10. Tools for Search-Tree Visualization: The APT Tool

M. Carro and M. Hermenegildo

Technical University of Madrid, 28660-Madrid, Spain.
{mcarro,herme}@fi.upm.es

Summary.
The control part of the execution of a constraint logic program can be conceptually shown as a search-tree, where nodes correspond to calls, and whose branches represent conjunctions and disjunctions. This tree represents the search space traversed by the program, and has also a direct relationship with the amount of work performed by the program. The nodes of the tree can be used to display information regarding the state and origin of instantiation of the variables involved in each call. This depiction can also be used for the enumeration process. These are the features implemented in APT, a tool which runs constraint logic programs while depicting a (modified) search-tree, keeping at the same time information about the state of the variables at every moment in the execution. This information can be used to replay the execution at will, both forwards and backwards in time. These views can be abstracted when the size of the execution requires it. The search-tree view is used as a framework onto which constraint-level visualizations (such as those presented in the following chapter) can be attached.

10.1 Introduction

Visualization of CLP executions is receiving much attention recently, since it appears that classical visualizations are often too dependent on the programming paradigms they were devised for, and do not adapt well to the nature of the computations performed by CLP programs. Also, the needs of CLP programmers are quite different [10.14]. Basic applications of visualization in the context of CLP, as well as Logic Programming (LP), include:

- Debugging. In this case it is often crucial that the programmer obtain a clear view of the program *state* (including, if possible, the program point) from the picture displayed. In this application, visualization is clearly complementary to other methods such as assertions [10.2, 10.10, 10.5] or text-based debugging [10.6, 10.12, 10.15]). In fact, many proposed visualizations designed for debugging purposes can be seen as a graphical front-end to text-based debuggers [10.11].
- Tuning and optimizing programs and programming systems (which may be termed—and we will refer to it with this name—as *performance debugging*). This is an application where visualization can have a major impact, possibly in combination with other well-established methods as, for example, profiling statistics.

– Teaching and education. Some applications to this end have already been developed and tested, using different approaches (see, for example, [10.13, 10.16]).

In all of the above situations, a good pictorial representation is fundamental for achieving a useful visualization. Thus, it is important to devise representations that are well suited to the characteristics of CLP data and control. In addition, a recurring problem in the graphical representations of even medium-sized executions is the huge amount of information that is usually available to represent. To cope successfully with these undoubtedly relevant cases, *abstractions* of the representations are also needed. Ideally, such abstractions should show the most interesting characteristics (according to the particular objectives of the visualization process, which may be different in each case), without cluttering the display with unneeded details.

The aim of the visualization paradigms we discuss is quite broad—i.e., we are not committing exclusively to teaching, or to debugging—, but our focus is debugging for correctness and, mainly, for performance. Visualization paradigms can be divided into three categories (which can coexist together seamlessly, and even be used together to achieve a better visualization): visualizing the execution flow / control of the program, visualizing the actual variables (i.e., representing their run-time values), and visualizing constraints among variables. The three views are amenable to abstraction.

In this chapter we will focus on the task of visualizing the execution flow of constraint logic programs. This is complementary to the discussion on depiction of items of data and their relationship (e.g., the constraints themselves), that will be discussed in Chapter 11. Herein, we will discuss general ways in which the control aspects of CLP execution can be visualized. Also, we will briefly describe APT, a prototype visualizer developed at UPM which implements some of the ideas presented in the chapter.

10.2 Visualizing Control

Program flow visualization (using flowcharts and block diagrams, for example) has been one of the classical targets of visualization in Computer Science. See, for example, [10.3] or the multiple versions of animated sorting algorithms (e.g., those under `http://reality.sgi.com/austern/java/demo/demo.html` or `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`), for some depictions of data evolution in procedural and object-oriented paradigms. The examples shown there have a strong educational component, but at the same time can be used for studying performance problems in the algorithms under consideration.

