Combining Algorithmic Debugging and Program Slicing *

Josep Silva
DSIC, T.U. Valencia
Camino de Vera s/n
46022 Valencia, Spain
jsilva@dsic.upv.es

Olaf Chitil
Computing Laboratory
University of Kent
CT2 7NZ, Canterbury, UK
o.chitil@kent.ac.uk

Abstract

Currently, program slicing and algorithmic debugging are two of the most relevant debugging techniques for declarative languages. They help programmers to find bugs in a semiautomatic manner. On the one hand, program slicing is a technique to extract those program fragments that (potentially) affect the values computed at some point of interest. On the other hand, algorithmic debugging is able to locate a bug by automatically generating a series of questions and processing the programmer's answers. In this work, we show for functional languages how the combination of both techniques produces a more powerful debugging schema that reduces the number of questions that programmers must answer to locate a bug.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and meaning of programs—specifying and verifying and reasoning about programs; D.3.1 [*Software*]: Programming Languages—formal definitions and theory

General Terms Languages, Theory, Algorithms

Keywords Program Slicing, Algorithmic Debugging

1. Introduction

Algorithmic debugging [20] is a debugging technique which relies on the programmer having an *intended interpretation* of the program. That is, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Algorithmic debugging was originally developed for logical languages and later transferred to other language paradigms, including (lazy) functional languages [14].

Essentially, algorithmic debugging is a two phase process: An *execution tree* (see, e.g., [12]) is built during the first phase where each node in the execution tree is labeled with an equation which consists of a function call whose arguments and result are in their final state of evaluation. In the second phase, the debugger traverses the execution tree asking the programmer whether equations are correct or wrong w.r.t. the intended semantics. When all the children of a wrong equation (if any) are correct, the debugger locates the bug in the function definition of this node [14]. Let us illustrate this process with an example.

EXAMPLE 1.1. *Consider the buggy program in Figure 1 (a), adapted from [7]. This program sums a list of integers ([1,2]) and computes the square of the result with three different methods. If the three methods compute the same result, the program returns* $True$*, if not, it returns* $False$*. Here, one of the three methods—the one adding the partial sums of its input number—contains a bug. From this program, an algorithmic debugger can automatically generate the execution tree of Figure 1 (c) (built-in functions such as + and == are assumed to be correct and hence their reductions are omitted from the tree; for the time being, the reader can ignore the distinction between white and dark nodes). This tree, in turn, can be used to produce a debugging session as depicted in Figure 1 (b). During the debugging session, the system asks the programmer about the correctness of some execution tree nodes w.r.t. the intended semantics. At the end of the debugging session, the algorithmic debugger determines that the bug of the program is located in function* $sum2$*. The definition of function* $sum2$ *should be:* $sum2\ x = div\ (x * (decr\ x))\ 2$.

Unfortunately, in practice—for real programs—algorithmic debugging can produce long series of questions which are semantically loosely connected (i.e. consecutive questions refers to independent parts of the computation) making the process very complicated indeed for the programmers of the program being debugged. In addition, questions can also be very complex. For instance, during a debugging session of a compiler, the algorithmic debugger for the logic-functional language Mercury [11]—currently, one of the most advanced algorithmic debuggers—asked a question of more than 1400 lines.

Hence new techniques and strategies to reduce the number of questions, to simplify them and to improve the order in which they are asked are a necessity to make algorithmic debuggers usable in practice.

To overcome such problems, in this paper we combine algorithmic debugging with program slicing [23]. Program slicing is a technique for decomposing programs based on data and control flow information. In particular, given an arbitrary expression in a program, program slicing can determine which slices, that is, program fragments, can (potentially) affect this expression. By combining algorithmic debugging and program slicing the debugging process

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, by the Vicerrectorado de Innovación y Desarrollo de la UPV under project TAMAT ref 5771, and by the United Kingdom under EPSRC grant EP/C516605/1.

PPDP'06 July 10–12, 2006, Venice, Italy.

```
main = sqrtest [1,2]

sqrtest x = test (computs (listsum x))

test (x,y,z) = (x==y) && (y==z)

listsum [] = 0
listsum (x:xs) = x + (listsum xs)

computs x = ((comput1 x),(comput2 x),(comput3 x))

comput1 x = square x

square x = x*x

comput2 x = listsum (list x x)

list x y | y==0      = []
         | otherwise = x:list x (y-1)

comput3 x = listsum (partialsums x)

partialsums x = [(sum1 x),(sum2 x)]

sum1 x = div (x * (incr x)) 2
sum2 x = div (x + (decr x)) 2

incr x = x + 1
decr x = x - 1
```

