Electronic Notes in Theoretical Computer Science 76 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume76.html 13 pages

Declarative Debugging for Encapsulated Search

Rafael Caballero [1]

Dep. Sistemas Informáticos y Programación
Universidad Complutense Madrid
Madrid, Spain

Wolfgang Lux [2]

Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Münster, Germany

Abstract

Declarative debugging has been proposed as a suitable technique for developing debuggers in the context of declarative languages. However, to become really useful debuggers must be able to deal with those parts of the languages that have no clear declarative semantics but are widely used in realistic programs. We explain in this paper how a declarative debugger of wrong answers for the lazy functional-logic language Curry can be extended to debug programs including the encapsulated search, an important feature of this language designed to control non-deterministic computation steps. We show how this can be done without introducing any changes in the compiler.

1 Introduction

Declarative debugging, also known as *algorithmic debugging*, was first introduced in [Sha82] in the context of the logic programming language Prolog, and later (see [Nai97]) presented as a general debugging technique. The overall idea is to introduce a suitable *computation tree* associated to any erroneous computation (the *initial symptom*). Different kinds of errors will have different types of associated computation trees.

The nodes of such computation trees must contain the results of auxiliary subcomputations, each one obtained by means of some logical inference from the results found at its children. Also, any node must have a possible *cause*

[1] Email: rafa@sip.ucm.es
[2] Email: wlux@uni-muenster.de

of the error associated. A *buggy node* is then a node with an erroneous result and with no erroneous results at its children, corresponding therefore to an erroneous subcomputation (i.e. it produces an erroneous output starting with correct inputs). Since the root of the tree is erroneous because it is the result of the main computation, it is easy to prove that in such tree at least one buggy node must exist, and therefore a cause of the error can be found.

In order to detect erroneous nodes an *intended meaning* of the program is assumed. This intended meaning must be known by an external *oracle* which is consulted by the debugger during the debugging process. Usually this oracle is the user, and hence the number and simplicity of the questions become a key concept for the usefulness of the debugger.

Due to its abstraction from the operational mechanisms of the languages, declarative debugging has been particularly helpful in the context of declarative programming, where the built-in control complicates the use of traditional debugging techniques such as tracing. This is particularly true in the case of lazy functional and functional-logic languages [NB96,NF97,NS97,Pop98] where the evaluation order must be transparent to the programmer. A known extension of declarative debugging is *abstract diagnosis* [CL+99,ACF01], leading to equivalent bottom-up and top-down diagnosis methods which do not require error symptoms to be given in advance. In order to be effectively implemented, abstract diagnosis uses abstract interpretation techniques to build a finite abstraction of the intended program semantics. These methods are outside the scope of this paper.

To become really helpful declarative debuggers cannot be constrained to deal only with the semantically declarative parts of the language. For instance input/output and graphical interfaces are part of realistic applications and hence any debugger tool should allow programs with these features to become fully applicable.

This is also the case of the *encapsulated search* [HS98], a feature of the language Curry [Han00] introduced to control non-deterministic computation steps. The encapsulated search can be used to implement search strategies different from the usual *depth-first* search, such as *bounded search* or *breadth-first search*. Also, it allows the programmer to encapsulate non-deterministic computations in programs using I/O facilities. For all these reasons, encapsulated search is part of many Curry programs and a realistic debugger for Curry should be able to debug programs including it.

Unfortunately, this is not trivial. On one hand, the encapsulated search is based on a built-in primitive *try* which is described in terms of its operational behaviour, thus having no clear declarative semantics. On the other hand, *try* cannot be considered as a usual primitive (like, for instance, *sin*). Indeed, in declarative debugging primitives are automatically assumed to be correct and a void computation tree associated to them. However, *try* is a higher-order primitive, which means that the computation of *try* can have associated erroneous subcomputations, not because the primitive itself is wrong, but due

to the evaluation of terms involving other functions of the program passed as parameters.

We present in this paper a declarative debugger for Curry which deals with programs including encapsulated search and show how this can be done without affecting the built-in code for *try*. The key idea is to replace all the calls to the primitive *try* in the debugged program by calls to a new function *try'*. This function is defined in Curry and will return the trees corresponding to the subcomputations carried out by the higher order parameters inside *try*. We show how this can be done easily in a declarative debugger based on *program transformation*. The ideas presented in this paper are part of the declarative debugger tool of wrong answers recently incorporated into the Curry system developed at the University of Münster [Lux99].