One of the main characteristics of declarative programming is the absence of explicit control. Although this theoretical property results in many advantages regarding, for example, program analysis and transformation, programs are executed in practice with fixed evaluation rules,[1] and different declaratively correct programs aimed at the same task can show wide differences in efficiency (including termination, which obviously affects total correctness). These differences are often related to the evaluation order. Understanding those evaluation rules is important in order to write efficient programs. In this context, a good visualization of the program execution (probably combined with other tools) can help to uncover performance (or even correctness) bugs which might otherwise be very difficult to locate.

In CLP programs (especially in those using finite domains) it is possible to distinguish two execution phases from the point of view of control flow: the *programmed search* which results from the actual program steps encoded in the program clauses, and the *solver operations*, which encompass the steps performed inside the solver when adding equations or when calling the (generally built-in) enumeration predicates. These two phases can be freely interleaved during the execution.

10.3 The Programmed Search as a Search Tree

The programmed search part of CLP execution is similar in many ways to that of LP execution. The visualization of this part of (C)LP program execution traditionally takes the form of a direct representation of the search-tree, whose nodes stand for `calls`, `successes`, `redos` and `failures`—i.e., the events which take place during execution. Classical LP visualization tools, of which the Transparent Prolog Machine (TPM [10.13]) is paradigmatic, are based on this representation. In particular, the TPM uses an augmented AND-OR tree (AORTA), in which AND and OR branches are compressed and take up less vertical space, but the information conveyed by it is basically the same as in a usual AND-OR tree (see below a more complete description of the AORTA tree). This tree can be used both for displaying the search as explicitly coded by the programmer, and for representing the search implicitly performed by enumeration predicates.

It is true that in CLP programs the control part has usually less importance than in LP, since most of the time is spent in equation solving and enumeration. However, note that one of the main differences between C(L)P and, e.g., Operations Research, is the ability to set up equations in an algorithmic fashion, and to search for the right set of equations. Although this part may sometimes be short (and perhaps deterministic), it may also be quite large and it is in any case relevant, for performance debugging, to be able to represent and understand its control flow.

Given the previous considerations, a first approach which can be used in order to visualize CLP executions is to represent the part corresponding

[1] By "fixed" we mean that these rules can be deterministically known at run-time, although maybe not known statically.

to the execution of the program clauses (the programmed search) using a search-tree depiction. Note that the constraint-related operations of a CLP execution (enumeration/propagation) typically occur in "bursts" which can be associated to points of the search-tree. Thus, the search tree depiction can be seen as primary view or a skeleton onto which other views of the state of the constraint store during enumeration and propagation (and which we will address in Section 10.4) can be grafted or to which they can be related.

10.4 Representing the Enumeration Process

The enumeration process typically performed by finite domain solvers (involving, e.g., domain splitting, choosing paths for constraint propagation, and heuristics for enumeration) often affects performance critically. Observing the behavior of this process in a given problem (or class of problems) can help to understand the source of performance flaws and reveal that a different set of constraints or a different enumeration strategy would improve the efficiency of the program.

The enumeration phase can be seen as a search for a mapping of values to variables which satisfy all the constraints set up so far. It takes the form of (or can be modeled as) a search which non-deterministically narrows the domains of the variables and, as a result of the propagation of these changes, updates the domains of other variables. Each of these steps results in either failure (in which case another branch of the search is chosen by setting the domain of the selected variable differently or by picking another variable to update) or in a new state with updated domains for the variables.[2] Thus, one approach in order to depict this process is to use the same representation proposed for the programmed search, i.e., to use a tree representation, in either time or event space. In this case nodes concerning the selection of variables and selection of domains should be clearly distinguished, as they represent radically different choices. Another alternative, which focuses more on data evolution than on control flow, is to simply visualize those steps as a succession of states for all the variables (as shown in Section 11.2.1 and Figures 11.3 and 11.4 of Chapter 11), or show an altogether *ad-hoc* representation of enumeration.

