Observing Functional Logic Computations*

Bernd Braßel[1] Olaf Chitil[2] Michael Hanus[1] Frank Huch[1]

[1] Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
{bbr,mh,fhu}@informatik.uni-kiel.de
[2] Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK
O.Chitil@kent.ac.uk

Abstract. A lightweight approach to debugging functional logic programs by observations is presented, implemented for the language Curry. The Curry Object Observation System (COOSy) comprises a portable library plus a viewing tool. A programmer can observe data structures and functions by annotating expressions in his program. The possibly partial values of observed expressions that are computed during program execution are recorded in a trace file, including information on non-deterministic choices and logical variables. A separate viewing tool displays the trace content. COOSy covers all aspects of modern functional logic multiparadigm languages such as lazy evaluation, higher order functions, non-deterministic search, logical variables, concurrency and constraints. Both use and implementation of COOSy are described.

1 Introduction

With improvements in the implementation of declarative languages and computer architecture, applications written in declarative languages have become larger. Because of this increase in application size, the need for debugging tools has become crucial. The step-by-step style of imperative debugging is not sufficient for declarative programs (first detailed discussion in [16]). This is especially true for the complex operational behaviour of lazy evaluation [11]. The programmer needs a debugging model that matches the high-level programming model of the declarative language.

Gill introduced a method for observing the values that expressions evaluate to during the execution of a lazy functional program [5]. These values give insight into how a program works and thus help the programmer locating bugs. Not only data structures but also functions can be observed. Observing does not change the normal behaviour of a program. If an expression is only partially evaluated, then exactly this partial value is observed. The most distinguishing feature of the method is that it can be implemented for a full lazy functional language by a small portable library. The programmer just imports the library and annotates expressions of interest with an observation function. Gill's Haskell Object Observation Debugger (HOOD)[3] has become a valuable tool for Haskell [13] programmers and was integrated into the popular Haskell system Hugs[4].

* This work has been partially supported by the DFG under grant Ha 2457/1-2.

[3] http://www.haskell.org/hood

[4] http://www.haskell.org/hugs

In this paper we extend the observation method to the more demanding setting of functional logic programming in the language Curry [7, 10]. The implementation as a library of observation combinators proves to be flexible in that it is not necessary to deal specially with some language features such as concurrency and constraints. However, two logical language features, namely non-determinism and logical variables, do require fundamental extensions of the implementation. In return for these complications we are able to observe more than just values. We observe additional information about non-determinism and the binding of logical variables that proves to be helpful for locating bugs in functional logic programs. In this paper we describe both the use and the implementation of the Curry Object Observation System (COOSy).

The next section gives a short demonstration of observations in purely functional Curry programs. In Section 3 we look at a number of Curry programs using non-determinism and logical variables; we see which information can be obtained by observations and how this information can be used to locate bugs. Sections 4 and 5 present the implementation of COOSy in Curry. In Section 6 we relate our work to others'. Section 7 concludes. We assume familiarity with Haskell [13] and Curry [7, 10] or the basic ideas of functional logic programming (see [6] for a survey).

2 Observations in Functional Programs

We review the idea of debugging by observation at the hand of a tiny program written in the purely functional subset of Curry:

Example 1 (A first observation).

```
max x y | x > y = x                    maxList = foldl max 0
        | x < y = y                    main = maxList [1,7,3,7,8]
```

Instead of the expected value `8`, we get the message `"No more solutions"` from the run-time system when evaluating `main`. Why has the computation failed?

First we may want to know what happened to the argument list of the function `maxList`. Using COOSy (i.e., importing the COOSy library), we can obtain this information by putting an `observe` expression around the list. The first argument of `observe` is an expression built from combinators provided by COOSy that describes the type of the observed expression. The second argument is a label to help matching observed values with observed expressions. The third argument is the expression to be observed.

```
import COOSy
...
main = maxList (observe (oList oInt) "ArgList" [1,7,3,7,8])
```

We execute the program. The function `observe` behaves like an identity on its third argument and thus does not change the program behaviour, *except* that the value of the third argument is also recorded in a trace file. We then view the information contained in the trace file with a separate COOSy tool shown in Figure 1.

