition of the function applied at the topmost buggy node is displayed at the end of the debugging session to assist the location of the error in the program text. There may be many logical errors in a given program definition. The debugging technique described previously requires each logical error to be located and corrected separately before the next. It is possible that multiple sources of errors may be located during the one debugging session through a fairly simple extension to the technique just described. We cover alternate EDT navigations and uses in section 7.5.

```
insort :: Ord a => [a] -> [a]
insort [] = []
insort (x:xs) = insert x (insort xs)

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

Figure 4: Buggy definition of insertion sort

Consider the (buggy) definition of insertion sort in figure 4[11]. The bug is due to an error in the first guard of the second equation defining the `insert` function. The intended meaning of the program is that given an unsorted list of objects satisfying the type constraints of the function, it will evaluate to a list containing those objects in ascending sorted order. The actual meaning of the program as it is currently defined is something different and therefore logically incorrect. For the actual meaning of the program to be equivalent to the intended meaning, the erroneous guard in `insert` could be changed to "`| x >= y = y : (insert x ys)`" (and the third guard should be removed).

An EDT for the evaluation of `insort [2,1,3]` is given in figure 5. Erroneous nodes are highlighted by a dark rectangle. Of the erroneous nodes, two topmost buggy nodes are identified by underlines within the node.

An example debugging session applied to the EDT generated by `insort [2,1,3]` is given in figure 6. The responses of the oracle are either '`y`' (for yes) to specify the application was correct or '`n`' (for no) to specify the application was incorrect. Notice that the questions asked by the debugger correspond to a partial depth–first traversal of the EDT, such that the children of correct nodes are not visited. The traversal terminates upon the location of the first topmost buggy node. When the node containing the application `insert 1 [3] = [3, 1]` is encountered, it is deemed to be incorrect by the oracle, and so the debugger recursively moves to the child of that node. The child is correct, and since the node is incorrect but only has correct children it is returned as a topmost buggy node. The definition of the function applied at the node is then presented at the end of the session.

[10] Note, however, that many topmost buggy nodes may correspond to the same function definition in the program text.

[11] We take this example from Hannan [3], however, it is possible that it was derived from earlier works such as Westman and Fritzson [22].

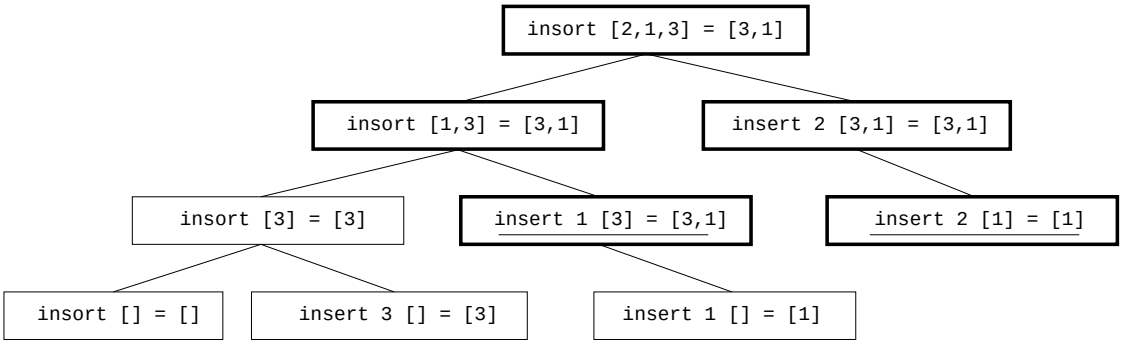

Figure 5: Evaluation dependence tree of `insort [2,1,3]` with a bug

```
buddha (insort [2,1,3])
final result = [3, 1]

insort [2, 1, 3] = [3, 1]
Correct - Y/N/Q? n

insort [1, 3] = [3, 1]
Correct - Y/N/Q? n

insort [3] = [3]
Correct - Y/N/Q? y

insert 1 [3] = [3, 1]
Correct - Y/N/Q? n

insert 1 [] = [1]
Correct - Y/N/Q? y

The erroneous equation is:
insert x (y:ys)
    | y > x = y : (insert x ys)
    | x < y = x : y : ys
    | otherwise = y : ys
```

Figure 6: Example interaction with the debugger

3.4 Limitations and requirements of the debugger

Logical errors in a program definition are not always manifest in the output of an execution of that program. For example, when applied to the empty list or lists of length one, the `insort` function evaluates to the correct sorted list. It is only when lists of length greater than one are supplied as arguments that the incorrectness of the program will be apparent in its output. For the declarative debugging technique (as described above) to be of any use in locating bugs in a program, the user must first identify example inputs upon which the program returns incorrect results. This limitation makes it clear that the role of a declarative debugger (or any debugger for that matter) is not to prove the correctness of a program definition, but rather to locate the source of errors in instances of the program execution that generate incorrect results.

In the declarative debugging scheme described above we take an original program definition and transform it into a new program definition which models the evaluation behaviour of the original program. The success of the debugging technique depends upon the ability of the transformed program to correctly preserve the behaviour of the original program. It is important to note that the original function and its transformed counterpart do not have the same declarative meaning. Furthermore, where higher–order functions are concerned, the domain of the original function is not the same as that of the transformed function: when functions are passed as arguments in the original program, transformed versions of those functions must be passed in the transformed version of the program.

The relationship between a function definition, its potential arguments and its result when applied,

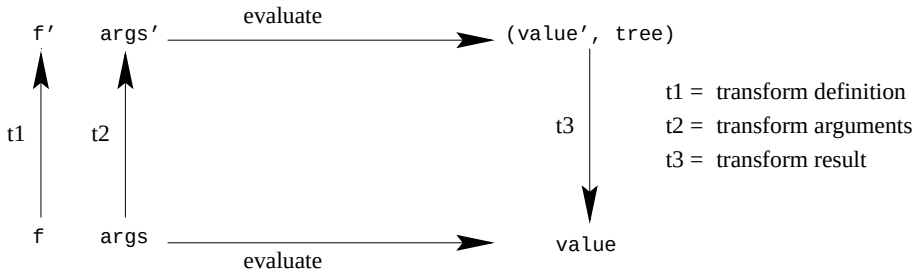

Figure 7: Commuting diagram, illustrating the mapping between functions and their transformed counterparts

and their transformed counterparts is illustrated by the commuting diagram in figure 7. The relationship is defined by three operators (t1, t2 and t3). We can think of the operation performed by t1 as the syntactic transformation performed on function definitions, or we can think of it in another sense as a mapping of abstract functions. Indeed, it is common for the distinction between functions (which are abstract entities) and their definitions (which are syntactic entities) to be blurred. In section 4.1 we describe the transformation of definitions, and thus refer to t1 in the syntactic sense, however, for the present discussion it is preferable to think of it as a mapping of functions as abstract entities. The second, t2, maps the arguments of the original function into arguments of the transformed function. For arguments which are not higher–order the transformation is the identity function. If the arguments are higher–order, then the transformation is more complicated. We discuss a transformation for curried function applications as arguments and the requirements for a more general transformation of higher–order arguments in section 4.1.6. The third, t3, maps the value of an application of the transformed function to the value of an application of the original function.

Earlier, in this section, we said that for the declarative debugging technique to be successful, the transformed version of a program must preserve the behaviour of the original program. By this we mean that, given the operators t1 and t2, we require the existence of the transformation operator t3 such that the value of `(t3 (f' args'))` is equal to the value of `(f args)`, *and* `(f args)` is terminating if (and only if) `(t3 (f' args'))` is terminating. In the current version of the debugger we do not allow higher–order results, and hence t3 is equivalent to `fst`.

For the debugger to be a useful tool we must ensure that the results it returns are correct. The preservation of the original program behaviour in the transformed version of a program is a necessary (but insufficient) property to ensure the correctness of the debugging technique. What does it mean for the results of the debugger to be correct? If the topmost application in the EDT is erroneous, several logical errors in the program definition may contribute to that error. The error in the topmost application is a symptom of one or more logical errors in the program definition. Therefore, we cannot make a direct causal connection between a topmost buggy node and the error in the topmost application of the program, since there may be several factors contributing to the error. However, we would like to be certain that the node returned by the debugger is actually incorrect in the intended interpretation of the program.

We define the correctness of the debugger to consist of two properties: soundness and completeness. In the trivial case that the topmost application of the program is correct, the debugger is sound and complete if it does not return any nodes from the EDT. The more important case occurs when the topmost application is incorrect. The debugger is sound if it returns an EDT node such that the right hand side of the equation corresponding to the node evaluated correctly in the intended interpretation but did not equal the intended value of the left hand side of the equation. If the debugger is guaranteed to *always* find a topmost buggy node with the desired soundness property, then it is considered complete.

We have described the relationship between the evaluation of a program and its corresponding EDT above. Providing that the tree is finite and that the top node in the tree is erroneous, there must be at

least one topmost buggy node in the tree. Given that the debugger traverses the tree beginning at the top node, following the erroneous nodes in a depth–first order, we can argue (informally) that it is guaranteed to find a topmost buggy node. Furthermore, by its very definition, a topmost buggy node corresponds to a function equation whose right hand side evaluated correctly but did not equal the intended value of its left hand side. Here we rely on an intuitive argument for the correctness of the debugging technique. We do not provide a formal proof of the soundness and completeness of the Naish and Barbour declarative debugging technique in this paper. For a more thorough exposition of correctness, the reader is referred to *A Declarative Debugging Scheme* [9]. In [9], Naish illustrates that for debugging *wrong answers* in Prolog programs, the EDT is an instance of the proof tree for the program being debugged, which is well established in the theory of logic programming, The formal proof of soundness and completeness requires a formalisation of the relationship (outlined in section 3.1) between the EDT and the evaluation of programs in the language in question. For Prolog, this is covered thoroughly in [7, 8] (cited in [9]), from which a similar method of proof can be derived for lazy functional languages.

4 The implementation of Buddha

4.1 The source to source transformation of programs

The first phase of debugging programs with Buddha involves a transformation on the source text of the program. In this section we describe how the transformation is implemented in Buddha. We begin by presenting a Haskell definition of an EDT node. We then outline the Naish and Barbour transformation algorithm which forms the basis for the complete transformation. Following the presentation of the algorithm, we briefly discuss the transformation of Nilsson and Sparud. We then illustrate how the transformation can be extended to incorporate where clauses and guarded equations. At the end of this section we discuss the implications that higher–order programming has on the transformation. In particular we show how curried function definitions and applications can be handled in the transformation. We then provide a description of what is needed for higher–order arguments to be handled in a more general manner.

4.1.1 Implementation of the EDT

In section 3.1 a high–level description was given of the EDT used in many declarative debugging techniques for lazy functional languages. Here we describe the Haskell implementation of the EDT used in Buddha. Recall that the EDT represents a particular instance of the evaluation of a program. Nodes of the tree represent function applications that occurred during that evaluation. Each node contains the following information: a string which identifies the function that was applied; a list of representations of the arguments that the function was applied to; a representation of the result of the application; a list containing the children of the node, such that each child corresponds to a function application from the right hand side of the function definition that contributed to the result; and a reference to the text of the function definition (for simplicity a string[12]).

The arguments and result of function applications can be of any arbitrary type. We want to keep a record of the arguments and the result of each application so that during the debugging session we can print a representation of them to the oracle. Haskell's strong typing will not allow mixed types to be stored at the same position in an EDT node. Sparud and Nilsson [21] solve this problem by using existential types, which allow objects of different types at the same position in a data structure providing such objects are a member of a certain type class. The disadvantages of this approach are that existential types are not currently part of the Haskell standard, and are not supported by all Haskell implementations. Furthermore, every possible type used in a program must be made a member of a

[12]When we present the transformed version of a function we only show the beginning of the string which represents the original definition of the function. This is to reduce the length of code listings in the paper.

particular type class.[13] We convert all function arguments and results into a representation of type `Term` by application of the impure function `dirt`, described in section 4.2. In the current version of Buddha, `Term` is a synonym for `String`, however lower level representations are possible. A Haskell definition of the EDT type is given in figure 8.

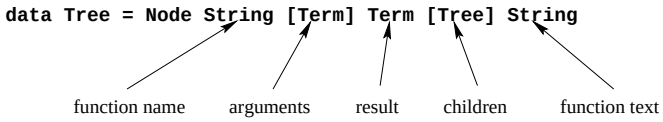

Figure 8: Haskell definition of the EDT used in Buddha

4.1.2 The transformation algorithm of Naish and Barbour

The original transformation algorithm of Naish and Barbour is defined for a simple functional language syntax. The intention of the algorithm is to capture the essence of the transformation for a broad class of functional languages. We use the algorithm as the basis for the transformation of Haskell functions within Buddha. In this section we define the abstract transformation algorithm, and compare it to the transformation of Nilsson and Sparud. In later sections we refine the transformation to handle some important syntactic constructs of Haskell.

The Naish and Barbour transformation algorithm is defined as follows:

For all equations of the form $f\ a_1\ a_2 \ldots a_n = r$, the following three transformation steps are performed:

1. Rewrite the equation as

   $f\ a_1\ a_2 \ldots a_n$
   $= (v_0,\ t_0)$
   `where`
   $v_0 = r$

2. Repeat the following step until there are no further applications in r.
   Let g be a function of arity m. If $g\ b_1\ b_2 \ldots b_m$ is an innermost function application in r, replace the application by v_i and add the following equation under the top `where` clause of f (v_i and t_i are new variables, unique in the scope of f)

   $(v_i,\ t_i) = g\ b_1\ b_2 \ldots b_m$

3. Add the following equation to the top where clause of f. ($t_1, \ldots, t_p$ correspond to the t_i in the previous step)

   $t_0 =$ `Node "`f`"` $[$`dirt` $a_1,$ `dirt` $a_2, \ldots,$ `dirt` $a_n]$ (`dirt` v_0) $[t_1, \ldots, t_p]$ (*text reference*)

The definition of an *innermost application* is important in the above algorithm. It essentially means the most nested function application within an expression in the right hand side of an equation. The application of built in operators are not considered as candidates for innermost function applications, since the evaluation of such operators is trusted to be correct. The right hand side of an equation may consist of several independently nested expressions each with their own innermost application. When such a situation arises we choose the leftmost of the innermost applications, however this is merely out

[13] Although this problem can be somewhat alleviated by automatic type class instance generation from the compiler.

of convention rather than necessity, as selection of any of the possible innermost applications would be sufficient. Examples of candidate innermost applications are presented as underlined expressions in figure 9.