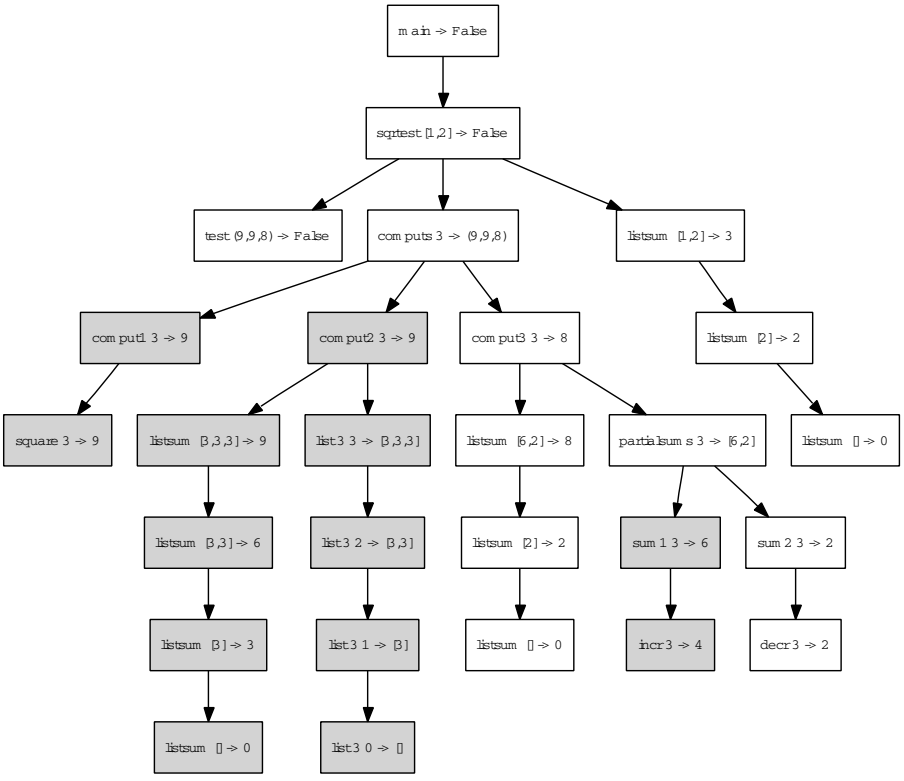
(a) Example program

```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test [9,9,8] = False? YES
(4)  computs 3 = [9,9,8]? NO
(5)  comput1 3 = 9? YES
(6)  comput2 3 = 9? YES
(7)  comput3 3 = 8? NO
(8)  listsum [6,2] = 8? YES
(9)  partialsums 3 = [6,2]? NO
(10) sum1 3 = 6? YES
(11) sum2 3 = 2? NO
(12) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

(b) Debugging session for program (a)



(c) Execution tree of program (a)

Figure 1. Example program (a) and its associated debugging session (b) and execution tree (c)

becomes more interactive, allowing the user to provide more information to the debugger. This information reduces the number of questions and guides the process, thus making the sequence of questions semantically related.

The main contributions of this paper can be summarised as follows:

1. We combine both *program slicing* and *algorithmic debugging* in a single, more powerful framework in the context of functional programming.
2. We adapt the well-known program slicing concepts of slice, slicing criterion, complete slice, and so on to the Augmented Redex Trail, a trace structure that can be the basis for both algorithmic debugging and program slicing, and we define and prove properties for this formalism.
3. We introduce an algorithm for slicing execution trees which is proven complete, in the sense that every slice produced contains all the nodes of the tree that are needed to locate a bug.

The rest of the paper is organised as follows. In the next section we outline how algorithmic debugging can be improved by letting the user provide more information. In Section 3 we define the Augmented Redex Trail and in Section 4 we define which kind of slices of an Augmented Redex Trail we are interested in. In Section 5 we combine these slicing concepts with algorithmic debugging and introduce an algorithm to compute the slices we need. Next, we give an example of a debugging session that uses our technique in Section 6. In Section 7 we discuss related debugging techniques. Finally, Section 8 concludes.

2. The Idea: Improved Algorithmic debugging

As shown in Example 1.1, a conventional algorithmic debugger will ask the programmer whether an equation

$$foo\ a_1 \ldots a_n\ =\ e$$

is correct or wrong; the programmer will answer "*yes*", "*no*" or "*maybe*" depending on whether they think that it is respectively correct, wrong or they just don't know.

Our main objective in this work is to be able to get more information from the programmer and consequently reduce the number of questions needed to locate a bug. For this purpose, we consider the following cases:

1. **The programmer answers "*maybe*".** This is the case when the programmer cannot provide any information about the correctness of the equation.
2. **The programmer answers "*no*".** The programmer can disagree with an equation and provide the system with additional information related to the wrong result.

   When the result is wrong, the programmer could say just "*no*", but they could also be more specific. They could exactly point to the parts of the result which are wrong. For instance, in the equation above, they could specify that only a subexpression of e is wrong, and the rest of the result is correct. This information is useful because it could help the system to avoid questions about correct parts of the computation.
3. **The programmer answers "*yes*".** This is the case when the programmer agrees with both the complete result and all the expressions in the left hand side of the equation; or when some precondition of the equation has been violated (i.e. some parts of the left hand side were not expected).
   - The programmer could detect that some expression inside the arguments of the left hand side of the equation should not have been computed. For instance, consider the equation $insert\ 'b'\ \text{"}cc\text{"}\ =\ \text{"}bcc\text{"}$, where function $insert$ inserts the first argument in a list of mutually different characters (the second argument). Clearly the result of the equation is correct; but the programmer could detect here that the second argument should not have been computed. This means that even for a correct equation the programmer could provide the system with information about bugs. In this case, the programmer could mark the second argument ("cc") as wrong (i.e. *inadmissible* [13] for this function because it violates a precondition).
   - Finally, both, the result and the arguments, could be correct, but the function name could be wrong[1], in the sense that it should have never been called. In this case, the programmer could mark the function name as wrong.

The information provided by the programmer allows the system to slice the execution tree, thus reducing the number of questions. The more information they provide, the smaller the execution tree becomes. For instance, the slice computed if the programmer answers "*no*", is usually larger than the one computed if they answer "*no, and this function should not have been computed*". Of course, as is the case in conventional algorithmic debugging, the programmer may only give information about the wrong expressions they are able to detect. In the worst case, they can always answer "*yes*", "*no*" or "*maybe*".

In summary, in order to provide information about errors, the programmer can specify that some part of the result or the arguments, or the function name is wrong. The slicing process needed for each case is different.

3. The Augmented Redex Trail

In the first phase of algorithmic debugging the program is executed and the execution tree is built. However, the execution tree is not actually built directly as described. Direct construction of the execution tree during the computation would not be very efficient; in particular, the nodes of the execution tree are labelled with equations that contain many repeated subexpressions and, for a lazy functional language, the tree nodes are constructed in a rather complex order. So instead of the final execution tree a more elaborated trace structure is constructed that avoids duplication of expressions through sharing.

The *Augmented Redex Trail (ART)* [21, 22] is such a trace structure. The ART is used in the Haskell tracer Hat [21, 22] and has similarities with the trace structure constructed by the earlier algorithmic debugger Freja [15]. The ART has been designed so that it can be constructed efficiently and after its complete construction the execution tree needed for algorithmic debugging can be constructed from it easily. In addition, an ART contains more information than an execution tree. An execution tree can be constructed from an ART but not vice versa. The ART supports several other views of a computation besides the execution tree. Nonetheless an ART is not larger than an execution tree with sharing. An ART is a complex graph of expression components that can represent both eager or lazy computations.

DEFINITION 3.1 (Augmented Redex Trail). *A **node** n is a sequence of the letters l, r and t. There is also a special node $\perp$.*

*A **node expression** $\mathcal{T}$ is a function symbol, a data constructor, a node or an application of two nodes.*

*A **trace graph** is a partial function $\mathcal{G} : n \mapsto \mathcal{T}$ such that its domain is prefix-closed (i.e., if $ni \in dom(\mathcal{G})$, then $n \in dom(\mathcal{G})$),*

[1] As we consider a higher order language, this is a special case of a wrong argument, the function name being the first (wrong) part of the function application.

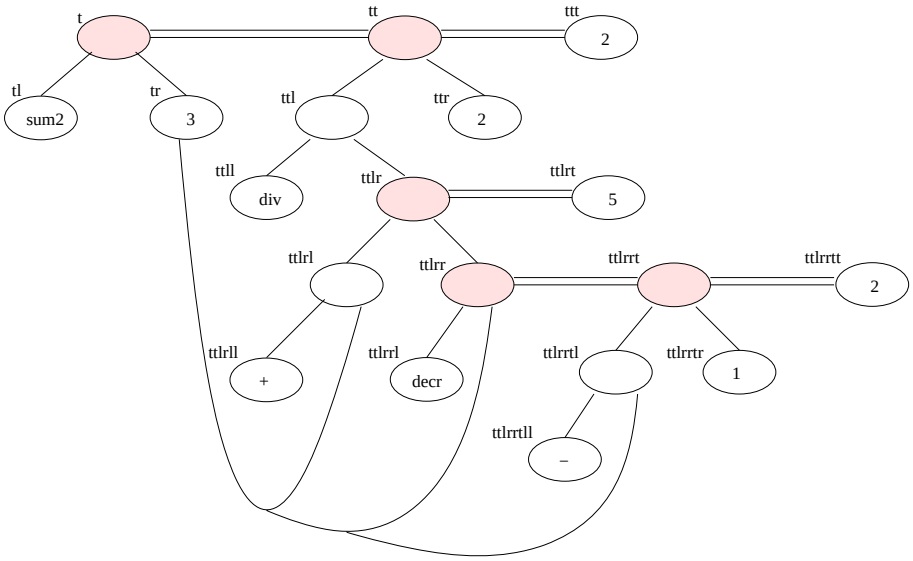


Figure 2. Example of ART

but $\bot \notin \mathcal{D}om(\mathcal{G})$. We often regard a trace graph as a set of tuples $\{(n, \mathcal{G}(n)) \mid n \in dom(\mathcal{G})\}$

*A node can **represent** several terms, because any nodes n and nt are considered equal. A node **approximates** a term if it represents this term, except that some subterms may be replaced by $\bot$.*

*A node n plus a node nt represent a **reduction step** of the program, if node n represents the redex and node nt approximates the reduct, and for each variable of the corresponding program rule there is one shared node.*

