$\mathcal{DDT}$: A Declarative Debugging Tool for Functional-Logic Languages

Rafael Caballero and Mario Rodríguez-Artalejo

Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid
e-mail: {rafa,mario}@sip.ucm.es *

Abstract. We present a graphical tool for the declarative debugging of wrong answers in functional-logic languages. The tool, integrated in the system $\mathcal{TOY}$, can be used to navigate a computation tree corresponding to some erroneous computation. During the navigation the users can either follow a fixed strategy or move freely providing information about the validity of the nodes as they like. We show by means of examples how this flexibility can reduce both the number and the complexity of the questions that the user must consider w.r.t. the usual top-down navigation strategy. Moreover, the tool includes some extra features that can be used to automatically simplify the computation trees.

1 Introduction

The idea of *declarative debugging* was first proposed by E.Y. Shapiro [18] in the field of Logic Programming, and has been developed not only in this paradigm [11, 7] but also in constraint-logic programming [20], functional programming [15, 14, 17], and functional-logic programming [5, 4, 6].

The overall idea in all the cases is the same, as pointed out by Lee Naish in [13]. The debugging starts when some erroneous computation, the so-called *initial symptom*, is found, and can be described as a two stages process:

- First, the declarative debugger builds a suitable *computation tree* for the initial symptom. Each node in this tree keeps the result of some subcomputation and can be associated to the fragment of code responsible for it. In particular, the root represents the (wrong) result of the main computation. The children of a given node must correspond to the intermediate subcomputations needed for obtaining the result at such node. This phase is automatically performed by the debugger; the user's assistance is only required to detect the initial symptom.

- Once the tree is obtained it is *navigated* looking for a *buggy* node, i.e. a node with an erroneous result whose children nodes have correct results. Such node is associated to a fragment of code that has produced an erroneous output from correct inputs, and will be pointed out by the debugger as one bug in the program. In order to check whether the nodes are correct or not, some external *oracle*, generally the user, is needed.

* Work partially supported by the Spanish CYCIT (project TIC2002-01167 'MELODIAS').

One of the main criticisms about the use of this technique w.r.t. other debugging methods such as abstract diagnosis [2, 1], is the amount and complexity of the questions that the user must answer during the navigation phase. The problem has been considered in (constraint) logic programming [18, 20], but has received little attention in functional and functional-logic programming, where most of the works are devoted to the definition of suitable computation trees and to devise mechanisms for their implementation. The declarative debuggers proposed in functional and functional-logic programs, to the best of our knowledge, perform a top-down traversal of the tree during the navigation phase. As we will see this is only satisfactory for trees containing a buggy node near the root; otherwise the number and complexity of the questions can make the debugging process unrealistic.

In this paper we $\mathcal{DDT}$ (an acronym for $\mathcal{D}$eclarative $\mathcal{D}$ebugging $\mathcal{T}$ool), a graphical declarative debugger included as part of the lazy functional-logic system $\mathcal{TOY}$ [12]. However, the ideas and techniques presented here are also valid for the declarative debugging of wrong answers in other lazy functional-logic languages such as Curry [10] or lazy functional languages as Haskell [16].

$\mathcal{DDT}$ allows the user either to navigate freely the computation tree or to select one of the default strategies provided by the tool to guide the navigation. In the paper we show how these possibilities can be used to reduce both the number and the complexity of the questions that the user must consider during the debugging process. Moreover, $\mathcal{DDT}$ also incorporates two techniques for simplifying the computation tree, the first one based on the notion of *entailment* proposed in [6], and the second one based on the use of another program as a (generally partial) correct specification of the intended program semantics. These two features can be used to determine the validity of some nodes of the computation tree in advance, thus simplifying the role of the user during the navigation phase.

The structure of the paper is as follows: next Section introduces some preliminary concepts and presents the general aspect of the tool. Section 3 explains by means of an example how the flexibility of the navigation in $\mathcal{DDT}$ can be used to detect buggy nodes more easily. Section 4 discusses the strategies provided by the system, while Section 5 presents the two techniques used to simplify the computation tree mentioned above. Finally Section 6 concludes and points to some planned future work.

$\mathcal{DDT}$ is part of the distribution of the $\mathcal{TOY}$ system, which is available at `http://titan.sip.ucm.es`.

2 Initial Concepts

As we said in the previous section, $\mathcal{DDT}$ is integrated in the lazy FLP language $\mathcal{TOY}$ [12]. In this section we first recall some basics about the language and illustrate them with an example. Then some basic notions and properties regarding computation trees are presented.

2.1 The $\mathcal{TOY}$ language

Programs in $\mathcal{TOY}$ can include data type declarations, type alias, infix operators declarations, function type declarations, and defining rules for functions symbols. Before describing the structure of the defining rules we must define some initial notions such as expressions and patterns. A more detailed description of the syntax of the language can be found in [12].

