A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs [*]

System Demonstration

Rafael Caballero

Facultad de Informática. Univ. Complutense de Madrid

rafa@sip.ucm.es

Abstract

Debugging is one of the essential parts of the software development cycle. However, the usual debugging techniques used in imperative languages such as the *step by step* execution often are not suitable for debugging declarative programming languages. We present here a graphical debugging environment for constraint lazy functional-logic programs based on *declarative debugging*. The debugger displays the computation tree associated with a computation which has produced an incorrect answer, and navigates it with the assistance of the user until the error, an incorrect program rule, is found out. The debugger supports programs including equality and disequality constraints.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language Classifications—Multiparadigm languages; H.5.2 [*Information Interfaces and Presentation*]: User Interfaces—Graphical user interfaces (GUI)

General Terms Languages

Keywords Functional-Logic Languages, Declarative Debugging

1. Declarative Debugging

The lack of auxiliary tools such as debuggers has been pointed out in [11] as a possible impediment to the success of functional languages like Haskell [8]. The same arguments can be applied as well to the case of functional-logic languages [6]. However, implementing debuggers for these languages is not an easy task. The approach followed traditionally in imperative programming, based on the *step by step* execution of the program, is not suitable in declarative languages, particularly in the case of the lazy functional and functional-logic languages, where features like the higher order, the polymorphism or the lazy evaluation makes the computation more difficult to observe.

Declarative debugging was first proposed by E. Y. Shapiro [10] as an alternative in the field of logic programming, and has been later applied in functional and functional-logic programming as well. A declarative debugging session starts when the user observes an unexpected behavior of the program which is regarded as wrong, and can be divided in two phases:

1. The declarative debugger generates a suitable *computation tree* (CT in short) representing the behavior of the program during the computation. Each node of the tree represents the result of some subcomputation; the value at the root of the tree corresponds to the result of the overall computation, which is called the *initial symptom*, while the results at the children of each node N must correspond to the subcomputations necessary to obtain the result at N. Moreover, each node must have some associated fragment of code, which is responsible for the computation result stored at the node.
2. The debugger *navigates* tries to locate a node whose result is incorrect but with correct results at all its children nodes. Such a node is called a *buggy node*, and corresponds to a fragment of code which has produced an erroneous result from correct inputs, and is therefore incorrect. In order to check the correctness of the nodes, the debugger relies on an external oracle, usually the user.

Here we present $\mathcal{DDT}$, a graphical declarative debugger of incorrect answers for the constraint lazy functional-logic programming language $\mathcal{TOY}$ [1]. The prototype is based on a previous version described in [5], the main novelty of the new version being the possibility of debugging programs including equality and disequality constraints. The system can be downloaded from: http://toy.sourceforge.net.

In the next section the programs considered in our setting are presented, together with a small example of incorrect program which will be used in the rest of the paper. The section 3 presents informally the computation trees handled by the debugger, while section 4 provide some details about the implementation of the program transformation employed by the tool. A short debugging session for our example program is shown in section 5, where some of the features of the graphical interface are introduced. The efficiency of the system is informally discussed in section 6. The paper ends in section 7 with some conclusions and some proposals of future work.

2. Toy programs

A $\mathcal{TOY}$ program can include declarations of data, function types, type alias, infix operator declarations, and defining rules for functions symbols. Two important syntactic categories in this setting are *expressions* and *patterns*. The possible forms of an expression

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

```
onlyOne::[A] → bool
onlyOne L           = size L==1

size:: [A] → int
size [ ]            = 0
size [X|L]          = if (member X L) then N
                        else 1+N
                      where N = size L

member:: A → [A] → bool
member U [ ]        = false
member U [V|L]      = if (U == V) then true
                        else member V L
```

Figure 1. Example of $\mathcal{TOY}$ program

e are:

$$e ::= \bot \ \mid X \mid h \mid (e\ e')$$

where X is a variable, h either a function symbol or a data constructor, and $\bot$ is a symbol representing the undefined value. This symbol cannot be written directly in the programs, but it is important for representing correctly the lazy semantics of the programs, and will be used by the debugger to indicate that a function call was not demanded during the analyzed computation. The expression $(e\ e')$ stands for the application of expression e to expression e'. We use the notation $e\ e_1\ e_2 \ldots e_n$ or $e\ \overline{e}_n$ as a shorthand for $((\ldots((e\ e_1)\ e_2)\ldots)\ e_n)$. Similarly, the possible forms of a pattern t are:

$$t ::= \bot \ \mid X \mid c\ \bar{t}_m \mid f\ \bar{t}_m$$

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, with the t_i patterns for $1 \le i \le m$. Hence, $\mathcal{TOY}$ considers partial applications of functions as patterns. Patterns without any occurrence of $\bot$ are called *total* patterns.