```
take 3 (drop 5 (sort xs))

(sum xs) + (average (map (+3) (reverse ys)))

MyData (plus 1 (sqrt x)) (round y)
```

Figure 9: Candidate innermost applications in example expressions

In Haskell, functions are defined by one or more equations. The above algorithm transforms the entire program function by function, such that each function is transformed equation by equation. There are three important stages in the transformation of an equation in the algorithm. Firstly, the top–level of the equation is rewritten to reflect the fact that the transformed version evaluates to a tuple containing the original value of the equation and a EDT node. Secondly, the right hand side of the equation is unravelled so that each application is made distinct. Having unravelled the applications on the right hand side, we can access the trees generated when those applications are evaluated. The trees for each application (denoted t_i in the algorithm) constitute the children of EDT node for the equation being transformed. To be more precise, they are *potential* children, since some of them may not be evaluated for a given execution of the program: we only consider those applications that are eventually evaluated to be children of an EDT node. However, since we cannot tell in advance which applications will be evaluated for all possible executions of the program, the tree for each application is inserted into the list of children. Later, in section 4.2 we will see that the debugger can determine which children of a node were not evaluated and can safely skip over them in its traversal of the EDT. Thirdly, the definition of the EDT node for the equation is added. Note that the function `dirt` is applied to the arguments and the result of the transformed equation in the definition of the node. This guarantees that a representation of the arguments and result of all functions can be stored within the same EDT.

Throughout the algorithm we have assumed that the extra identifiers that are introduced (the v_i and t_i) are unique in the scope of the equation being transformed. This is not trivial in Haskell, because the identifiers within the scope of an equation include all top–level identifiers (including those imported from libraries and other modules), and all identifiers local to the equation (including identifiers within scope of the top–level where clause of the equation). To guarantee the uniqueness of the introduced identifiers we must first determine all identifiers that are within scope, and then choose a number for each new identifier such that the concatentation of the identifier prefix (v or t) with the number is unique within the scope. A related problem occurs with the definition of the EDT node type. In figure 8 we define the EDT node type `Node` using the data constructor `Tree`. If the program being transformed makes use of the same type constructor or data constructor as in the definition of the EDT node type then an error will occur in the transformed program. A similar technique to that described above for generating unique identifiers could be used to avoid name clashes in type constructors and data constructors. However, a simpler method is to choose names for the type constructor and data constructor used in the EDT node type such that it is unlikely that a programmer would use those names within their own type definitions. In the current version of Buddha we do not guarantee the uniquness of introduced identifiers. However, for simplicty, we assume throughout the rest of the paper that all additional identifiers that are introduced by the transformation and the type and data constructors of the EDT node type have the desired uniqueness properties.

An example transformation of the second equation for the `insort` function of figure 4 is given step by step in figure 10. Step 1 of the transformation corresponds to the first part of the algorithm, steps 2 through 6 correspond to the second part of the algorithm and step 7 corresponds to the third part of

1. Rewrite the equation as in the first step

```
insort (x:xs)
  = (v0, t0)
  where
  v0 = insert x (insort xs)
```

2. Locate the innermost application

```
v0 = insert x (insort xs)
```

3. Replace it with a variable and add a new equation to the where clause

```
v0 = insert x v1
(v1, t1) = insort xs
```

4. Locate the next innermost application

```
v0 = insert x v1
```

5. Replace it with a variable and add a new equation to the where clause

```
v0 = v2
(v1, t1) = insort xs
(v2, t2) = insert x v1
```

6. Notice that there are no more applications in the right hand side

7. Add the definition of the EDT node to the where clause

```
insort (x:xs)
  = (v0, t0)
  where
  v0 = v2
  (v1, t1) = insort xs
  (v2, t2) = insert x v1
  t0 = Node "insort" [dirt (x : xs)] \
     (dirt v0) [t1, t2] "insort ..."
```

Figure 10: Example transformation of the second equation of the `insort` function from figure 4

the algorithm. For simplicity we show the transformation as performed on Haskell code, whereas inside Buddha it is performed on the parse tree of the code. Step 7 illustrates the final transformed version of the equation. Note that the calls to `insort` (recursively) and `insert` are to the transformed versions of those functions.

As can be seen from the final transformed version of the `insort` function in figure 10, the passing of additional parameters results in a significant growth in the size of the transformed code. Nilsson and Sparud [21] describe an almost identical transformation that hides the passing of values and EDT nodes inside a continuation based application operator. The benefits of using such an operator are that the resulting transformed code is more concise than that of the transformation used in Buddha, and also the need for unique identifiers is reduced. The key difference between the Nilsson and Sparud debugging technique and that used in Buddha is the method for representing the value of arguments and results of function applications in the EDT. In the former method, a run–time tree is generated by the transformed program. The arguments and results of applications are stored directly in the nodes of the run–time tree, using existential types to overcome the strong typing restrictions of Haskell. The nodes of the run–time tree are converted by the debugger upon traversal of the tree to make printable versions of the arguments and results of applications available. During the conversion of the tree, impure functions are required to determine whether the applications were evaluated during the execution of the program, and whether the arguments and result of an application were fully or partly evaluated. The transformation of Naish and Barbour avoids the need for two trees to be created and also avoids the necessity of existential types to hold values in the EDT.

4.1.3 Transformation of where clauses

Haskell allows local definitions to be made inside the definition of an equation using a where clause. The main benefits afforded by such syntax are the provision for local scoping of identifiers and the simplification

of the definition of equations into smaller sub–expressions. Haskell equations are not limited to a single top level where clause: they can be arbitrarily nested. However, deep nesting of where clauses can make definitions difficult to read, therefore more than two levels of nesting is considered unusual.

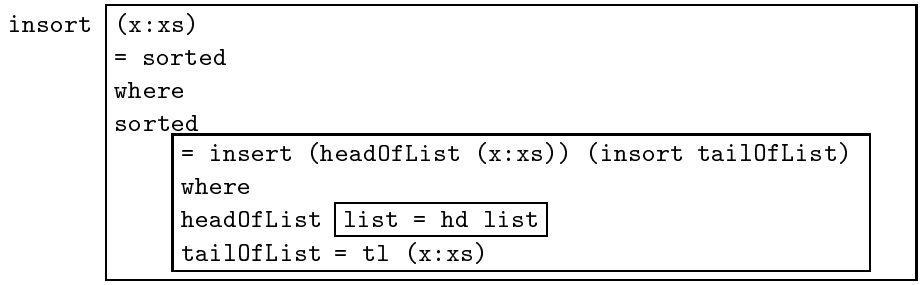

Figure 11: Scoping of identifiers in where clauses

A slightly convoluted definition of the second equation for the function `insort` is given in figure 11. It has the same intended meaning as the definition of `insort` introduced in section 3.3. The scope (or visibility) of identifiers is illustrated by the boxes surrounding sections of code. Identifiers that are introduced within an enclosing box are only visible inside that box (and any box contained within it). The only identifier visible outside the definition of the equation is the name of the function. Within the outermost box, the function name, the arguments to the function and the identifier `sorted` are visible. Within the second where clause, the identifiers `headOfList` and `tailOfList` are introduced, and are only visible within the second inner box. The local function `headOfList` is defined in the second where clause, and its argument `list` is only visible within the definition of `headOfList` (inside the innermost box).

Where clauses behave in much the same way as top–level equations. This presents a problem for the transformation algorithm of Naish and Barbour because it is defined for a syntax that does not support local definitions.

We would like the debugger to be able to ask questions about the correctness of local definitions for a given execution of the program. One possible solution, which is the approach implemented in the current version of Buddha, is to elevate all local definitions in where clauses to top–level function definitions. This is not as trivial as it may first seem. Consider the equation in figure 11. The identifiers `x` and `xs`, which are arguments to the top–level equation, are within the scope of local definition of `tailOfList`. If we were to elevate `tailOfList` to a top–level equation, we would have to explicitly pass the values of `x` and `xs` to it when called, making `tailOfList` a function of two arguments.

We do not alter the final value of the program by elevating local definitions, however, we do significantly alter the structure of the program from its original definition. This may cause some confusion to the oracle when the program is being debugged. Although `tailOfList` has many characteristics of a function, the programmer may not think of it as a function in its own right, but rather as a sub–part of the `insort` function, or simply as a macro that expands to a particular value. Therefore, to ask a question about the correctness of `tailOfList` when applied to two arguments would betray the definition of the program. On the other hand, elevating `headOfList` to a top–level definition is likely to cause less confusion when debugging because it is defined explicitly as a function of one argument, and upon elevation it will remain a function of one argument.

The transformation of the equation in figure 11 after all local definitions are elevated to the top–level is given below:

```
insort (x:xs)
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = sorted x xs
   t0 = Node "insort" [dirt (x : xs)] (dirt v0) [t1] "insort ..."

sorted x xs
   = (v0, t0)
   where
   v0 = v4
   (v1, t1) = headOfList (x : xs)
   (v2, t2) = tailOfList x xs
   (v3, t3) = insort v2
   (v4, t4) = insert v1 v3
   t0 = Node "sorted" [dirt x, dirt xs] (dirt v0) [t1, t2, t3, t4] "sorted ..."

headOfList list
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = hd list
   t0 = Node "headOfList" [dirt list] (dirt v0) [t1] "headOfList ..."

tailOfList x xs
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = tl (x : xs)
   t0 = Node "tailOfList" [dirt x, dirt xs] (dirt v0) [t1] "tailOfList ..."
```

After transformation, the (once local) identifiers `sorted`, `headOfList`, and `tailOfList` appear as if they were top–level function definitions in the original version of the program. Currently, for simplicity, we employ a transformation based on this approach, however, the questions that are generated by the debugger on this transformed code are hard to reconcile with the program as it was orginally defined. A further potential complication exists with this approach due to the scoping rules of Haskell. When elevating a local definition we must be careful not to cause a name–clash with an already existing top–level definition. If a name–clash does occur we have to rename the elevated expression to a unique identifier. This will introduce additional confusion into the use of the debugger when names that were not defined in the orginal program appear as part of questions during the debugging session.

Thankfully there is a rather simple solution to the problems identified above, that can be implemented with only a small extension to the transformation described thus far. Essentially we could tag any definitions that were raised from where clauses to top-level definitions, and provide extra information in the EDT node for the definition such that the debugger can generate more sensible questions for the oracle. Essentially the extra information we need to know is the name of the top–level equation that the local equation came from, the original textual definition of the local equation, the number of arguments the local equation orginally had, and if the local equation had to be renamed to avoid a nameclash, its original name. To implement this solution we could extend the definition of the EDT node type to include

a new node type for this special case of elevated definition, within which we would store the required extra information. A possible extended definition of the EDT node type is shown in figure 12.

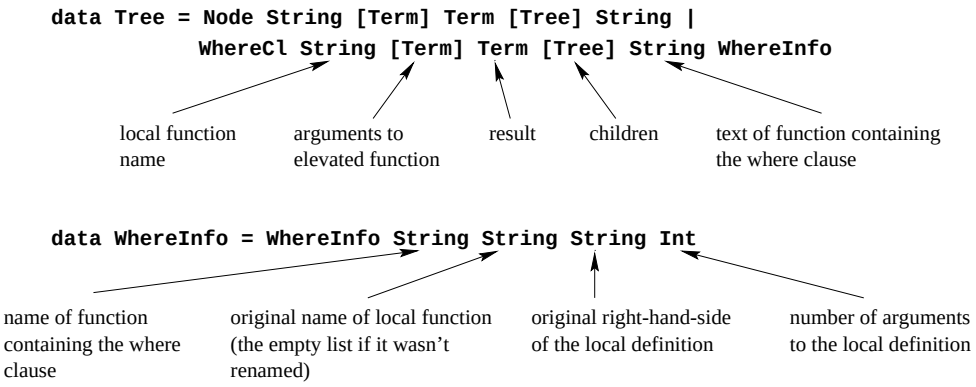

Figure 12: Possible extension of the EDT node type to incorporate extra information for elevated where clauses

Using the extended definition of an EDT node from figure 12, the transformation of the elevated definition of `tailOfList` is given below.

```
tailOfList x xs
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = tl (x : xs)
   t0 = WhereCl "tailOfList" [dirt x, dirt xs] (dirt v0) [t1] wi
   wi = WhereInfo "insort" [] "tl (x:xs)" 0
```

During a debugging session, when the debugger encounters a node in the EDT corresponding to an instance of an elevated where clause, it can make use of the extra in the node to generate more meaningful questions, as in the following example.

```
          :
          :
{where clause in insort}
headOfList [1,2,3,4] = 1
Correct - Y/N/Q? y

{where clause in insort}
let (x, xs) = (1, [2,3,4]) in tailOfList = tl (x:xs)
tailOfList = [2,3,4]
Correct - Y/N/Q? y
          :
          :
```

If the debugger discovers that the number of arguments of an elevated local definition differs from the number of arguments that it had when defined locally, the extra information stored in the EDT node for an application of that definition could be used to provide suitable context for questions presented to the oracle. This is illustrated by the example question asked about the correctness of the local definition `tailOfList`. No further context is required for the question asked about the correctness of `headOfList`

because it was defined locally as a function of one argument, and when elevated it remained a function of one argument.

4.1.4 Transformation of guarded equations

Haskell allows the right hand side of an equation to have multiple possible definitions through the use of guards. The potential definitions for the top–level equation are each preceded by a guard which is simply a boolean expression. During the evaluation of the top–level equation the guards are tested in top–down textual order until one of them evaluates to true.[14] The corresponding equation for the first true guard is selected as the definition of the top–level equation. Haskell also provides the keyword `otherwise` which will always be selected if all the preceeding guards are false.[15] We have already seen the use of guards in the second equation for the `insert` function in figure 4.

The Naish and Barbour transformation algorithm must be extended to support guarded equations because they are not part of the syntax for which it is defined. Guarded equations pose an interesting problem for the transformation of equations because the final definition of the equation is decided during its evaluation. Guards are expressions that evaluate to a boolean value, and just like any other expression they may contain function applications. Therefore, the testing of guards will in some cases cause function applications to be evaluated. For successful debugging to take place, such applications must be represented by nodes in the EDT. In general, some number of false guards will be evaluated followed by a final guard which is true (which may be the catch–all `otherwise` guard). Function applications may be made during the evaluation of any preceding false guards, during the evaluation of the true guard, and finally during the evaluation of the right hand side corresponding to the true guard. The potential children for a node whose definition uses guards must include all the function applications just mentioned. Therefore we cannot statically define the complete EDT node for an equation with guards as in the third step of the transformation algorithm, because the potential children of the node will depend on the evaluation of guards at run–time.

Consider the redefinition of the second equation for the `insert` function in figure 13. Note that in the new definition the first comparison of `x` and `y` is now made in the function `gt`. We have intentionally shifted the error in the program to the definition of `gt`, which is intended to be true if its first argument is greater than its second argument. However, as it is defined, `gt` does not reflect its intended meaning. The important point to note from this new definition is that determination of the truth of the first guard requires the application of `gt` to its arguments. It is essential that we generate a node in the EDT for this application, because the actual error in the program is in the definition of `gt`. The application of `gt` to its arguments must therefore appear as a child of the EDT node for an application of this equation for `insert`.

```
insert x (y:ys)
   | gt x y = y : (insert x ys)
   | x < y  = x:y:ys
   | otherwise = y:ys

gt x y = x < y
```

Figure 13: Redefinition of the buggy insert function, introduced in figure 4

Within Buddha we employ a two stage process for transforming guarded equations. The first stage consists of an application unravelling process similar to the second step of the transformation algorithm.

[14] It is possible there will be no true guards for a given equation. In such a case, alternative equations may be applied. In the absence of alternative equations a program exception will occur, causing the execution of the program to halt.

[15] For convenience, the keyword `otherwise` is simply another name for the boolean value `True`.

However, in this case we must unravel the applications made in all guards, and all their corresponding equations. The unravelling of the first guard of `insert` and its corresponding equation is shown below. The identifier `gval0` represents the value of the first guard (either true or false) and the identifier `gtree0` represents the EDT node for the application performed in the guard. The identifier `eval0` represents the value of the equation following the first guard, whilst `eval1` and `etree1` represent the value and the EDT node for the innermost application in the equation respectively.