The display of the enumeration process can have different degrees of faithfulness to what actually happens internally in the solver. Showing the internal behavior of the solver is not always possible, since in some CLP systems the enumeration and propagation parts of the execution are performed at a level not accessible from user code. This complicates the program visualization, since in order to gather data, either the system itself has to be instrumented

[2] This enumeration can often be encoded as a Prolog-like search procedure which selects a variable, inspects its domain, and narrows it, with failure as a possible result. The inspection and setting of the domains of the variables are typically primitive operations of the underlying system.

to produce the data (as in the CHIP Tree Visualizer [10.1], Chapter 7), or sufficient knowledge about the solver operation must be available so that its operation can be mimicked externally in a meta-interpreter inside the visualizer, and inserted transparently between the user-perceived execution steps.

Other types of visualization concerned with the internal work performed by the solver need low-level support from the constraint solver. They are very useful for system implementors who have access to the system internals, and for the programmer who wants to really fine-tune a program to achieve superior performance in a given platform, but its own nature prevents them from being portable across platforms. Therefore we chose to move towards generalization at the expense of some losses, and base a more general, user-definable depiction, on simpler, portable primitives, while possible. These may not have access to all the internal characteristics of a programming system, but in turn can be used in a variety of environments.

10.5 Coupling Control Visualization with Assertions

One of the techniques used frequently for program verification and correctness debugging is to use assertions which (partially) describe the specification and check the program against these assertions (see, e.g., [10.2, 10.10, 10.5], and Chapters 1, 2, 3, and 4 and their references). The program can sometimes be checked statically for compliance with the assertions, and when this cannot be ensured, run-time tests can be automatically incorporated into the program. Typically, a warning is issued if any of these run-time tests fail, flagging an error in the program, since it has reached a state not allowed by the specification. It appears useful to couple this kind of run-time testing with control visualization. Nodes which correspond to run-time tests can be, for example, color coded to reflect whether the associated check succeeded or failed; the latter case may not necessarily mean that the branch being executed has to fail as well. This allows the programmer to easily pinpoint the state of the execution that results in the violation of an assertion (and, thus, of the specification) and, by clicking on the nodes associated to the run-time checks, to explore for the reason of the error by following the source of instantiation of the variables. As mentioned before, the design of the tree is independent from the constraint domain, and so the user should be able to click on a node and bring up a window (perhaps under the control of a different application) which shows the variables / constraints active at the moment in which the node was clicked. This window allows the programmer to peruse the state of the variables and detect which are the precise sources of the values of the variables involved in the faulty assertion.

This does not mean, of course, that assertion checking at compile time should be looked on as opposed to visualization: rather, visualization can be effectively used as user interface, with interesting characteristics of its own, to assertion-based debugging methods.

10.6 The APT Tool

In order to test the basic ideas of the previous sections, we extended the APT tool (*A Prolog Tracer* [10.17]) to serve as a CLP control visualizer. APT is essentially a TPM-based search-tree visualizer,[3] and inherits many characteristics from the TPM. However, APT also adds some interesting new features. APT is built around a meta-interpreter coded in Prolog which rewrites the source program and runs it, gathering information about the goals executed and the state of the store at run-time. This execution can be performed depth-first or breadth-first, and can be replayed at will, using the collected information. All APT windows are animated, and are updated as the (re)execution of the program proceeds.

```
a(X,Y):- b(X,Z), c(Z,Y).
a(X,Y):- X=Y.
b(1,2).
c(2,4).
```

Fig. 10.1. Sample code

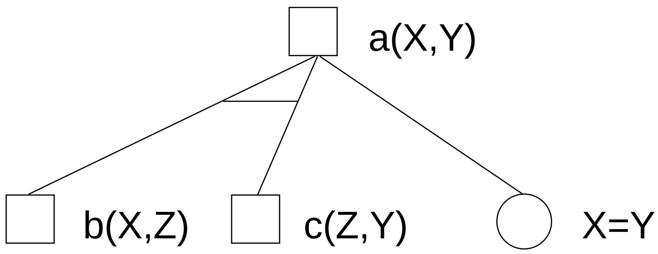

Fig. 10.2. AORTA execution tree for the program in Figure 10.1

The main visualization of APT offers a tree-like depiction, in which nodes from calls to user code are represented by squares and are adorned optionally with the name of the predicate being called. Nodes corresponding to built-ins appear as circles. Figure 10.1 shows a small program, and Figure 10.2 the execution tree corresponding to this program with the query `a(2, 2)`. Goals in the body of the first clause of `a/2` (`b(X, Z)` and `c(Z,Y)`) are shown as nodes whose edges to their parent are crossed with a line — these are AND-branches corresponding to the goals inside the clause. The goals in the body

