Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 177 (2007) 107–122
www.elsevier.com/locate/entcs

The Interactive Curry Observation Debugger iCODE [1]

Parissa H. Sadeghi and Frank Huch

{phsa,fhu}@informatik.uni-kiel.de
Institute of Computer Science
University of Kiel
Olshausenstr. 40, 24098 Kiel, Germany

Abstract

Debugging by observing the evaluation of expressions and functions is a useful approach for finding bugs in lazy functional and functional logic programs. However, adding and removing observation annotations to a program is an effort making the use of this debugging technique in practice uncomfortable. Having tool support for managing observations is desirable. We developed a tool that provides this ability for programmers. Without annotating expressions in a program, the evaluation of functions, data structures and arbitrary subexpressions can be observed by selecting them from a tree-structure representing the whole program. Furthermore, the tool provides a step by step performing of observations where each observation is shown in a separated viewer. Beside searching bugs, the tool can be used to assist beginners in learning the non-deterministic behavior of lazy functional logic programs. To find a surrounding area that contains the failure, the tool can furthermore show the executed part of the program by marking the expressions that are activated during program execution.

Keywords: Curry, debugging, functional logic languages, observation, tool

1 Introduction

One of the original problems of programming is locating bugs in programs. This is especially true for declarative programming languages which make it difficult to predict the evaluation order. There is a variety of methods for locating bugs in a program (related approaches are discussed in Section 7). Although the tools implementing these methods do usually not remove the bugs, they are often called *debuggers*. The Curry Object Observation SYstem (*COOSY*) is one of them [2]. This debugger is a tool to observe data structures and functions of a program written in the declarative multi-paradigm language Curry [9]. COOSY extends Gill's idea [1] of observing expressions in lazy functional programs to the lazy functional logic setting.

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2007.01.007

In COOSY, a programmer annotates expressions in her/his program to observe the evaluation of the program execution in the desired position. However, in practice adding and removing observations to/from the program is some kind of effort which keeps people away from using tools like COOSY. Instead, people prefer printing data structures using `trace`, which influences the evaluation behavior and does not provide the power of observing functions. Another problem exists for beginners still fighting the type system, currying, and lazy evaluation: understanding COOSY would mean another burden to them (especially since Curry does not provide type classes and observations have to be annotated with special observers related to the type of the observed values).

Our solution to this problem is a convenient graphical interactive tool called *iCODE* (Interactive Curry Observation DEbugger) which supports debugging by observations with the following features:

- marking expressions representing the executed part of the program to locate a surrounding area which contains the failure.
- representing the whole program as a tree-structure with the selection ability on each expression or function in a graphical user interface.
- showing the observation steps of each desired part of the program in separated viewers, with the backward and forward stepping ability on the steps.
- automatic generation of observers for all user defined data types.

All source code modifications are performed automatically before executing the code. A step by step presentation of the evaluation of selected data structures or functions in a separate window can help beginners to understand the lazy execution semantics of Curry. This presentation provides another advantage: all views are implemented by independent system processes which may even be distributed on multiple computers to avoid a slow-down or shortage of memory of observed executions. Finally, observation sessions can be saved and reloaded when debugging is necessary again. The tool is implemented in Curry using libraries for GUI programming [10,11] and meta programming which are available for the Curry system PAKCS [6].

2 A Review on Observations in COOSY

COOSY [2] is based on the idea to observe data structures and functions during program execution. The programmer can import the module `Observe` into her/his program and annotate expressions with applications of the function `observe`:

```
observe :: Observer a -> String -> a -> a
```

The function `observe` behaves as an identity on its third argument. Additionally, it generates, as a hidden side effect, a trail file representing the evaluated part of the observed data. To distinguish different observations from each other, `observe` takes a label as its second argument. After program termination (including run-time errors and aborts), all observations are presented to the user, with respect to their different labels. Finally, `observe` demands an observer as its first argument

which defines the special observation behavior for the type of the value `observe` is applied to. For each predefined type τ such an observer is defined as `o`τ. For example, for expressions of type `Int` the observer `oInt` should be used and for `[Int]` the observer `oList oInt`. Note, that observers for polymorphic type constructors (e.g., `[]`) are functions taking as many arguments as the type constructor.