An important predefined function symbol is $==$, which represents *strict equality*. An expression $e_1 == e_2$ is evaluated to true if e_1 and e_2 can be reduced to some common total pattern, and to false if e_1 and e_2 can be reduced to incompatible patterns. In $\mathcal{TOY}$ $e_1 == e_2$ also represents an atomic constraint which is satisfied when $e_1 == e_2$ is evaluated to true. Both meanings of $==$ can be distinguished from the context. Similarly, the notation $e_1 /= e_2$ is used for representing an atomic constraint which is satisfied when $e_1 == e_2$ is evaluated to false. The defining rules for a function f have the form:

$$f\ t_1 \ldots t_n = r \ \Leftarrow\ C\ \ where\ LD$$

where the t_i are patterns, r (the *right-hand side*) is an expression, C (the *condition*) is a conjunction of atomic constraints, and LD is a list of *local conditions*, each one of the form $t = e$. A *goal* in $\mathcal{TOY}$ has the same form as a rule condition C. The computed *answers* for a goal G must be of the form $\sigma \,\square\, C$, with σ a substitution and C a constraint in solved form. For instance a simple goal like $X == Y$ has one answer: $\{X \mapsto Y\} \,\square\, \{tot(X)\}$, meaning that the goal holds if X and Y represent the same, total value.

Let us consider now the small example program of figure 1. The function onlyOne checks whether a list contains only one element, perhaps repeated many times. This function relies on the function size, which calculates the number of different elements that can be found in a given list. size only counts the last occurrence of an element in the list, using member for checking this condition. A goal like onlyOne [X,Y,Z]==true will obtain two different answers:

- {Y ↦ X, Z ↦ X } □ {tot(X) }
- {Z ↦ Y } □ {tot(Y), Y /= X }

The first answer is correct, but the second one is an incorrect answer (if Y/=X then [X,Y,Z] has more than one element), showing that something is wrong in the program. After detecting this *initial symptom* the user would start the debugger to locate the error.

3. Computation Trees

In [9] a logic calculus *CRWL*($\mathcal{D}$) has been proposed as a suitable semantics for constraint functional-logic programs over a parametric domain $\mathcal{D}$. Given a program P, a goal G, and a computed answer $\sigma \,\square\, C$, there exists always a proof tree, whose nodes correspond to the application of the *CRWL*($\mathcal{D}$) logic inference rules, which proves the logical implication $G\sigma \Leftarrow C$ w.r.t. P. From the theoretical point of view, the CT used by our debugger can be seen as a simplification of this proof tree, obtained by removing all the nodes associated to *CRWL*($\mathcal{D}$)-inferences that do not depend on the program definition rules (and hence cannot be incorrect). A similar approach was used in [3, 4] to define the computation trees suitable for debugging incorrect answers in programs without constraints.

More precisely, the root of the CT is of the form $G\sigma \Leftarrow C$, with G the initial goal and $\sigma \,\square\, C$ the incorrect answer being debugged. The other nodes of the tree contain *constraint basic facts* (CBFs from now on) of the form

$$f\ \bar{t}_n\ \rightarrow\ t\ \Leftarrow\ C$$

with t_i, t patterns, and C a conjunction of atomic constraints, meaning that a call $f\ \bar{t}_n$ can return the result t whenever C holds. Each CBF corresponds to a function call evaluated during the computation. The children of each node N contain the CBFs associated to the evaluation of function calls occurring in the right-hand side, the conditions, or in the local definitions of the program rule used for evaluating the function call associated to N. All the values in the CBFs of the tree appear evaluated as much as they were demanded by the computation, and are affected by the substitution σ. The occurrences of function calls which were not needed during the computation are replaced by the symbol _, standing for $\bot$. The constraint C at the CBFs correspond to the constraint store obtained at the end of the computation, and hence is the same for all the nodes. However, in order to simplify the questions asked by the debugger to the user the constraint C of a CBF $f\ \bar{t}_n \rightarrow t \Leftarrow C$ will be simplified by removing all the atomic constraints whose variables do not occur in $f\ \bar{t}_n \rightarrow t$. In principle this could be unsafe, because reducing the number of constraints could convert a non-satisfiable constraint C in a satisfiable one. But this is not the case, because C is a constraint obtained as part of a computed answer and hence it must be satisfiable. Thus the simplified CBF will have the same declarative meaning as the original one.