[3] Another CLP visualizer that depicts the control part as a tree in the TPM spirit is the one developed by PrologIA [10.19], Chapter 6.

of the second clause (only one, in this case) are linked to the parent node with a separate set of edges. The actual run-time arguments are not shown at this level, but nodes can be blown up for more detail, as we will see later.

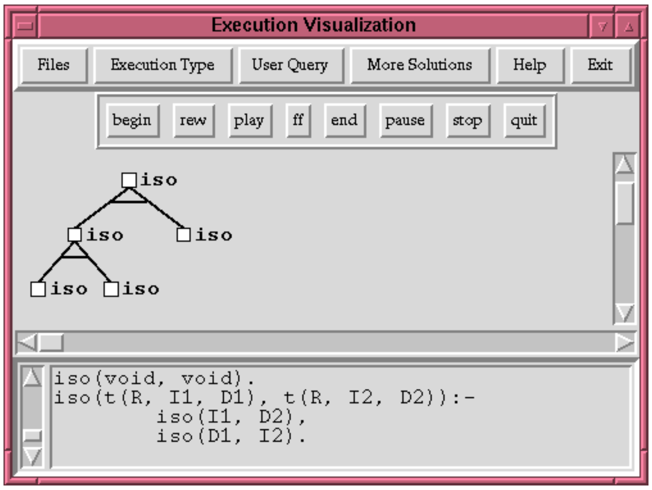

Fig. 10.3. A small execution tree, as shown by APT

Figure 10.3 shows a view of an actual APT window. In a real execution, the state of each node (*not yet called, called but not yet exited, exited, failed*) is shown by means of a color code. Clicking on a node opens a different window in which the relevant part of the program source, i.e., the calling body atom and the matching clause head, is represented together with the (run-time) state of the variables in that node (Figure 10.4).

The presentation of these node views depends on the type of data (e.g., the constraint domain) used. This is one of the most useful general concepts underlying the design of APT: the graphical display of control is logically separated from that of data. This allows developing data visualizations independently from the control visualization, and using them together. The data visualization can then be taken care of by a variety of tools, depending on the data to be visualized. Following the proposal outlined in the previous section, this allows using APT as a control skeleton for visualizing CLP execution. In this case, the windows which are opened when clicking on the tree nodes offer views of the constraint store in the state represented by the selected node. These views vary depending on the constraint domain used, or even for the same domain, depending on the data visualization paradigm used.

The figure at the left of the program text (still in Figure 10.4) represents the state of the call: marked with a tick ($\surd$) for success (as in this case), crossed ($\times$) for failure, and signaled with a question mark if the call has not

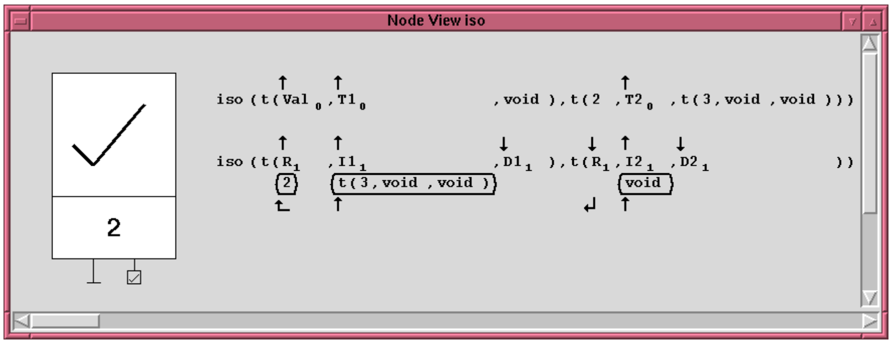

Fig. 10.4. Detailed view of a node (Herbrand domain)