The syntax of *partial expressions* $e \in Exp_\perp$ is $e ::= \perp \mid X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). In the rest of the paper the notation $e\ e_1\ e_2 \ldots e_n$ (or even $e\ \overline{e}_n$) is used as a shorthand for $(\ldots((e\ e_1)\ e_2)\ldots)e_n)$. Similarly, the syntax of *partial patterns* $t \in Pat_\perp \subset Exp_\perp$ can be defined as $t ::= \perp \mid X \mid c\ t_1\ \ldots\ t_m \mid f\ t_1\ \ldots\ t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are partial patterns for all $1 \le i \le m$. We define the *approximation ordering* $\sqsubseteq$ as the least partial ordering over $Pat_\perp$ satisfying the two following properties:

- $\perp \sqsubseteq t$, for all $t \in Pat_\perp$.
- $h\ \overline{t}_m \sqsubseteq h\ \overline{s}_m$ if $h\ \overline{t}_m, h\ \overline{s}_m \in Pat_\perp$ and $t_i \sqsubseteq s_i$ for all $1 \le i \le m$.

Expressions and patterns without any occurrence of $\perp$ are called *total*.

The defining rules for a function f are composed of a *left-hand side*, a *right-hand side*, an optional *condition*, and some optional *local definitions*:

$$(R) \quad \underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \;=\; \underbrace{r}_{\text{right-hand side}} \;\Leftarrow\; \underbrace{C}_{\text{condition}} \quad where \quad \underbrace{LD}_{\text{local definitions}}$$

the condition has the form $C \equiv e_1 == e'_1, \ldots, e_k == e'_k$, while the local definitions are $LD \equiv \{s_1 \leftarrow a_1;\ \ldots;\ s_m \leftarrow a_m\}$, where e_i, e'_i, a_i and r are total expressions, the t_j, s_j are total patterns with no variable occurring more than once in different t_k, t_l or in different s_k, s_l, and no variable in s_i occurring in a_j for $1 \le j < i \le m$. Roughly, the intended meaning of a program rule like (R) is that a call to the function f can be reduced to r whenever the actual parameters match the patterns t_i, using the local definitions LD, and ensuring that the conditions $e_i == e'_i$ are satisfied. A condition $e == e'$ is satisfied by evaluating e and e' to some common total pattern.

A formal semantic calculus for $\mathcal{TOY}$ programs is described in [8, 9], and has been adapted to the declarative debugging of wrong answers in [5, 6]. As we proved in [5, 6] a simplification of the proof trees in this semantic calculus can be employed as suitable computation trees for the declarative debugging of wrong answers in lazy functional-logic languages. The nodes of such computation trees are always *basic facts* of the form $f\ t_1 \ldots t_n \to t$, with f a function symbol of arity n and $t, t_1, \ldots t_n \in Pat_\perp$. The idea is that a basic fact $f\ t_1 \ldots t_n \to t$ can be proved w.r.t. some program P iff the pattern t approximates the value of the function call $(f\ t_1 \ldots t_n)$ in P. Since the value $\perp$ approximates all the values

```
fib                     = [1, 1 | fibAux 1 1]
fibAux N M              = [N+M | fibAux N (N+M)]

goldenApprox            = (tail fib) ./. fib

infixr 20 ./.
[X | Xs]  ./. [Y | Ys]  = [ X/Y | Xs ./. Ys]

tail [X|Xs]             = Xs

take 0 L                = []
take N [X|Xs]           = [X| take (N-1) Xs] <== N>0

main R                  = true <== take 5 goldenApprox == R
```

Fig. 1. Approximating the Golden Ratio

in our semantics, trivial basic facts of the form $f\ t_1 \dots t_n \to \bot$ always can be proved. In [5, 6] we also proved that any buggy node in these computation trees its associated with some *erroneous function rule* of the program, in fact with the function rule used to prove the basic fact labelling the node. The *intended model* of a program P is the set $\mathcal{I}$ of basic facts that the user expects to be provable w.r.t. P. A node of the computation tree is correct if its basic fact belongs to $\mathcal{I}$, and incorrect otherwise. Correct nodes are also called valid w.r.t. $\mathcal{I}$.

We will assume that every program P includes a special program rule *main* of the form $main\ X_1\ \dots X_k\ = true\ <==\ C$, where $\{X_1, \dots, X_k\}$, $k \geq 0$, is the set of variables occurring in the condition C. The system will compute substitutions σ, called *answers*, of patterns for variables such that $dom(\sigma) \subseteq \{\ \mathrm{X}_1,\ \dots,\ \mathrm{X}_k\}$, meaning that the basic fact $(main\ X_1 \dots X_k)\sigma \to true$ can be proved w.r.t. P. Notice that this notion of goal, suitable for this work, is compatible with actual goals in $\mathcal{TOY}$ which are of the form: $e_1 == e'_1, \dots, e_k == e'_k$, simply by assuming that the goal is the condition of an implicit program rule for *main*.