The figure 2 shows the CT corresponding to the second (wrong) answer of our example goal onlyOne [X,Y,Z]==true. Notice that the tree does not include nodes for predefined functions such as $+$ or if then else, because such nodes are assumed to be valid and have been removed in advance.

4. Implementing the program transformation

The implementation of the debugger can be divided in two different phases, following closely the two stages of a declarative debugging process described in Section 1:

1. The debugger uses a program transformation (described in [4]) in order to generate a program whose functions return, as part of their results, the CTs corresponding to the computations in the original program. The debugger uses the transformed program to repeat the incorrect computation, obtaining in this way

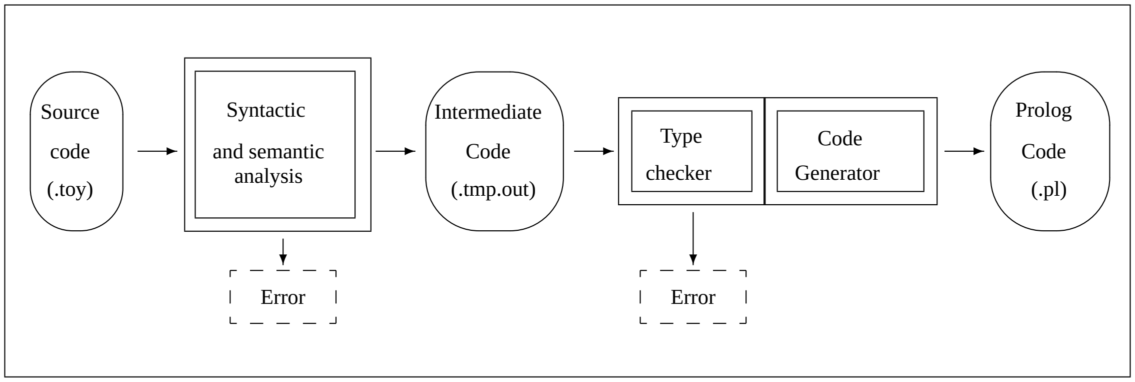
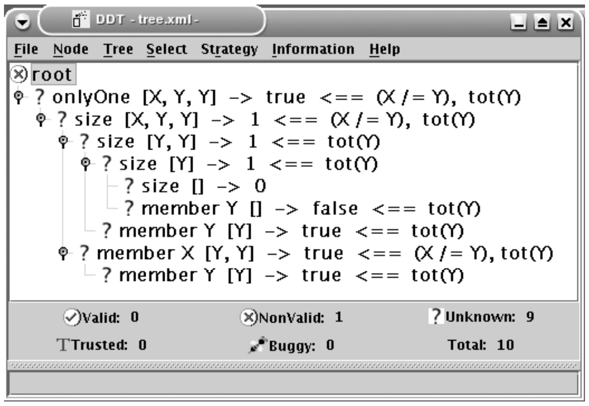


Figure 3. Compiling a $\mathcal{TOY}$ program



```
root
? onlyOne [X, Y, Y] -> true <== (X /= Y), tot(Y)
  ? size [X, Y, Y] -> 1 <== (X /= Y), tot(Y)
    ? size [Y, Y] -> 1 <== tot(Y)
      ? size [Y] -> 1 <== tot(Y)
        ? size [] -> 0
        ? member Y [] -> false <== tot(Y)
      ? member Y [Y] -> true <== tot(Y)
    ? member X [Y, Y] -> true <== (X /= Y), tot(Y)
      ? member Y [Y] -> true <== tot(Y)

Valid: 0        NonValid: 1     Unknown: 9
Trusted: 0      Buggy: 0        Total: 10
```

Figure 2. Computation tree displayed by $\mathcal{DDT}$

the CT. The constraint C is not included in the tree returned by the transformed program, and is incorporated afterwards by the debugger, which obtains it through a primitive getConstraintStore. The CT is then written in a file, which will be loaded by the graphical interface.

