Display of Functional Values for Debugging

Thomas Davie and Olaf Chitil

Computing Laboratory, University of Kent, CT2 7NF, UK
{tatd2, o.chitil}@kent.ac.uk,
Web: http://www.cs.kent.ac.uk/people/{rpg/tatd2, staff/oc}

Abstract. Functional values are used naturally in higher order functional programs, as they are commonly passed around or returned by other function. As such any debugger for these languages must be capable of conveying information about functional values to the user. Current debuggers are limited to displaying the name of the function, or a partial application. In this paper we study two alternative methods of displaying functional values, and their effect on a popular debugging method (Algorithmic Debugging).

1 Introduction and Motivation

Many debugging techniques for higher order functional languages are based on displaying equations to the user. Algorithmic debugging [8–10], making observations [7, 5], and redex trails [1, 11] all rely on displaying information as an equation. As such it is important that we are able to display these equations in a clear and easy to understand way.

Values have a clear and unambiguous meaning, and are usually smaller than the expressions from which they were computed. Hence, in order to keep the display of equations clear most debuggers try to display values as much as possible rather than equations. The simplest equations a debugger can display therefore contain only one expression, limiting everything else to values:

$$f v_{a_1}..v_{a_n} = v_r$$

That is some function, applied to a number of arguments (as values), reducing to a result as a value.

In the context of first order languages, this is perhaps the end of the story, in that we have a clear way of displaying all equations. However, when higher order functions are involved, displaying values becomes much harder. A debugger for higher order languages must be able to clearly and unambiguously display a functional value. Debuggers up until now have simply displayed the name of the function, however we will argue that this is neither clear, nor unambiguous. Given the program shown in figure 1, we might reasonably present this equation:

```
tMap g (Branch (Leaf 5) (Leaf 2))
  = Branch (Leaf False) (Leaf True)
```

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)

main = tMap g (Branch (Leaf 5) (Leaf 2))

tMap g (Leaf x) = Leaf (g x)
tMap g (Branch l r) = Branch (tMap g l) (tMap g r)

-- G should test if it’s input is 5.
g = (==2)
```

Fig. 1. A sample buggy program

The display of the function name g is ambiguous. The user intends the program to perform a specific purpose and has a concept of how the program should achieve this purpose. The user has an intended semantics for the function g. The user’s intended semantics do not necessarily agree however with the semantics of the function written in the program. Because of this, we can ask, is g referring to the function the programmer’s intended semantics, or the actual semantics of g?

Further to this ambiguity, the question may become extremely unclear. When debugging the program shown in Figure 2, the user is presented with the equation

```
(recogniseOpen
  (acceptClose (acceptEmpty <|> recogniseOpen ...)
           <|> recogniseOpen
                  (acceptEmpty <|> recogniseOpen ...))) "()())"
  = False
```

this equation is so complex that the user has trouble taking its meaning in. This equation is taken from a debugging session for a simple recogniser, when applied to a larger scale program this problem becomes many times worse. In this paper we present a new display mechanism that helps to disambiguate whether the equation refers to the intended or actual semantics, and makes equation like the one shown above much more simple.

2 Displaying Functional Values

When we display an equation we must be able to easily differentiate between two types of functional values. We must be able to display both a value representation for the user’s intended semantics, and also a representation for the actual semantics. We propose to maintain the current display of function names when representing the intended semantics of a function. This requires us to provide a representation for a function’s actual semantics. We propose two new methods of displaying a function. First by displaying the result of applying the function

```
(<|>) :: (String -> Bool) -> (String -> Bool) -> String -> Bool
(<|>) f g s = f s || g s

recognise :: String -> Bool
recognise = recogniseOpen (acceptEmpty <|> recognise)

acceptEmpty :: String -> Bool
acceptEmpty [] = True
acceptEmpty _ = False

acceptClose :: (String -> Bool) -> String -> Bool
acceptClose c (')':xs) = c xs
acceptClose c _ = False

recogniseOpen :: (String -> Bool) -> String -> Bool
recogniseOpen c ('(':xs) = ((acceptClose c) <|> (recogniseInner c)) xs
recogniseOpen c xs = False

recogniseInner :: (String -> Bool) -> String -> Bool
recogniseInner c = recogniseOpen ((acceptClose c) <|> (recogniseOpen c))
```

Note, the bug is in the last line, `recogniseOpen c` should be `recogniseInner c`.

Fig. 2. A buggy recogniser

to different input values. Secondly, we attempt to reproduce the code used to define the function.

2.1 Functions as Finite Mappings

Functions can be represented as infinite mappings from input to result. However for obvious computational and space reasons, displaying an infinite mapping is inappropriate. This is especially so, when the user is only likely to be interested in what happens with very specific inputs. Because of this, we consider displaying functions as finite mappings.

This choice brings up the question of exactly what mappings to display. It makes sense to only display the effects of the function in cases that were actually executed. We can narrow this further, to the user only being interested in applications that were caused by the computation we are asking about.