2.2 An Example

Figure 1 shows a $\mathcal{TOY}$ program whose purpose is to approximate the number $\frac{1+\sqrt{5}}{2}$, known as the *golden ratio*, by using the Fibonacci sequence $1,\ 1,\ 2,\ 3,\ 5, \dots$, where each term in the sequence (after the second) is the sum of the two that immediately precede it. If we call $fib(i)$ to the i-th term of this sequence, the following property holds:

$$\lim_{n\to\infty} \frac{fib(n+1)}{fib(n)} = \frac{1+\sqrt{5}}{2}$$

The program contains a function fib that represents an infinite list containing the Fibonacci sequence. This function uses an auxiliary function fibAux. Given two integer values X_0 and X_1, the function call fibAux X_0 X_1 is expected to compute the infinite list [X_2, X_3, ...], such that $X_k = X_{k-2} + X_{k-1}$, for any $k \geq 2$. Function goldenApprox computes the infinite list [fib(2)/fib(1), fib(3)/fib(2), ...] using the infix operator ./., that returns the result of dividing two infinite lists term by term. The meaning of the rest of the functions should be clear from the context. Some basic facts included in the intended model $\mathcal{I}$ of the program are:

$$\mathcal{I} = \{ \ldots, \text{fib} \rightarrow \bot, \ldots, \text{fib} \rightarrow [1 \mid \bot], \ldots, \text{fib} \rightarrow [1,1,2,3,5,8 \mid \bot], \ldots,$$
$$\ldots, \text{fibAux } 1\ 1 \rightarrow [2,3,5, \mid \bot], \ldots, \text{fibAux } 10\ 20 \rightarrow [30,50,80,120 \mid \bot], \ldots,$$
$$\ldots, \text{main } [1,2,1.5,1.66,1.6] \rightarrow \text{true}, \ldots \}$$

among others. In particular, the basic fact main $[1, 2, 1.5, 1.66, 1.6] \rightarrow$ true is expected since [1/1, 2/1, 3/2, 5/3, 8/5] = [1, 2, 1.5, 1.66, 1.6] (rounding to two decimals for simplicity) is the list of the fifth first approximations to the golden ratio using the Fibonacci sequence. However, the system computes the answer $\sigma = \{\text{R} \mapsto [1, 2, 1.5, 1.33, 1.25\ ]\}$. This is a *wrong answer*, since main R$\sigma \rightarrow$ true is not in the intended model of the program, and constitutes the initial symptom showing that there is some bug in the program.

The computation tree for this wrong computation can be seen in Figure 2, as displayed by $\mathcal{DDT}$. The root of the tree corresponds to the initial symptom and the children of each node correspond to the function calls needed for computing the basic fact at the node. For instance the root has two children:

(1) goldenApprox $\rightarrow [1, 2, 1.5, 1.33, 1.25|_\ ]$
(2) take 5 $[1, 2, 1.5, 1.33, 1.25|_\ ] \rightarrow [1, 2, 1.5, 1.33, 1.25\ ]$

corresponding to the two function calls in the condition of main instantiated with the values used during the computation. The character _ in the display represents the symbol $\bot$, and stands in place of some value whose evaluation was not needed during the computation. The basic fact (2) is valid in the intended model, but the basic fact (1) is not, since the fourth and fifth members of the list at the right-hand side of the basic fact should be $\frac{fib(5)}{fib(4)} = 1.66$ and $\frac{fib(6)}{fib(5)} = 1.6$ respectively.

In the debugging session of the figure we have provided information about the validity of all the nodes, although usually this is not necessary as we will see in sections 3 and 4. At the bottom of the display $\mathcal{DDT}$ shows data about the amount of different kinds of nodes, including *unknown* nodes, corresponding to basic facts whose validity has not yet determined, and *trusted* nodes, which correspond to basic facts associated to trusted functions. In this example the user has decided that functions take and tail are trusted and hence all the basic facts corresponding to calls of these functions will be considered valid by the debugger. The computation tree has two buggy nodes (one appears selected in the figure), both of them corresponding to the application of the single function rule for fibAux, which is therefore a wrong rule and will be pointed out by the