The explicit annotation of the observer for each type is necessary, since Curry, in contrast to Haskell, does not provide type classes which hide these observers from the user in HOOD. However, there is also a benefit of these explicit annotations. It is possible to use different observers for the same type which allows selective masking of substructures in large observed data structures, e.g. by the predefined observer `oOpaque` [2] which presents every data structure by the symbol `#`.

As a small example, we consider a function which computes all sublists of a given list (here with elements of type `Int`):

```
sublists :: [Int] -> [[Int]]
sublists xs = let (ready,extend) = sublists' xs in ready++extend

sublists' :: [Int] -> ([[Int]],[[Int]])
sublists' [] = ([[]],[[]])
sublists' (x:xs) = let (ready,extend) = sublists' xs in
                     (ready++extend,[x]:map (x:) extend)
```

The idea is to distinguish lists which are already closed sublists and lists which may be extended with the actual list element `x`. Unfortunately, this program contains a little bug. `sublists [1,2,3]` yields:

```
[[],[],[3],[3],[2],[2,3],[2,3],[1],[1,2],[1,2,3],[1,2,3]]
```

Some elements occur twice in the result. To find this bug, we first observe the two results of `sublists'` in `sublists` and obtain:

```
sublists xs =
  let (ready,extend) = observe (oPair (oList (oList oInt))
                                      (oList (oList oInt)))
                               "result" (sublists' xs) in
     ready++extend

result
------
([[],[],[3],[3],[2],[2,3],[2,3]],[[1],[1,2],[1,2,3],[1,2,3]])
```

The bug seems to result from the first pair component because the replication appears here. Hence, we observe this component within the right-hand side of `sublists'` and obtain:

```
sublists' (x:xs) =
  let (ready,extend) = sublists' xs in
    (observe (oList (oList oInt)) "first component" (ready++extend),
     [x]:map (x:) extend)
```

```
first component
---------------
[[],[]]
[[],[],[3],[3]]
[[],[],[3],[3],[2],[2,3],[2,3]]
```

This observation still shows the bug, but does not help to locate it, since we cannot distinguish the values of `ready` and `extend`. A better observation point would have been the result of `sublists'` during recursion. Hence, we again change the source code and add an observer to another place:

```
sublists' (x:xs) =
  let (ready,extend) =
        observe (oPair (oList (oList oInt)) (oList (oList oInt)))
                "sublists'" (sublists' xs) in
    (ready++extend, [x]:map (x:) extend)

sublists'
---------
([[]],[[]])
([[],[]],[[3],[3]])
([[],[],[3],[3]],[[2],[2,3],[2,3]])
```

In the second line of this observation, we see that the bug is located in the second pair component. Thinking about this observation, we see that the expression `[x]:map (x:)  extend` adds the list `[3]` twice, since the empty list is contained in `extend`. The bug is located in the base case which should be corrected to:

```
sublists' [] = ([[]],[])
```

Observing data structures can help finding a bug. However, a program consists functions and it is more interesting to observe functions which COOSY provides as well. Observers for functions can be constructed by means of the right associative operator:

```
(~>) :: Observer a -> Observer b -> Observer (a -> b)
```

In our example, we could have used a functional observer to observe the recursive calls of `sublists'`:

```
sublists' (x:xs) =
  let (ready,extend) =
        observe (oList oInt ~> oPair (oList (oList oInt))
                                     (oList (oList oInt)))
                "sublists'" sublists' xs in
    (ready++extend, [x]:map (x:) extend)

sublists'
```