*A trace graph $\mathcal{G}$ where node ϵ (empty sequence) approximates the start term M and every pair of nodes n and nt represents a reduction step with respect to the program P, is an **Augmented Redex Trail (ART)** for start term M and program P.*

More detailed definitions are given in [4, 3].

For example, consider the graph depicted in Figure 2. This graph is a part of the ART generated from the program in Figure 1 (a). In particular, it corresponds to the function call "$sum2\ 3$" producing the result "2". As specified in Definition 3.1, each node is identified by a sequence of letters 't', 'l' and 'r'[2]. Every node contains the expression it represents, except for a node which represents an application of two expressions connected to it by simple lines. A double line represents a single reduction step. For instance, node $ttlrrt$ represents the function call "$3 - 1$" that has been reduced to "2".

In the following a (atom) denotes a function symbol or constructor, and n, m and o denote nodes. We usually write $n \in \mathcal{G}$ instead of $n \in \mathcal{D}om(\mathcal{G})$ when it is clear from the context. We also say that a node $n \in \mathcal{G}$ is reduced iff $nt \in \mathcal{G}$. Given a node $n \in \mathcal{G}$ where $\mathcal{G}(n)$ is an application of nodes $m \cdot o$, we respectively refer to m and o as $left(n)$ and $right(n)$.

The parent of a node is the node which created it. Formally:

$$parent(nl) = parent(n)$$
$$parent(nr) = parent(n)$$
$$parent(nt) = n$$

The recursive application of function $parent$ produces the ancestors of a node:

$$ancestors(n) = \begin{cases} \{m\} \cup ancestors(m) & \text{if } m \text{ defined} \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{where } m = parent(n)$$

Finally, to extract the most evaluated form represented by a node, function mef is defined by:

$$mef(\mathcal{G}, n) = \begin{cases} a & \text{if } \mathcal{G}(n) = a \\ mef(\mathcal{G}, m) & \text{if } \mathcal{G}(n) = m \\ mef(\mathcal{G}, m) \cdot mef(\mathcal{G}, o) & \text{if } \mathcal{G}(n) = m \cdot o \text{ and } nt \notin \mathcal{G} \\ mef(\mathcal{G}, nt) & \text{if } nt \in \mathcal{G} \end{cases}$$

Intuitively, function mef traverses the graph $\mathcal{G}$ following reduction edges (n to nt) as far as possible.

Now we see how easily the execution tree is obtained from the ART: The reduced nodes of the ART become the nodes of the execution tree. The function $parent$ expresses the child–parent relationship between these nodes in the execution tree. The equation of a node n can be reconstructed by applying the function mef to $left(n)$ and $right(n)$ to form the left-hand-side, and to nt to form the right-hand-side. For instance, for the ART in Figure 2 the algorithmic debugger produces an execution tree with five questions, one for each reduction step (reduced nodes have been shaded). For example, the question asked for node $ttlr$ is "$3 + 2 = 5$?".

The actual implementation of ARTs includes more detailed information than described here. In particular, the execution tree in Figure 1 (c) was generated from the graph shown in Figure 5.

4. Slicing the ART

Program slicing techniques have been based on tree-like data structures such as the *Program Dependence Graphs* [6]. In the context of execution trees we want to define the slicing mechanism over their underlying ART. This provides us with two advantages:

1. At this level we have much more information about the computation which allows production of more precise slices. It is straightforward to extract an execution tree from the sliced ART [2].

[2] For easier reading we here call the first node t; within the ART of the complete computation the node is actually $trtrtrtrlr$.

2. When we define the slicing technique at the ART level, it can be used for any other purpose (e.g. for proper program slicing which has not been defined for these graphs, for program specialisation, etc.).

Now, we define the notion of *slice* and some properties in the context of ARTs.

DEFINITION 4.1 (slice). *Let $\mathcal{G}$ be an ART. A slice of $\mathcal{G}$ is a set of nodes $\{n_1, \ldots, n_k\} \subseteq \mathcal{D}om(\mathcal{G})$.*

To compute slices w.r.t. a particular node of interest we need to identify which nodes in the computation have influenced this node of interest. We introduce the following definition:

DEFINITION 4.2 (Influence between Nodes). *Given an ART $\mathcal{G}$ for start term M and program P. Node n_1 influences node n_2, written $n_1 \rightrightarrows n_2$ iff for all ARTs $\mathcal{G}'$ for start term M and program P with $n_2 \in \mathcal{G}'$ we have $n_1 \in \mathcal{G}'$.*

Note that we have not fixed any evaluation order for constructing ARTs, but functional programs are deterministic. Hence different ARTs differ in which parts of the computation are how much evaluated.

Influence is transitive and reflexive.

LEMMA 4.3 (Transitivity and Reflexivity $\rightrightarrows$). *Let $\mathcal{G}$ be an ART for start term M and program P. For nodes n_1, n_2 and n_3 in $\mathcal{G}$:*

- *$n_1 \rightrightarrows n_2$ and $n_2 \rightrightarrows n_3$ implies $n_1 \rightrightarrows n_3$*
- *$n_1 \rightrightarrows n_1$*

PROOF 4.4. *We consider influence for a graph $\mathcal{G}$ with start term M and program P. Let $\mathcal{G}'$ be any ART for the same start term and program such that $n_3 \in \mathcal{G}'$. With $n_2 \rightrightarrows n_3$ follows that node n_2 is in $\mathcal{G}'$. Then with $n_1 \rightrightarrows n_2$ follows that node n_1 is in $\mathcal{G}'$. So according to Definition 4.2, $n_1 \rightrightarrows n_3$. Reflexivity trivially holds by definition of influence.*

We perform program slicing on the ART $\mathcal{G}$ without knowing the program P whose computation is described by $\mathcal{G}$. So we cannot compute other ARTs $\mathcal{G}'$ for start term M and program P; furthermore, there is a large, possibly infinite number of such $\mathcal{G}'$s. Our aim is to find an algorithm that obtains a slice from the one given ART $\mathcal{G}$ ensuring completeness (i.e., all nodes that influence the node of interest belong to the slice). Therefore we have to approximate the slice that influences a given node:

DEFINITION 4.5 (Weak Influence between Nodes). *Given an ART $\mathcal{G}$. Reduction n_1 weakly influences node n_2, written $n_1 \rightsquigarrow n_2$ iff*

1. *$n_2 = n_1 o$ for some $o \in \{l, r, t\}^*$, or*
2. *there exist $m, o \in \{l, r, t\}^*$ such that $m \in ancestors(n_2)$, $n_1 = mo$ and o does not start with t.*

The idea is to include all prefixes of the given node and all nodes that may be needed for the reduction step of an ancestor.

LEMMA 4.6 (Transitivity and Reflexivity of $\rightsquigarrow$). *Let $\mathcal{G}$ be an ART for start term M and program P. For nodes n_1, n_2 and n_3 in $\mathcal{G}$:*

- *$n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$ implies $n_1 \rightsquigarrow n_3$*
- *$n_1 \rightsquigarrow n_1$*

PROOF 4.7. *Reflexivity is trivial. For transitivity let $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$. We perform case analysis according to the two conditions of Definition 4.5:*

- *$n_2 = n_1 o_1$ and $n_3 = n_2 o_2$. Then $n_3 = n_1 o_1 o_2$, so $n_1 \rightsquigarrow n_3$.*
- *$n_2 = n_1 o_1$ and $n_2 = m o_2$ with $m \in ancestors(n_3)$. Either $m = n_1 p$, then $n_1 p p' = n_3$ and thus $n_1 \rightsquigarrow n_3$. Or $n_1 = mp$. Then $n_2 = mpo_1$. Because o_2 does not start with t, p cannot start with t. So $n_1 \rightsquigarrow n_3$.*
- *$n_1 = mo_1$ where $m \in ancestors(n_2)$ and $n_3 = n_2 o_2$. Because $ancestors(n_2) \subseteq ancestors(n_3)$, we have $m \in ancestors(n_3)$. So $n_1 \rightsquigarrow n_3$.*
- *$n_1 = m_1 o_1$ where $m_1 \in ancestors(n_2)$ and $n_2 = m_2 o_2$ where $m_2 \in ancestors(n_3)$. So $m_1 o' = n_2 = m_2 o_2$. Either $m_2 = m_1 o$, then $m_1 \in ancestors(n_3)$, so $n_1 \rightsquigarrow n_3$. Or $m_1 = m_2 o$. Because o_2 does not start with t, also o does not start with t. Furthermore, $n_1 = m_2 o o_1$ and $m_2 \in ancestors(n_3)$. So $n_1 \rightsquigarrow n_3$.*

DEFINITION 4.8 (Slice of Influence). *Given an ART $\mathcal{G}$ and a node $n \in \mathcal{G}$ the slice of $\mathcal{G}$ w.r.t. node n is $slice(\mathcal{G}, n) = \{m \in \mathcal{G} \mid m \rightsquigarrow n\}$.*