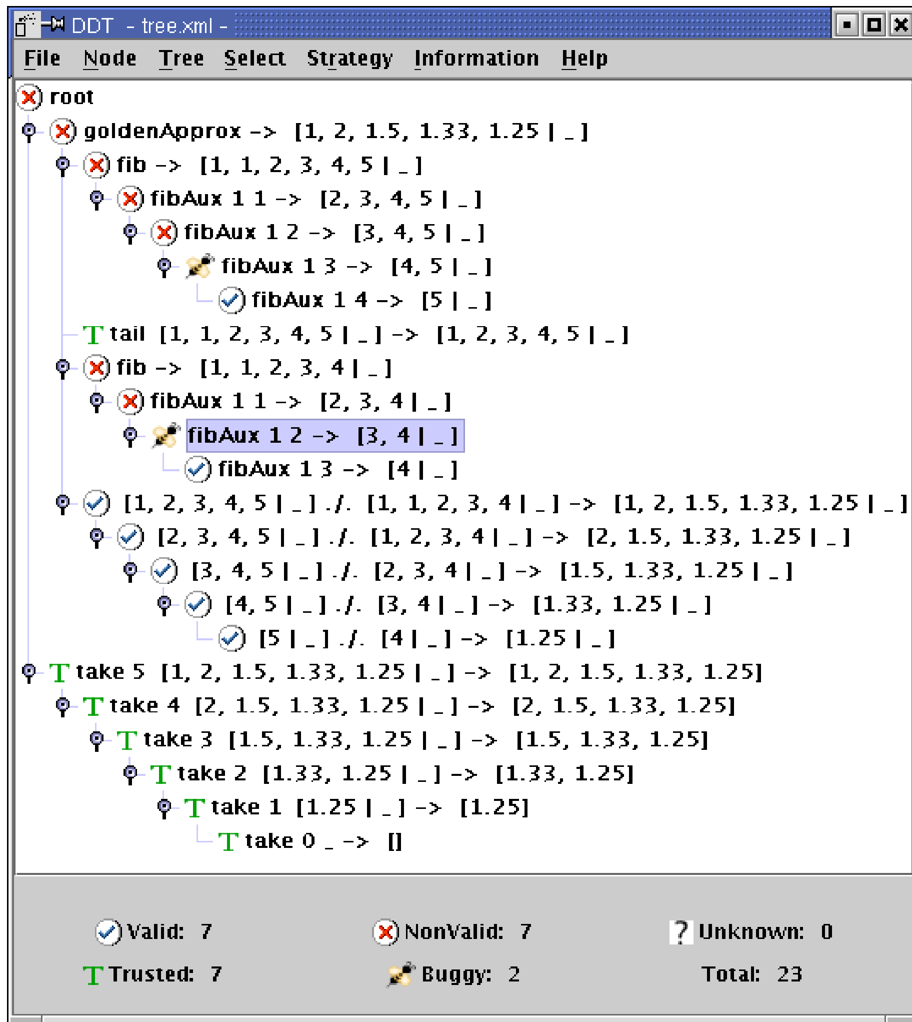


Fig. 2. Computation Tree for the program of Figure 1

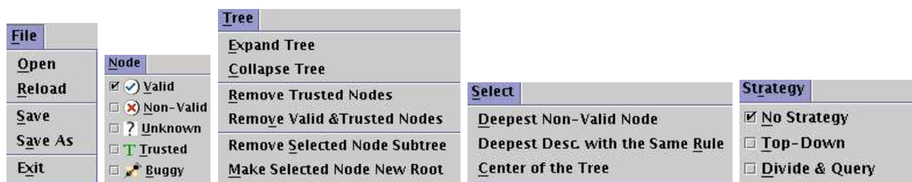


Fig. 3. Some menu options in $\mathcal{DDT}$

debugger as the cause of the bug. The bug in this rule is in the first argument of the recursive call, that should be M instead of N. The correct rule is then: fibAux N M = [N+M | fibAux **M** (N+M)].

2.3 Computation Trees

Next we present some definitions and auxiliary notation about computation trees (shortly CT's). Notice that although some basic facts can occur repeatedly in a CT, any node can be identified by its path to the root. Given a CT T and a node $N \in T$, we will use the notation $root(T)$ to represent the root of the tree, $subtree(T, N)$ for the subtree of T whose root is N, and $children(T, N)$ to represent a list with the children nodes of N in T. If N is valid w.r.t. the intended interpretation $\mathcal{I}$ we will write $valid(N)$, assuming that $\mathcal{I}$ is clear from the context. Analogously $nonvalid(N)$ will represent a non-valid node, while $buggy(T, N)$ will mean that $nonvalid(N)$ and $valid(N')$ for all $N' \in children(T, N)$. Finally, $buggy(T)$ will indicate that there exists a node $N \in T$ such that $buggy(T, N)$.

The number of nodes in a computation tree T will be represented as $|T|$. Let N be a node in T such that $N \neq root(T)$, and let P be the parent of N in T. Then the notation $T - N$ represents the new tree obtained by removing N from T and letting the children of N become children of P. Hence, if $N_1, \ldots, N_{i-1}, N, N_{i+1}, \ldots N_m$ are the children of P in T and $N'_1, \ldots, N'_m$ are the children of N in T, the children of P in $T - N$ will be $N_1, \ldots, N_{i-1}, N'_1, \ldots, N'_m, N_{i+1} \ldots N_m$. With these definitions two interesting properties of computation trees can be proved. Given a computation tree T:

- P1 If $N \in T$ and $nonvalid(N)$, then there is some node $N' \in subtree(T, N)$ such that $buggy(T, N')$ and the path from N to N' only has non-valid nodes.
- P2 if $N \in T$ and $valid(N)$, then $buggy(T)$ iff $buggy(T - N)$, and for every $N' \in T - N$ such that $buggy(T - N, N')$, $buggy(T, N')$ holds.

At several places in the rest of the paper we will use these properties for justifying the correctness of various $\mathcal{DDT}$ features.

3 Free Navigation

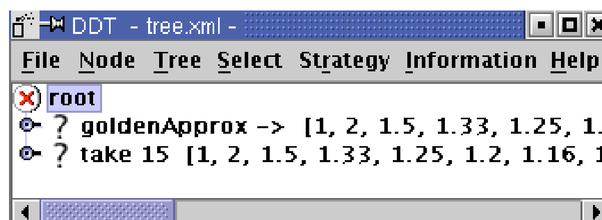
The $\mathcal{TOY}$ system includes currently two declarative debugging navigators: a textual top-down navigator similar to those of *Buddha* [17] and *Freja* [14], and the graphical navigator $\mathcal{DDT}$. This section shows by means of an example how the flexibility allowed by $\mathcal{DDT}$ can reduce the number and complexity of the nodes that the user must examine in comparison to the top-down navigators.