The next section introduces the declarative debugger for wrong answers included in the Münster Curry system. Section 3 introduces the encapsulated search in Curry, and section 4 presents our solution for extending the debugger to programs using this feature, showing the debugging sessions obtained for the examples of section 2. Finally, section 5 presents some conclusions and future work.

2 Declarative Debugging of Wrong Answers in Lazy Functional-Logic Programs

In this section we introduce a declarative debugger of wrong answers for Curry, but still without considering encapsulated search. First we will introduce the computation trees used by our debugger and then explain briefly the program transformation mechanism used to implement the tool. All the concepts are introduced informally and by means of examples. See [CLR00,CR02] for a complete theoretical presentation on these aspects. Both works are presented in the context of the lazy functional-logic language $\mathcal{TOY}$ but are also valid for Curry with minor changes. Notice that differences at the operational level (such as the availability of residuation in addition to narrowing for function reduction in Curry), do not affect the semantics and therefore have no influence on the debugger.

All examples are presented in the syntax of the lazy functional-logic language Curry (see [Han00] for a complete description). We will consider initial *goals* as general expressions e, and *computed answers* as pairs (e', σ), meaning that $e\sigma$ can be evaluated producing e' as result. Whenever $\sigma = id$ we will write the computed answer simply as e'. Notice that Curry allows non-deterministic computations and therefore a goal can have several answers.

Computation Trees

As we said in the introduction, the debugging process starts with an initial symptom detected while evaluating some goal. Two different types of symptoms are possible, each one corresponding to a different sort of bug:

Positive symptom: An unexpected answer is obtained as the result of the computation. This answer is called a *wrong answer*.
Negative symptom: An expected answer is missing in the multiset of results obtained as the result of the computation. This answer is called a *missing answer*. A particular (but usual) case of missing answers is obtained when a goal fails (i.e. computes no answer) unexpectedly.

Consider for instance, the Curry program in Fig. 1. Function *append* repre-

```
append eval flex

append::  [A] → [A]→ [A]

append []       ys  = ys

append (x:xs)   ys  = x :  append xs ys


last::  [A] → A

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

Fig. 1. Last element in a list

sents the concatenation of two lists, while function *last* is intended to compute the last element in a list. However, the goal

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

returns `1` as the only computed answer. Here we have both a missing answer (`3`) and a wrong answer (`1`).

In order to detect wrong answers, the oracle (the user in our debugger) must know the intended meaning $\mathcal{I}$ of the program. As proved in [CLR00], it is enough to consider $\mathcal{I}$ as a set of *basic facts* of the form $f\ t_1 \ldots t_n \rightarrow t$, where $t_1 \ldots t_n, t$ are constructor terms, meaning that function f produces t as a result when applied to $t_1 \ldots t_n$. All questions to the user will be about whether a basic fact is in $\mathcal{I}$ or not. Notice that this ensures that the questions asked to the user are as simple as possible. This is done by replacing nested function calls by their results obtained during the computation. A suspended function call not evaluated at the end of the computation is denoted by a special constructor term $\perp$, represented in the debugger questions by the symbol `_`. Thus we can say that `last [1,2,3]` $\rightarrow$ `1` $\notin \mathcal{I}$ while `last [1,2,3]` $\rightarrow$ `3` $\in \mathcal{I}$, with $\mathcal{I}$ the intended meaning of the program in Fig. 1.

The *computation trees* used by the debugger will have basic facts at their nodes. Each node has an associated program rule, the program rule used at that computation step. Thus the debugger will point out the program rule associated to a buggy node as an incorrect program rule. The children of a node correspond to the subcomputations carried out while evaluating

the conditions (guards and local declarations) and the right-hand side of the program rule used at that point. The results of soundness and completeness presented in [CLR00] ensure that given a wrong answer an incorrect program rule is detected by the debugger.

Implementing the Debugger

Several strategies have been presented to create and navigate the computation tree. A well-known approach widely employed in Logic Programming uses meta-interpreters to re-execute the goal during the debugging phase. Thus, the computation tree is not constructed explicitely, and both wrong and missing answers are easily handled. This idea has been extended in the case of NUE-Prolog to Functional-Logic Languages [NB95].

However, this solution is not feasible in languages that do not provide these built-in meta-instructions, as Haskell or Curry. To the best of our knowledge, only declarative debuggers for wrong answers are currently available for these languages. The reason is twofold: first, the computation trees necessary for detecting missing answers are much more complicated. Second, often wrong and missing answers occur simultaneously, i.e. we obtain an unexpected answer instead of the expected one. In these cases usually it is enough to find out the reason for the wrong answer to get rid of both errors. This is also the case of our debugger: only wrong answers are treated.

Two different techniques have been proposed in related papers for producing a computation tree associated to a wrong computation (see [NS97] for a comparison). These two proposals are also valid for the case of lazy functional-logic programming:

- Modify the implementation of the abstract machine to produce the computation tree.
- Transform the program to be debugged P into a new program P' in which all the functions return the same result as in P but paired with their corresponding computation tree.

We have adopted the second alternative based on program transformation because of its flexibility and portability. Therefore, a function like