PROPOSITION 4.9 (Slice of Influence).
$\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$ is an ART for the same start term and program as $\mathcal{G}$.

PROOF 4.10. *According to the subsequent lemmas $slice(\mathcal{G}, n)$ is prefix-closed and for every node $nt \in \mathcal{D}om(\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\})$, the node n represents the redex and node nt approximates the reduct of the reduction step n to nt.*

LEMMA 4.11 (Prefix-closed). *Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. Then $slice(\mathcal{G}, n)$ is prefix-closed.*

PROOF 4.12. *Let us assume that $mo \in slice(\mathcal{G}, n)$, i.e., $mo \rightsquigarrow n$. Because m is a prefix of mo, it follows with the first condition in Definition 4.5 that $m \rightsquigarrow mo$. Then, by Lemma 4.6, $m \rightsquigarrow n$. So $m \in slice(\mathcal{G}, n)$.*

LEMMA 4.13 (Reductions in Slice). *Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. For any reduction step m to mt in $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$, node m represents the same redex and node mt approximates the same reduct as in $\mathcal{G}$.*

PROOF 4.14. *We show that all relevant nodes exist unchanged in $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$: From the second part of Definition 4.5 follows that all nodes reachable from m in $\mathcal{G}$ are also in $slice(\mathcal{G}, n)$. The fact that $mt \in slice(\mathcal{G}, n)$ suffices already for mt to approximate the reduct; because $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$ is a subset of $\mathcal{G}$ it must be the same reduct.*

Weak influence approximates influence:

PROPOSITION 4.15 (Influence between Nodes).
Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. Then $slice(\mathcal{G}, n) \supseteq \{m \in \mathcal{G} \mid m \rightrightarrows n\}$.

PROOF 4.16. *We prove that $m \not\rightsquigarrow n$ implies $m \not\rightrightarrows n$. According to Proposition 4.9 $slice(\mathcal{G}, n)$ is an ART for the same start term and program as $\mathcal{G}$. From $m \not\rightsquigarrow n$ follows that $m \notin slice(\mathcal{G}, n)$. However, $n \in slice(\mathcal{G}, n)$. Hence according to the definition of influence $m \not\rightrightarrows n$.*

So we will compute slices of an ART $\mathcal{G}$ that weakly influence a node n.

5. Combining algorithmic debugging and program slicing

The combination of algorithmic debugging and program slicing can lead to a much more accurate debugging session. An algorithmic debugger can interpret information from the programmer about the correctness of different parts of equations, and, by means of a program slicer, use this information to slice execution trees, thus reducing the number of questions needed to find a bug. In

this section, we show how program slicing can be combined with algorithmic debugging.

5.1 The Slicing Criterion

To define the slicing technique we need to define first a slicing criterion in this context which specifies the information provided by the programmer. During algorithmic debugging, the programmer is prompted and asked about the correctness of some series of equations such as the following:

$$\begin{array}{lcl} f_1\ v_1^1 \ldots v_i^1 & = & val_1 \\ f_2\ v_1^2 \ldots v_j^2 & = & val_2 \\ \ldots & & \\ f_n\ v_1^n \ldots v_k^n & = & val_n \end{array}$$

Then, the programmer determines for each equation whether it is correct or wrong. In particular, when the programmer identifies a wrong (sub)expression inside an equation they may produce a slice of the execution tree w.r.t. this wrong expression. Therefore, from the point of view of the programmer, a slicing criterion is a wrong (sub)expression: they should be able to specify (i.e. mark) which parts of the equations shown during algorithmic debugging are wrong. However, from the point of view of the debugger, the internal representation of the execution tree is an ART and thus a slicing criterion should be a pair of nodes $\langle m, n\rangle$ in the ART denoting where the slice starts (m) and where it ends (n).

To automatically produce a slicing criterion from the expression marked by the programmer, we use slicing patterns. The domain Pat of slicing patterns is defined as follows:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid \pi_1\pi_2$$

where $\bot$ denotes a subexpression that is irrelevant and $\top$ a relevant subexpression.

Note that a (partially or completely evaluated) most evaluated form is an instance of a slicing pattern where atoms have been replaced by $\bot$ and $\top$. Patterns are used to distinguish between correct and wrong parts of equations, and hence finding the nodes of the ART which could be the cause of the bug. For instance, once the programmer has selected some (wrong) subexpression, the system automatically produces a pattern from the whole expression where the subexpression marked by the programmer have been replaced by $\top$, and the rest of subexpressions have been replaced by $\bot$. Because algorithmic debugging finds a single bug at a time, we only consider a single expression marked in π; different wrong expressions could be produced by different bugs.

EXAMPLE 5.1. *Consider the following equation from the execution tree of Figure 1 (c):* `computs 3 = (9,9,8)`

If the user marks `8` *as wrong, the pattern generated for the expression* `(9,9,8)`*, internally represented as* `(((T·9)·9)·8)`*, would be* `(⊥· ⊤)`*, where* `T` *is the tuple constructor.*

As shown in the previous example, the pattern allows to discard those subexpressions which are correct. With the information provided by the pattern it is easy to find the nodes in the ART which are associated with wrong expressions. The following function identifies the node in the ART associated with the wrong expression marked by the programmer:

$$patNodes(\mathcal{G}, \pi, n) = \begin{cases} \{\} & \text{if } \pi = \bot \\ patNodes(\mathcal{G}, \pi, nt) & \text{if } nt \in \mathcal{G} \text{ and } \pi \neq \bot \\ patNodes(\mathcal{G}, \pi, m) & \text{if } nt \notin \mathcal{G}, \mathcal{G}(n) = m \\ \{n\} & \text{if } nt \notin \mathcal{G}, \mathcal{G}(n) \neq m \text{ and } \pi = \top \\ patNodes(\mathcal{G}, \pi_1, n_1) \cup & \text{if } nt \notin \mathcal{G}, \mathcal{G}(n) = n_1 \cdot n_2 \\ \quad patNodes(\mathcal{G}, \pi_2, n_2) & \text{and } \pi = \pi_1 \cdot \pi_2 \end{cases}$$

Given an ART $\mathcal{G}$, a pattern π for an expression M and a node n such that $mef(\mathcal{G}, n) = M$, $patNodes(\mathcal{G}, \pi, n)$ finds in the ART the node associated with the wrong expression specified in π. Then, the slicer produces a slice for this node which contains all those nodes in $\mathcal{G}$ that could influence the wrong expression.

For instance, given the equation $foo\ M = N$, the programmer should be able to specify that they want to produce a slice from either foo, some subexpression of M or some subexpression of N. In consequence, we distinguish between three possible ways of defining a slicing criterion:

- The programmer marks a subexpression of N and the associated pattern π is generated; then, the system computes the tuple $\langle m, n\rangle$ where $\{n\} = patNodes(\mathcal{G}, \pi, m)$ such that $mef(\mathcal{G}, left(m))mef(\mathcal{G}, right(m)) = foo\ M$ and, in consequence, m must be a reduced node.
- The programmer marks a subexpression of M and the associated pattern π is generated; then, the system computes the tuple $\langle \epsilon, n\rangle$ where $\{n\} = patNodes(\mathcal{G}, \pi, m)$ such that $mef(\mathcal{G}, m) = M$ is the second argument of the application $foo\ M$.
- The programmer marks the function name foo, and the system produces a tuple $\langle \epsilon, n\rangle$ where $mef(\mathcal{G}, n) = foo$ is the first argument of the application $foo\ M$.

We need to distinguish between errors in the right hand side of an equation and errors in the left hand side because the slices for them are essentially different. In the former case, we only care about new nodes, because we trust the arguments of the function call in the left hand side of the equation and, in consequence, all the nodes which were computed before this function call could not cause the bug; but, in the later case, the error could be in any previous node that *could have influenced* (rather than *had influenced*) the value of the incorrect argument or function name.