```
(gval0, gtree0) = gt x y
eval0 = y : eval1
(eval1, etree1) = insert x ys
```

After all the guards and their corresponding right hand sides have been unravelled we move to the second stage of the transformation involving guarded equations. A mechanism is needed which can process the transformed guarded equations, determine the successful guard (if there was one) and return the value of its corresponding equation along with the EDT nodes for all applications made for each tested guard and the evaluation of the equation. We represent each guard and its corresponding equation by a four element tuple. The first element of the tuple contains the value of the guard which can be either true or false. The second element of the tuple contains a list of EDT nodes that represent potential applications in the evaluation of the guard. The third element of the tuple contains the value of the corresponding equation (if it were to be evaluated). The fourth element of the tuple contains a list of the EDT nodes that represent potential applications in the evaluation of the equation. Therefore, the type of the tuple is `(Bool, [Tree], a, [Tree])`. We generate a list of these tuples for every guard–equation pair in top–down textual order. The list is passed to a new function called `guards` which is defined below.

```
guards :: [(Bool, [Tree], a, [Tree])] -> Maybe (a, [Tree])
guards gs = guardsAcc [] gs

guardsAcc :: [Tree] -> [(Bool, [Tree], a, [Tree])] -> Maybe (a, [Tree])
guardsAcc _ [] = Nothing

guardsAcc acc ((False, gts, _, _):gs)
   = guardsAcc (acc ++ gts) gs

guardsAcc acc ((True, gts, ev, ets):gs)
   = Just (ev, (ets ++ gts ++ acc))
```

The processing of the list is performed in the accumulator function `guardsAcc`. Essentially, `guardsAcc` traverses the list from left to right accumulating the EDT nodes that are potentially evaluated up to and including the first true guard and its corresponding right hand side. The traversal of the list stops if a tuple representing a true guard is found (denoted by the value `True` in the first element of the tuple). If a successful guard is found, `guardsAcc` returns the value `Just` (*val*, *nodes*) such that *val* correpsonds to the value of the selected equation, and *nodes* corresponds to the accumulated list of EDT nodes for all tested guards concatenated with the list of EDT nodes for the selected equation. It is possible that none of the guards are successful. In this case `guardsAcc` returns the value `Nothing` indicating this failure.

The only remaining thing to do in the transformation of a guarded equation is to tie the results of the `guards` function into the results of the transformed equation. If a successful guard was found, then the value of the transformed equation is equal to the first element of the tuple returned by `guards` and the children of the EDT node for the equation are the second element of the same tuple. However, if no successful guards were found then we must ensure that the overall evaluation of the transformed equation does not proceed. The complete transformation of the `insert` function from figure 13 is somewhat more complicated than the transformations presented earlier. Readers interested in the complete transformation of `insert` are referred to appendix A.

There is one flaw in the transformation of guarded equations presented above. Haskell allows a function to be defined with multiple equations, where the patterns for those equations are overlapping. In cases where such overlapping equations also involve guards, the above transformation may fail to generate a sufficient EDT. Consider the simple program below.

```
foo n | isOdd n = []
foo n | isEven n = [n]

isOdd x = (x `mod` 2) /= 0

isEven x = (x `mod` 2) == 0

isOddOrZero x = (isOdd x) || (x == 0)
```

Let us assume that the intended meaning of the function `foo` is to return the empty list for arguments that are either the number `0` or odd numbers, and for all even arguments (except `0`) it should return a single element list containing that argument. As it happens, this does not equate with the actual meaning of `foo` as it is defined. For the intended meaning to correspond with the actual meaning the first equation for `foo` should be re–written as "`foo n | isOddOrZero n = []`". Therefore the error is in the first definition of `foo` because the wrong function is applied in the guard. For example, evaluation of `foo 0` results in the value `[0]`, whereas it is intended to evaluate to the empty list. The transformation for guarded equations presented above will not allow the debugger to locate the correct source of the bug in the application `foo 0`. The error is in the first equation for `foo`, however since all guards in that equation fail for the argument `0`, no nodes are generated in the EDT for the application of this equation or the failed guard. Instead the second equation is evaluated, and nodes for its application and the application of `isEven 0` are stored in the EDT. Since `isEven` is correct, the debugger will conclude that the error is in the second equation for `foo` when in fact the error is in the first equation.

In the current version of Buddha we disallow functions defined by multiple equations with overlapping patterns if any of those equations uses guards. In future versions of the debugger we hope to extend the transformation to handle such function definitions.

4.1.5 Transformation of curried function definitions

Suppose we define a function which evaluates to one plus the value of its numerical argument. A simple definition of such a function is given for `inc` below:

```
plus x y = x + y
inc x = plus 1 x
```

If we apply our transformation to this program, the transformed version of `inc` would look like:

```
inc x
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = plus 1 x
   t0 = Node "inc" [dirt x] (dirt v0) [t1] "inc ..."
```

The resulting type of the transformed version of `inc` is `Num a => a -> (a, Tree)`, just as we would have expected it to be. However, it is also possible to define a curried version of `inc` as in the following:

```
inc = plus 1
```

The curried version has the same type and equivalent meaning to the uncurried version above. However, when we apply our existing transformation to the curried version of `inc` we get:

```
inc
   = (v0, t0)
   where
   v0 = v1
   (v1, t1) = plus 1
   t0 = Node "inc" [] (dirt v0) [t1] "inc ..."
```

The transformed version of the curried `inc` is not type correct. Notice that the type of the expression `plus 1` is functional, however the transformed version expects the expression to a non–functional value (namely a two element tuple). From this simple example it can be seen that the current transformation is insufficient for curried function definitions.

This problem can be easily solved if the arities of curried function definitions are known in advance of the transformation. If the arities of curried function definitions are known, we can compare the number of formal arguments in the definition with the number of arguments that an uncurried version of the function would need. If the number of formal arguments in the definition is less than the arity of the function, we can supply additional arguments to the function definition such that the difference is met. In the example above, this would simply mean converting the curried definition of `inc` into its non–curried definition before it is transformed. It is safe to add additional parameters to curried function definitions because the curried and uncurried versions of the same function definition have the same meaning.

Correct determination of the arity of a function is difficult without type information for the function. It is possible to automatically infer the types of functions in a Haskell program using the type analysis of the compiler or interpreter. In the current version of Buddha we do not use type analysis to determine the types of functions, and hence their arities. Instead we require that curried function definitions be supplied with correct type signatures in the source of the program. Where type signatures are provided the arities can be calculated directly. Functions without type signatures are assumed to be defined with a full set of formal arguments. In such cases the arity of a function is calculated from the number of formal arguments in its definition.

4.1.6 Transformation of higher–order arguments

Previously, in section 3.4, we proposed that the domain of functions and their transformed counterparts are not the same. Where functions appear in the original program, transformed versions of those functions appear in the transformed program, including functions in higher–order arguments. This is demonstrated using the program in figure 14.

```
incs :: [Int] -> [Int]
incs xs = map (plus 1) xs

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (i:is)
   = (f i) : (map f is)

plus :: (Num a) => a -> a -> a
plus x y = x + y
```

Figure 14: Definition of the higher–order function `map`

The function `incs` increments each element in a list of integers by one. It does so by passing the function `plus 1` as a higher–order argument to `map`, which recursively applies its first argument to every element of the list as its second argument. All non–functional objects in the original program retain

their value (and hence type) in the transformed version of the program. However, functional objects are transformed and hence do not retain their original value (or type) in the transformed program. The type of `plus 1` is `Num a => a -> a`, however in the transformed version of the program the type of `plus 1` is `Num a => a -> (a, Tree)`.

The difference in the domains between functions and their transformed versions does not in itself pose a problem for the debugging scheme provided that there is a one–to–one mapping between them. The true difficulty of higher–order arguments was highlighted in section 2.3: functions are abstract objects which do not have inherent printable representations. However, for the oracle to decide upon the correctness of a function application we must display a representation of the arguments that were involved in the application. We cannot rely on the names of functions to be retained during the execution of the program, and hence `dirt` may not always generate suitable representations of higher–order arguments. Therefore, we need to transform higher–order arguments so that we can generate a printable representation of them.

Nilsson and Sparud [15] propose that functional arguments be transformed into a tuple such that the first element of the tuple is the function itself and the second element is a printable representation of the function. For example, the function `plus 1` which is passed as an argument to the function `map` in figure 14, would be transformed into the tuple `(plus 1, "(plus 1)")`, and the type of the transformed version of map will become `(a -> (b, Tree), String) -> [a] -> ([b], Tree)`. We adopt this approach in Buddha for higher–order arguments that are curried function applications (such as `plus 1`).

To incorporate the transformation of curried functions as arguments, we make the following three modifications to the algorithm presented in section 4.1.2.

1. Alter the definition of the second step to read:

   > Repeat the following step until there are no further applications in r.
   > Let g be a function of arity m.
   > If $g\ b_1\ b_2 \ldots b_k$ is an innermost function application in r, and $k < m$, replace the application by $func_i$ and add the following equation to the top `where` clause of f
   >
   > $func_i\ =\ (g\ b_1\ b_2 \ldots b_k, \texttt{"("}\ \texttt{++}\ \texttt{"}g\texttt{"}\ \texttt{++}\ args\ \texttt{++}\ \texttt{")"})$
   >
   > ($func_i$ is a unique identifier in the scope of f and $args$ is the concatenated list of string representations for $b_1\ b_2 \ldots b_k$).
   > Else if $k = m$, then perform the operation as originally defined in this step.

2. Add the following new stage to the second step of the algorithm:

   > If argument a_i of f is functional (i.e. has arity > 0), rewrite any occurrence of a_i in the left hand side of the definition and any non–application of a_i in the right hand side of the definition as (a_i, rep_i). Whenever a_i is applied in the right hand side it remains textually unchanged.

3. Re–define the third step to read:

   > Add the following equation to the top where clause of f. ($t_1, \ldots, t_p$ correspond to the t_i in the previous step)
   >
   > $t_0 = \texttt{Node "}f\texttt{"}\ [argrep_1,\ argrep_2, \ldots argrep_3]\ (\texttt{dirt}\ v_0)\ [t_1, \ldots, t_p]$ (*text reference*)
   >
   > If argument a_i is functional then $argrep_i$ is rep_i, otherwise it is `dirt` a_i.

Applying these extensions to the algorithm, the equation for `incs` in figure 14 transforms into the following definition:

```
incs xs
   = (v0, t0)
   where
   v0 = v1
   func1 = (plus 1, "(" ++ "plus " ++ "1" ++ ")")
   (v1, t1) = map func1 xs
   t0 = Node "incs" [dirt xs] (dirt v0) [t1] "incs ..."
```

And the second equation for `map` transforms into the following definition:

```
map (f, rep1) (i:is)
   = (v0, t0)
   where
   v0 = v1 : v2
   (v1, t1) = f i
   (v2, t2) = map (f, rep1) is
   t0 = Node "map" [rep1, dirt (i : is)] (dirt v0) [t1, t2] "map ..."
```

Unfortunately this method is not a complete solution for higher–order arguments. Consider the small program in figure 15. Under the current transformation, it is assumed that functional arguments will be apparent from the type signature of the function. However, this is not the case for the polymorphic function `reverse`, which is defined over lists of any type. In order to obtain a printable representation of the argument to `reverse`, the debugger will cause `dirt` to be applied to that argument. However, in the case that the elements of the list are functional, `dirt` cannot be guaranteed to supply a meaningful representation of those elements. As we can see from the function `main` reverse can be applied to a list of functions.