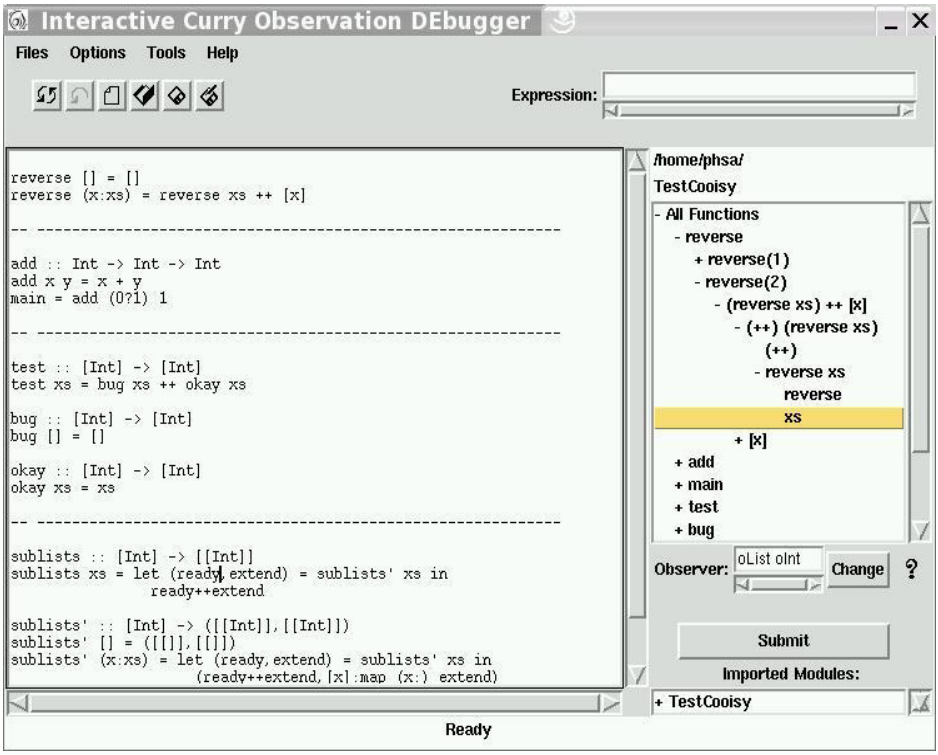


Fig. 1. A tree of all program expressions in the main window

```
---------
[] -> ([[]],[[]])
[3] -> ([[],[]],[[3],[3]])
[2,3] -> ([[],[],[3],[3]],[[2],[2,3],[2,3]])
```

In this observation, it is also possible to detect the bug and in practice it is often easier to find bugs by observing functions. However, in larger programs it is still an iteration of adding and removing observers to find the location of a bug, similar to the debugging session sketch for the `sublists` example. A tool which supports the programmer in adding and removing observers is desired.

3 Tree Presentation of a Program

iCODE is a small portable Curry program that provides a graphical interface debugger for the Curry programs. It uses the meta-programming library of Curry [6] and presents the whole program as a tree which contains functions, data structures and all defined subexpressions of the program which may be necessary to be observed for finding bugs. By means of Curry's Tcl/Tk library [10,11] we provide convenient access to this tree. By default all functions of a program (module) are available. On selection of a corresponding rule the user can access the right-hand side of a function definition and descend into the tree representing all its subexpressions. On the other hand, for a concise presentation, initially local definitions and

all subexpressions are hidden and can be opened on demand by the user. She/he can also select and deselect arbitrary expressions for being observed.

Let us consider the following simple program that offers the reverse presentation of a list:

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

The function `reverse` is defined by two rules which we present as `reverse(1)` and `reverse(2)` to the user. Each of these rules can be selected for observation. All expressions within each of these rules have to be presented in the tree. In the right-hand side of the first rule the only expression is the empty list. In other words, only the expression `[]` is presented to the user. The second rule contains three function calls: `(++)`, `reverse` and `(:)`. Each function takes two expressions as parameters. Furthermore, every function call itself and all partial applications are represented (see Figure 1).

Now by the selection of the expression that is done by only a mouse-click on the expression, iCODE automatically adds necessary observe function to the source code as described in the following section and loads the changed program automatically in a new PAKCS-Shell which is provided for the programmer to perform request to the program. Advance can be triggered automatically in separate viewers which are distinguished by different labels they belong to.

4 Automatic Observations in iCODE

In this section we show how iCODE helps programmers during debugging by automatically adding the necessary observers to selected expressions and functions.

4.1 Observing Functions

The most important feature of a convenient observation tool, is the observation of top-level functions. In Curry, these functions can be defined by one or more rules and may behave non-deterministically. The idea of observing such a function should be that every call to this function is observed. The easiest way to realize this behavior is to add a wrapper function which adds the observation to the original function. In our example from Section 2, an observation of all calls to `sublists'` can be obtained as follows:

```
sublists' = observe (oList oInt ~> oPair (oList (oList oInt))
                                         (oList (oList oInt)))
                    "sublists'"
                    helpSublists'
   where
         helpSublists' []     = ([[]],[[]])
         helpSublists' (x:xs) =
              let (ready,extend) = sublists' xs in
```

```
(ready++extend,[x]:map (x:) extend)
```

Note, that we reuse the original function name for the wrapper function. By leaving the recursive calls in the right hand sides of the original function definition unchanged, we guarantee that iCODE observes each application of `sublists'`. This technique can also be applied to locally defined functions and is provided by our tool.