2. The CT is then navigated by the graphical interface, which has been implemented in Java, using the *JTree* component of the package *Swing* for displaying the tree.

Let us look to the first phase, the program transformation, more carefully. The figure 3 represents briefly the normal compilation process of a $\mathcal{TOY}$ program P.toy. First, the source code is parsed, syntactic and semantically analyzed (excluding the type checking phase, which is carried out afterwards), and all the "syntactic sugar" is eliminated. If no error is found the result, an intermediate code, is stored in an intermediate temporary file P.tmp.out. This temporary file, somehow similar to the *Flat Curry* file used by Curry [7] but represented through Prolog clauses, is then loaded by the second part, which does the type checking and, if no type error is found, the code generation which produces a Prolog file P.pl.

When implementing the program transformation, the first idea is to perform a source-to-source transformation, hence producing a transformed source program $\mathsf{P}^{\mathcal{T}}$.toy from the original program P.toy. However this means that the file P.toy must be parsed again. Moreover, the transformed program $\mathsf{P}^{\mathcal{T}}$.toy also will have to be parsed in order to produce a compiled transformed program $\mathsf{P}^{\mathcal{T}}$.pl, which is the final goal of this process. For these reasons we have preferred to apply the transformation to the intermediate file P.tmp.

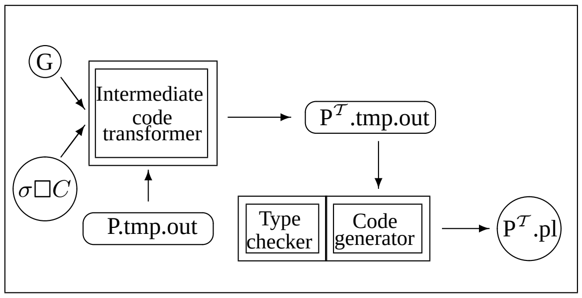

G
$\sigma\Box C$
Intermediate code transformer
P.tmp.out
$\mathsf{P}^{\mathcal{T}}$.tmp.out
Type checker
Code generator
$\mathsf{P}^{\mathcal{T}}$.pl

Figure 4. Transformed program generation

The transformation process is described in the figure 4. It starts with the intermediate temporary file P.tmp.out, the goal G and the incorrect answer $\sigma\Box C$. Notice that the file P.tmp.out always exists because the debugger is used after an unexpected computation is found out, and hence the original program P has been already compiled. The file is read by the intermediate code transformer, which produces a transformed intermediate file $\mathsf{P}^{\mathcal{T}}$.tmp.out, which in turn is used by the code generator in order to produce the final transformed code $\mathsf{P}^{\mathcal{T}}$.pl. Observe that the syntactic and semantic analysis phases are in this approach avoided, since we now that the original program contains no errors (it was compiled and used already) and therefore the same holds for the transformed program. The same occurs with the type checking phase (we have proved in work [4] that the transformed program has no type errors if the original one was well-typed). At this point the transformed program can be executed in order to obtain the computation tree. Following this approach the generation of the transformed program can be done in very few seconds even for large, realistic programs.

5. A debugging session

The purpose of the debugger is to locate a buggy node in the CT. For this reason the user must determine the validity or non-validity of some nodes of the CT w.r.t. the expected meaning of the program. Only the root, which corresponds to the initial symptom, and whose content is not displayed by the debugger, is marked at the beginning as non-valid. The other nodes have a symbol '?' at the left, meaning that their validity is still not known.