Let us consider again the program of Figure 1, but replacing the rule of the function main by main R = true <== take 15 goldenApprox == R. The answer computed by the system is again wrong and therefore the declarative debugger can be employed. By using the top-down navigator of $\mathcal{TOY}$ we can obtain the following debugging session, where we have replaced part of the lists by dots for the sake of saving space in this presentation.

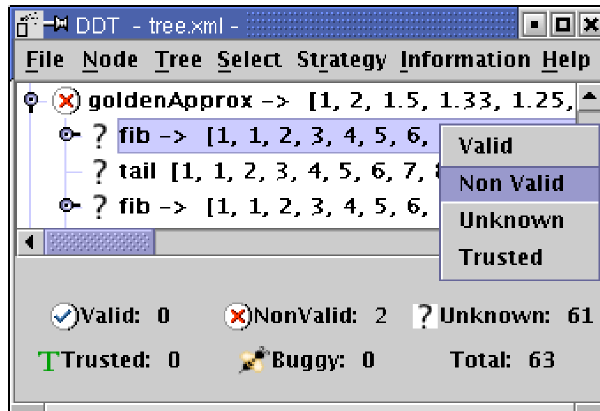
```
Consider the following facts:
1: goldenApprox → [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ]
2: take 15 [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ] → [1, 2, 1.5, 1.33, 1.25, 1.2, ...]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
2: tail [1, 1, 2, 3, 4, 5, ... | _ ] → [1, 2, 3, 4, 5, ... | _ ]
3: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 | _ ]
4: [1, 2, 3, 4, ... | _ ] ./. [1, 1, 2, 3, ... | _ ] → [1, 2, 1.5, 1.33, 1.25, ... | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fibAux 1 1 → [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 2 → [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
..... (12 similar questions more involving fibAux )
Rule number 1 of the function fibAux is wrong.
```

We have omitted 12 additional questions about the validity of basic facts involving fibAux. Even assuming that the user knows the answer to these questions, the process can be a bit boring. With greater values in the argument of take the use of the top-down textual navigator will become unrealistic. It could be argued that the user should stop after a few questions about fibAux and try some easier goal, but in general is not always feasible to replace the goal which has produced an error symptom by a simpler one.

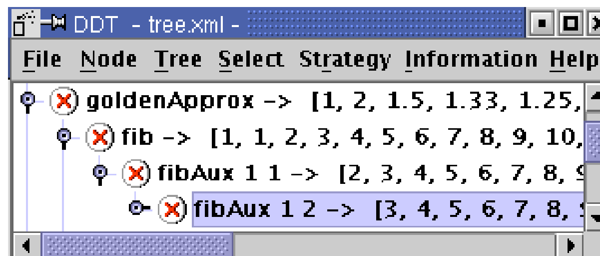
Let us examine now a possible debugging session for the same program using $\mathcal{DDT}$. The display is not completely shown in some of the images due to the lack of space. In the initial state of the debugger only the first level of the tree, with the two children of the root, is expanded:



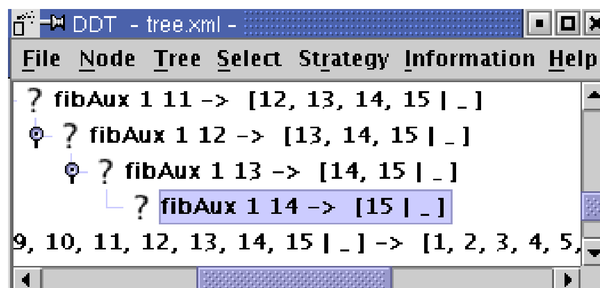
The user observes that the node corresponding to goldenApprox is not valid, changes its state to *nonvalid* and expands the node to examine its children. The state of a node can be changed in $\mathcal{DDT}$ either by using the option menu "Nodes" (see Figure 3), or by right-clicking over the node and selecting the state from the menu that appears, as shows the next image:



The next image shows the debugging session after descending in this way four levels in the tree:



At this point the user, maybe getting bored, can check that the next nodes in the same path correspond to the function fibAux, and seem to be non-valid. Then the option "Deepest Descendant with the Same Rule" of the menu "Select" can be used. This option looks automatically for the deepest use of fibAux in the subtree whose root is selected. In this example, the system finds out and selects the node containing the basic fact fibAux 1 14 $\rightarrow$ [15 | _]:



The user can check readily that this node is valid and change its state consequently. Moving now bottom-up, the user detects that the parent of the selected node contains the non-valid basic fact fibAux 1 13 $\rightarrow$ [14,15 | _] (the second element of the list should be $14 + 13 = 27$ instead of 15) and changes its state

accordingly. In this moment $\mathcal{DDT}$ detects that this node is buggy and shows an message reporting to the user that the only program rule for fibAux, used at the buggy node, has been detected as incorrect.

Due to the recursive structure of function definitions, the option "Deepest Descendant with the Same Rule" will often render a new current node N whose basic fact is smaller and thus simpler to analyze. Navigation can then proceed by moving down to N's children in case that N is non-valid, and moving up to N's ancestors otherwise. In both cases, property P1 guarantees that a buggy node can be eventually found, because N is known to have an invalid ancestor.