4.2 Observing Data Types

The most problematic part of using COOSY (especially for beginners) is the definition of observers for newly introduced data types, although COOSY provides useful abstractions for this task. For every user defined data type, corresponding observers have to be defined to observe values of this type. Our tool provides an automatic derivation of these observers, not only for defined data types in the program, but also for data types which are imported from other modules. We sketch the idea by means of an example. Consider the data type for natural numbers:

```
data Nat = O | S Nat
```

It defines two constructors, the constructor `O :: Nat` with arity 0 and the constructor `S :: Nat -> Nat` with arity 1. The observer for each type τ (e.g., `Int`) should be available as function `o`τ (e.g., `oInt`). Hence, we define an observer `oNat`. COOSY already provides generic observers `o1`, `o2`, `o3`,... by which the observer `oNat` can easily be defined as follows:

```
oNat :: Observer Nat
oNat O     = o0 "O" O
oNat (S x) = o1 oNat "S" S x
```

For polymorphic data types an observer needs observers for the polymorphic arguments as well, like `oList`. The construction should become clear from the following example:

```
data Tree a b = Branch a b [Tree a b] [Tree a b]

oTree :: Observer x1 -> Observer x2 -> Observer (Tree x1 x2)
oTree oa ob (Branch x1 x2 x3 x4) =
    o4 oa ob (oList (oTree oa ob)) (oList (oTree oa ob)) "Branch"
       Branch x1 x2 x3 x4
```

In this way, generic observers for all data structures defined in the program are generated and added automatically to the program. These observers are used for observations of functions and expressions over values of this type. This method is applied to the imported data types, so that for all imported modules the data types observers can be generated and automatically imported to the program.

However, polymorphism brings up a problem. How can polymorphic functions be observed? The function can be used in different type instantiations. Hence, the only type we can assign to its polymorphic arguments is `oOpaque`.

4.3 Observing Expressions

Sometimes it is not sufficient to observe functions defined in a program. Observations of subexpressions in the right-hand sides of rules can become necessary to find a bug. A user can also select (sub-)expressions from right-hand sides of function definitions. For this purpose iCODE provides a tree representation of the whole program, in which the user can select arbitrary (sub-)expressions of the program to be observed. Corresponding calls of `observe` are automatically added as presented in Section 2. The type of each selected expression is inferred and a corresponding observer is generated.

iCODE also automatically generates labels for the observers which helps the programmer to later identify the observations. Top-level functions are simply labeled with their name. Local functions are labeled with a colon-separated list of function names leading to the declaration of the observed function. Finally, expressions are labeled with the corresponding function name and a `String` representation of the selected expression.

4.4 Observing Imported Modules

iCODE supports adding observations to different modules of a project. When the user selects functions or expressions of a module to be observed a new module is generated which contains the observer calls. Since observer calls in (even indirectly) imported modules must also be executed, iCODE can check for each imported module whether an observer version is available and uses this for execution.

5 Design of iCODE

With the advance of modern computer technology, distributed programming is becoming more and more popular. Instead of storing huge amounts of data redundantly in many places, we use a client/server architecture, and typically, we have many clients connected to many servers. For the communication between a client and a server in Curry, TCP communication can be used. In this section we briefly review the client/server architecture of iCODE for showing the observation steps in separate viewing tools.

5.1 Architecture

Originally in COOSY, each time the computation made progress the information about observed values was recorded in a separate trace file as *events*. There are two kinds of events to distinguish unevaluated expressions from failed or non-terminated computations: Demand and Value. A demand event shows that a value is needed for the computation. A value event shows that the computation of a value has succeeded [2]. These events were shown with a textual visualization in the viewer of COOSY.