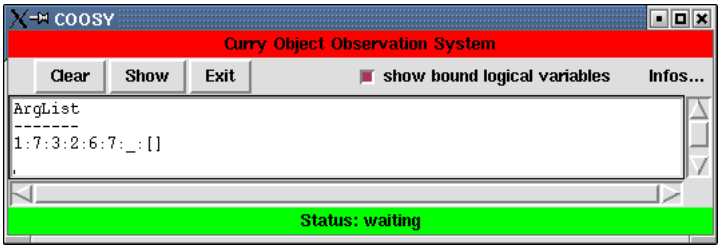

Fig. 1. Screenshot of the COOSy GUI

When we press the "Show" button, the GUI shows the human-readable representation, called *protocol*, of our first example observation. Note that observations are recorded even though the program execution fails. This is crucial for debugging purposes. In Figure 1 we can see that the argument list was evaluated to `1:7:3:7:_:[]`. The underscore means that the last number in the list has not been evaluated. Why is that? Trusting good old `foldl`, we turn to examine the call to `max`:

```
maxList = foldl (observe (oInt ~> oInt ~> oInt) "max" max) 0
```

The symbol `~>` denotes the observation of functions, here of type `Int->Int->Int`.

```
{ \ ! _  -> !
, \ 7 7  -> !
, \ 7 3  -> 7
, \ 1 7  -> 7
, \ 0 1  -> 1}
```

The result of the observation is presented on the left. The observable value of a function is a set of mappings from arguments to results. Each single mapping is written as a pseudo-lambda expression. The symbol `!` denotes a failed computation.

Now the error can be clearly located: Whereas many calls to `max` give satisfactorily results, the call "`max 7 7`" fails. Our definition of `max` does not handle equal arguments! Note that the first mapping, "`\ ! _ -> !`", is perfectly sensible: with a fail in the first argument `max` fails as well without evaluating the second argument.

We can use the type descriptor `oOpaque` in the first argument of `observe` to blend out (large) data structure components:

Example 2 (oOpaque — blending out information).

```
main = map maxList (observe (oList oOpaque) "xs" [[1..n]| n <-[1..10]])
```

The corresponding observation is

```
#:#:#:#:#:#:#:#:#:#:[]
```

The type descriptor `oOpaque` becomes vital when we observe polymorphic values so that the types of components are statically unknown at the observation point:

Example 3 (oOpaque — observing polymorphic values).

```
myLength = observe (oList oOpaque ~> oInt) "length" length
main     = myLength [1..10] + myLength "Hello"
```

We still observe the structure of the argument lists:

```
{ \ (_:_:_:_:_:_:_:_:_:_:[])  -> 10 }

{ \ (_:_:_:_:_:[])  -> 5 }
```

Curry provides type descriptors for all pre-defined types. For new data types or for the purpose of displaying values in our own way we can define our own type descriptors; see Section 4.2. Alternatively, we can use the `derive` function provided by COOSy that takes a Curry program and adds type descriptors for all user-defined algebraic data types.

3 Observations in Functional Logic Programs

In addition to the functional features of Haskell, Curry provides also features for logic programming. In particular, a function can be non-deterministic, i.e., have more than one result for a given argument, because it can be defined by a set of non-confluent rules. Furthermore, Curry expressions can contain logical (existentially quantified) variables. Both features give rise to new observations.

3.1 Observing Non-deterministic Functions

Consider the following program which uses a non-deterministic function to compute all permutations of a given list:

Example 4 (Observing non-determinism).

```
perm []     = []                        insert x []     = [x]
perm (x:xs) = insert x (perm xs)        insert x (y:ys) = y:insert x ys
main = perm [1,2,3]                     insert x xs     = x:xs
```

Naturally, we implanted a little bug you will have spotted already. The effect of this bug is not a wrong result but too many of the right ones. Correctly the program computes the lists `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, and `[3,2,1]`, but these lists are computed twice, thrice, twice, thrice, four, and eight times, respectively. To isolate the bug, we set an observation point at the call of the non-deterministic function of our program:

```
perm (x:xs) = observe oInsert "insert" insert x (perm xs)
      where oInsert = oInt ~> oList oInt ~> oList oInt
```

Now, we can make use of another nice feature of COOSy: observations can be viewed before the end of the whole computation. To debug Example 4, we view the observation only when a few solutions are computed. For the first result `[3,2,1]` we get the following protocol for the label `insert`:

```
{\ 3 [] -> 3:[]}

{\ 2 (3:[]) -> 3:2:[]}

{\ 1 (3:2:[]) -> 3:2:1:[]}
```

The blank lines show that all three entries belong to different observations (one for each recursive call to `perm`). Of course, this protocol yields no hint to the bug, as we have only seen a single solution so far. We prompt the system for the next solution, which is again `[3,2,1]`. This could give us a hint towards the error.