```
append::[a] → [a] → [a]
```

will be included in the transformed program with a type

```
append'::[a] → [a] → ([a],CTree)
```

where `CTree` is a datatype defined to represent computation trees:

```
data CTree = Void | Node String [String] String [CTree]
```

Void trees are used for auxiliary functions whose computations cannot be the cause of the error (*trusted* functions). The last argument of `Node` are the children trees, while the strings represent, respectively, the name of the program rule, its arguments and the produced result. An impure function

`dval::a` $\rightarrow$ `String` is used in the transformed program (see the example below) to convert any value into its `String` representation.

Observe that this type transformation must be applied recursively in order to change also the type of functional types occurring as parameters. In general each n-ary function

$$f :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$$

is transformed into a function

$$f' :: \tau_1\prime \rightarrow \ldots \rightarrow \tau_n\prime \rightarrow (\tau\prime, \text{CTree})$$

with the type transformation $\prime$ defined as:

$$\alpha\prime = \alpha \qquad (C\ \overline{\tau}_n)\prime = C\ \overline{\tau}_n\prime \qquad (\mu \rightarrow \nu)\prime = \mu\prime \rightarrow (\nu\prime,\ \text{CTree})$$

where α is a *type variable* and C a *type constructor* of arity n. Actually, as explained in [CR02] a n-ary function f is transformed into $n+1$ functions in the transformed program, including n functions for the case of partial applications of f. However, we will skip these auxiliary functions here for the sake of simplicity. As an example, the second rule of `append` will appear in the transformed programs as:

```
append'  (x:xs) ys = (result,tree)
  where  (r,t)    = append' xs ys
         result   = x :  r
         tree     = Node "append" [dval (x:xs), dval ys]
                         (dval result) [t]
```

In this case there is only one subcomputation, `append' xs ys`, and that it produces a result `r` and a tree `t` which will be the only child of a computation step using this program rule.

3 Encapsulated Search

The encapsulated search [HS98] in Curry allows to explore the search space of a predicate with a user defined search strategy. This is useful if the default strategy is not suitable. In addition, it allows to encapsulate non-deterministic computations such that they can be used in a program that interacts with the external world through Curry's monadic I/O system [PW93].

Programatically the encapsulated search is available through the primitive function *try* with type

$$(\alpha \rightarrow \texttt{Success}) \rightarrow [\alpha \rightarrow \texttt{Success}]$$

The argument of *try* is the (unary) predicate whose solutions are searched. The evaluation of an expression *try* g for some search goal g starts by applying

g to a fresh variable x and then reduces this application until one of the following happens.

(i) The evaluation fails, i.e. there is no solution to $g\ x$. In this case the encapsulated search returns an empty list.

(ii) The application succeeds, taking only deterministic computation steps. In this case *try* g returns a singleton list `[`g'`]`, where g' is the solved form of g. For instance, given the definitions from Fig. 1, `try (λx → x =:= last [1,2,3])` returns the list `[λx → x =:= 1]`. Notice that the evaluation of `last [1,2,3]` is completely deterministic.

(iii) A non-deterministic computation step occurs. In this case *try* g returns a list of all possible continuations after this step in the form of partially solved search goals.
Thus `try (λx → append x [] =:= [1,2,3])` evaluates to the list [3]