Instead, in iCODE, we use a `Socket` establishing a connection between the main window of the tool and the observed application (PAKCS-System). All events of the

observed application are sent to the iCODE's main window. Each event contains the label of the observation it belongs to. Using this architecture we can also

Fig. 2. A Socket to connect the observe applications and the main window

present observations in a single-step mode which helps beginners to understand the evaluation order of a computation. By pressing the forward button in this mode, the user can delay the client for an acknowledging message from the server before the computation continues and the next message is sent to the server (see Figure 2). The received messages in the server are forwarded to the trace windows, for showing all observations of a special label. When started, each trace window generates a socket and waits to receive the events from the main window, see Figure 3.

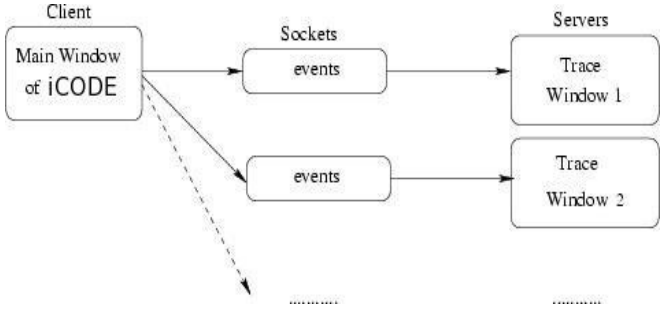


Fig. 3. Sockets to connect the main window and the trace windows

In Section 2 we have seen that for each observed function/expression a label is needed to match the observed values with the observed expressions. These labels group the progress of the execution for each observed expression in separate trace windows. Each of these windows which is named with the corresponding label receives messages from the main window through a socket.

Now, by sending events, the observed values are shown to the programmer with a textual visualization in the related trace window. The programmer may conveniently arrange these windows on her/his screen and even close observations she/he is not interested in anymore.

5.2 Surfing Observation

Originally in COOSY, the information about the observed expression was recorded as a list of events in a trace file which was considered to be shown after the program execution. That means the programmer could observe the evaluation of selected expressions only when the execution was terminated. Our aim in the new version (iCODE) is the ability to also show intermediate steps of the evaluation with the possibility of forward and backward stepping. For this purpose we use TCP communication and change the list data-structure to a tree structure which is stored in a

dynamic predicate [5]. Dynamic predicates are similar to `external` functions, whose code is not contained in the program, but is dynamically computed/extended, like the meta predicates `assert` and `retract` in Prolog.

In COOSY the generation of the observations for each observation label contained in the trace file works in a bottom-up manner. For a continuous update of observed data terms this algorithm is of no use, since in each step the whole observed data structure has to be re-constructed. We need a top-down algorithm which allows extensions in all possible leaf positions of the presented data structure. For instance, during the computation non-evaluated arguments (represented by underscores) may flip into values, but values within a data structure will not change anymore. However, we must consider the non-determinism within Curry by which values may later be related to different non-deterministic computations. Our new representation of events is stored in the following dynamic predicate:

```
TreesTable :: [([Index],EvalTree)] -> Dynamic
TreesTable = dynamic

data EvalTree = Open Int
              | Value  Arity String Index [EvalTree]
              | Demand ArgNr        Index [EvalTree]
              | Fun                 Index [EvalTree]
              | LogVar              Index [EvalTree]

type Index  = Int
type ArgNr  = Int
type Arity  = Int
```

The dynamic predicate `TreesTable` is a kind of global state accessible and modifiable within the whole program. The indices represent all nodes occuring in the corresponding evaluation tree (`EvalTree`) with respect to the order in which they were added. This is necessary since the evaluation order is not statically fixed.

In Section 5 we have seen that events are sent via a socket connection from the main window of iCODE to each trace window. Each event contains a logical parent showing in which order values are constructed. Hence, within one non-deterministic branching the index list `[Index]` is extended in its head position whenever the evaluation tree is extended, usually in an `Open` leaf.

If part of a value is used in more than one non-deterministic computation, then the logical parent indicates which part of an evaluation tree is shared within two non-deterministic computations. We only consider the subtree consisting of nodes from the index list up to the logical parent of the new event. This subtree with the corresponding indices is copied as an additional evaluation tree to the global `TreesTable`.

As an example we consider the following simple program that performs a non-deterministic computation:

```
add :: Int -> Int -> Int
add x y = x + y
main = add (0?1) 1
```

The expression `(0?1)` either yields 0 or 1. That means the function `main` offers two different results `0+1` and `1+1`.