The resulting protocol is:

```
{\ 3 []  -> 3:[]}

{\ 2 (3:[]) -> 3:2:[]}
{\ 2 (3:[]) -> 3:2:[]}

{\ 1 (3:2:[]) -> 3:2:1:[]}
{\ 1 (3:2:[]) -> 3:2:1:[]}
```

The observations of `insert` are ordered in three packages of calls belonging together. The lines within one package are non-deterministic alternatives of a single call to `insert`. The colouring provides the information about what was already computed for the previous solution and what is newly constructed. For instance, in the second package the alternative solution only constructed a new end of the list. This is a very good hint towards finding the error, but it can get even more obvious if we look at the protocol for the first three solutions. The third solution is again `[3,2,1]` and the protocol starts like this:

```
{\ 3 [] -> 3:[]}
{\ 3 [] -> 3:[]}
```

Now in these first two lines it is clear that `insert` yields two results for the very same arguments `3` and `[]`. The source of this is easy to spot in the program and we conclude that the two rules "`insert x xs = x:xs`" and "`insert x [] = [x]`" overlap more than they should. Deleting the latter yields the correct program which computes all and only the desired solutions.

3.2 Observing Logical Variables

Curry features another extension of functional programming: the use of logical variables. We first demonstrate how COOSy represents logical variables.

Example 5 (Representation of logical variables).

```
main = observe oInt "Logical Variable" x where x free
```

The clause "`where x free`" declares the variable `x` as a logical (existentially quantified, free) variable. Running the program, we obtain the observation on the left in which we can see that COOSy represents unbound logical variables as "`?`".

```
Logical Variable
----------------
?
```

Let us observe a logical variable which is bound in the course of the computation:

```
f 1 = 1
main = f (observe oInt "Variable + Binding" x) where x free
```

Now the result is `(?/1)`. Changing `f` in this example to `f 1 = 1` and `f 2 = 2` we get the result on the left (after computing all possible solutions). Note that the colouring indicates that the same logical variable was bound to different values.

```
(?/1)
(?/2)
```

The next example includes a new little bug:

Example 6 (Searching a bug in a function defined with logical features).

```
last xs | append ys [y] =:= xs = y
      where y,ys free

append []     ys = ys
append (x:xs) ys = x:append xs xs
```

The function `last` is intended to compute the last element of a list by solving the equation `append ys [y] =:= xs`. When considering the call `last [1,2,3]`, this equation is (or rather *should be*) only solvable if `ys` is bound to `[1,2]` and `y` is bound to `3`, as "`append [1,2] [3]`" should equal `[1,2,3]`.

However, testing our little program, the result of `last [1,2,3]` is not the expected `3` but rather a logical variable. Where should we begin to find the bug? Let us be cautious about this error and start at the top, slowly advancing down the program. First we shall look at the whole list computed in the guard of `last`:

```
last xs | observe oInts "as" (append ys [y]) =:= xs = y where y,ys free
```

```
?/1:[]
?/1:[]
?/1:?/2:[]
?/1:?/2:?/3:[]
?/1:?/2:?/3:_:_
```

With `oInts = oList oInt`. The result of the observation is on the left. This result is peculiar. Knowing how the program should work, it is okay that there are three states of evaluation where the system guesses that `1`, `2` and `3` might be the last element of the list. But why are there two versions of "`?/1:[]`"? We should find out more about the arguments of `append`:

```
last xs | append (observe oInts "ys" ys) [y] =:= xs = y where y,ys free
```

This yields the observation:

```
(?/[])
(?/(?/1:(?/[])))
(?/(?/1:(?/(?/2:(?/[])))))
(?/(?/1:(?/(?/2:(?/(?/3:(?/[]))))))
(?/(?/1:(?/(?/2:(?/(?/3:(?/(?:?)))))))
```

We see immediately why COOSy has the option to turn off the representation of bound variables as the result after selecting it is so much more readable.

```
[]
1:[]
1:2:[]
1:2:3:[]
1:2:3:?:?
```

This looks quite correct, with the small exception that `ys` should not be bound to the whole list `[1,2,3]` but maximally to `[1,2]`. A similar observation of the second argument `y` resulting in `(?/1)` is also wrong of course, but we already know that `y` is not bound correctly, or the final result could never be a logical variable.