```
main :: [a -> b]
main = reverse [plus x | x <- [1,2,3]]

plus :: Num a => a -> a -> a
plus x y = x + y

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = (rev xs) ++ [x]
```

Figure 15: Polymorphic function `reverse` and higher–order arguments

This problem can be solved if we know the type of every identifier in the program text. Essentially, we can wrap every object that is passed as an argument to a function in a new data constructor. The new data–type should distinguish between functional values and non–functional values. For example it might look like:

```
data Arg a = Fun a String | NonFun a
```

For functional values we store the function and a string representation of the function, and for non–functional values we simply store the value itself. We then define a function `dirt'` to be used in place of `dirt` in the usual transformation. The operation of `dirt'` is such that it first checks to see whether its argument is wrapped in the constructor `Fun`, if so it simply uses the string representation of the function, otherwise it proceeds with the usual lower level generation of a representation employed by the current version of `dirt`.

We do not have sufficient type information available in the current version of Buddha to perform this more thorough treatment of higher–order arguments, however we hope to incorporate the technique in future versions of the debugger.

4.2 The impure function `dirt`

The primitive function `dirt`[16] is central to the success of the Naish and Barbour debugging technique. In this section we discuss the implementation of `dirt` using the internal interface of the Hugs interpreted Haskell system. We discuss the fundamental requirements of `dirt`, and describe how they are satisfied in our system.

The impure function `dirt` serves two important purposes within the debugger. Firstly, it provides a method for representing all arguments and results of functions under a single type. This means that no matter what type an object is in the original program we can always store a representation of it in the EDT such that the strong typing constraints of Haskell are satisfied without the need for existential types. This property is evidenced by the type signature of `dirt` which is `a -> Term`. Secondly, it provides a representation of arguments and results of functions which reflects the extent to which they are evaluated when it is applied to them. In other words, `dirt` returns a representation of the state of evaluation of its argument which may not be equivalent to its declarative value.

Naish and Barbour describe the following three fundamental requirements that must hold for an implementation of `dirt`:

1. It must not be applied to its arguments until they are in their final state of evaluation which is equal to their state of evaluation at the end of the execution of the original program. In other words, applications of `dirt` must be evaluated lazily.

2. It must not force the further evaluation of its argument.

3. It must return a representation of its argument that contains sufficient information for the oracle to decide whether an application in an EDT node was correct or erroneous.

The first requirement constrains the time at which `dirt` is applied to its argument. This is ultimately determined by the evaluation of the debugger, and so we cannot control this property from within the implementation of `dirt` itself. To illustrate that this first requirement can be satisfied within Buddha, we must first describe the order of evaluation that occurs inside the debugger. The debugger is written in Haskell, and so its evaluation is lazy. The laziness of the evaluation means that all computations are driven by the need to produce output to the oracle. We rely on this laziness to ensure that `dirt` is applied to its argument at the desired time.

Recall that each function in the transformed program returns a tuple containing the value of the original function when applied, and an EDT representing the computation of that value. The debugger first prints the final result of the original program, which drives the complete evaluation of the first element of the tuple returned by the topmost application in the transformed program. The evaluation of the first element of this tuple is equivalent to the evaluation of the original program: the EDT is not needed to compute this value, and so up to this point it is not evaluated. After the final value of the original program is printed, the debugger then begins to traverse the EDT starting at the node corresponding to the topmost application. The nodes at each level in the EDT are evaluated as the tree is traversed. Each node that is visited is evaluated to the extent that sufficient information can be presented to the oracle in order for the correctness of the application to be determined. The information presented to the oracle at each node includes the name of the function that was applied and the arguments and result of that function (to the extent that they were evaluated at the end of the original program). Therefore, the evaluation of each application of `dirt` is delayed until the value of that application is needed to be

[16] Display Intermediate Reduced Term.

displayed to the oracle. Since the evaluation of the original program proceeds to completion before the debugger begins to traverse the EDT we can be sure that `dirt` will be applied to its arguments in their most evaluated form.

The second requirement ensures that the evaluation of the original program is preserved in the evaluation of the transformed program. To demonstrate the necessity of this constraint we present the example program in figure 16.

```
take n as
   | as == []  = []
   | n  == 0   = []
   | otherwise = head as : (take (n - 1) (tail as))

main = take 3 [1..]
```

Figure 16: Example program where an infinite argument is only partially evaluated

During the evaluation of `main`, `take` is applied to an infinite list of integers. At the end of the computation of `main`, the infinite list is only evaluated three elements deep: the remainder of the list is left unevaluated. In the transformed version of `take`, `dirt` will be applied to the arguments `n` and `as`. If `dirt` were to force the evaluation of its argument, the application `dirt as` would loop indefinitely because the full evaluation of `as` is an infinite computation. This is obviously an undesirable property since the original program was terminating on this input. It is also possible that (for some reason) `dirt` may only cause a partial further evaluation it argument, and hence avoid non–termination. However, it is still desirable to avoid *any* further computation if `dirt` is to reliably represent objects only to the extent that they were evaluated in the original program.

In order for `dirt` to satisfy this second requirement, it must be able to inspect the internal Haskell run–time representation of an object without causing any further evaluation of the object. Such an operation is impossible to perform within the confines of the Haskell language. This is because within Haskell we can only reason about the value of an object, we cannot reason about its state of evaluation. Therefore, `dirt` breaks the declarative semantics of Haskell. However, for the purposes of providing an adequate debugger, this is a necessary evil.

The third requirement is difficult to define precisely. Obviously, arguments to `dirt` that are fully evaluated should retain their usual textual representation. However, objects that are only partially evaluated are harder to represent because the unevaluated parts have no syntactic counterpart within Haskell. The key observation to be made here is that unevaluated expressions at the end of the original program execution had no influence over the final value of the program. Therefore, the value of unevaluated expressions is not important in determining the correctness of an application. The function `dirt` can safely represent the unevaluated parts of its argument with a suitable symbol[17], indicating that they were unevaluated, and that the correctness of the application does not depend of their value. The value that `dirt` returns depends on the extent to which its argument is evaluated rather than the value of its argument, and so `dirt` is not referentially transparent.

To understand our implementation of `dirt` it is helpful if one has a moderate conception of the evaluation mechanism employed in Hugs. Our description will necessarily be brief and incomplete, however, the reader is directed to *The implementation of the Gofer functional programming system* [6] which describes in detail the implementation of the programming system upon which Hugs is based.

The entire Hugs system is implemented in the C programming language. The execution of a Haskell program in Hugs consists of four stages. In the first stage the Haskell text of the program is parsed and checked for syntactic correctness. In the second stage various static analyses are performed on the parsed program including type analysis. If the static analysis stage is satisfied the program is translated

[17] In many cases an underscore will be sufficient.

in the third stage into a simple internal language. Finally, in the fourth stage the translated program is evaluated by an execution mechanism which imitates the operation of an abstract machine defined on the internal language.

During the execution of a program the Hugs evaluation mechanism manipulates two memory segments. The first segment is called the stack. During program execution the stack is used to hold sub–expressions that are being evaluated, intermediate values and function arguments. The second segment is called the heap. The heap is used to store both program expressions (represented as graphs) and program data. Initially, the top–level expression to be evaluated in a program resides in the heap. It is copied into the stack, and the internal function `eval()` reduces the expression into a simplified form called *weak head normal form*[18]. For example, the value `12` is the weak head normal form of the expression `6 * 2`. The reduction of an expression may cause further expressions to be copied onto the stack, which in turn must be reduced into weak head normal form by `eval()`. Program expressions and program data are represented by compositions of an internal abstract data type called a *cell*. Almost every object from a Haskell program is mapped to a cell composition within Hugs, including all the basic types, composite data structures such as tuples and lists, and abstract objects such as expressions (or function applications). Indeed, the Hugs heap and stack are simply arrays of cells. For efficiency reasons, the low–level implementation of the cell data type is rather complicated. Within the rest of this section we reason about cells in an abstract manner to simplify the discussion.

The operation of `dirt` when applied to a Haskell object is broken into two steps. In the first step a copy of the cell representation of the object is obtained from the Hugs heap. In the second step the cell is dissected to reveal its internal contents. Composite cell structures are dissected recursively until the cell structure has been completely inspected, or sufficient information about the Haskell object has been obtained. To implement these two steps we make use of two primitive functions from the Hugs specific library `HugsInternals.hs`: `getCell` and `classifyCell`. Both functions are called from within the Haskell code of the debugger, however, in order for them to interface with the internal Hugs system they are implemented in C and are compiled into the Hugs executable program.

The task performed by `getCell` is simple: given an argument of any Haskell type, it will return a value of type `Cell` that corresponds to the internal representation of the argument. The type `Cell` is built into the Hugs system, and is essentially a convenient Haskell wrapper for the internal representation of a cell. Therefore, `getCell` provides the required mechanism for implementing the first stage of `dirt`. Once we obtain the internal representation of a Haskell object we must implement a mechanism for inspecting its contents. This is the task performed by `classifyCell`. Given an argument of type `Cell`, `classifyCell` will return a value of type `CellKind` which identifies what type of internal structure the cell represents and what its contents are. The `CellKind` data type is implemented in Haskell, however, it is partially defined in terms of the type `Cell` to allow for composite objects such as data structures and function applications. Below is the Haskell definition of the `CellKind` type, as provided in the `HugsInternals.hs` library.