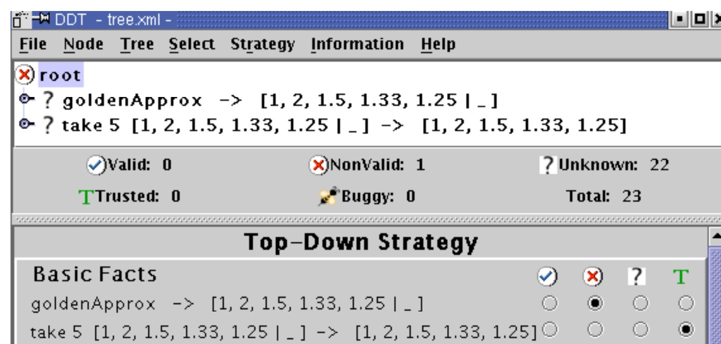
Of course the textual debugger could be enhanced with more sophisticated options, similar to those of $\mathcal{DDT}$ mentioned here. However we think that the graphical displaying provides a more general perspective of the CTs that allows, in many cases, a quicker detection of the buggy nodes.

4 Strategy-guided Navigation: *Top-down* Versus *Divide-and-query*

Although free navigation can be used to reduce the number of nodes considered during the debugging process, $\mathcal{DDT}$ also includes the possibility of using a strategy-guided navigation and provides two possibilities: the *top-down* and the *divide-and-query* strategies.

The top-down strategy behaves essentially like the textual debugger presented in the previous section. The process starts with a computation tree whose root is considered non-valid. Then the children of the root are examined looking for some non-valid child. If such child is found the debugging continues examining its corresponding subtree. Otherwise all the children are valid and the root of the tree is pointed out as buggy, finishing the debugging.

The next display shows the starting point of the a debugging session using the top-down strategy, where the user has marked the first node as non-valid and the second one as trusted:



Notice that after each subsequent step the selected subtree has a smaller size and an invalid root. Hence, as a consequence of property P1, a buggy node is eventually reached.

	G_1	G_2	G_3	G_4	G_5	G_6
Nodes	403	6725	963	600	257	731
Top-Down	102 (106)	83 (165)	3 (3)	102 (204)	10 (33)	39 (71)
Divide-and-Query	10 (10)	13 (13)	10 (10)	10 (10)	8 (8)	7 (7)

Fig. 4. Number of steps and nodes examined with the two strategies

The divide-and-query strategy was presented in [18] and has been included also in the system *TkCalypso* [20]. As in the top-down strategy, debugging starts with a computation tree whose root is not valid. The idea is to choose a node N such that the number of nodes inside and outside of the subtree rooted by N are the same. Although such node (called the *center* of the tree) does not exist in most of the cases, the system looks for the node that better approximates the condition. Then the user is queried about the validity of the basic fact labelling this node. If the node is non-valid its subtree will be considered at the next step. If it is valid then its subtree is deleted from the tree and debugging continues. The process ends when the subtree considered has been reduced to a single non-valid node.

Is easy to observe that, as in the top-down strategy, the number of nodes in the tree T considered is reduced after each step, and that $nonvalid(root(T))$ holds. To check that the strategy will find some buggy node we must examine the two actions that it can perform depending on the validity of the selected node N. First, if $nonvalid(N)$, substituting the whole tree by $subtree(T, N)$ is safe due to property P1. If $valid(N)$ we must ensure that the deleting of $subtree(T, N)$ will not delete all the buggy nodes. This holds again by Property P1, since the tree must have a buggy node B with a path of non-valid nodes from $root(T)$ to B. Therefore N cannot be part of this path and B is not in $subtree(T, N)$.

Since these strategies modify the structure of the tree, $\mathcal{DDT}$ includes options to save and load computation trees (see the options of the menu "File"). The files are stored in XML format. These options can be also used to restore a previous version of the debugging session if the user realizes after some steps that she or he made a mistake when providing information about the validity of the nodes, a situation that often arises.

Figure 4 shows a comparison of the number of steps and the number of nodes examined (between round brackets) during some debugging sessions with the two strategies. The first row of the table shows the total number of nodes of the computation tree considered in each example. Goal G_1 corresponds to our example of Figure 1 taking the 100 first approximations of the golden ratio. Goal G_2 uses a buggy program for computing prime numbers presented in [6]. G_3 uses a program with arithmetic in Peano's representation. G_4 corresponds to a program for sorting numbers using a functional-logic programming technique called "lazy generate-and-test" in [8]. Goal G_5 uses a program for the symbolic derivation of expressions, while G_6 corresponds to a program implementing a queue.

The table shows a clear advantage of the divide-and-query strategy w.r.t. the top-down strategy. In general, the number of steps in a debugging session with a CT of size n is $O(log\ n)$ when using the divide and query and $O(n)$ when using the top-down strategy. However, these are worst-case estimations. The top-down strategy can behave more efficiently whenever there is a buggy node close to the root as it is the case for goal G_3.