Seeing all this, we cannot help but search for our error in `append`:

```
last xs | observe oApp "App" append ys [y] =:= xs = y where y,ys free
```

where `oApp = oList oInt ~> oList oInt ~> oList oInt`. The observation yields (with representation of bound logical variables turned off again):

```
{\[]          (1:[]) -> 1:[]}
{\(1:[])       _     -> 1:[]}
{\(1:2:[])     _     -> 1:2:[]}
{\(1:2:3:[])   _     -> 1:2:3:[]}
{\(1:2:3:?:?) _      -> 1:2:3:_:_}
```

We can clearly see that whenever the first argument of append is a non-empty list, the second argument is not evaluated. Looking into the rule `append (x:xs) ys = x:append xs xs`, we now easily spot the bug where our eyes were clouded only seconds ago.

For the purpose of this paper, we have only presented very small buggy programs. Similarly to other language implementations, Curry implementations [9] provide warnings for some of these bugs (e.g., single occurrences of a variable in a defining rule), but it should be clear that there are also many larger wrong programs where such compiler warnings are not produced.

4 Implementation for Functional Programs

Now we delve into the depths of our first implementation for purely functional Curry programs, before we consider non-determinism and logical variables in the next section. Our implementation roughly follows Gill's HOOD implementation, outlined in [5]. We start with the format of our protocol files and then describe how these files are written.

4.1 Format of the Trace Files

Each time the computation makes progress with the evaluation of an observed expression an event is recorded in the trace file. To distinguish unevaluated expressions from failed or non-terminating computations, we need to trace two types of events:

Demand The incident that a value is needed for a computation and consequently its evaluation is started is called a *Demand Event*.

Value The incident that the computation of a value (to head normal form) has succeeded is called a *Value Event*.

From the nature of evaluation in Curry we can conclude:

- No Value without Demand: If a value was computed, there must have been a corresponding Demand Event for the data.
- A Demand Event without a corresponding Value Event indicates a failed or non-terminated computation.
- The correspondence between Value and Demand Events has to be recorded in the protocol.

Keeping these facts in mind, we can now turn to the format of the trace files. A *trace* is a list of events, each of which is either a Demand or a Value Event. To model the correspondence between Demands and Values, we construct an identification of the events by numbering them: `type EventID = Int`. For each Value one of its arguments has to be its own ID and another one is the ID of the corresponding Demand, which we will call its *parent*. There are two more important things we have to know about a Value Event: (1) Because a value is built from constructor symbols, we need the arity of the constructor. (2) We need a string indicating what the value should be printed like in the GUI.

Recording the arity of a value is important for another task of the COOSy system: The traced data structures have to be reconstructed correctly. In order to achieve this, more than the arity has to be recorded. A Demand has to be tagged with the information which argument of which value it demands. This means that two numbers have to be added: (1) the `EventID` of the Value Event, which is the parent of the Demand and (2) a number between 1 and the arity of that Value. The first Demand of an observation is tagged with a parent ID of `-1` and argument number of `0` to indicate that it has no parent and denotes the root of the data structure. Putting it all together we have:

```
data Event = Value  Int    String                EventID  EventID
             --     arity  value representation  own ID   parent ID
           | Demand Int                          EventID  EventID
             --     number of argument           own ID   parent ID
```

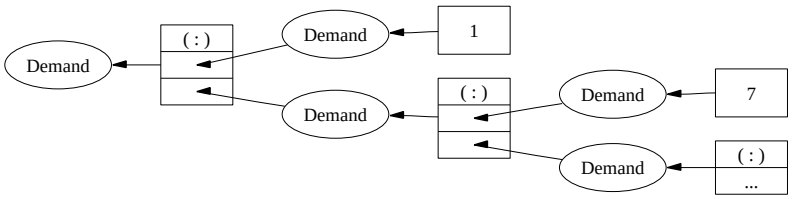

Fig. 2. Graphical representation of COOSy trace of Example 7

Example 7 (Trace representation). The trace of Example 1 begins like this:

```
[Demand 0 0 -1, Value 2 "(:)" 1  0,Demand 2  2 1, Value 2 "(:)" 3  2,
 Demand 2 4  3, Value 2 "(:)" 5  4,...
 Demand 1 16 1, Value 0 "1"  17 16,Demand 1 18 3, Value 0 "7"  19 18,...]
```

Note that the pattern of alternating Demands and Values is not necessary in general.

To make traces more comprehensible, we use a graphical representation, where ellipses denote Demands and Values are denoted by rectangular records which contain a field for each argument. The parent and structural information is visualised by arrows, see Figure 2.