DEFINITION 5.2 (slicing criterion).
Let $\mathcal{G}$ be an ART. A slicing criterion for $\mathcal{G}$ is a tuple $\langle m, n\rangle$; such that $m, n \in \mathcal{G}$ and $n = mo$, where $o \in \{l, r, t\}^*$.

A slicing criterion points out two nodes in the graph. These nodes delimit the area of the graph where the bug must be. The objective of the slicer is to find these nodes and collect all the nodes which are weakly influenced by the first node and that weakly influence the second node. This kind of slice is known in the literature as a *chop* [8]:

DEFINITION 5.3 (Chop of Influence).
Let $\mathcal{G}$ be an ART and $\langle m, n\rangle$ a slicing criterion for $\mathcal{G}$. The slice (or chop) $\mathcal{S}$ of $\mathcal{G}$ w.r.t. $\langle m, n\rangle$ is $slice(\mathcal{G}, m, n) = \{n' \in \mathcal{G} \mid m \rightsquigarrow n'$ and $n' \rightsquigarrow n\}$.

Obviously $slice(\mathcal{G}, n) = slice(\mathcal{G}, \epsilon, n)$.

5.2 The Slicing Algorithm

Figure 3 shows the formal definition of the slicing algorithm for ARTs. For the sake of convenience, we give in this figure two versions of the algorithm. On the one hand, this algorithm is able to compute a slice from an ART. On the other hand, we present a conveniently modified version which only computes those nodes that are useful during algorithmic debugging. We first describe the algorithm to slice ARTs (for the time being the reader can ignore the underlining):

The function *collect* computes the slice $slice(\mathcal{G}, m, n)$ backwards from n to m. It collects all the nodes for which m is a prefix, and that belong to one of the following sets:

- The nodes which are in the path from m to n

$$collect(\mathcal{G},m,n) = \begin{cases} \underline{path(\mathcal{G},m,n)} & \text{if } parent(m) = parent(n) \\ & \text{or if } parent(m) \text{ and } parent(n) \text{ are undefined} \\ \{p\} \cup collect(\mathcal{G},m,p) \cup subnodes(\mathcal{G},p) & \text{otherwise, where } parent(n) = p \\ \underline{\cup\, path(\mathcal{G},pt,n)} & \end{cases}$$

$$subnodes(\mathcal{G},n) = \begin{cases} subn(\mathcal{G},m) \cup subn(\mathcal{G},o) & \text{if } \mathcal{G}(n) = m \cdot o, m = nl \text{ and } o = nr \\ subn(\mathcal{G},m) & \text{if } \mathcal{G}(n) = m \cdot o, m = nl \text{ and } o \neq nr \\ subn(\mathcal{G},o) & \text{if } \mathcal{G}(n) = m \cdot o, m \neq nl \text{ and } o = nr \\ \{\} & \text{otherwise} \end{cases}$$

$$subn(\mathcal{G},n) = \begin{cases} \{n\} \cup subnodes(\mathcal{G},n) \cup subn(\mathcal{G},nt) & \text{if } nt \in \mathcal{G} \\ \underline{\{n\} \cup}\ subnodes(\mathcal{G},n) & \text{if } nt \notin \mathcal{G} \end{cases}$$

$$path(\mathcal{G},n,n') = \begin{cases} \{n\} & \text{if } n = n' \\ \{n'\} \cup path(\mathcal{G},n,m) & \text{if } n' = mi, m \text{ is a sequence of letters and } i \text{ is a letter} \end{cases}$$

Figure 3. Function $collect$ to slice ARTs

- The ancestors of n
- All the nodes which completely evaluated the arguments of the ancestors of n (we call such set of nodes the *subnodes*).

The function $collect$ proceeds by identifying the parent of the incorrect node (i.e., the function that introduced it) and its subnodes; then, it recursively collects all the nodes which could influence the parent (all its ancestors). Functions $subnodes$ and $subn$ are used to compute all those nodes which completely evaluated the arguments of a given application node n. Finally, function $path$ computes the path between two given nodes.

To the best of our knowledge, the algorithm presented in Figure 3 is the first program slicing technique for ARTs. However, the slices it produces contain nodes which are useless for algorithmic debugging. In algorithmic debugging we are only interested in reductions, i.e., reduced nodes, because they are the only ones which yield questions during the algorithmic debugging process [2]. For this reason we also introduce a slightly modified version which only computes those nodes from the ART which are needed to produce the execution trees used during algorithmic debugging.

Firstly, only reduced nodes must be computed, thus the underlined expressions in Figure 3 should be omitted from the algorithm. And secondly, we must ensure that the produced slice is not empty. This is ensured by the fact that the slicing criterion defined for algorithmic debugging always takes a reduced node as the starting point. Therefore, the slice is never empty, because at least the (reduced) node associated with the left hand side of the wrong equation is included in the slice.

5.3 Completeness of the Slicing Algorithm

The following result states the completeness of the slicing algorithm:

THEOREM 5.4 (completeness). *Given an ART $\mathcal{G}$ and a slicing criterion $\langle m,n\rangle$ for $\mathcal{G}$, $collect(\mathcal{G},m,n) = slice(\mathcal{G},m,n)$.*

PROOF 5.5. *We prove the theorem by showing that:*

1. *$slice(\mathcal{G},m,n) \subseteq collect(\mathcal{G},m,n)$ and*
2. *$collect(\mathcal{G},m,n) \subseteq slice(\mathcal{G},m,n)$*

We consider two possible cases according to $collect$'s definition. Let us assume first that either $parent(m) = parent(n)$ or $parent(m)$ and $parent(n)$ are undefined. In this case, $collect(\mathcal{G},m,n) = path(\mathcal{G},m,n)$, thus it only collects (first rule) the nodes which are in the path from m to n.

Now, we prove that $slice(\mathcal{G},m,n) = path(\mathcal{G},m,n)$. Because m and n have the same parent, and m is a prefix of n (by Definition 5.2), we know that $n = mo$ where $o \in \{l,r\}^*$. Then, by Definitions 4.5 and 5.3 only the prefixes of n with m as prefix belong to $slice(\mathcal{G},m,n)$. These nodes form the path between m and n. Then, $slice(\mathcal{G},m,n) = path(\mathcal{G},m,n) = collect(\mathcal{G},m,n)$. Therefore, in this case both claims hold.

If, on the contrary, $parent(m) \neq parent(n) = p$ or $parent(m)$ is undefined and $parent(n) = p$, rule 2 from $collect$ is applied. In this case, all the nodes that belong to the set $\mathcal{S} = \{p\} \cup collect(\mathcal{G},m,p) \cup subnodes(\mathcal{G},p) \cup path(\mathcal{G},pt,n)$ are collected.

We prove the first claim by contradiction assuming that $\exists n' \in slice(\mathcal{G},m,n)$ and $n' \notin \mathcal{S}$. Because $n' \notin \mathcal{S}$, we know that n' is not an ancestor of n nor a subnode of an ancestor of n. Thus, by Definition 4.5, n' must be a prefix of n. Moreover, we know that n' is not in the path between m and n, thus it must be a prefix of both n and m. But this is a contradiction w.r.t. the definition of chop (Definition 5.3), because $m \rightsquigarrow n'$ and thus n' cannot be a prefix of m. Then, $slice(\mathcal{G},m,n) \subseteq collect(\mathcal{G},m,n)$

In order to prove the second claim, we assume that $\exists n' \in \mathcal{S}$ and $n' \notin slice(\mathcal{G},m,n)$. Then, either p or a node in $collect(\mathcal{G},m,p)$, $subnodes(\mathcal{G},p)$ or $path(\mathcal{G},pt,n)$ does not belong to $slice(\mathcal{G},m,n)$. First, we know that m is a prefix of n in the initial call (by Definition 5.2) and in all the recursive calls by rule 2 of $collect(\mathcal{G},m,n)$); and $parent(m) \neq parent(n) = p$ thus $n = mo$ being o a sequence of letters containing at least one t. Then, we know that $m = p$ or m is a prefix of p. Therefore $p \in slice(\mathcal{G},m,n)$ and $n' \neq p$. In addition, by rule 2 of Definition 4.5, we know that the subnodes of p weakly influence n, and they are weakly influenced by m because m is their prefix. Hence $subnodes(\mathcal{G},p) \subseteq slice(\mathcal{G},m,n)$ and thus $n' \notin subnodes(\mathcal{G},p)$. Moreover, because m is a prefix of n and by rule 1 of Definition 4.5 we have that $\forall m' \in path(\mathcal{G},m,n), m \rightsquigarrow m' \rightsquigarrow n$ and consequently $path(\mathcal{G},m,n) \subseteq slice(\mathcal{G},m,n)$, thus $n' \neq m'$. Then, n' does not belong to $collect(\mathcal{G},m,p)$, but this is a contradiction by the inductive hypothesis on the length of p.