The source code $\mathcal{DDT}$ consists of 2900 lines of Java code. We have used the Java language for two reasons: firstly, Java provides several libraries for designing graphical interfaces, and in particular some specific classes for representing trees graphically, and secondly because the Prolog system in which $\mathcal{TOY}$ is based, SICStus Prolog [19], includes an interface for interacting with Java.

5 Simplification of Computation Trees

Next we present two techniques incorporated in $\mathcal{DDT}$ that can provide automatically information about the validity of some nodes in the computation tree.

5.1 Entailment

In [5, 6] we presented an *entailment* relation between basic facts based on the *approximation ordering* $\sqsubseteq$ defined in Section 2.1. A basic fact $f\ \bar{t}_n \rightarrow t$ *entails* another basic fact $f\ \bar{s}_n \rightarrow s$ (written as $f\ \bar{t}_n \rightarrow t\ \succeq\ f\ \bar{s}_n \rightarrow s$) iff there is some total substitution $\theta \in Subst$ such that $t_1\theta \sqsubseteq s_1, \ \dots\ , t_n\theta \sqsubseteq s_n, s \sqsubseteq t\theta$.

Notice that the entailment property is covariant in the arguments but contravariant in the result. For instance, considering the basic facts of Figure 2, it can be easily proved that

$$\mathsf{fib} \rightarrow [1,1,2,3,4,5 \mid \bot\,] \succeq \mathsf{fib} \rightarrow [1,1,2,3,4 \mid \bot\,]$$
$$\mathsf{fibAux}\ 1\ 2 \rightarrow [3,4,5 \mid \bot] \succeq \mathsf{fibAux}\ 1\ 2 \rightarrow [3,4 \mid \bot]$$

with θ as the identity substitution in both cases. As shown in [6], entailment between basic facts is a decidable relation, and intended program models are closed under entailment, i.e. if $f\ \bar{t}_n \rightarrow t\ \succeq\ f\ \bar{s}_n \rightarrow s$ and $(f\ \bar{t}_n \rightarrow t) \in \mathcal{I}$ then $(f\ \bar{s}_n \rightarrow s) \in \mathcal{I}$. This means that if $f\ \bar{t}_n \rightarrow t$ is valid then $f\ \bar{s}_n \rightarrow s$ will be also valid, and conversely that if $f\ \bar{s}_n \rightarrow s$ is not valid then $f\ \bar{t}_n \rightarrow t$ is not valid.

$\mathcal{DDT}$ uses this property for changing automatically the state of some nodes when the user provides information about others. For instance when the node containing fib $\rightarrow$ [1,1,2,3,4,5 | $\bot$] is marked as *valid*, the state of the node containing fib $\rightarrow$ [1,1,2,3,4 | $\bot$] will be changed accordingly to *valid*, while marking the node corresponding to fibAux 1 2 $\rightarrow$ [3,4 | $\bot$] as *nonvalid* will automatically change to *nonvalid* of the state of the node containing fibAux 1 2 $\rightarrow$ [3,4,5 | $\bot$].

Every basic fact obviously entails itself. Therefore, any user-given change in the state of a node propagates automatically to all the other nodes containing the same basic fact.

5.2 Trusted Specifications

As explained in previous sections, $\mathcal{DDT}$ users can mark some CT nodes as *trusted* during a debugging session. All the nodes whose basic facts correspond to the function used at the trusted node are automatically considered as valid. A more automatic way of declaring trusted functions is by means of trusted specifications. This idea is not new and was introduced in the seminal work of E.Y. Shapiro [18].

We say that a $\mathcal{TOY}$ program S is a *trusted specification* if the user assumes as valid all the basic facts that can be derived from S. Let P be a buggy program, T a CT corresponding to some initial symptom for P, and S_P some trusted specification for *some* of the functions occurring in P. Then the following procedure can be used to provide some information about the validity of the nodes in T:

For each basic fact $\varphi : f\ \bar{t}_n \rightarrow t$ *labelling some node* N *of* T*:*
If valid?$(S_p, \varphi) =$ *yes* *then delete* N*.*
If valid?$(S_p, \varphi) =$ *no* *then mark* N *as non-valid.*
If valid?$(S_p, \varphi) =$ *don't-know then mark* N *as unknown.*

$\mathcal{DDT}$ incorporates an algorithm for computing *valid?*(S_p, φ) in a correct way. If the result is *yes* then φ can be derived from S_p and deleting N is safe because of Property P2. If the result is *no* then φ cannot be derived from S_p and marking N as non-valid is correct. Otherwise N is marked as unknown. There are two possible situations where the algorithm returns *don't-know*:

- If the function used in φ is not defined in S_p.
- If the time required for deciding if φ can be derived from S_p exceeds a certain time-out constant. This is done to avoid possible problems of non-termination, since the set of basic facts derivable from a given program is undecidable in general.

In each debugging session $\mathcal{DDT}$ asks the user whether this simplification should be performed. If the answer is affirmative the tool asks for the name of the $\mathcal{TOY}$ program which contains the trusted specification, and simplifies the tree before the navigation phase.

For instance, the program of Figure 5 is a trusted specification for the example program of the golden ratio, using a different method for generating the infinite list with the Fibonacci sequence. For saving space we don't include the definitions of take, tail, goldenApprox, ./. and main which are the same of the Figure 1.