Functions are treated in the same way as data structures: A functional value is built like any binary constructor, connecting its argument with its result. To represent a function with more than one argument, the result is a functional value as well (curryfied representation of functions). The only special treatment of functions is in the pretty printer.

4.2 Writing the Trace

When the function `observe` is initially called, it

1. gets a new `EventID` from a global state (to make sure that the ID is unambiguous),
2. writes a Demand Event into the trace file as a side effect (by `unsafePerformIO`),
3. evaluates the observed value to head normal form,
4. deconstructs the result, if it is not a constant (constructor with arity 0),
5. constructs a new term with the same head symbol where `observe` is called for each argument of the result constructor together with the information which of the arguments it observes.

Example 8 (Observing user-defined data structures). We want to formulate an observer for the data type `data Nat = O | S Nat` for which an observation might be "`main = observe oNat "Nat" (S (S O))`".

The function `observe` calls the actual observing function `observer` with the initial values for parent and argument number `-1` and `0`, respectively. Both functions are independent of the observed type.

```
observe oType label x = observer oType label x (-1) 0

observer oType label x parent argNr = unsafePerformIO $
      do eventID <- recordEvent label (Demand argNr) parent
         return (seq x (oType x label eventID))
```

The function `seq` evaluates `x` to head normal form and then returns its second argument. When this evaluation of `x` succeeds, the function `oType` is responsible for recording the value in the trace file. The function `recordEvent` first obtains a new `EventID` and then writes the Event to a trace file. There is a separate trace file for each user-defined label, to simplify filtering events from different observations directly when they occur.

```
recordEvent label event parent =
      do eventID <- getNewID
         writeToTraceFile label (event eventID parent)
         return eventID
```

As an example for a function `oType`, observing the type `Nat` could be implemented as follows:

```
oNat O label parent = unsafePerformIO $
      do recordEvent label (Value 0 "O") parent
         return O
```

This is the code for the constant `O`. For the unary constructor `S` we need to call the function `observer` again, recursively starting the observation of the arguments by means of the observer `oNat`:

```
oNat (S x) label parent = unsafePerformIO $
      do eventID <- recordEvent label (Value 1 "S") parent
         return (S (observer x oNat eventID 1))
```

These functions are fairly schematic for each type, depending only on the arity of the constructors. Therefore, COOSy provides functions `o0, o1, o2...` for constructors of arity 0,1,2... Using these, the definition of `oNat` simply looks like this:

```
oNat O     = o0 "O" O
oNat (S x) = o1 oNat "S" S x
```

The function `oNat` is the *standard observer* for the type `Nat`. Standard observers are automatically derivable as mentioned at the end of Section 2.

5 Extending COOSy for Logical Language Features

To handle the full language Curry, we have to extend our implementation

- to put together information from observations of non-deterministic computations correctly
- and to observe logical variables.

5.1 Non-Deterministic Functions

In the extended setting, the number of values corresponding to a single Demand Event is not limited. The problem is that we need some kind of separation; which values belong to which part of the computation? The following program demonstrates the problem:

Example 9 (Showing the need of predecessors).

```
coin = O          plus O     x = x                    main = plus O coin
coin = S O        plus (S x) y = S (plus x y)
```

Observing `plus` in `main` using `observe (oNat ~> oNat ~> oNat) " + " plus`,

```
{\ O O     -> O}
{\ O (S O) -> S O}
```

we would like to obtain the observation on the left. However, looking at the trace of Example 9 (Figure 3), we spot a problem. The function `plus` has two arguments and therefore is represented as argument1 `->` (argument2 `->` result). We see in Figure 3 that the first argument of `plus` was simply evaluated to `O`. Because the second argument is the non-deterministic function `coin`, its demand yields two values! In consequence, there are two results. The problem is: *Which argument belongs to which result?* There is simply not enough information in the trace of