```
Starting Debugging Session...

(1) main = False? NO
(2) sqrtest [1,2] = False? NO
(3) test (9,9,8) = False? YES (The user selects "8")
(4) computs 3 = (9,9,8)? NO
(5) comput3 3 = 8? NO
(6) listsum [6,2] = 8? YES
(7) partialsums 3 = [6,2]? NO (The user selects "2")
(8) sum2 3 = 2? NO
(9) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Figure 4. Debugging session

As discussed before, we cannot guarantee minimality (i.e., ensure that only nodes that influence the slicing criterion are collected) w.r.t. Definition 4.2, but completeness. This loss of precision is unavoidable because during algorithmic debugging we have available only one ART (the wrong computation). However, the algorithm in Figure 3 is minimal w.r.t. Definition 4.5 (i.e., ensure that only nodes that weakly influence the slicing criterion are collected). However, even if the underlying ART corresponds to a lazy evaluation language (this is the case, for instance, of a debugging session in Haskell) the slice produced is not minimal w.r.t. Definition 4.2. This phenomenon is due to *sharing*: A previously computed reduced node can appear as argument in a non-strict position of a function call. In consequence, only when the slicing criterion starts at the beginning of the computation (classical backward slicing) the slice produced is minimal, and thus it contains all and only the nodes that could influence the slicing criterion.