For the first result after selecting the function `add` to be observed, ten events are sent from the observe application to the main window. The first event is a `Demand` that is stored in the above defined dynamic tree with the index `0` as:

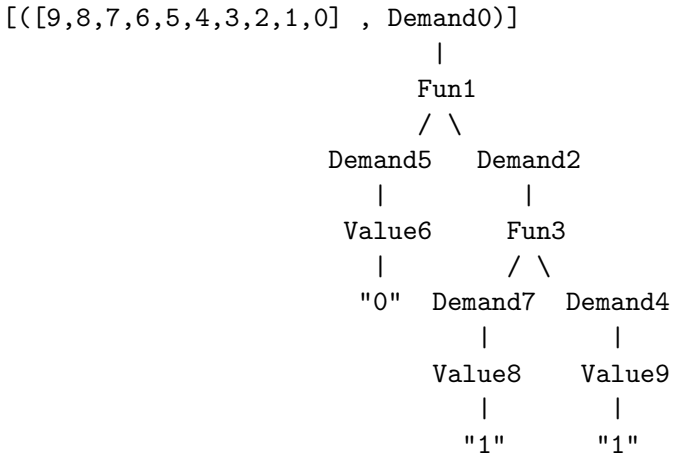
```
[([0], Demand 0 0 [(Open 1)])]
```

The second received message is a `Fun` event with the logical parent `0` that should be substituted in the open-subtree of its parent:

```
[([1,0], Demand 0 0 [Fun 1 [(Open 1),(Open 2)]])]
```

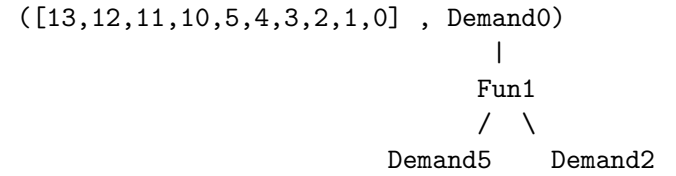
This `Fun` event is stored as a node with two subtrees presenting the argument and the result of the corresponding function. Functions are represented in curried form, i.e. the result of a function with arity two is again a function.

After adding the remaining eight events to the evaluation tree we obtain



which is shown to the user as `{\0 1 -> 1}`.

The next incoming event is a value event with the logical parent `5`. This index does not occur in the head position of the index list. Hence, we detect a non-deterministic computation. The observed value of this computation shares the nodes with indices 0 to 5 with the first tree. Hence we copy this part and extend it with the new event 10, which means the same function is called with another argument (`1`) in this non-deterministic branch. After adding three further events we obtain



```
    |          |
Value10      Fun3
    |          / \
  "1"  Demand11  Demand4
         |         |
       Value12    Value13
         |         |
        "1"       "2"
```

which is shown to the user as `{\1 1 -> 2}`.

This method helps us to provide fast pretty printing for each intermediate step of observations in the trace windows. Furthermore, we can present shared parts of the evaluation trees in the second presentation by a lighter color, which helps to understand non-deterministic computations. Figure 4 shows the last four steps of the example. While, the underscore represents a non-evaluated expression, the exclamation mark stands for an initiated but not jet finished computation.

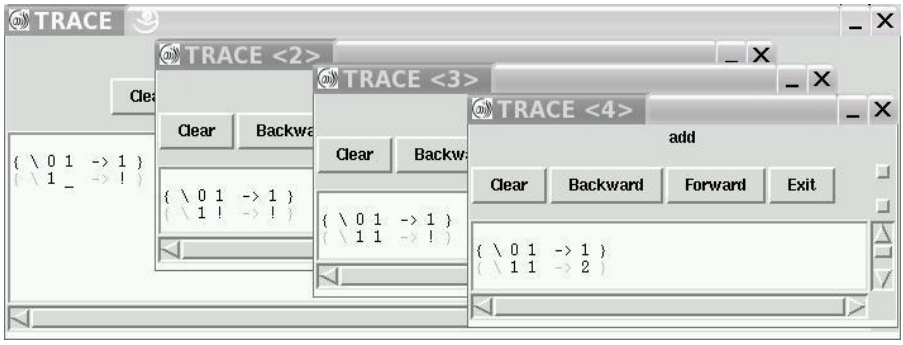


Fig. 4. A trace window

By storing the number of incoming events in a list we can also perform backward and forward stepping through the observations presented in one observation window by filtering the `TreesTable` with respect to a subsets of considered indices.