```
data CellKind
  = Apply   Cell [Cell]
  | Fun     Name
  | Con     Name
  | Tuple   Int
  | Int     Int
  | Integer Integer
  | Float   Float
  | Char    Char
  | Prim    String
  | Error   Cell
```

[18] Note that not all expressions can be reduced to weak head normal form. For example, the application of `inf` (from figure 1) to a numeric argument is non–terminating because the application does not have a weak head normal form.

The `Apply` constructor allows for applications to be represented within the `CellKind` type. At first sight, the internal representation of objects appears to allow illegal constructs. For example, it seems plausible from the definition of `CellKind` that a number could be applied to a character, which is not a valid or sensible operation from a Haskell perspective. However, the strong typing constraints on the original Haskell program ensure that only type correct structures will appear in the internal representation of objects. Therefore we do not need to perform any type checking when we inspect the internal representation of objects. Within `dirt` we completely unfold the `CellKind` data structure into a new data structure, which is identical to the `CellKind` structure, except that all `Cell` elements are recursively expanded.

Consider the internal representation of the infinite list which is passed as an argument to `take` in figure 16. At the end of the evaluation of the function `main` the list is only evaluated three elements deep. A visual representation of the list at the end of the evaluation of `main` is provided in figure 17. Essentially the figure illustrates the `CellKind` representation of the list such that all `Cell` elements are expanded.

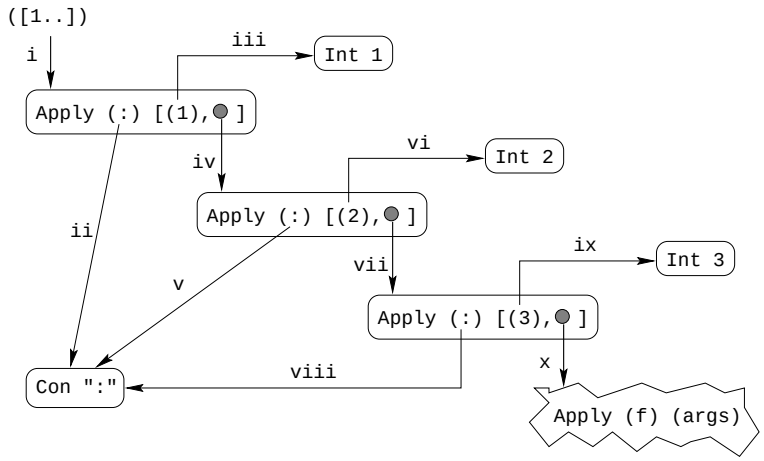

Figure 17: Internal cell structure of the list `[1..]` when evaluated three levels deep

The rounded boxes and the jagged box in the diagram represent each unique cell in the internal representation of the list. The outermost cell representing the entire list resides at the top left corner of the diagram. The list is constructed by a series of applications of the list constructor cell (“`:`”) to two argument cells. The first argument to each application of the list constructor cell is an element of the list (in this case an integer cell) and the second argument of the list constructor cell is an application of cells that represents the tail of the list. Notice that the last element of the list does not contain an application of the list constructor cell, but rather an application of a function. This cell represents an unevaluated expression (hence the jagged box), and corresponds to the unevaluated tail of the infinite list. The function applied in this cell is an internal Hugs function that generates the consecutive elements of an infinite series of integers.

The cell representation of the list is dissected from the top left corner to the bottom right corner of the diagram. Each labelled arc in the diagram represents the expansion of a cell into an object of type `CellKind`. The cell structure of the list is expanded according to the ascending order of arc labels. The recursive expansion of the structure halts when the application of a function is discovered. The presence of a function application within a data structure is conclusive evidence that this branch of the data structure has not been evaluated, and so the expansion of the cell structure corresponding to this branch may safely stop at this point. As `dirt` traverses the cell structure it generates a representation of the structure. At present, this representation is a string, however, this is only for convenience, and

other representations are possible. The representation of the cell objects corresponding to values of basic type are simply the string representation of the data inside the cell. For example, the cell containing the integer 12 (`Integer 12`), is simply represented as the string "`12`". Function applications are `CellKind` structures of the form: `Apply (Fun` f`)` `[`arg_1, $arg_2, \ldots, arg_n$`]`. The presence of a function application at the end of the computation means that the application was not evaluated, hence function applications are represented by an underscore character ("`_`") (except when at the end of a list, for which we use another notation described below). Applications of data constructors (except the list constructor) are represented by the name of the constructor (as a string) followed by the string representations of the arguments to the constructor, which are parenthesised in the case that the arguments are also applications of data constructors. Applications of the list constructor are treated differently, so that the Haskell short–hand notation for lists is generated. Where the tail of a list is found to be unevaluated, a special (non–Haskell) notation is used to indicate this occurrence. So, for example, the representation of the list `[1..]` when evaluated three elements deep is the string "`[1, 2, 3, ...?]`". In appendix B we demonstrate the transformation of a buggy version of `take` along with an example debugging session applied to the transformed program. The debugging session illustrates the questions that the debugger presents to the oracle when partially evaluated data structures are involved.

There is one final concern in the implementation of `dirt` that we have failed to mention. It is possible that the cell structure for a Haskell object contains cycles. For example, in the definition of the infinite list of ones "`ones = 1:ones`", both occurrences of the name `ones` are represented by the same cell internally. We solve this problem by keeping a list of the predecessor cells for each cell that is expanded. When a cell is encountered, we compare it to each element of its list of predecessors. If a match occurs we have found a cycle in the cell structure. Cycles in the cell structure are tagged with a special label and are not further traversed.

4.3 Traversal of the EDT in search of topmost buggy nodes

When the transformation of the original program is complete the transformed code is loaded into the interpreter, executed, and the EDT that results from the execution is traversed by the debugger in search of a topmost buggy node. We refer to this process as *error diagnosis*, the implementation of which corresponds to the second level of the Nilsson and Sparud debugging architecture described in section 3.3. In this section we present a simple algorithm for traversing the EDT which is guaranteed to find at least one topmost buggy node if the top node in the EDT is erroneous. Naish and Barbour present an error diagnosis algorithm in [11], however it does not give an accurate account of how the interaction with the oracle ought to be implemented. In our algorithm, the interaction with the oracle is explicit. To reduce the length of this section we use the Haskell code taken directly from the implementation of Buddha to describe the algorithm. This is possible due to the high–level nature of the Haskell language. Although the code makes use of monadic I/O to interact with the oracle, it is not necessary for the reader to have knowledge in this area.

The definition of the code for error diagnosis is provided in a module called `StdBuddhaLib.hs`. This module is imported in the definition of the transformed program, so that the error diagnosis code can execute the transformed program to generate the EDT.

Let us consider the structure of the EDT again. Each node represents an application of a function. The children of a node are stored as a list inside the node. The top node of the EDT represents the topmost application in the executed program, and the leaves of the EDT are nodes which do not have any children (the absence of which is indicated by an empty list inside the node). Error diagnosis consists of a partial depth–first traversal of the EDT which terminates upon the location of a topmost buggy node. The children of correct nodes are not visited in the traversal. Nodes representing applications that were not evaluated (and hence the sub–trees defined by those nodes) are ignored in the traversal.

The error diagnosis algorithm recurses through three stages. The operation of each stage is described in the following list, along with its implementation in Haskell.

1. The first stage of the diagnosis is implemented by the function `debug`, which checks if its argument node was evaluated. The determination is based on the result of the application, and is performed in the function `isEvaluated`. If the node was not evaluated, the flag `Nothing` is returned to indicate that this branch of the EDT should be ignored. If the node was (at least partially) evaluated the node is passed to the second stage of the diagnosis (implemented by the function `debugNode`) to be checked for correctness.

```
debug node
   = do
        isEvaluated node >>= \e ->
           case e of
               True  -> debugNode node
               False -> return Nothing
```

   In the current implementation of the function `dirt`, unevaluated values are represented by the underscore character. Therefore, the determination of whether a function was evaluated is simply a comparison of the result of the application against the string containing only the underscore character.

```
isEvaluated node@(Node _ _ result _ _)
   = do
        case result of
           "_" -> return False
           _   -> return True
```

2. The second stage of the diagnosis is implemented in the function `debugNode`, which presents the node to the oracle. If the oracle deems the node to be correct, then the flag `Nothing` is returned to indicate that this branch of the EDT should not be further traversed. If the oracle deems the node to be incorrect, the children of the node are checked for correctness in the third stage of the diagnosis (implemented by the function `debugChildren`). The interaction with the oracle is provided by the function `correct` and the presentation of the node is implemented by the function `showNode` (which is not defined here).

```
debugNode node@(Node _ _ _ children _)
   = do
        (showNode node)
        putStr "\n"
        correct >>= \r ->
           case r of
             Yes -> return Nothing
             No  -> debugChildren node children
```

   The interaction with the oracle is rather simplistic. The function `showNode` displays the contents of the node to the oracle. It is at precisely this stage that the applications of `dirt` in the node are evaluated. Essentially, `showNode` prints the name of the function, the representations of its arguments and result to the display. The oracle is queried about the correctness of the application by the function `correct`, which loops indefinitely until either a "`y`" (for yes) or a "`n`" (for no) is supplied by the oracle.

```
correct
  = do
       putStr "Correct - Y/N/Q?\n"
       getLine >>= \r ->
          case r of
             "y" -> (return Yes)
             "Y" -> (return Yes)
             "n" -> (return No)
             "N" -> (return No)
             a   -> correct
```

3. The third stage of the diagnosis is implemented by the function `debugChildren` which checks the children of an erroneous node for correctness. Its arguments are a single node (called the parent) and a list of nodes which are the children of the parent. Starting from the front of the list the children are passed one by one to the first stage of the diagnosis to be checked for correctness. If the function `debug` returns the flag `Nothing` then either that child was not evaluated or it was deemed correct. In such a case the next child in the list of children is passed to `debug`. If a child is deemed erroneous, then the sub–tree defined by that node is recursively traversed by `debug` which will eventually return a topmost buggy node. If all the children of a parent node are deemed to be correct, or the parent does not have any children, then that node must be a topmost buggy node, so it is returned by the function `debugChildren` as the source of the error in the EDT.

```
debugChildren parentBug []
   = return (Just parentBug)

debugChildren parentBug (c:cs)
   = do
        (debug c) >>= \r ->
           case r of
             Nothing -> (debugChildren parentBug cs)
             _       -> (return r)
```

The whole diagnosis is driven by the function `buddha`.