```
main = myMap g [1,2,3] ++ myMap g [4,5]

myMap _ [] = []
myMap g (x:xs) = g x : myMap g xs

g = (==2)
```

If we trace the above program, we might show the equation `myMap g [1,2,3] = [False,True,False]`. This equation considers the intended semantics of `g`. If we want to display an equation showing the actual semantics of `g` we must instead display `myMap {1 -> False,2 -> True,3 -> False} [1,2,3] = [False,True,False]`. Note that we do not display the argument/result pair for `g 4` or `g 5`. This is because they were not involved in the computation of `myMap g [1,2,3]`.

If the mapping takes up significant amounts of space, we might want to rearange the layout of this equation:

```
myMap {f1} [1,2,3] = [False,True,False]
  where
    {f1} 1 = False
    {f1} 2 = True
    {f1} 3 = False
```

2.2 Displaying Function Code

A second method of displaying the actual semantics of a function would be to display the code used on the right hand side of the function definition. In the example used above we would display the equation `myMap (==2) [1,2,3] = [False,True,False]`.

```
main = myMap g [[1,2,3],[4,5]]

myMap _ [] = []
myMap g (x:xs) = g x : f g xs

g [] = False
g (x:xs) = x == 2 || g xs
```

This display has a problem however. In the above example, we cannot display `g` as the right hand side of it's definition for two reasons. First, we have no way of representing the two lines of pattern match as one line. Secondly, `g` is recursive, we cannot display it infinitely. As the tracing tool is run after the program has already terminated, we can guarantee that there are a finite number of applications, and thus display all of them, but this would use up far too much screen space. Thus we make a similar refinement to the one made earlier, and we display the semantics required after the equation:

```
myMap {f1} [[1,2,3],[4,5]] = [True,False]
  where
    {f1} []     = False
    {f1} (x:xs) = x == 2 || {f1} xs
```

The problem of recursive functions is a specialised case of a more general problem. If the function we are attempting to display, contains calls to other functions, how are they displayed? We have two options: First, we may expand all further function definitions, asking about the actual semantics for these functions too; or secondly, we may leave them as function names, and ask about their intended semantics. If we choose to expand all the function definitions, we may end up displaying a large proportion of the program's code, and the equation becoming too long for the user to easily take in. The choice between these two displays is left as an implementation detail.

Local Function Definitions When functions are defined locally, we have another problem when displaying the function code. The original function code may have contained variables taken from the parent scope. These are not available to us except as their bound values. Thus, our only option for displaying such variables is to display them as their runtime value.

```
main = f 3

f x = myMap g [1,2,3]
      where
        g y = x * y

myMap _ [] = []
myMap g (x:xs) = g x : f g xs
```

In the above example we might want to display an equation about the application `myMap g [1,2,3]` with the actual semantics of `g`. This equation would look like this:

```
myMap {f1} [1,2,3]
where
  {f1} x = 2 * x
```

2.3 Display of Lambda Expressions

Both methods of displaying functional values described here are appropriate for use when displaying lambda expressions as demonstrated by these two example questions:

```
myMap {\1} [1,2,3] = [False,True,False]
  where
    {\1} 1 -> False
    {\1} 2 -> True
    {\1} 3 -> False
```

or:

```
myMap (\x -> x==2) [1,2,3] = [False,True,False]
```

```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

Fig. 3. Buggy insertion sort program

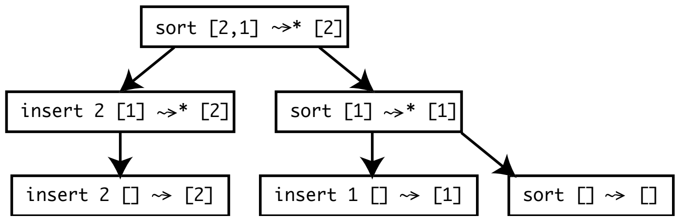

Fig. 4. EDT of a computation of the buggy insertion sort program in Figure 3

3 Implications of Proposals on Algorithmic Debugging

Algorithmic debugging [8–10] is a semi-automatic method for locating bugs in programs that is based on the compositional representation of a computation as a tree. Many different tree layouts are possible. Figure 4 shows one such layout (the *Evaluation Dependancy Tree* [9]) for a computation of the Haskell program in Figure 3. Each node of the tree is labelled with a computation, that is, a big-step reduction of a redex to its value. The computation of a node is fully determined by the computations of all its children and the use of a small slice[1] of the program.

For example, the computation `sort [2,1]` $\rightsquigarrow^*$ `[2]` is composed of the computations `insert 2 [1]` $\rightsquigarrow^*$ `[2]` and `sort [1]` $\rightsquigarrow^*$ `[1]` and the single-step reduction `sort [2,1]` $\rightsquigarrow$ `insert 2 (sort [1])`:

`sort [2,1]` $\rightsquigarrow$ `insert 2 (sort [1])` $\rightsquigarrow^*$ `insert 2 [1]` $\rightsquigarrow^*$ `[2]`

The single-step reduction uses an instance of the program slice

```
sort (x:xs) = insert x (sort xs)
```

Another example is the computation `insert 1 []` $\rightsquigarrow$ `[1]`. The node has no children. The single step reduction only uses an instance of the program slice

```
insert x [] = [x]
```

[1] A program slice is made up of “parts of a program that affect [...] some point of interest”[12]

An algorithmic debugger asks the user whether computations of tree nodes agree with the intentions of the user. If the computation of a node agrees with the user's intentions, then the node is called *correct*, otherwise it is called *erroneous*. A node is *buggy*, if it is erroneous and all its children are correct. Because of the compositional definition of the EDT, the program slice associated with the buggy node contains a bug. That buggy program slice has to be modified to make the computation of the buggy node agree with the user's intentions. Algorithmic debugging is given some theoretical foundations by Caballeroe et al. and Chitil et al. [2, 3].

An example session with an algorithmic debugger:

```
sort [2,1] = [2]     The system asks about the top level node.
> no                 The user says it is incorrect.
sort [1] = [1]       The system asks about an erroneous
> yes                node's child.
insert 2 [1] = [2]
> no                 One of the children is erroneous.
insert 2 [] = [2]    The system investigates that node's children.
> yes                All those children are correct, so the
                     node is buggy and its buggy slice is shown.

Bug identified:
  insert x (y:ys)
    | x < y     = ...
    | otherwise = insert x ys
```

As shown in Section 1 the questions algorithmic debugging ask can become complex, and hard for the user to disambiguate. We therefore consider using the functional value representations explained above. This however changes the semantics of the question.

Changing the semantics of the question obviously has an effect on what the users answer means for the debugger. Thus, we must rearrange the EDT appropriately when we decide to display a function using one of these methods.

Using the example from earlier, if we ask the question `myMap {1 -> False,2 -> True,3 -> False} [1,2,3] = [False,True,False]?` rather than `myMap g [1,2,3] = [False,True,False]?`, we are no longer asking anything about the function `g`. Thus, applications of `g` should not appear below this node in the EDT. Instead, the applications are moved up to be siblings of the application in which the function was called.

The EDT must be rearranged in this way not only when asking questions where functional values are represented as finite mappings, but for any question in which the debugger asks about the actual semantics of a piece of code. We must simply apply this transformation for each application in which we consider the actual semantics. This means when using the finite mapping display, we must recursively apply the EDT transformation for every application within those being displayed. Similarly, this process must be carried out when displaying function code. However when displaying function code, we have the option to