The options in the menu, shown in figure 5, can be used to manage and simplify the tree. For instance, the option *Remove Valid & Trusted Nodes* of the menu *Tree* automatically all the nodes marked as *valid* or *trusted*. It can be proved that this operation is

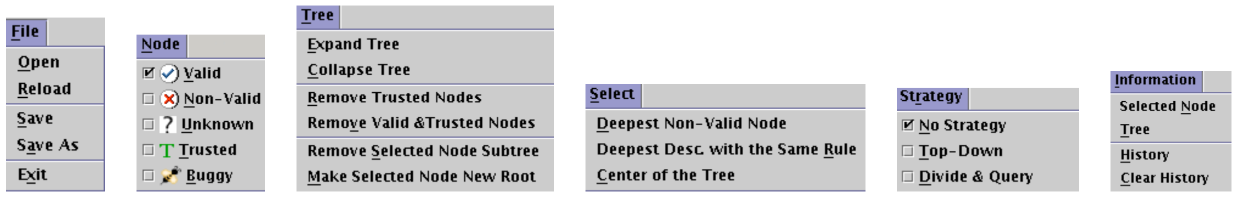
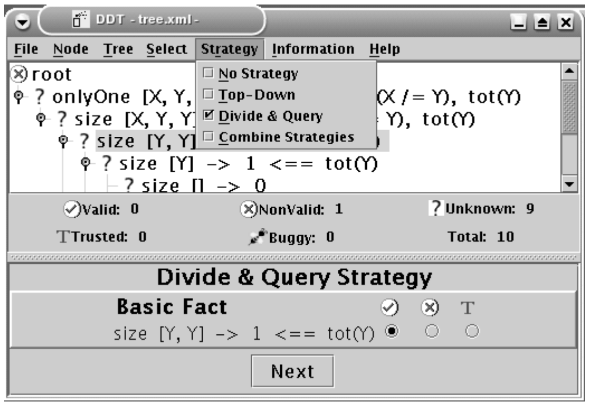


Figure 5. Menu options in $\mathcal{DDT}$

safe w.r.t. the existence of a buggy node, in the sense that if the original tree T had some buggy node, then the simplified tree T' also has a buggy node which was buggy in T as well. Deleting a node N of the tree means in this context that their children trees will become children of the parent of N after the deletion. Therefore the operation is not defined for the root node, but in our case this is not a problem since the root node is always non-valid. At any point the current CT can be saved/loaded in XML format using the options of the menu *File*.

The validity of any node can be changed by clicking the right-button of the mouse over the node and choosing a value among *valid, non-valid, don't-know*, and *trusted* in the menu *Nodes*. Marking a node as *trusted* makes all the nodes associated to the node's function automatically valid. Although the use can move throughout the tree, providing information freely until a buggy node is located, for large trees it is recommended to follow some fixed navigation strategy. $\mathcal{DDT}$ includes two strategies: top-down and divide-and-query. In the top-down strategy 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 process.

The divide-and-query strategy looks at each step for 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 singles out 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. Both strategies are complete, in the sense that given any CT with a non-valid root a buggy node is always found if the user answers correctly to que questions. The next figure shows the first question asked by the debugger after selecting the strategy *divide-and query*:



```
root
? onlyOne [X, Y,            (X /= Y), tot(Y)
  ? size [X, Y, Y]          Y), tot(Y)
    ? size [Y, Y]
      ? size [Y] -> 1 <== tot(Y)
        ? size [] -> 0
```

Valid: 0   NonValid: 1   Unknown: 9
Trusted: 0   Buggy: 0   Total: 10

Divide & Query Strategy

Basic Fact
size [Y, Y] -> 1 <== tot(Y) [✓ ✓] ⊗ T

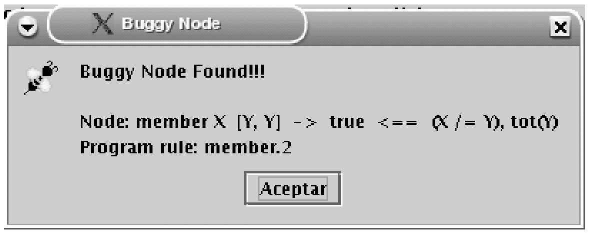
This constraint basic fact is valid; given any total value Y, the list [Y,Y] only contains one element. The next questions will be:

- size [X,Y,Y] → 1 ⇐ (X/=Y), tot(Y): non-valid; the list has at list two elements since (X/=Y) holds.

- member X [Y,Y] → true ⇐ (X/=Y), tot(Y): non-valid; if (X/=Y) then X is not a member of the list.

- member Y [Y] → true ⇐ tot(Y): valid.

After answering these questions the buggy node has been found:



Buggy Node Found!!!