COROLLARY 5.6 (completeness of execution trees).
Given an execution tree $\mathcal{E}$, an equation $l = r \in \mathcal{E}$ and an expression e that is a subexpression of l or r, marked as wrong during an algorithmic debugging session, then, a slice of $\mathcal{E}$ produced by function $slice$ w.r.t. e contains the (buggy) nodes which caused e.

PROOF 5.7. *Let $\mathcal{G}$ be the ART associated with $\mathcal{E}$, and let $\langle m, n\rangle$ be the slicing criterion produced for the expression e. Let $\mathcal{E}'$ be the slice of $\mathcal{E}$ which contains the (buggy) nodes which caused e and let $\mathcal{S}$ be the ART associated to $\mathcal{E}'$. Then, the node associated with e is $n' \in \mathcal{G} \mid mef(\mathcal{G}, n') = e$, and the node associated with l is $n'' \in \mathcal{G} \mid mef(\mathcal{G}, n'') = l$. First, by Theorem 5.4, $\mathcal{S}$ is a chop of $\mathcal{G}$ w.r.t. $\langle n'', n'\rangle$, and we know that only the nodes that influenced n' could produce the bug in e; then, we want to prove that $\forall v \in \mathcal{G} \mid v \rightsquigarrow n', v \in \mathcal{S}$.*

If $e \subseteq l$ then $m = t$. If $e \subseteq r$ then $m = n''$ and by Proposition 1 in [12] the bug must be in the execution subtree that has $l = r$ as root. Therefore, the buggy node which caused e must be $v = n''o$ for any sequence o. Because $\mathcal{S}$ is a chop of $\mathcal{G}$ w.r.t. $\langle n'', n'\rangle$, in both cases, we have that $\forall v' \in \mathcal{G} \mid v' = n''o$ and $v' \rightsquigarrow n', v' \in \mathcal{S}$; and hence, by Proposition 4.15 if $v \rightsquigarrow n'$ then $v \in \mathcal{S}$.

6. The technique in practice

In this section we show an example of how to integrate our technique with an algorithmic debugger. Consider again the example program in Figure 1 (a) and its associated execution tree in Figure 1 (c). The debugging session when using our technique is depicted in Figure 4.

This debugging session is very similar to the one in Figure 1 (b). There are only two differences: The programmer is allowed to provide the system with more information and they do that in questions '3' and '7'; and there are less questions to answer, the latter being the consequence of the former.

Internally, when the programmer selects an expression, they are indirectly defining a slicing criterion. Then, the system uses this slicing criterion to slice the execution tree, thus reducing the number of questions. In the example, when the programmer selected the expression "8"[3], the system automatically removed from the execution tree eleven possible questions, all the subtrees of the equations "$comput1\ 3 = 9$" and "$comput2\ 3 = 9$" (see dark nodes on the left of Figure 1 (c)). Similarly, when the programmer selected the expression "2", the system automatically removed from the execution tree two possible questions, the subtree of the equation "$sum1\ 3 = 6$" (see dark nodes on the right of Figure 1 (c)). Note that the slicing technique is completely transparent for the user.

In general, the amount of information deleted from the tree is directly related to the size of the data structures in the equations, because when the programmer selects one subexpression, the computation of the other subexpressions can be avoided. Another important advantage of combining program slicing and algorithmic debugging is that it allows the programmer to guide the questions of the algorithmic debugger. Traditionally, the programmer had to answer the questions without taking part in the process. Thanks to the slicing mechanism, the user can select which (sub)expressions they want to inspect (i.e., because they are wrong or just suspicious), thus guiding the algorithmic debugger and producing a sequence of questions semantically related for the programmer.

For large computations a trace is huge. Hat stores it's ART on disk, so that the size of the ART is not limited by the size of the main memory. The Mercury debugger materialises parts of the execution tree at a time, by rerunning the computation. Because a slice can be a large part of the ART, collecting all its nodes could be very expensive for both implementation choices. However, for directing the algorithmic debugging process we do not need all nodes at once. We only have to determine them one at a time to determine the next question. We are in a different situation when we want to highlight in the program source the program slice associated with the trace slice. Such an operation can take time linear in the size of the trace and is thus unsuitable for an interactive tool, as the source-based algorithmic debugger of Hat demonstrates (see related work).

7. Related Work

The idea of improving algorithmic debugging by allowing the user to provide specific information about the location of errors is not new. In fact, it was introduced by Pereira almost two decades ago [18]. The original work of Pereira defined—in the context of Prolog—how an algorithmic debugger could take advantage of additional information from the user in order to reduce the number of questions. Surprisingly, posterior research in this subject practically ignored this work (6 cites in the *CiteSeer* [1] portal and only 4 cites in the *ACM* [5] portal). The first attempt to adapt the ideas of Pereira to the imperative programming paradigm was [7] by Fritzson et al. In this work, the authors propose the use of program slicing in order to reduce execution trees, thus reducing the number of questions in algorithmic debugging. Dynamic program slicing was just the technique that Pereira needed in his development, but his work was done in 1986. Even though program slicing was published in 1984 [23], dynamic program slicing was introduced only in 1988 [10].

Although the use of program slicing in algorithmic debugging was shown to be very useful, the technique by Fritzson et al. is still limited, because the user is only allowed to specify which vari-

[3] Note that, even when the equation is correct, the user can mark an expression as inadmissible, providing the system with useful information about the bug.

ables have a wrong value, but they are not allowed to specify which part of the value is wrong. This information would notably improve the effectiveness of the technique. In addition, the authors do not give details about the dynamic slicing algorithm used, so it is not clear whether the variables selected by the user must be output variables, because they only considered the use of program slicing in wrong equations (the concept of *inadmissibility* was lost [18]). Later the ideas introduced by Fritzson et al. have been applied to the logic language paradigm [17]. In particular, Kókai et al. [9] have defined the IDTS system which integrates algorithmic debugging, testing and program slicing techniques in a single framework. Our work can be seen as an adaptation of these ideas to the functional paradigm. Functional programs impose different technical challenges (i.e., generalised use of higher order functions, absence of nondeterminism and unification) which implies the use of different formalisms compared to logic programs. For instance, while ARTs directly represent the sequence of function unfoldings, which allows to trace data flows, the slicing of Prolog proof trees requires explicit annotation of the tree with information about the relations between terms. These differences produce an essentially different slicing process.

Recently, MacLarty et al. [11] have implemented an algorithmic debugger for the functional logic language Mercury which allows the user to specify which parts of an equation are wrong (including subterms). In particular, they have defined a strategy called "*Subterm Dependency Tracking*" which, given a wrong term, finds the 'origin' of this subterm in the execution tree. This information is used by the tool for determining which questions are more likely to be wrong, thus improving the algorithmic debugging process. Although this process can be considered as a kind of program slicing, they haven't defined it formally, and, for the case of wrong input arguments, the method has only been proposed but not implemented nor proved correct.

The source-based trace explorer of Hat [2] implements algorithmic debugging and can display a simple program slice that contains the bug location and that shrinks when more questions are answered. However, the user cannot indicate wrong subexpressions and the slice does not direct the algorithmic debugging process.