finished yet. The number below the symbol denotes the number of clauses in the predicate. For each of these clauses there is a small segment sticking out from the bottom of the box. Clauses tried and failed have a "bottom" ($\perp$) sign; the clause (if any) currently under execution, but not yet finished, has a small box with a question mark; and a clause finished with success is marked with a tick.

Visualizations for constraints and constrained data are discussed elsewhere (for example, in Chapters 11, 11.5, and 12, and their references). As an example of such a visualization, Figure 10.4 shows a depiction used for the Herbrand domain (which is built into APT as the default node depiction, given that most CLP systems include the Herbrand domain). The node blow-up shows the run-time call on top and the matching head below it. The answer substitution (i.e., the result of head unification and/or body execution) is shown enclosed by rounded rectangles. The arrows represent the source and target of the substitution, i.e., the data flow. In the example shown, variable $\mathtt{R}_1$ received a value 2 from a call, and this value is communicated through head unification to variable $\mathtt{Val}_0$, which returns it to the caller. On the other hand, variable $\mathtt{T1}_0$ in the call unifies with variable $\mathtt{I1}_1$ in the head, and returns an output value `t(3, void, void)` which comes from some internal body call ($\mathtt{I1}_1$ does not appear anywhere else in the head of the clause).

APT is able to show the origin of the instantiation of any variables at any moment in the execution. In order to do that, APT keeps track of the point in the tree in which the (current) substitution of a variable was generated. Clicking on a substitution causes a line in the main tree to be drawn from the current node to the node where the substitution was generated. This is a very powerful feature which helps in correctness debugging, as the source of a (presumably) wrong instantiation (causing, for example an unexpected failure or a wrong answer) can be easily located. The culprit node can in turn be blown up and inspected to find out the cause of the generation of those values.[4]

Some More Details on The APT Tool. The tool reads and executes programs, generating an internal trace with information about the search-tree, the variables in each call, and the run-time (Herbrand) constraints associated. The execution can then be replayed, either automatically or step-by-step, and the user can move forwards and backwards in time. More detailed information about each invocation can be requested. The tool has a built-in text editor, with a full range of editing commands, most of them compatible with Emacs. Files open from the visualizer are loaded into the editor.

Execution can be performed either in depth-first or breadth-first mode. In the case of depth-first search, the user can specify a maximum depth to search; when this depth has been reached, the user is warned and prompted to decide whether to stop executing, or to search with a new, deeper maximum level. The search mode and search depth are controlled by the metainterpreter built in APT, so that no special characteristic is required from the underlying CLP system.

The queries to the program are entered in the window APT was launched from. Once a query has been finished, the user can ask for another solution to the same query. As in a toplevel, this is performed by forcing backtracking after a simulated failure. If the tree to be visualized is too large to fit in the window (which is often the case), slide bars make it possible to navigate through the execution.

APT uses Tcl/Tk [10.18] to provide the graphical interface. The original implementation of APT was developed under SICStus Prolog. It has also been ported to the clp(fd)/Calypso system developed at INRIA [10.9].

10.7 Event-based and Time-based Depiction of Control

In our experience, tree-based representations such as those of the TPM, APT, and similar tools are certainly quite useful in education and for correctness and performance debugging. In some cases, the shape of the search-tree can help in tracking down sources of unexpected low performance, showing, for example, which computation patterns have been executed more often, or which parts dominate the execution. However, the lack of a representation of time (or, in general, of resource consumption) greatly hinders the use of simple search-trees in performance debugging. AND-OR trees, as those used in APT, do not depict usually time (or in general, resource) consumption; they need to be adorned with more information.