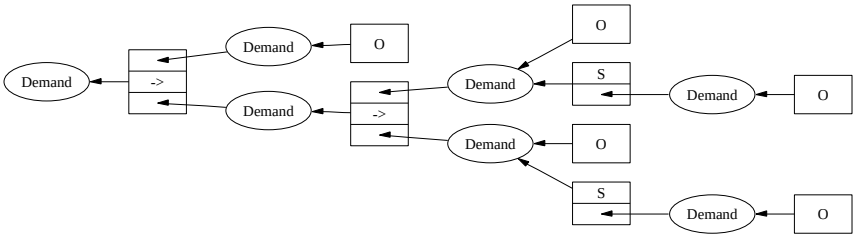

Fig. 3. COOSy trace of Example 9

Figure 3 to answer this question. So we have to store more information in the COOSy trace. In addition to the reference chain denoting the structural information (the *parents*), we also have to trace a reference chain denoting which event belongs to which branch of the computation. Therefore, each Demand and each Value Event is extended by a further `EventID`: the ID of the event which occurred just before in the same branch of the computation, called the *predecessor*. We have extended Figure 3 with this new reference chain, denoted by dotted arrows, and the result is shown in Figure 4. The extended output can be separated correctly into the non-deterministic computations by following the predecessor chain starting at each result. The predecessor chain comprises exactly the parts of the trace that correspond to the computation yielding the result.

5.2 Writing the Predecessor Chain

The predecessor in a given computation is a matter of the rewrite or narrowing strategy. Consequently, it is clear that we can only obtain this information by exploiting non-declarative features.

It is also clear that the observer functions introduced in Section 4.2 have to be extended by a further argument providing the information about the predecessors:

```
observer oType l x parent argNr preds = unsafePerformIO $
         do eventID <- recordEvent l (Demand argNr) parent preds
            return (seq x (oType x l eventID preds))
```

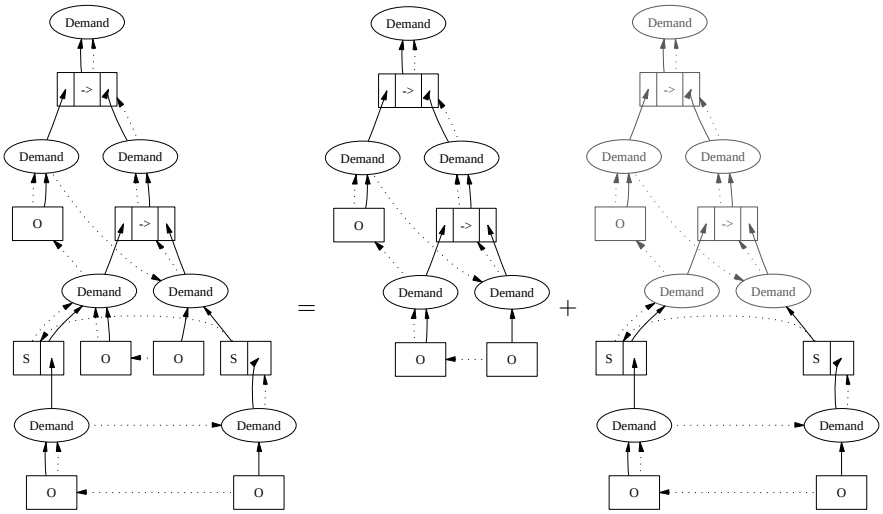

Fig. 4. Separating the extended COOSy trace of Example 9

The functions `oType` for observing the different types of data have to be extended analogously. Still unclear is the type of this new argument `preds`. As it is impossible to know beforehand which part of a given term will be evaluated next without implementing the whole search strategy, we need some means of propagating information across the term structure. Fortunately, logical variables provide such a tool. If the new argument of the function `observer` is a logical variable, its binding will be visible throughout the whole observation. Thus, whenever a new event is written to the trace file, the variable can be bound to the corresponding `EventID`. However, because the order of evaluation is unknown, we need instead of a single `EventID` a list of IDs terminated by a logical variable to be bound to the next event in line.

This is where we have to use a non-declarative feature. Whenever we want to write a new event to the trace file, we need to go through the list of predecessors until we find the logical variable which marks its end. This variable is then bound to the new `EventID` followed by a new logical variable. In order to find the variable marking the end of the list, we need a test function `isVar` which returns `True` whenever its argument is a logical variable and `False` otherwise. While this function can easily be implemented in any Curry implementation, this test is clearly non-declarative.[5]

The extended function `recordEvent` can now be defined as

```
recordEvent label event parent preds
  = do eventID <- getNewID
       let (pred,logVar) = getPred parent preds
```

[5] Consider the definition `rightFirst | x=:=0 & y=:=isVar x = y where x,y free` The function `rightFirst` returns `True` whenever the right constraint `y=:=isVar x` was evaluated before the left one. This should not be possible because `&` denotes the concurrent conjunction of constraints, i.e., the combined constraint c_1 `&` c_2 is solved by solving the constraints c_1 and c_2 *in any order*.

```
        newLogVar free
    doSolve (logVar =:= (eventID:newLogVar))
    writeToTraceFile label (event eventID parent pred)
    return eventID
```

where `doSolve` solves the given constraint and `getPred` is defined as

```
getPred p xs = if isVar xs then (p,xs) else getPred (head xs) (tail xs)
```

5.3 Information Concerning Logical Variables

As Section 3.2 demonstrated, COOSy also allows observation of logical variables. This task is more complex than it sounds.