```
[(λx → x =:= [] & append [] [] =:= [1,2,3]),
 (λx → let y,ys free in
       x =:= (y:ys) & append (y:ys) [] =:= [1,2,3])]
```

With the help of the *try* primitive, we can implement different strategies to explore the search space of a predicate. The necessary functions to implement a breadth first search and unpack its solutions are shown in Fig. 2. The *bfs*

```
bfs g =  step [g]
  where  step     []          = []
         step     (g:gs)      = collect (try g) gs
         collect  []          gs   = step gs
         collect  [g]         gs   = g :  step gs
         collect  (g1:g2:gs)  gs'  = step (gs' ++ g1:g2:gs)

unpack g | g x = x where x free
```

Fig. 2. Breadth first search

function maintains a list of search goals. The local *step* function takes the next goal from this list and applies *try* to it. If there is no solution to this goal it is discarded. Otherwise, if the goal can be solved deterministically its solved form is returned. Finally, if *try* returns a list of alternatives the local *collect* function appends them to the list of search goals. In any case, *step* is called again for the remaining goals. The *unpack* function is useful for extracting the solution of a search goal.

Due to the lazy evaluation strategy of Curry only those solutions are com-

[3] In Curry the operator `&` denotes the (concurrent) conjunction of two predicates.

puted that are actually needed, making it possible to use *bfs* for a goal which has infinitely many solutions. Consider for instance the program of Fig. 3 for finding paths in a directed graph. The existence of cycles in the graph can

```
data Node = A | B | C | D


graph::  [(Node,Node)]
graph = [(A,B), (B,A), (B,D), (D,C), (C,B)]


path::  [(Node,Node)] → Node → Node → [Node]
path g x  y | x =:= y = [x]
path g x  y | (member (x,z) g) & (path g y z =:= l) = x:l
                where z,l free


member::  a → [a] → Success
member x  (y:ys) = x=:=y
member x  (y:ys) = member x ys
```

Fig. 3. Paths in a directed graph

produce non-terminating computations using the default depth-first search. However by using *bfs* in combination with *head* we can try a goal like

```
unpack (head (bfs (λl → l =:= path graph B C)))
```

which computes the shortest path from `B` to `C`. Nevertheless the result obtained is `[B, C, A, B]` which is a wrong answer in the sense of Sect. 2, and constitutes an example showing the convenience of debugging programs that include encapsulated search.

4 Declarative Debugging of Encapsulated Search

A declarative debugger has no option than trusting the implementation of primitive functions. In the case of first order primitives the debugger can associate a void computation tree with them. However, the situation is more complicated for a higher order primitive like *try*.

Because of the functional types occurring in the argument and result types of *try*, it does not suffice to pair the result of *try* with a void computation tree. In addition, the arguments and the results of *try* have to be transformed to match the types expected by the debugger, following the type transformation

described in Sect. 2:

```
try'::(a → (Success,CTree)) → ([a → (Success,CTree)],CTree)
try' g = (map wrap (try (unwrap g)),Void)

unwrap  g  = λx → let (r,t) = g x in r
wrap    g  = λx → (g x,Void)
```

The auxiliary function *unwrap* transforms the argument into a regular search goal, i.e. a unary predicate, by discarding its computation tree. On the other hand, the *wrap* function invents a computation tree (*Void*) to be associated to goals returned by the *try* primitive. The use of a void computation tree is justified by the fact that all elements of the list returned by *try* are (equivalent to) lambda abstractions.

However, this transformation is too naive, as can be seen if we use the following main function as a goal with the program in Fig. 1.

```
main = unpack (head (bfs (λx → x =:= last [1,2,3])))
```

When executed this program returns the wrong result 1. But we are unable to find the source of this error in function *last* using the above transformation of *try*:[4]

```
Entering debugger...