[4] APT uses a "rich" meta-interpreter, in the sense that it keeps track of a large amount of information. In retrospect, the "rich meta-interpreter" approach has advantages and disadvantages. On one hand it allows determining very interesting information such, e.g., the origin of a given binding mentioned previously. On the other hand, it cannot cope with large executions.

One approach in order to remedy this is to incorporate resource-related information into the depiction itself, for example by making the distance between a node and its children reflect the elapsed time (or amount of resource consumed). Such a representation in *time space* provides insight into the cost of different parts of the execution: in a CLP language not all user-perceived steps have the same cost, and therefore they should be represented with a different associated height. For example, constraint addition, removal, unification, backtracking, etc. can have different associated penalties for different programs, and, even for the same program, the very same operation can cause a different overhead at different points in the execution.

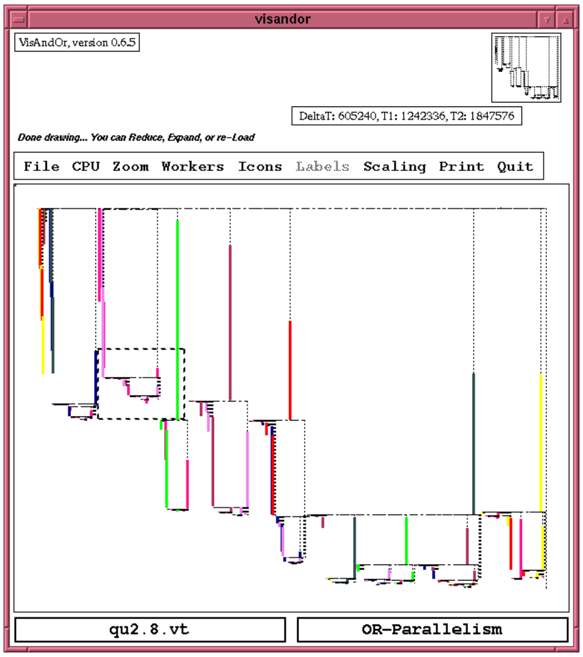

Fig. 10.5. VisAndOr showing an execution in time space

Time-oriented views have been used in several other (C)LP visualization tools, such as VisAndOr [10.7] (included with recent distributions of SICStus [10.21]) and VISTA [10.22]. VisAndOr is a graphical tool aimed at displaying and understanding the performance of parallel execution of logic programs, while VISTA focuses on concurrent logic programs. In VisAndOr, time runs from top to bottom, and parallel tasks are drawn as vertical lines (see Figure 10.5). These lines use different colors and thickness to represent which processor executes each task and the state of the task: running, waiting to

Fig. 10.6. VisAndOr showing an execution in event space

be executed, or finished. In this case, the length of the vertical lines reflects accurately a measure of the time spent.

A VisAndOr view can be described as an skeletal depiction of a logic program execution in which only nodes with relevance to the parallelism (forks & joins) were chosen to be displayed. And, as an interesting feature, the tree is adorned with tags (colors and line thickness) which add information without cluttering the display. VisAndOr allows also switching to an *event space*, in which every event in the execution (say, the creation, the start, and the end of a task, among others) takes the same amount of space—see Figure 10.6, where the same execution as in Figure 10.5 is depicted. Note that in this view the structure of the execution is easier to see, but the notion of time is lost—or, better, traded off for an alternate view. This event-oriented visualization is the one usually portrayed in the tree-like representation for the execution of logic programs: events are associated to the `calls` made in the program, and space is evenly divided among those events. Thus, event- and time-based visualization are not exclusive, but rather complementary to each other, and it is worth having both in a visualization tool aimed at program debugging.