For removing the evaluation trees from the `TreesTable` we have defined a clear-button in each trace window. Furthermore when the observed program is restarted the `TreesTable` is cleared automatically.

6 Executed Parts of the Program

In some cases programmers prefer to follow the order of program execution instead of observing functions to see the program behavior during the execution. Furthermore, knowing which parts of the program have been executed is an interesting information for the programmer, because this restricts the possible locations of a bug to the executed parts. Observers should only be added to executed code. iCODE provides such a feature which can also be useful for testing small separate functions of a program and focus on a small environment of the program for being observed.

Another nice feature of constantly showing the executed parts of a program is

that in case of the program yielding `No more solutions`, the last marked expression usually shows where the computation finally failed. In many cases, the last marked expression determines the reason for an unexpected program failure or run-time error. To keep the result view of our tool small (c.f. Figure 5), we take the following artificial program as an example:

```
test :: [Int] -> [Int]
test xs = bug xs ++ okay xs

bug :: [Int] -> [Int]
bug [] = []

okay :: [Int] -> [Int]
okay xs = xs
```

The function `bug` represents a failing computation which might be much more complex in a real application. iCODE's presentation of the execution of `test [1]` is shown in Figure 5. The program is again represented as a tree (Section 3), in which executed parts are marked green and the last executed expression is marked red. We can see that the function `okay` is never applied. The bug may either be located in the application of `bug` to `xs` or within the function `bug`. Furthermore, we can see that the program finally failed when the function `bug` was applied to `xs`.

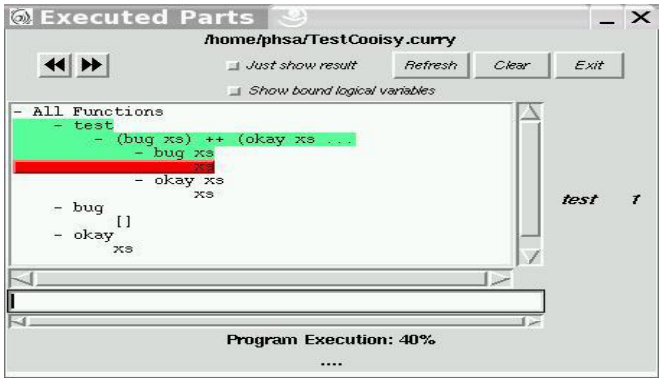


Fig. 5. Marking the executed part of the program

The viewer also shows how many times the executed functions are called. For a non-terminating computation, this information can be helpful to find the non-terminating recursion.

For marking expressions, iCODE adds calls to the function `markLineNumber` applied to the position of the actual expression in a flat-tree of the curry program:

```
markLineNumber :: String -> Int -> a -> a
```

To distinguish the expressions of imported modules from the main module, the function takes the name of the actual module as its first argument. The second argument is the position of the actual expression in a flat-tree representing the whole program and the third argument is the executed expression that the function

`markLineNumber` behaves as an identity function on. Executing this function, the first and second argument are sent as a message from the executed application to the main window of iCODE (Section 5). In the main window process, the message initiates a marking of the corresponding position of the actual expression in the viewer with the ability of backward and forward stepping on the marked expressions. This technique is a light-weight implementation of program slicing as defined in [3]. Furthermore, it will be interesting to investigate how this kind of slicing can be used for improving debugging like done in [4].

7 Related Work

iCODE is an observational debugger for the functional logic language Curry which extends Gill's idea (HOOD) [1] to observe data structures of program expressions. It is an improvement of COOSY that covers all aspects of modern functional logic languages such as lazy evaluation, higher order functions, non-deterministic search, logical variables, concurrency and constraints.

iCODE offers a comfortable graphical user interface that helps the user to conveniently and automatically observe the evaluation of arbitrary program expressions. It displays the observation steps as a comprehensive summary, based on pretty-printing.

The graphical visualization of HOOD (GHOOD) [7] also uses Gill's idea to observe the expressions of a program. In contrast to HOODs comprehensive summary as a textual visualization, GHOOD offers a graphical visualization, based on a tree-layout algorithm which displays the structure of the selected expression of a program as a tree. However, also in GHOOD observers have to be added manually which still means more effort than using iCODE.