No error has been found
```

This outcome results from the fact that we discard computation tree associated with the computations performed inside the encapsulated search and at the same time return a void computation tree from *try'*. Thus, our transformation not only trusts the *try* primitive itself but also all computations carried out by *try* as well.

As *try* performs internal computations which can be wrong, it seems that we should return a non-void computation tree together with the list search goals. But we cannot look inside the operation of the *try* primitive in order to determine which operations have been performed within the encapsulated search. Even worse, if a non-deterministic computation step has caused *try* to return, no computation tree has been built for the search goal but only (unrelated) parts of this tree exist.

Fortunately, a careful analysis reveals that it is not necessary to extract such information at all if we want to diagnose *wrong answers*. In the presence of meta functions like *try* we have to be more precise about which errors can

[4] In all the debugging sessions presented in this section functions *bfs* and *unpack* are considered *trusted*. Thus the navigator skips automatically the questions about their validity, going directly to their children.

be considered as wrong answers because the encapsulated search may turn missing answers into wrong answers. Consider for instance the simple goal

```
main = bfs (λx → last [1,2,3] =:= 3 & x =:= 1)
```

For the intended meaning of *last* we expect this function to return a list containing a λ-abstraction equivalent to `λx → x =:= 1`. However, given the definition of *last* from Fig. 1 the program will return an empty list, which is a wrong answer. On the other hand, if we consider the search goal passed to *try* alone, it has no solution at all but fails. Thus, we have a missing answer. We cannot expect our debugger to diagnose such errors which were originally missing answers and manifest themselves as wrong answers only by the use of a meta function.

Having thus restricted the set of errors we can detect, it turns out that we only need the computation trees of solved search goals and can trust all intermediate computations performed by *try*. The only kind of wrong answer that is possible for a unary predicate p is $p\,t$ being satisfied for some term t such that $p\ t \rightarrow \mathit{Success} \notin \mathcal{I}$. Such an error can be observed only if the (solved) search goal returned from *try* is applied to an argument in a code similar to *unpack*. The computation tree associated to the search goal thus becomes a child of the node associated to the function which unpacks the solution.

We cannot return the computation tree computed by the transformed search goal directly because the *try* primitive can only handle (unary) predicates. Instead we can instantiate an additional argument to the search goal with the computation tree when the computation succeeds. Thus our improved *wrap* and *unwrap* functions become

```
unwrap g  = λ(x,t) → let (r,t') = g x in r & t =:= t'

wrap g x  | g (x,t) = (Success,t) where t free
```

With these definitions, *try'* still returns results of the form (*value*,`Void`) as any other primitive, but now each element in the list *value* (of type $\alpha \rightarrow$ `(Success,CTree)`) will produce the computed tree associated to its corresponding subcomputation, if it succeeds.

Note that by the use of an equality constraint, the program becomes more strict. Our transformation ensures that all computation trees built for a successful computation are finite. Therefore the use of the strict equality in the improved *unwrap* function will not introduce termination problems.

Using the improved definitions of *wrap* and *unwrap* the debugger now correctly spots the wrong definition of *last*.

```
Entering debugger...

Considering the following basic facts:
1. main#lambda1(1)  -> Success
```

```
Are all of them valid? (y/n)
n

Considering the following basic facts:
1. last([1,2,3])  -> 1
Are all of them valid? (y/n)
n

Considering the following basic facts:
1. append([1], [2,3])  -> [1,2,3]
Are all of them valid? (y/n)
y

 ** Function last is incorrect (last([1,2,3])  -> 1) **
```

A more realistic example, which involves a non-trivial search, is the goal

```
main = unpack (head (bfs (λl → l =:= path graph B C)))
```

for the program Fig. 3.

```
Entering debugger...

Considering the following basic facts:
1. main#lambda1([B,C,A,B])  -> Success
Are all of them valid? (y/n)
n

Considering the following basic facts:
1. graph()  -> [(A,B),(B,A),(B,D),(D,C),(C,B)]
2. path([(A,B),(B,A),(B,D),(D,C),(C,B)], B, C)  -> [B,C,A,B]
Are all of them valid? (y/n)
n
Write the number of an erroneous basic fact in the list
2

Considering the following basic facts:
1. member((B,A), [(A,B),(B,A),(B,D),(D,C),(C,B)])  -> Success
2. path([(A,B),(B,A),(B,D),(D,C),(C,B)], C, A)  -> [C,A,B]
Are all of them valid? (y/n)
n
Write the number of an erroneous basic fact in the list
2