```
buddha (result, tree)
   = do
       finalResult result
       buggyNode <- debug tree
       outputResult buggyNode
```

The argument to `buddha` is the top level application of the transformed program. The evaluation of the original program is forced to completion by the function `finalResult` which prints first element of the tuple returned by the application. The top node of the EDT (which is the second element of the tuple returned by the application) is then passed to the function `debug` which begins the first stage of the recursive diagnosis algorithm. The topmost buggy node returned by `debug` is passed to the function `outputResult` which displays the erroneous node and the textual definition of the equation that was applied the node.

5 Related work

The development of declarative debugging techniques has progressed significantly since the seminal work of Shapiro [18]. Originally, declarative debugging was applied to Prolog programs. Recently, much research has been invested into adapting the ideas of declarative debugging to other programming languages.

The motivation for this research has been provided by the success of the original Prolog debugging systems and the impracticality of traditional debugging techniques for non–imperative languages. Nilsson and Sparud [15] provide a detailed historical perspective of debugging lazy functional programming languages. In particular they note that some of the earliest work in the field was performed by Hall and O'Donnell [2], however, many of the problems discussed in our paper are avoided in the work of Hall and O'Donnell because their target language was untyped. Although there has been a reasonable amount of effort directed at debugging lazy functional languages, there remains a dearth of suitable debugging systems for languages such as Haskell. In this section we discuss three approaches to satisfying the need for debugging systems for lazy functional languages. The first two approaches are based on declarative debugging, and have been referred to significantly throughout this paper. The third approach marks a departure from declarative debugging, and represents an alternative direction of research in the field.

5.1 Naish and Barbour

The work by Naish and Barbour [11, 10] provides the basis upon which Buddha is implemented. The specification of a declarative debugger for a logical–functional language NUE–Prolog is given in [10]. Many issues that effect the implementation of a debugger for Haskell are not encountered by the NUE–Prolog debugger. This is because NUE–Prolog and the debugger are both implemented in Prolog, which simplifies the creation of the EDT and the provision of low–level primitives. The specification of a more portable declarative debugging system for lazy functional languages is given in [11], which provides the algorithm presented in section 4.1.2, and a high–level outline of the function `dirt`. Naish and Barbour also briefly discuss means for improving the memory efficiency of their debugging technique, which we hope to incorporate into future versions of Buddha.[19]

5.2 Nilsson and Sparud

Nilsson and Sparud have contributed significantly to the field of declarative debugging for lazy functional languages both independently [12, 13, 14, 19, 20] and in collaboration [15, 21]. The extent of their work covers the generation of EDTs via modified language implementation and source to source transformation of program definitions. The work of Nilsson and Sparud has been referred to throughout this paper: in section 3.2 we discussed both of their EDT generation methods; in section 4.1.1 we compared the structure of their EDT to that used in Buddha; and in section 4.1.2 we briefly mentioned the differences between their source transformation and that of Naish and Barbour.

The objective of Nilsson and Sparud is roughly equivalent to that of Naish and Barbour, which is to provide a declarative debugging technique suitable for a large class of modern lazy functional languages. Ultimately, the main difference between the two approaches is the way in which representations of objects are stored in the EDT. This is evidenced by the low level primitives that each approach requires. Nilsson and Sparud [15] suggest that their technique may be more portable than that of Naish and Barbour because the success of `dirt` when applied to higher–order arguments requires the names of functions to be maintained during the execution of programs, which is not done in some Haskell implementations.[20] We acknowledge this criticism, and in the implementation of Buddha we adopt a similar source level transformation of higher–order arguments as that proposed by Nilsson and Sparud, thus improving the portability of the debugger to other Haskell implementations.

An attempt to implement a debugger for Haskell based on the work of Nilsson and Sparud is presented by Hannan [3]. The current implementation of Buddha makes use of the Haskell parser and arity analyser from this work. Unfortunately, the required low–level primitives of the technique were not implemented,

[19] For details of the memory usage of Buddha, and potential improvements, see section 7.2.

[20] In particular, the names of user–defined functions are not maintained by the run–time system of the Chalmers Haskell implementation HBC.

thus limiting the use of the debugger to a subset of Haskell that does not allow lazy functions or data constructors.

5.3 Kishon and Hudak

A completely different approach to debugging lazy functional languages than the previous two mentioned is taken by Kishon and Hudak [1]. They define a method for semi–automatically generating a family of source–level program monitors such as profilers, tracers and debuggers, by combining the continuation semantics of the language with a specification for the desired monitor. The work of Kishon and Hudak represents a formal approach to the development of program monitors, and makes use of the similarities in various program transformation techniques to produce an overall method which is both provably correct and flexible. The usability of their resulting debugger is hindered by the fact that it reflects the lazy evaluation of programs, making the order in which questions are asked rather confusing. Furthermore, the difficulty of both developing continuation semantics for a large language like Haskell, and specifying the desired monitor properties of a complete debugging system may further prohibit that practical use of their approach.

6 Contribution

We believe that Buddha is a significant advancement of debugging technology for the Haskell programming language. Our implementation compares favourably to the debugging systems described by Nilsson and Sparud [15]. The advantage of our system over theirs is that we only require the construction of a single EDT, which does not need existential types to satisfy the type constraints of Haskell. The closest known implementation to Buddha is the debugger for the language Gofer, by Hannan [3], which is based on the work of Nilsson and Sparud. Hannan's system supports more syntax than Buddha, but does not handle lazy functions correctly, and therefore is severely limited in its application.

Our biggest achievement was the provision of the low level function `dirt`. This is the crux of the Naish and Barbour technique, which previously had only been implemented in a *toy* language (namely NUE–Prolog). With the aid of the internal interface library provided by the standard Hugs distribution, we were able to code `dirt` without any modifications to the underlying language implementation. This allows Buddha to be executed on any Hugs interpreter that supports the internal interface library.

The development of a debugger for a moderate subset of the Haskell language is the most tangible outcome of our work. However, our contribution to the field of debugging lazy functional languages extends further than the implemented system. We have taken the initial work of Naish and Barbour and shown that an extension of their technique is applicable to a complex modern lazy functional language. We have demonstrated how the two important syntactic constructs of where clauses and guarded equations can be supported in the transformation of programs, and identified the difficulties of transforming functions with higher–order arguments. Although such details are specific to the target language, we believe that they have not been adequately covered in the literature. We have also attempted to argue the correctness of the declarative debugging technique for lazy functional languages, which is an issue that we feel deserves more formal attention than it is currently given by researchers in the field.

The current version of Buddha consists of approximately 4000 lines of Haskell code, nearly 1000 of which are due to the parser and arity analyser of Hannan [3].[21] The remaining 3000 lines of code were written entirely by myself over a period of about six months. The underlying transformation algorithm used in Buddha is due to Naish and Barbour [11], which also incorporates a high–level specification for the impure function `dirt`. However, the implementation of the transformation and the extensions discussed in this paper are due to my myself, as is the implementation of `dirt`. The approach taken to

[21] Minor modifications were made to the parser to assist its integration into the Buddha implementation.

transforming higher–order arguments is derived from the work of Nilsson and Sparud [15], although the adaptation to the transformation employed in Buddha is due to myself.

7 Further work

We believe that Buddha provides a substantial proof of the declarative debugging concept for the Haskell language, however there are many modifications that can be made in order to make it a more useful system. In this section we discuss various extensions to the work described in this paper, which we hope to incorporate in future versions of Buddha.

7.1 Supporting the full Haskell syntax

In section 2.6 we mentioned that Haskell is a syntactically rich language, and that various syntactic constructs are not yet supported by the transformation employed in the debugger. The most notable of the unsupported syntactic constructs are lambda expressions, let expressions, case expressions and list comprehensions. Furthermore, we currently disallow functions defined by multiple equations with overlapping patterns if any of those equations uses guards[22]. Obviously, for Buddha to be regarded as a complete debugger for Haskell, such syntactic restrictions must be eliminated.

7.2 Improving the memory consumption of Buddha

There are two aspects of memory usage that may prohibit the use of Buddha as it is currently implemented. The first aspect of memory consumption involves the size of the EDT that is generated during the debugging session. The full EDT must contain a node for every application that was made during the evaluation of the program, however, due to lazy evaluation only those nodes which are actually traversed by the debugger will be constructed. If the number of nodes traversed in order to locate a topmost buggy node is small, then the memory consumed by the EDT may be significantly less than expected. Naish and Barbour [11] suggest that we can further reduce the size of the EDT by avoiding the creation of nodes for functions which are trusted to be correct, for example, functions included from standard libraries. The second aspect of memory consumption is more severe than the first. During the execution of the original program, intermediate data structures may be created which are needed only temporarily. Garbage collection can free the memory used by such data structures after they are no longer needed, thus minimising the amount of memory used by the program at any instant. In the transformed version of the program, such garbage collection of intermediate data structures is prohibited because the EDT must maintain a reference to them in case they are needed to be displayed in a question to the oracle. Therefore, memory which would ordinarily be reclaimed by the garbage collector is kept live by the EDT, which introduces a *space leak* into the transformed program. The memory consumption of the debugger is thus proportional to the length of the program evaluation, and is at least as expensive as executing the original program without garbage collection.

Naish and Barbour [11] propose to solve the space leak introduced by the transformed program by initially only creating the top few levels of the EDT. The lower levels of computation are provided by calls to the original code of the program, thus avoiding the generation of a complete EDT for the execution. When the leaves of the partial EDT are encountered, the code for the sub–computation at that point is called to further (partially) expand the EDT at the leaf. The main cost of this solution is that sub–computations may be repeatedly (partially) evaluated. If those sub–computations are expensive, the re–evaluation of them could dramatically slow the progress of the debugger.

It is proposed that the function `dirt` can be used to drive the computation to reconstruct the sub–tree at a leaf of the partial EDT. Consider a hypothetical node at the fringe of a partial EDT: by observing

[22] The reader is directed to section 4.1.4 for a discussion of this restriction.

the representation of the result of the application at the node, we can determine the extent to which the application was evaluated (`dirt` already provides us with this information). The sub–tree at the node can be expanded by forcing the re–evaluation of the application to the extent that the result of the re–evaluation is equivalent to the representation of the original evaluation at the node. For the re–computation to be forced to the correct extent, we must be able to manipulate the reduction mechanism used by the underlying language implementation. Within Hugs this could be achieved by stepping through the reduction loop performed by the internal function `eval()`, and continually comparing the state of reduction of the re–evaluated application against the representation of its original state of reduction in the EDT. We are confident that this can be achieved on top of the current version of `dirt`, however, we are currently uncertain how to ensure efficiency and safety of the re–evaluation process.

7.3 Monadic style programs and I/O

Monadic programming poses a problem for the declarative debugging technique. Sparud [20] proposes that declarative debugging, in the style discussed in this paper, is incompatible with monadic programming because the use of monads builds a new language on top of the lazy functional language. Certain constructs that are hidden in the new *monadic* language (such as a representation of state) will be exposed by the debugger. Determining the correctness of the hidden constructs may be difficult or impossible for the oracle.

In Haskell, I/O is performed using the I/O monad. As we demonstrated in section 4.3, the error diagnosis stage of the debugger uses the I/O monad to interact with the oracle. The use of monadic I/O in programs that are being debugged causes an additional problem for the debugging technique, on top of the difficulties mentioned above. The I/O of the program being debugged must be threaded within the I/O performed by the debugger itself. Currently, there does not appear to be a simple solution for this problem. To avoid the difficulties of monadic I/O, we assume that the debugger will only be applied to the sub–parts of programs that do not perform I/O operations.

Further research into debugging monadic programs is required. We believe that a solution to the general problem of debugging monadic style programming will assist the special case of debugging programs that use the I/O monad.

7.4 Using type analysis in the transformation of programs

The difficulty of correctly transforming higher–order functions was discussed in section 4.1.6. Currently we rely on the arity of functions to transform curried function definitions and curried applications. In cases where type signatures are not available, the arity information may be incorrect, leading to errors in the transformed code. More exact type information about expressions in the original program would greatly assist the transformation of higher–order functions. In future versions of Buddha we hope to make use of type inference to supply the required type information, which may be obtainable from the type analysis stage of the underlying language implementation.

7.5 Improving the interaction with the oracle, and alternative uses of the EDT

The current interaction between the debugger and the oracle is rather naive:

- Large representations of arguments and results of functions are printed in full, which can be both annoying and confusing for a human oracle.
- Where functions are repeatedly applied to the same arguments in the original program, the debugger will repeatedly ask (redundant) questions about the correctness of the same application.

- Questions are asked about the application of functions which are previously known to be correct on all (or a subset of) their inputs.
- The oracle must know the intended meaning of all applications made in a program, and so must always be able to answer either *yes* or *no* to the questions presented to it by the debugger.

In future versions of Buddha we hope to remedy these failings and provide a more flexible and usable interface to the debugger. In particular we hope to provide the following improvements: a browsing mechanism to allow the incremental presentation of large data structures; a mechanism for remembering the questions that have been previously asked by the debugger so that they are not repeated; the ability to make assertions about functions which are considered correct on all (or a subset of) their inputs to reduce or eliminate questions asked about their application; and the provision of a *don't know* response which allows the oracle to indicate that it is uncertain about the correctness of an application, allowing the debugging session to proceed further in the hope that more certain errors may be located.

Previously in section 3.3, we made the observation that the EDT generated by the transformed program may be useful for tasks other than debugging programs. In particular, rather than determine the correctness of applications, users may simply want to browse through the EDT to view the applications made in an execution of the program. This may be done to obtain a greater understanding of the program, or perhaps to gain a high–level profile of its execution. We could provide a browsing mode in the debugger which simply allows the user to interactively traverse the EDT in any order they choose, or perhaps a graphical representation of the EDT, which could also be used in combination with the error diagnosis algorithm.[23]

8 Conclusion

Declarative debugging is a proven method for developing systems that assist the location of bugs in programs written in relational languages. The adaption of declarative debugging techniques to functional languages is an active area of research. Laziness, referential transparency and strong static type systems are features which make the implementation of declarative debuggers difficult in pure lazy functional languages.

We have demonstrated the implementation of Buddha, a declarative debugger for the programming language Haskell. Our implementation demonstrates the suitability of the Naish and Barbour declarative debugging technique, and shows how it can be extended to incorporate typical syntactic constructs of modern functional languages. We have illustrated how the required low–level primitives of the technique can be implemented within the Hugs environment, and we have increased the ability of the technique to support higher–order functions.

As a prototype system, Buddha provides a sound basis upon which a more complete debugging system for Haskell may be built. Complete support for higher–order programming and improved memory consumption are two main extensions described in this report that will be included in future versions of the debugger.

Acknowledgements

I would like to thank Lee Naish for supervising this project. Without his wisdom, support and encouragement none of the work presented in this paper would have been possible. I would like to thank Harald Søndergaard for introducing me to the Haskell programming language, and for encouraging me to pursue a project in functional programming languages. I would like to thank my parents, Jan and Brian, for supporting me through my undergraduate career, and providing me with endless inspiration and guidance.

[23] Techniques for applying graphical interfaces to declarative debuggers for lazy functional languages are described in [22] and [20].

The implementation of Buddha was greatly assisted by the valuable advice of Alistair Reid regarding the implementation of Hugs, and by the Haskell parser and arity analyser written by Miles Hannan.

Bibliography

[1] Kishon A and P. Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–574, 1995.

[2] C. Hall and J. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, volume 20, pages 60–68, Washington, USA, 1985.

[3] M. Hannan. Debugging systems for lazy functional languages. Master's thesis, Schools of Electrical Engineering and Computer Science and Engineering, The University of New South Wales, 1995.

[4] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to Haskell, version 1.4. http://www.haskell.org/tutorial/index.html.

[5] J. Hughes. Why functional programming matters. *The Computer Journal*, 32, 1989.

[6] M. Jones. The implementation of the Gofer functional programming system. Technical report, Yale University, Department of Computer Science, Connecticut, USA, 1994.

[7] J. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, New York, 1984.

[8] J. Lloyd. Declarative error analysis. *New Generation Computing*, 5(2):133–154, 1987.

[9] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[10] L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In Graham Forsyth and Moonis Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers*, volume 2, pages 91–99, Melbourne, 1995. DSTO General Document 51.

[11] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.

[12] H. Nilsson. A declarative approach to debugging for lazy functional languages. Master's thesis, Linköping University, 1994.

[13] H. Nilsson and P. Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In P. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 117–134, Linköping, Sweden, 1993.

[14] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[15] H. Nilsson and J. Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical report, Department of Computer and Information Science, Linköping University, 1996.

[16] J. Peterson, K. Hammond, L. Augustsson, B. Boutel W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 1.4 Language Report. http://haskell.systemsz.cs.yale.edu/onlinereport/.

[17] A. Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.

[18] E. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.

[19] J. Sparud. Towards a Haskell debugger. Technical report, Chalmers University of Technology, Department of Computer Science, Göteborg, Sweden, 1995.

[20] J. Sparud. A transformational approach to debugging lazy functional programs. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 1996.

[21] J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional languages. In M. Ducassé, editor, *Proceedings of AADEBUG'95*, Saint-Malo, France, 1995.

[22] R. Westman and P. Fritzson. Graphical user interfaces for algorithmic debugging. In P. Fritzson, editor, *Lecture Notes in Computer Science*, volume 749, pages 273–286. Springer, May 1993.

A Example transformed guarded equation

Below is the transformed version of the `insert` function from figure 13, along with the code necessary to support its operation.