For having a suitable overview of large programs, GHOOD offers a graphical visualization instead of textual information. This is nice for educational purposes. However, for real application the textual representation seems more appropriate and we decided to keep COOSY's textual representation within iCODE. As an improvement we present the trace for each selected expression in a separate window which the user can conveniently move or even close within his graphical environment.

Also related to debugging lazy languages is the Haskell tracer Hat [8]. It is based on tracing the whole execution of a program, combined with different viewing tools supporting users to conveniently analyze the recorded trace. Although Hat is a powerful debugging tool, there are also some disadvantage of Hat compared to observation based debugging:

- Hat is restricted to a subset of Haskell. Extension of Haskell can not be covered easily and Hat cannot be used at all to analyze programs using such extensions.
- During the execution, large trace files are generated which may slow down using the tracer for debugging real applications a lot.

These disadvantages do not hold for iCODE which is still light-weight and works independently of Curry extension (at least for that parts of a program not using

the extension). On the other hand, having the whole program as a data structure in iCODE, some more global information like in Hat can be computed (like the line information discussed in Section 6). However, iCODE is supposed to stay a light-weight and easy to use debugger.

8 Conclusion

Sometimes it is hard to figure out what caused an unexpected output or program failure. A well implemented, easy to use debugger is needed to help the programmer in finding the position of the error in the program quickly and easily.

We have extended the Curry Object Observation System [2] in a new version to provide a comfortable graphical interface as *Interactive Curry Observation DEbugger*. It helps the programmer to observe data structures or functions of arbitrary expressions of her/his program to find bugs. Using iCODE is very simple and should be accessible to beginners which we want to investigate in our next lectures about declarative programming.

Distributed programming helps us to send the information about the observed expressions through a socket and to show each computed expression in a trace window in parallel. The trace windows separate the display of observation steps for selected expressions and offer an understandable result for programmers. The information about observed expressions/functions is collected in each trace window and the ability of going forward and backward on the collected information is provided for the programmer.

The programmer does not need to add annotations to her/his program to observe the desired expressions. These annotations are added automatically by iCODE. A tree containing all program expressions (i.e. global and local functions, patterns, variables and all subexpressions) is provided for the programmer. Each selection in this tree activates iCODE to write the annotations in an extra file automatically, without changing the original program. Also larger projects consisting of different modules are supported.

For future work, we want to improve observations of polymorphic functions by generating specialized versions for each usage of observed polymorphic functions. Furthermore, we plan to investigate, how our tool can also be used as a platform for other development tools for Curry, like refactoring, test environment and program analysis. Another possible future work could result from the fact that our tool holds a lot of meta information about debugged programs. Hence, it could be possible to add observations to every/many program functions automatically and derive information about the connection between different observations which may improve debugging.

References

[1] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[2] B. Braßel, O. Chitil, M. Hanus and F. Huch. Observing Functional Logic Computations. *In Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193-208. Springer LNCS 3057, 2004.

[3] C. Ochoa, J. Silva and G. Vidal. Lightweight Program Specialization via Dynamic Slicing. *In Proc. of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1-7. ACM Press, 2005.

[4] O. Chitil. Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson and P. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474,pages 126–141. Springer, March 2005.

[5] M. Hanus. Dynamic Predicates in Functional Logic Programs. In *Journal of Functional and Logic Programming,* volume 5. EAPLS, 2004.

[6] M.Hanus et. al. PAKCS: The Portland Aachen Kiel Curry System, 2004. Available at *http://www.informatik.uni-kiel.de/~pakcs.*

[7] C. Reinke. GHood – Graphical Visualisation and Animation of Haskell Object Observations. In Ralf Hinze, editor, *ACM SIGPLAN Haskell Workshop, Firenze, Italy, volume 59 of Electronic Notes in Theoretical Computer Science*, page 29. Elsevier Science, September 2001. Preliminary Proceedings have appeared as Technical Report UU-CS-2001-23, Institute of Information and Computing Sciences, Utrecht University. Final proceedings to appear in ENTCS.

[8] M. Wallace, O. Chitil, T. Brehm and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).

[9] M. Hanus. Curry: An Integrated Functional Logic Language, 2006.

[10] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *PADL '00: Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, pages 4762, London, UK, 2000. Springer-Verlag.

[11] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley Longman,Inc., 1998.