Node: member X [Y, Y] -> true <== (X /= Y), tot(Y)
Program rule: member.2

The debugger indicates that the cause of the error is at the second rule of member. Examining more carefully the CBF we find that since X /= Y, it is the else part of the rule which has been used and therefore is wrong. Now is easy to find out that the V in member V L should be a U.

In this simple session can be observed that although the constraint in all the CBFs is the same (except for the simplifications) it must be considered in each case, since it becomes meaningful only when is part of a CBF.

6. Efficiency

We have seen already a debugging session for a small example program. Of course the question is whether this technique can be also useful for larger, more realistic programs. We discuss in this sections three issues related with the efficiency and applicability of the tool: the complexity of the questions, the number of questions, and the resources (memory and time) required by the transformed program to produce the CT.

Complexity of the questions. We have seen already that que debugger always asks questions about the validity of constraint basic facts, which cannot included nested calls because they have been replaced by their result at the end of the computation, or by _ if they were not evaluated. In spite of this, the validity of some CBFs can still be difficult to determine when they involve very large structures in the parameters or in the result. However, and due to the recursive nature of programs in functional logic languages, it is usual that a complex wrong computation requires some simpler, but also wrong subcomputations. Hence, a heuristic that often works in

	CT		Top-Down		D& Q
Program	Nodes	Depth	Q.	Nodes	Q.
P1	201	68	68	134	9
P2	194	14	12	45	8
P3	206	18	17	49	8
P4	191	19	19	36	8
P5	198	15	15	29	8
P6	195	44	6	10	7
P7	191	14	13	24	8
P8	200	42	42	83	8
P9	196	16	12	23	8
P10	197	31	9	39	7

Table 1. Comparison of strategies (I)

these cases is to look for some non-valid node in the bottom part of the tree, where the constraint basic facts are usually simpler. Then, we can select the subtree rooted by such a node for debugging, by using the option *Make Selected Node New Root* of the menu *Tree* (see figure 5). The situation can become harder when more complex constraints are involved, such as arithmetic constraints.

Number of questions. The number of questions performed by the debugger depends on the strategy. The table 1 shows a comparison between the top-down strategy and the divide and query (called *D&Q* in the table) for 10 different programs. The first two numbers correspond to the number of nodes and the depth of the CT. The two columns corresponding to the top-down strategy show the total number of questions (represented by *Q.*) and the total number of nodes that must be examined by the user during the debugging session (in top-down each question includes all the children nodes of an non-valid node). Similarly, the number in the column labelled by *Q.* of the divide and query part stands for the number of questions asked by the tool during the debugging session following this strategy (which is the same as the number of examined nodes in this strategy). We have tried in each case the goal that better approximates the number of 200 nodes in the CT. This is a brief description of the programs:

P1: Incorrect program for reversing a list.

P2: Erroneous quick-sort algorithm.

P3: Erroneous bubble-sort algorithm.

P4: Erroneous permutation-sort algorithm.

P5: Obtaining prime numbers.

P6: Arithmetic with Peano numbers.

P7: Sorting a tree.

P8: Computing the golden ratio.

P9: N-Queens problem.

P10: Using functional-logic *extensions*.

All of them can be located at the *examples/debugger* folder of the $\mathcal{TOY}$ distribution. In the table it can be seen that the top-down strategy, adopted by several declarative debuggers, does not seem to be suitable, since the number of questions is very often directly related to the depth of the CT. On the contrary, the experimental results indicate that the divide-and-query strategy works well in most of the cases, requiring a number of questions close to $\log_2 N$, with N the number of nodes in the CT. This is expected, since this strategy roughly divides by 2 the number of nodes of the CT after every question. The same situation holds when we try goals that require a CT with a greater number of nodes. In the table 2 we can observe the increment in the number of questions in the program P9. The goal S_i corresponds to try the solution of the N-Queens problem for $N = i$. The top-down strategy

	CT		Top-Down		D & Q
Goal	Nodes	Depth	Q	Nodes	Q
S_7	98	12	15	8	7
S_{11}	196	16	12	23	8
S_{14}	292	20	15	29	8
S_{17}	403	22	18	35	9
S_{19}	488	24	20	39	9
S_{26}	850	32	27	53	10
S_{30}	1100	36	31	61	11
S_{35}	1456	40	36	71	11
S_{40}	1865	46	41	81	11
S_{45}	2321	50	46	91	11
S_{50}	2830	56	51	101	12
S_{55}	3386	60	56	111	12