Considering the following basic facts:
1. member((C,B), [(A,B),(B,A),(B,D),(D,C),(C,B)])  -> Success
2. path([(A,B),(B,A),(B,D),(D,C),(C,B)], A, B)  -> [A,B]
```

```
Are all of them valid? (y/n)
y

 ** Function path is incorrect
    (path([(A,B),(B,A),(B,D),(D,C),(C,B)], C, A)  -> [C,A,B]) **
```

5 Conclusions

We have presented in this paper some ideas that can be used to integrate an important operational feature of the language Curry, the encapsulated search, into a declarative debugger for wrong answers. The solution presented is simple and has been introduced in the debugger readily and without affecting the rest of the system, in particular the built-in coding of the primitive *try*. The result of this integration can be found in the Münster Curry system, which is available at http://danae.uni-muenster.de/\char‘~lux/curry.

As future work, other features as input/output should be included as part of the compiler using similar ideas to those presented here.

Regarding the navigation phase, we plan to introduce an algorithm to reduce the number of questions asked to the oracle, not only avoiding repeated questions but also questions *entailed* by previous ones, following the ideas presented in [CR02]. At the moment only the erroneous instance of the erroneous program rule is displayed, which usually is enough to identify this rule in the program, but this information must still be completed by including its position.

An interesting point of research would be that of missing answers and its relation to encapsulated search. Since any goal with a missing answer can be converted easily to a failing goal, the primitive *try* can be used to convert any missing answer in a wrong answer (namely the answer `[]`) and maybe this could be exploited in a debugger for missing answers in Curry.

References

[ACF01] M. Alpuente, J. Correa and M. Falaschi. *A Debugging Scheme for Functional Logic Programs*. Proc. WFLP'2001, Kiel, Germany, September 13–15, 2001. 1

[CLR00] R. Caballero, F.J. López-Fraguas and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. Technical Report SIP-104-00, Dep. Lenguajes, Sistemas Informáticos y Programación, July 2000. 2, 2

[CL+99] M. Comini, G. Levi, M.C. Meo and G. Vitello. *Abstract Diagnosis*. J. of Logic Programming 39, 43–93, 1999. 1

[CR02] R. Caballero and M. Rodríguez-Artalejo. *A Declarative Debugging System*

for Lazy Functional Logic Programs. Electronic Notes in Theoretical Computer Science. Volume 64. To appear. 2, 2, 5

[Han00] M.Hanus. *Curry: An Integrated Functional Logic Language*. Version 0.7.1, June 2000. Available at http://www.informatik.uni-kiel.de/curry/report.html. 1, 2

[HS98] M.Hanus, F. Steiner. *Controlling Search in Declarative Programs*. In Proc. PLILP/ALP'98, LNCS 1490, pp.374–390,1998. 1, 3

[LS99] F.J. López-Fraguas, and Jaime Sánchez Heznández. *$\mathcal{TOY}$ a Multiparadigm Declarative System*, In Proc. RTA'99, LNCS 1631, Springer Verlag, 244–247, 1999.

[Lux99] W. Lux. *Implementing Encapsulated Search for a Lazy Functional Logic Language*. In Proc. FLOPS 99, LNCS 1722, 100–113, 1999 1

[Nai97] L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3. 1

[NB95] L. Naish, Timothy Barbour. *A Declarative debugger for a logical-functional language*. In Graham Forsyth and Moonis Ali, eds. Eight International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Invited and additional papers, Vol. 2, pp. 91–99, 1995. DSTO General Document 51. 2

[NB96] L. Naish and T. Barbour. *Towards a Portable Lazy Functional Declarative Debugger*. Australian Computer Science Communications, 18(1):401–408, 1996. 1

[NF97] H. Nilsson, P. Fritzson. *Algorithmic Debugging of Lazy Functional Languages*. The Journal of Functional Programming, 4(3):337–370, 1994. 1

[NS97] H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121–150, 1997. 1, 2

[PW93] S. Peyton Jones, P. Wadler. *Imperative Functional Programming*. In Proc. POPL'93, 123–127, ACM, 1993 3

[Pop98] B. Pope. *Buddha. A Declarative Debugger for Haskell*. Honors Thesis, Department of Computer Science, University of Melbourne, Australia, June 1998. 1

[Sha82] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Mass., 1982.

1