We can imagine this program as the first, naive, solution for the problem of the golden ratio approximations programmed by the user. It works correctly, but the generation of Fibonacci numbers is quite inefficient. Then the example of Figure 1 could be an attempt of improving the efficiency of this program. After trying the new version the user could observe that it returns a different answer, and decide that the first naive version was more likely to be correct. Then the declarative debugger could be started using this first version of Figure 5 as a trusted specification. The simplification will delete 12 nodes of the computation

```
fibN 1          = 1
fibN 2          = 1
fibN N          = if N>2 then (fibN (N-1))+(fibN (N-2))

fib             = map fibN (from 0)

map F []        = []
map F [X|Xs]    = [F X | map F Xs]

from N          = [N | from N+1]
```

Fig. 5. A Trusted Specification for the program of Figure 1

tree and mark 3 nodes more as non valid, hence reducing the number of *unknown* nodes in the initial CT from 23 to only 8.

6 Conclusions

In this paper we have described the declarative debugging tool $\mathcal{DDT}$, which is part of the functional-logic system $\mathcal{TOY}$. In comparison to the traditional top-down declarative debuggers, $\mathcal{DDT}$ gives more support for avoiding the complexity of oracle questions. This can be achieved either by skillful free CT navigation, or by using a divide-and-query navigation strategy. Additionally, $\mathcal{DDT}$ also offers two useful techniques for simplifying the CT prior to navigation.

In contrast to other debugging tools (as e.g. the recent visual debugger for Mercury [3]) $\mathcal{DDT}$ is an off-line tool: the computation tree must be completely generated before it can be displayed and navigated. Unfortunately, complete CT generation causes a considerable overhead w.r.t. to the original computation which led to the debugging session, both in terms of time and space resources. Related works on the implementation of declarative debuggers for lazy functional languages [14, 17] have proposed techniques for reducing the computational overhead caused by debugging. As far as we know, this kind of techniques have been worked out only for the top-down navigation strategy. They mainly rely on a lazy generation of the CT as demanded by navigation.

In spite of the computational overhead, we still believe that $\mathcal{DDT}$ offers better facilities for CT simplification and navigation, which means a crucial advantage in CTs with a large number of nodes, where top-down navigation produces too many (maybe complex) questions. As future work, we plan to revise the implementation of the $\mathcal{DDT}$ tool, looking for incremental CT simplification and navigation methods that can be made compatible with lazy CT generation.

Acknowledgements The authors are grateful to Wolfgang Lux and Francisco J. López-Fraguas for their useful comments about the paper, as well as to the anonymous referees for their constructive remarks.

References

1. M. Alpuente, F.J. Correa, and M. Falaschi. *A Debugging Scheme for Funcional Logic Programs*. Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.
2. M. Comini, G. Levi, M.C. Meo y G. Vitello. *Abstract Diagnosis*. J. of Logic Programming 39, 43–93, 1999.
3. M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. *ViMer: A Visual Debugger for Mercury*. In Proc. PPDP03, ACM Press, 56–66, 2003.
4. R. Caballero and W. Lux. *Declarative Debugging of Encapsulated Search*. Electronic Notes in Theoretical Computer Science. Vol. 76. Elsevier, 2002.
5. R. Caballero, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. In Proc. FLOPS'01, Springer LNCS 2024:170–184, 2001.
6. R. Caballero and M. Rodríguez Artalejo. *A Declarative Debugging System for Lazy Functional Logic Programs*. Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.
7. G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method*. The Journal of Logic Programming 4(3):177–198, 1987.
8. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*, Journal of Logic Programming 40(1) 44-87, 1999.
9. J.C. González-Moreno, M.T. Hortalá-González y M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. Available at: `http://danae.uni-muenster.de/lehre/kuchen/JFLP`
10. 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.
11. J.W. Lloyd. *Declarative Error Diagnosis*. New Generation Computing 5(2):133–154, 1987.
12. F.J. López-Fraguas, and J. Sánchez-Hernández. *$\mathcal{TOY}$ a Multiparadigm Declarative System*, In Proc. RTA'99, LNCS 1631, Springer Verlag, 244-247, 1999.
13. L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3.
14. H. Nilsson. *How to look busy while being lazy as ever: The implementation of a lazy functional debugger*. Journal of Functional Programming 11(6):629–671, 2001.
15. H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121–150, 1997.
16. S.L. Peyton Jones (ed.), J. Hughes (ed.), et al. *Report on the programming language Haskell 98: a non-strict, purely functional language*. Available at `http://www.haskell.org/onlinereport/`, 2002.
17. B. Pope and L. Naish, *Practical Aspects of Declarative Debugging in Haskell 98*, In Proc. PPDP03, ACM Press, 230–240, 2003.
18. E.Y.Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, 1982.
19. SICStus Prolog homepage: `http://www.sics.se/sicstus/`.
20. A. Tessier and G. Ferrand. *Declarative Diagnosis in the CLP Scheme*. In P. Deransart, M. Hermenegildo and J. Małuszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Chapter 5, 151–174. Springer LNCS 1870, 2000.