Table 2. Comparison of strategies (II)

requires the examination of 111 nodes for S_{55}, while the divide and query only requires 12. The situation is similar for other examples, although there exists also cases like P6, P10 where the number of questions performed by the top-down strategy is constant and does not depend on the number of nodes of the CT. Nevertheless we can resume the comparison indicating that the top-down strategy is more unpredictable, while the number of questions asked by the debugger using the divide and query is related to $\log_2 N$ in all the experiments. Thus we can expect an approximate number of 16 questions for a computation with a CT of 30000 nodes, which corresponds to a realistic computation size.

Memory overhead The memory overhead required by the generation of the CT is actually the major drawback of the technique. The size of the tree is so huge that the system usually is not able of producing CTs with more than 5000 nodes. In certain examples this number can be reduced even to a few hundreds of nodes. A possible solution would be to evaluate the CT lazily, but this alternative is difficult to conciliate with the use of the divide and query strategy, which traverses the whole tree before each question. Another alternative was proposed in [**?**], where H. Nilsson presented a declarative debugger for a subset of Haskell which was not based in a program transformation but on a modification of the underlying abstract machine. Adapting these ideas to a functional-logic language could improve the efficiency and make the tool suitable for debugging large computations.

7. Conclusions and future work

We have presented a graphical environment for finding incorrect program rules in constraint lazy functional-logic programs. The debugger is equipped with a graphical user interface and has been implemented for the language $\mathcal{TOY}$, although the same approach can be used for debugging similar languages such as Curry [7]. In fact, a textual version of the debugger, which uses a top-down strategy for detecting buggy nodes, is already part of the Curry system of the Münster University [2]. The debugger currently supports programs with equality and disequality constraints, but the implementation is almost independent of the constraint domain. The only change that must be done in order to apply the debugger to other constraint domains is the redefinition in each case of the primitive getConstraintStore, which is used by the debugger for obtaining the constraint store at the end of the computation. We plan to extend the debugger to programs including arithmetic constraints and also for programs allowing constraints over finite domains.

A different task is to extend the debugger for dealing with *missing answers*, which currently are not admitted by the prototype. A missing answer is obtained when the set of computed answers for a given goal does not cover some expected answer. In this case the computation tree is different and more complex, since it must represent all the computations that occur while solving the goal.

Acknowledgments

I would like to thank Mario Rodríguez-Artalejo for his useful suggestions.

References

[1] M. Abengózar-Carneros et al. Toy: A multiparadigm declarative language. Version 1.0. Technical Report SIP-119/00, Universidad Complutense de Madrid, February 2002.

[2] R. Caballero and W. Lux. Declarative Debugging of Encapsulated Search. Electronic Notes in Theoretical Computer Science, 76, pages 1–13 2002.

[3] R. Caballero, F. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proc. FLOPS'01*, number 2024 in LNCS, pages 170–184. Springer, 2001.

[4] R. Caballero and M. Rodríguez-Artalejo. A Declarative Debugging System for Lazy Functional Logic Programs. Electronic Notes in Theoretical Computer Science, 64, 2002.

[5] R. Caballero and M. Rodríguez-Artalejo. DDT: a Declarative Debugging Tool for Functional-Logic Languages. In *Proc. FLOPS'04*, number 2998 in LNCS, pages 70–84. Springer, 2004.

[6] M. Hanus. The Integration of Functions into Logic Programming: A Survey. J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", pages 583–628, 1994.

[7] M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8, April 15, 2003). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[8] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[9] F. López-Fraguas, M. Rodríguez-Artalejo, and R. d. Vado-Vírseda. Constraint functional logic programming revisited. In *Proc. WRLA'2004*, volume 117 of *Elec. Notes on Theor. Comp. Science*, pages 5–50, 2004.

[10] H. Nilsson. How to look busy while being lazy as ever: The implementation of a lazy functional debugger. Journal of Functional Programming 11(6), pages 629–671, 2001.

[11] E. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.

[12] P. Wadler. *Why no one uses Functional Languages*. SIGPLAN Notices 33(8), pages 23–27, 1998.