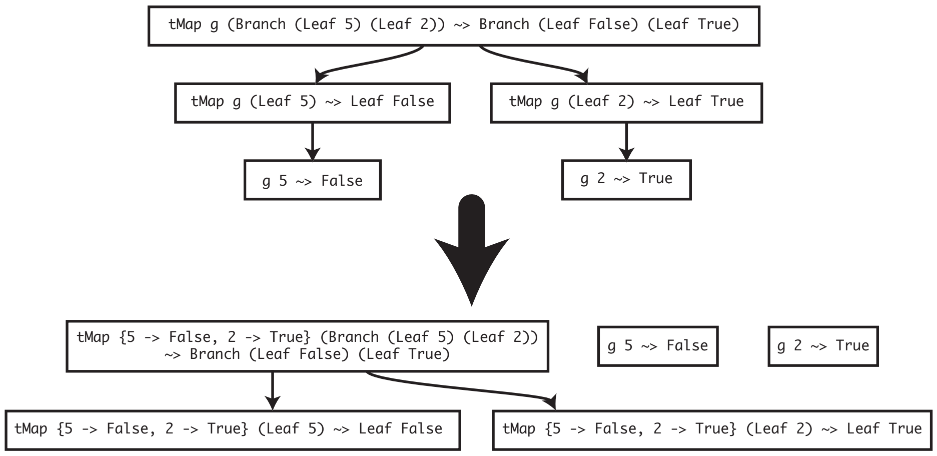
tMap g (Branch (Leaf 5) (Leaf 2)) ~> Branch (Leaf False) (Leaf True)
tMap g (Leaf 5) ~> Leaf False
tMap g (Leaf 2) ~> Leaf True
g 5 ~> False
g 2 ~> True
tMap {5 -> False, 2 -> True} (Branch (Leaf 5) (Leaf 2)) ~> Branch (Leaf False) (Leaf True)
g 5 ~> False
g 2 ~> True
tMap {5 -> False, 2 -> True} (Leaf 5) ~> Leaf False
tMap {5 -> False, 2 -> True} (Leaf 2) ~> Leaf True

Fig. 5. EDT rearrangement for the program shown in Figure 1 and function `g`

use named functions (and thus refer to the intended semantics of a function) within the functional value. This means that we can stop recursively applying the transformation at this point.

Figure 5 shows how the rearrangement is applied to the example we examined earlier. The function `g` is rewritten to represent the function's actual semantics. Because the applications of `tMap` no longer depend on evaluating `g`, but instead its actual semantics, we remove `g` from beneath those applications. The applications of `g` are reinserted next to `tMap` in the tree. Note that in a debugging session, the user must now answer distinct questions about `tMap` and `g`. This clarifies what the questions are about, and makes them easier to answer.

This can be applied repeatedly where we have higher order functions with higher order functions passed as arguments or returned. We must simply apply the transformation in stages and the position of the nodes is unfolded.

4 Comparison of Methods

We now have three different methods of displaying functions. As the function's name, as a finite mapping, or as it's definition.

Displaying function names is the only method suggested for use when asking about the intended semantics of a function. This is because the other two methods display exactly what did happen. Because the displays show exactly what did happen with no other point of reference, it is illogical to ask the user what they intended to happen. Asking this question would be akin to asking the user "what did you mean 6 to be?" Without any more context, the question is completely meaningless.

Displaying the function name is obviously inappropriate when asking questions about lambda expressions as there is no function name to display. However as discussed earlier, both displaying mappings, and displaying code, are able to deal with this situation.

When asked to display a large or complex function, displaying the code could become inapropriate, as we may end up displaying hundreds of lines of code, that the user must read through in order to answer one question. Similarly, if we apply a function many many times, it may be inappropriate to ask questions displaying finite mappings, as the number of mappings shown could become huge. If we only display a small selection of mappings it is likely that we will not display the mappings relevant to the question.

Table 1 provides a summary of this comparison.

Display Method	Function Name	Finite Map	Declaring Code
Semantics asked about	Intended	Actual	Actual
Capable of displaying labmda expressions	No	Yes	Yes
Capable of displaying large functions	Yes	Yes	No
Quick to read for small functions	Yes	No	Yes

Table 1. Function display methods

5 Implementation

An implementation of the above methods will be based on the Haskell tracer Hat [13, 6]. Hat provides a framework on which our debugger is built.

5.1 Hat

Tracing a computation with Hat consists of two phases: trace generation and trace viewing. First, a special version of the program runs. In addition to its normal input/output behaviour it writes a trace into a file. Secondly, after the program has terminated, the programmer studies the trace with a collection of viewing tools.

Importantly for us this means that we must only implement another view that uses the generated trace. An EDT can be reconstructed from the trace [13, 4]. This has already been implemented in the debuggers hat-delta and hat-detect. An extension of the debugger hat-delta will include the ability to use our new representations.