To the best of our knowledge, our technique is the first formal approach that 1) is able to slice execution trees for either input or output expressions, 2) copes with complete expressions and subexpressions and 3) is proven complete. We consider functional programs, a paradigm that completely lacked debugging techniques combining algorithmic debugging and program slicing.

8. Conclusions

We have presented a technique to improve algorithmic debugging by combining it with program slicing. Firstly, we have adapted some program slicing concepts (and proven some of their properties) to the context of Augmented Redex Trails. Based on this adaptation we have introduced an algorithm to slice ARTs and proven its completeness. The algorithm is generic enough to be used in different program slicing based applications. We have slightly modified the slicing algorithm for the case of algorithmic debugging in order to optimise the number of nodes in slices so that only profitable nodes to build execution trees are included in the slice. With the information provided by the user the introduced algorithm produces a slice which only contains the relevant parts of the execution tree, thus reducing the number of questions. However, this technique could be further improved by providing not only the relevant nodes from the execution tree, but also considering *how* relevant they are. As future work, we plan to extend our technique in this direction. We want to investigate the possibility of automatically generating heuristics from a given slicing criterion based on the probability of equations being wrong. These heuristics could help the algorithmic debugger to select the order of questions depending on which of them are more likely to be wrong. We are currently working on a prototype in Haskell [19], based on the Haskell tracer Hat [22] (http://www.haskell.org/hat/).

References

[1] Citeseer: The NECI Scientific Literature Digital Library. Accessed on March 27th 2006. URL: http://citeseer.ist.psu.edu.

[2] O. Chitil. Source-Based Trace Exploration. In *16th Int'l Workshop on Implementation of Functional Languages (IFL 2004)*. Springer LNCS, 2005.

[3] O. Chitil and Y. Luo. Proving the Correctness of Declarative Debugging for Functional Programs. In *Trends in Functional Programming (TFP 2006)*, 2006.

[4] O. Chitil and Y. Luo. Towards a Theory of Tracing for Functional Programs Based on Graph Rewriting. In *TERMGRAPH 2006*, 2006.

[5] The ACM Digital Library. Association for Computing Machinery. Accessed on March 27th 2006. URL: http://portal.acm.org.

[6] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[7] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.

[8] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical Report CS-94-169, 1994.

[9] Gabriella Kokai, L Harmath, and Tibor Gyimothy. Algorithmic debugging and testing of prolog programs. In *Workshop on Logic Programming Environments*, pages 14–21, 1997.

[10] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[11] I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.

[12] Lee Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[13] Lee Naish. A three-valued declarative debugging scheme. In *Workshop on Logic Programming Environments*, pages 1–12, 1997.

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

[15] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[16] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press, 2004.

[17] J. Paakki, T. Gyimóthy, and T. Horváth. Effective algorithmic debugging for inductive logic programming. volume 237 of *GMD-Studien*, pages 175–194. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[18] L. M. Pereira. Rational Debugging in Logic Programming. In *Proceedings on Third international conference on logic programming*, pages 203–210, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[19] S. L. Peyton-Jones, editor. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, 2003.

[20] E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[21] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

[22] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.

[23] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

A. Augmented Redex Trail

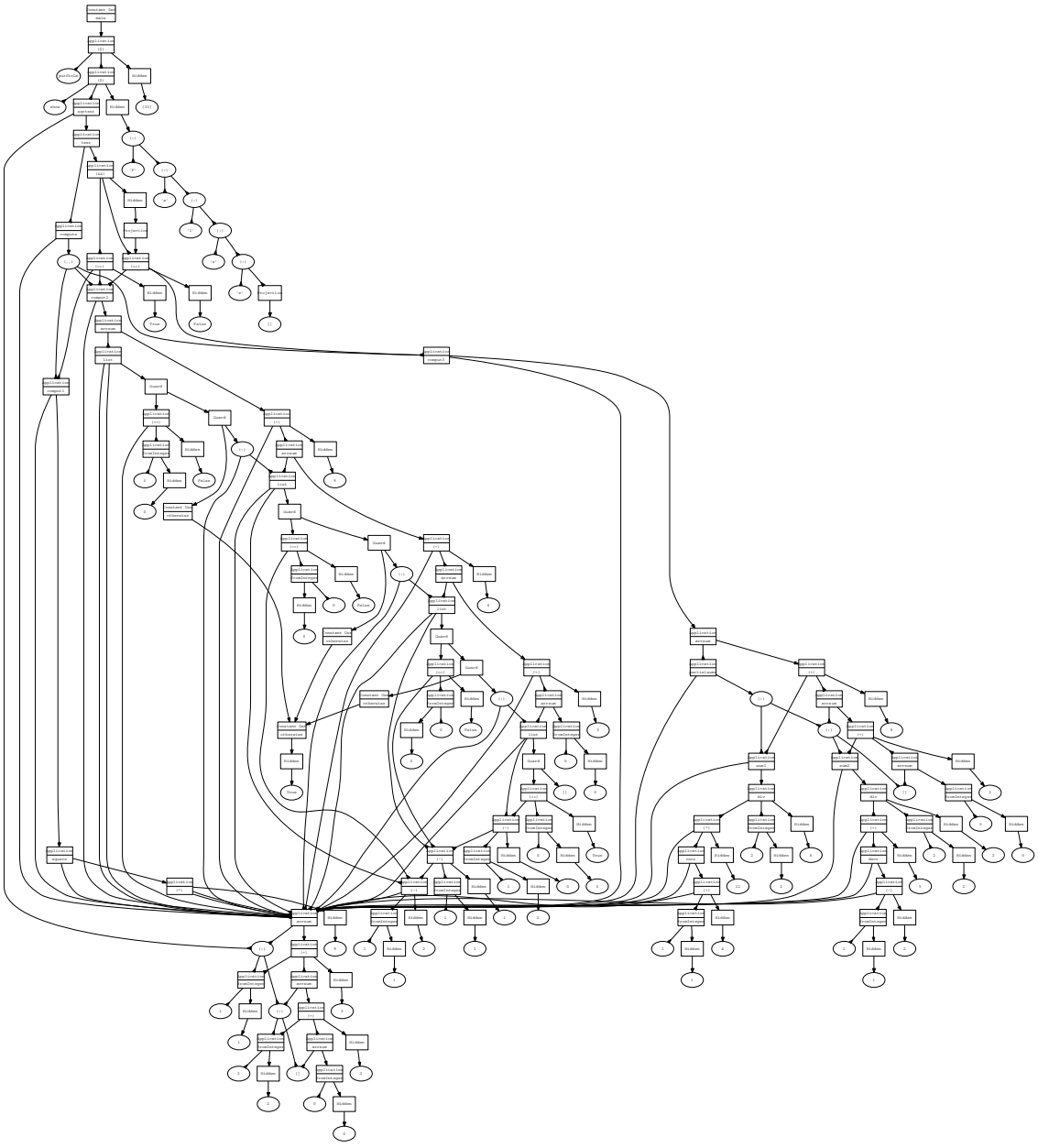


Figure 5. ART of the program in Figure 1 (a)