```
insert x (y:ys)
   | isaMatch matched = (v0, t0)
   where
   v0 = getValue matched
   children = getChildren matched
   (gval0, gtree0) = gt x y
   eval0 = y : eval1
   (eval1, etree1) = insert x ys
   gval2 = x < y
   eval2 = x : y : ys
   gval3 = otherwise
   eval3 = y : ys
   matched = guards [(gval0, [gtree0], eval0, [etree1]), \
                     (gval2, [], eval2, []), (gval3, [], eval3, [])]
   t0 = Node "insert" [dirt x, dirt (y : ys)] (dirt v0) children "insert ..."

-- these functions are used to support the transformed equation

isaMAtch :: Maybe a -> Bool
isaMatch Nothing  = False
isaMatch (Just _) = True

getChildren :: Maybe (a, [Tree]) -> [Tree]
getChildren Nothing = []
getChildren (Just (_, ts)) = ts

getValue ::  Maybe (a, [Tree]) -> a
getValue (Just (val, _)) = val
```

B Transformation and debugging session for a buggy version of take

Below is a buggy version of the function `take`. The bug is due to the definition of the function `isZero`, which should read "`isZero x = x == 0`".

```
take n as
   | as == [] = []
   | isZero n = []
   | otherwise = hd as : (take (n-1) (tl as))

hd (a:as) = a
tl (a:as) = as

isZero x = x == 1
```

Using the transformation described in this paper we get the following code:

```
take n as
   | isaMatch matched = (v0, t0)
   where
   v0 = getValue matched
   children = getChildren matched
   gval10 = as == []
   eval10 = []
   gval20 = gval21
   (gval21, gtree21) = isZero n
   eval20 = []
   gval30 = otherwise
   eval30 = eval31 : eval33
   (eval31, etree31) = hd as
   (eval32, etree32) = tl as
   (eval33, etree33) = take (n - 1) eval32
   matched = guards [(gval10, [], eval10, []), (gval20, [gtree21], eval20, []),\
                     (gval30, [], eval30, [etree31, etree32, etree33])]
   t0 = Node "take" [dirt n, dirt as] (dirt v0) children "take ..."

hd (a:as)
   = (v0, t0)
   where
   v0 = a
   t0 = Node "hd" [dirt (a : as)] (dirt v0) [] "hd ..."

tl (a:as)
   = (v0, t0)
   where
   v0 = as
   t0 = Node "tl" [dirt (a : as)] (dirt v0) [] "tl ..."
```

```
isZero x
   = (v0, t0)
   where
   v0 = x == 1
   t0 = Node "isZero" [dirt x] (dirt v0) [] "isZero ..."
```

An example debugging session is given below where the (transformed) function `take` is applied to an infinite list of numbers.

```
buddha (take 3 [1..])
final result = [1, 2]

take 3 [1, 2, 3, ...?] = [1, 2]
Correct - Y/N/Q? n

hd [1, 2, 3, ...?] = 1
Correct - Y/N/Q? y

tl [1, 2, 3, ...?] = [2, 3, ...?]
Correct - Y/N/Q? y

take 2 [2, 3, ...?] = [2]
Correct - Y/N/Q? n

hd [2, 3, ...?] = 2
Correct - Y/N/Q? y

tl [2, 3, ...?] = [3, ...?]
Correct - Y/N/Q? y

take 1 [3, ...?] = []
Correct - Y/N/Q? n

isZero 1 = True
Correct - Y/N/Q? n


The erroneous node is:
isZero 1 = True

The erroneous equation is:
isZero x = x == 1
```