Whenever a value is computed, we have to test if it is a logical variable. Otherwise, we would get into the unfortunate situation that the observing functions would either begin to guess solutions for the variable or suspend until it is bound. Influencing the computation in this way would of course be disastrous for COOSy as a debugging tool. Hence the function `observer` has to test the observed expression with `isVar` (Section 5.2) before proceeding. However, what to do if this test yields true? First of all, we need to write an event of a new kind to the trace file and, consequently, we have to extend the type of events:

```
data Event = Demand ... | Value ... | LogVar EventID EventID EventID
                                     --     ownID   parent  predecessor
```

Thus, for each logical variable we store its own ID as well as the ones of its parent and its predecessor. There is one serious problem left, though. What we would like to observe is not only the fact that a logical variable has been computed but also the values it will be bound to, see Examples 5 and 6. How are we going to do this? The values to which the variable will be bound are guessed by the search strategy or directly in the code of other parts of the program. Therefore we introduce one last unsafe feature: a special kind of constraint which suspends on the observed logical variable, while the rest of the computation continues unchanged. As soon as the variable is bound, the constraint is woken up and the new value can be recorded in the trace file. The "as soon as" is crucial because otherwise information about the binding might not be written at all in the case of a failed computation. The complete code is:

```
observer x oType parent argNr preds = unsafePerformIO $ do
 eventID <- recordEvent (Demand argNr) parent preds
 if isVar x
   then do idLogVar <- recordEvent LogVar eventID preds
           spawnConstraint (seq x (x =:= oType x label idLogVar preds))
                           (return x)
   else return (oType x label eventID preds)
```

Note that `isVar` already evaluates `x` to head normal form, so we do not need `seq` in the `else`-branch. However, we need `seq` to make sure that the spawned constraint suspends as long as `x` is not bound to a value. `spawnConstraint` is the last non-standard feature of the PAKCS [9] implementation of Curry we have to explain. It can be thought of as a normal guarded rule like `isZero x | x=:=0 = x` which could be written using `spawnConstraint` as

```
isZero x = spawnConstraint (x=:=0) x.
```

However, there are two important differences: (1) the right-hand side is evaluated even if the evaluation of the guard suspends and (2) the spawned constraint gets a higher priority than any normal constraint.

The Curry environment PAKCS prints a warning when suspended constraints are left at the end of an evaluation. This warning will occur in COOSy whenever an observed logical variable is never bound to a value. However, as the constraints spawned by COOSy are essentially identities (with the side effect of writing into the trace file), this warning does not restrict the soundness of the evaluation.

6 Related Work

COOSy extends Gill's idea of observing values in lazy functional languages to functional logic languages. COOSy also differs in that the function `observe` takes a type description as first argument whereas in HOOD there is no such argument because the function `observe` is overloaded, using the Haskell class system which is not yet provided by Curry. While overloading is convenient for simple examples, a type descriptor is more flexible as Example 3 demonstrates. It is a serious limitation of HOOD that it cannot observe polymorphic expressions. The version of HOOD integrated in Hugs overcomes this problem through a polymorphic `observe`. Its implementation, however, requires reflective features from the runtime system, which would be hard to provide for most compilers.

The most influential approach to debugging declarative programs is *declarative (or algorithmic) debugging*. It was originally introduced by Shapiro for logic programming languages [16]. The system asks questions about part of the computation such as "Should `factorial 3` yield `17`?" which the user has to answer with "yes" or "no". After a series of questions and answers, the system can give the location of the bug in the program. There exist numerous implementations and the approach has been extended to constraint logic programming [18], assertions [4], functional programming [11, 12, 14] and functional logic programming [2]. Declarative debugging relies on a simple big-step semantics for the programming language. Hence the approach is less suited for imperative languages, side effects in general and even the search strategies of logic languages.

A large number of methods for debugging lazy functional languages have been proposed [12, Chapter 11]. Most of these rely on following the sequence of actual reductions of the computation. However, for a human user the evaluation order of a lazy functional program is confusing. It is far easier—and also goal-oriented for debugging—to navigate *backwards* through a computation trace from an erroneous result to the cause of the bug [1, 17].