5.2 Finite Mappings

In order to find the mappings to display we must search the trace below the application the tracer is asking about, and retrieve all applications of the function in question. These mappings are then presented in the form shown above. The search for applications can be performed efficiently, by only searching certain parts of the trace as discussed in Section 5.4.

5.3 Declaring Code

In order to reconstruct the right hand side of a function definition, all we must do is create an expression based on the result pointer of the application of the function. However, as variable names are not stored in the trace, it is not possible to resurrect the names of the variables used. This means that when displaying a function, the variable names used probably will not be the same as those in the original code. Worse than this, the left hand side is in no way stored in the trace, and so if pattern matching is performed, we must attempt to resurrect the patterns from the trace. Where the pointers in the result NodeExp point **into** rather than to the arguments of the application, we must create a pattern match. This is not guaranteed to be syntactically identical to the original, but will be semantically identical.

This method at first glance only appears to require finding one application of the lambda. However, when branching code is considered, one application must be found for each code path in order to recover code for each part of the branch. If we use this code:

```
main = myMap (\x -> if x > 10 then x - 10 else x) [1,5,11]

myMap _ [] = []
myMap g (x:xs) = g x : myMap g xs
```

We would like to ask this question:

```
myMap (\x -> if x > 10 then x - 10 else x) [1,5,11] = [1,5,1]
```

Asking this question requires finding one application where x is greater than 10 to recreate the then branch, and one application where x is less than 10 to reconstruct the else branch. This search can be optimised using the method described in 5.4.

5.4 Performing Searches

In order to find applications of the function we must search the result of the application we are asking questions about. However we do not need to follow all result pointers. This is because the function cannot be called unless it is in scope, so when we follow a result pointer (i.e. enter another function call) we must do a scope check. The function is still in scope in two situations – first when it is passed as an argument, and secondly, when the new call is to a local function definition. We hope that in practice this restricts the search to a relatively small area of trace.

6 Summary and Future Work

We have presented two methods of representing the actual semantics of a functional value in an equational representation of an application. This disambiguates, and clarifies complex equations. Changing the question semantics has an effect on the layout of the EDT.

We hope to be able to prove that our changes to the EDT combined with the changes to the question semantics do not change the result of an algorithmic debugging session. This leaves us with three major areas of research for the future:

– An implementation, and evaluation of this methods effect on the length of the debugging session, and on how easy it is to answer the questions.
– A proof that algorithmic debugging is unaffected by this method.
– An investigation into this methods effects on other debugging techniques.

At least at first sight, it appears that this method makes questions significantly easier to answer. If we consider our earlier example from Figure 2. The debugger previously asked the question

```
(recogniseOpen
  (acceptClose (acceptEmpty <|> recogniseOpen ...)
           <|> recogniseOpen
                 (acceptEmpty <|> recogniseOpen ...))) "()())"
  = False?
```

Using our method, the user could chose to answer a much more simple question:

```
(recogniseOpen {")())" -> False} "()())" = False?
```

We hope that this method will help in debugging state based monadic code. Current algorithmic debuggers ask complex questions about a series bind operators interspersed with lambda expressions and user functions when used on stateful monadic code. We hope that with some refinement we will be able to debug stateful code in a much easier to follow manner.

References

1. Simon P. Booth and Simon B. Jones. Walk backwards to happiness – debugging by time travel. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 171–183, 1997.

2. Rafael Caballero, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In *FLOPS*, pages 170–184, 2001.
3. Olaf Chitil and Yong Lou. Proving the correctness of declarative debugging for functional programs. In *Draft Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP 2006*, pages 41–54, Nottingham, UK, April 2006.
4. Olaf Chitil and Yong Luo. Towards a theory of tracing for functional programs based on graph rewriting. In Ian Mackie, editor, *Draft Proceedings of the 3rd International Workshop on Term Graph Rewriting, Termgraph 2006*, page 10, April 2006.
5. Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, LNCS 2011, pages 176–193, Aachen, Germany, March 2001. Springer.
6. Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, number 2638 in LNCS, pages 59–99, Oxford, August 2003.
7. Andy Gill. Debugging haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2000.
8. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
9. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998.
10. Ehud Yehuda Shapiro. *Algorithmic program debugging.* MIT Press, 1982.
11. Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *PLILP '97: Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 291–308, London, UK, 1997. Springer-Verlag.
12. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
13. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).