10.8 Abstracting Control

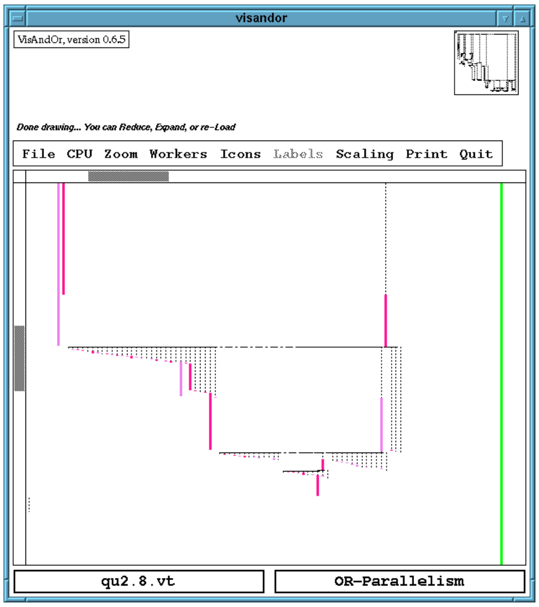

Fig. 10.7. Zoomed view in VisAndOr

The search-tree discussed throughout this chapter gives a good representation of the space being traversed. It also offers some degree of abstraction with respect to a classical search-tree by reusing the tree nodes during backtracking. But it has the drawback of being too explicit, taking up too much space, and showing too much detail to be useful in medium-sized computations, which can easily generate thousands of nodes. A means of abstracting this view is desirable.

An obvious way to cope with a very large number of objects (nodes and links) in the limited space provided by a screen is using a virtual canvas larger than the physical screen (as done by APT). However, this makes it difficult to perceive the "big picture". An alternative is simply squeezing the picture to fit into the available space; this can be made uniformly, or with a selection which changes the compression ratio in different parts of the image (this, in fact, is related to whether a time- or event-oriented view is used). The former has the drawback that we lose the capability to see the details of the execution when necessary. The latter seems more promising, since there might be parts of the tree which the user is not really interested in watching in detail (for example, because they belong to parts of the program which have already been tested).

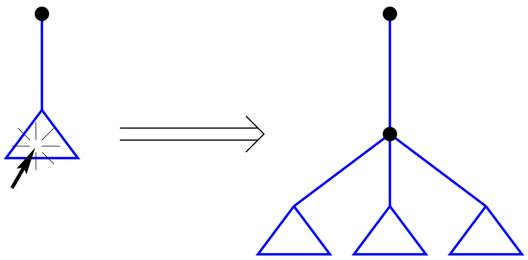

Fig. 10.8. Exposing hidden parts of a tree

An example of a tool which compresses automatically parts of the search-tree is the VISTA tool for the visualization of concurrent logic programs [10.22]. This compression is performed automatically at the points of greater density of objects—near the leaves. But this disallows blowing up those parts if a greater detail is needed. An alternative possibility is to allow the user to slide virtual magnifying lenses, which provide with a sort of fish-eye transformation and give both a global view (because the whole tree is shrunk to fit in a window) and a detailed view (because selected parts of the tree are zoomed out to greater detail). Providing at the same time a compressed view of the whole search-tree, in which the area being zoomed is clearly depicted, can also help to locate the place we are looking at; this option was already present in VisAndOr, where the canvas could be zoomed out, and the window on it was represented as a dotted square in a reduced view of the whole execution (Figure 10.7, corresponding to the dotted square in the top right corner of Figure 10.5).