Each approach to debugging has its strengths and weaknesses, as already a comparison of three systems shows [3]. Hence later versions of the Haskell tracing system Hat[6] enable the generation of a single trace from a computation which can be viewed in several different ways. Hat includes viewing tools for redex trailing, declarative debugging and observations [19].

[6] `http://www.haskell.org/hat`

Like HOOD, COOSy differs in a number of ways from most other debugging tools for declarative languages. First of all it is implemented by a small portable library, whereas most other tools are either implemented as full program transformations or even modifications of several phases of a compiler. In return, observations are limited to values and information about demand and logical variables. Further information via a library would require extensive reflective features in the programming language. Like most debugging tools, COOSy records a trace as a separate data structure. In this trace only information about expressions under observation are recorded. In contrast, most tools record information about the full computation. While full recording enables the user to inspect the full computation after a single run, it poses serious space problems. Observation with COOSy does slow down computation considerably but only proportionally to the size of the observed data and not to the length of the full computation. COOSy requires the user to annotate his program with applications of `observe`. While any program modification poses the danger of introducing additional bugs and substantial modifications would be tiresome, this does provide a simple and intuitive interface to using the system.

7 Conclusion

We presented a new lightweight method for detecting bugs in functional logic programs. The user of the Curry Object Observation System (COOSy) annotates expressions of interest in his program. During program execution information about the values of observed expressions are recorded in a trace file. A separate viewing tool displays this information in the form of mostly familiar expressions.

In this paper we extended the debugging-by-observation method as introduced and implemented by HOOD [5] for functional languages to functional logic languages. The challenge was the support of non-deterministic computations and logical variables. The implementation method of HOOD would record confusing information on values obtained in non-deterministic computations and no information on any parts of values whose computation involved logical variables. Our implementation even records additional information in the trace on non-determinism and logical variables which are displayed by the viewing tool in an easy-to-read way and which give the user vital clues for debugging.

It is essential for debugging that observations do not change the semantics of the program and work for computations that fail with run-time errors or have to be interrupted because of non-termination. The separation of faulty program and viewing tool, afforded by the trace files, enables debugging of programs that create dynamic web pages and which are run on a web server in batch mode [8].

A number of examples demonstrated how COOSy helps locating bugs in faulty programs. Through gaining more experience in using COOSy—especially on larger programs with real bugs—we hope to distill general strategies for debugging with COOSy. Furthermore, we intend to extend the viewing tool in the future. Currently, it gives only a static snapshot of the observed values. Instead, observed values could be shown and constantly updated concurrently to the computation. We could also take advantage of the sequential recording of

events in the trace. The viewing tool could allow the user to step forwards and backwards through an animation, similar to GHood [15]. Animations would give the programmer further insights into the workings of his program and might be particularly useful for educational purposes.

COOSy is written in Curry, requiring only a few non-declarative extensions that are easy to implement. COOSy is freely available for the Curry system PAKCS [9][7] and will be distributed with future versions of PAKCS.

References

1. Simon P. Booth and Simon B. Jones. Walk backwards to happiness - debugging by time travel. In *Automated and Algorithmic Debugging*, pages 171–183, 1997.
2. R. Caballero, F.J. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In *FLOPS 2001*, pages 170–184. Springer LNCS 2024, 2001.
3. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *IFL 2000*, pp. 176–193. Springer LNCS 2011, 2001.
4. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. The use of assertions in algorithmic debugging. In *Proc. FGCS'88*, pages 573–581, 1988.
5. Andy Gill. Debugging Haskell by observing intermediate data structures. In Graham Hutton, editor, *ENTCS*, volume 41. Elsevier, 2001.
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
7. M. Hanus. A unified computation model for functional and logic programming. In POPL '97, pages 80–93, 1997.
8. M. Hanus. High-level server side web scripting in Curry. In *PADL'01*, pages 76–92. Springer LNCS 1990, 2001.
9. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.
10. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2002.
11. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
12. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, 1998.
13. Simon Peyton Jones et al. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), 2003.
14. B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *ACM PPDP '03*, pages 230–240, 2003.
15. Claus Reinke. GHood – graphical visualisation and animation of Haskell object observations. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, 2001.
16. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
17. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. *LNCS*, 1292:291–308, 1997.
18. A. Tessier. Declarative debugging in constraint logic programming. *LNCS*, 1179:64–73, 1996.
19. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, 2001.

[7] `http://www.informatik.uni-kiel.de/~pakcs/COOSy`