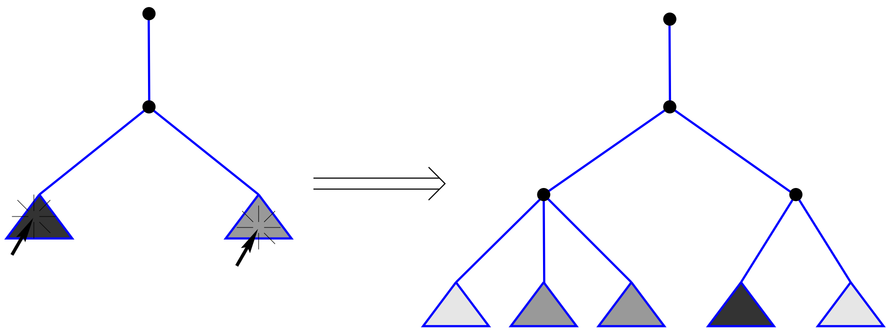

Fig. 10.9. Abstracting parts of a tree

Another possibility to avoid cluttering up the display is to allow the user to hide parts of the tree (see Figure 10.8 and [10.20]). This actually allows for the selective exploration of the tree (i.e., in the cases where a call is being made to a predicate known to be correct, or whose performance has already been tested). Whereas this avoids having too many objects at a time, feedback on the relative sizes of the subtrees is lost. It can be recovered, though, by tagging the collapsed subtrees with a mark which measures the relative importance of the subtrees. This "importance" can range from execution time to the number of nodes, number of calls, number of added constraints, number of fixpoint steps in the solver, etc.; different measures would lead to different abstraction points of view. Possible tagging schemes are raw numbers attached to the collapsed subtrees (indicating the concrete value measured under the subtree) or different shades of gray (which should be automatically re-scaled as subtrees are collapsed/expanded; see Figure 10.9). Representations of tree abstractions are currently being incorporated into APT.

10.9 Conclusions

We have presented some design considerations regarding the depiction of search-trees resulting from the execution of constraint logic programs. We have argued that these depictions are applicable to the programmed search part and to the enumeration parts of such executions. We have also presented a concrete tool, APT, based on these ideas. Two interesting characteristics of this tool are, first, the decoupling of the representation of control from the constraint domain used in the program (and from the representation of the store), and, second, the recording of the point in which every variable is created and assigned a value. The former allows visualizers for variables and constraints in different domains (such as those presented in Chapter 11) to be plugged and used when necessary. The latter allows tracking the source of unexpected values and failures.

APT has served mainly as an experimentation prototype for, on one hand, studying the viability of some of the depictions proposed, and, on the other, as a skeleton on which the constraint-level views presented in the following chapters can be attached. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed by Cosytec for the CHIP system, and which are described in other chapters.

Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments. Also thanks to Abder Aggoun and Helmut Simonis and other DiSCiPl project members for many discussions on constraint visualization.

The prototype implementation of APT was done by Ángel López [10.17].

The material in this chapter comes mainly from DiSCiPl project reports, parts of which were already presented at the AGP'98 Joint Conference on Declarative Programming [10.8].

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

10.1 A. Aggoun and H. Simonis. Search Tree Visualization. Technical Report D.WP1.1.M1.1-2, COSYTEC, June 1997. In the ESPRIT LTR Project 22352 DiSCiPl.

10.2 K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

10.3 R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

10.4 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.

10.5 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

10.6 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

10.7 M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

10.8 M. Carro and M. Hermenegildo. Some Design Issues in the Visualization of Constraint Program Execution. In *AGP'98 Joint Conference on Declarative Programming*, pages 71–86, July 1998.

10.9 Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

10.10 W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

10.11 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

10.12 Mireille Ducassé. A General Query Mechanism Based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Programming Language Implementation and Logic Programming, PLILP'92*, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, 1992.

10.13 M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

10.14 Massimo Fabris. CP Debugging Needs. Technical report, ICON s.r.l., April 1997. ESPRIT LTR Project 22352 DiSCiPl deliverable D.WP1.1.M1.1.

10.15 J.M. Fernández. Declarative debugging for babel. Master's thesis, School of Computer Science, Technical University of Madrid, October 1994.

10.16 K. Kahn. Drawing on Napkins, Video-game Animation, and Other ways to program Computers. *Communications of the ACM*, 39(8):49–59, August 1996.

10.17 A. López Luengo. Apt: Implementing a graphical visualizer of the execution of logic programs. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, October 1997.

10.18 John K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, 1994.

10.19 PrologIA. Visual tools for debugging of Prolog IV programs. Technical Report D.WP3.5.M2.2, ESPRIT LTR Project 22352 DiSCiPl, October 1998.

10.20 Christian Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *ICLP'97*. MIT Press, July 1997.

10.21 Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

10.22 Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.