facultad de informática

universidad politécnica de madrid

A User Guide to APT

A. López
alopez@dia.fi.upm.es

M. Carro
mcarro@fi.upm.es

TR Number CLIP6/97.1

A User Guide to APT

Technical Report Number: CLIP6/97.1

Authors

A. López
M. Carro
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid — Spain

Keywords

Constraint Logic Programming; Prolog; Program Visualization.

Acknowledgements

Thanks to all the members of the CLIP Group and the rest of the partners of the DISCIPL Project for useful discussions and feedback on the tool.
This work has been partially supported by the European ESPRIT LTR project DISCIPL # 22532 and Spanish CICYT Projects E96-1015-265 and TIC96-1012-C02-01.

Abstract

This is the abridged user manual of APT, a tool to visualize the execution of logic programs. APT is written in Prolog and Tcl/Tk, and runs under the X Window environment. It gives the user several facilities, including a built-in editor, the possibility of moving forwards and backwards in the execution, and the ability of providing a detailed look at the calls and runtime substitutions during the program execution.

Resumen

Este es el prontuario de APT, una herramienta de visualización de la ejecución de programas lógicos. APT está escrito en Prolog y Tcl/Tk, y se ejecuta bajo el sistema de ventanas X Window. APT ofrece diversas facilidades al usuario, incluyendo un editor integrado, la posibilidad de moverse hacia adelante y hacia atrás en la ejecución del programa, y la posibilidad de mostrar detalladamente las llamadas y las substituciones existentes en tiempo de ejecución.

Contents

1 Introduction

APT is a tool intended to show execution trees corresponding to (constraint) logic programs, including Prolog programs. It has been designed following principles similar to those of the TPM [EB88]. The current implementation supports only the Herbrand domain, but it is parametric with respect to the domain. Support for other systems, such as finite domains, is planned for the future. The tool can read and execute programs, generating a trace with information about the search tree, the variables in each call, and the 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 execution tree is represented by means of AND/OR trees, in a somewhat compacted representation. Nodes in the tree correspond to predicate calls. These nodes can be shown either in a short form (only predicate names are shown) or in expanded form (the source of the head of the matching clause and the variables and substitutions are presented as well). User-defined predicates are depicted as a box, while builtins are represented by a circle.

We will use the code in Figure 1 to illustrate how the tree representation maps to an execution. Let us assume we are making the query `a(2,2)`.

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

b(1,2).
b(2,3).

c(2,4).
c(4,6).
c(6,8).
```

Figure 1: Sample code

Figure 2 shows an execution tree corresponding to the query and program aforementioned. 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 second clause (in this case there is only one goal in that clause) are linked to the parent node with a separate set of edges. In general, edges which link nodes corresponding to goals in the same clause with the parent node (AND nodes) are crossed with a line; groups of edges like this correspond to different clauses.

The actual implementation (using Sicstus Prolog and Tcl/Tk, under X Window, Figure 3) has a graphical interface which allows the user to load and execute programs, view the resulting execution tree, and navigate through it. We will briefly describe the interface and the different actions the user can perform.

All buttons in the APT window have a character underlined, and can be can be activated by hitting the key combination *Alt+underlined character.*

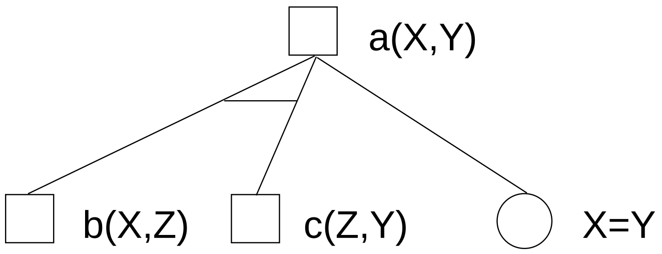
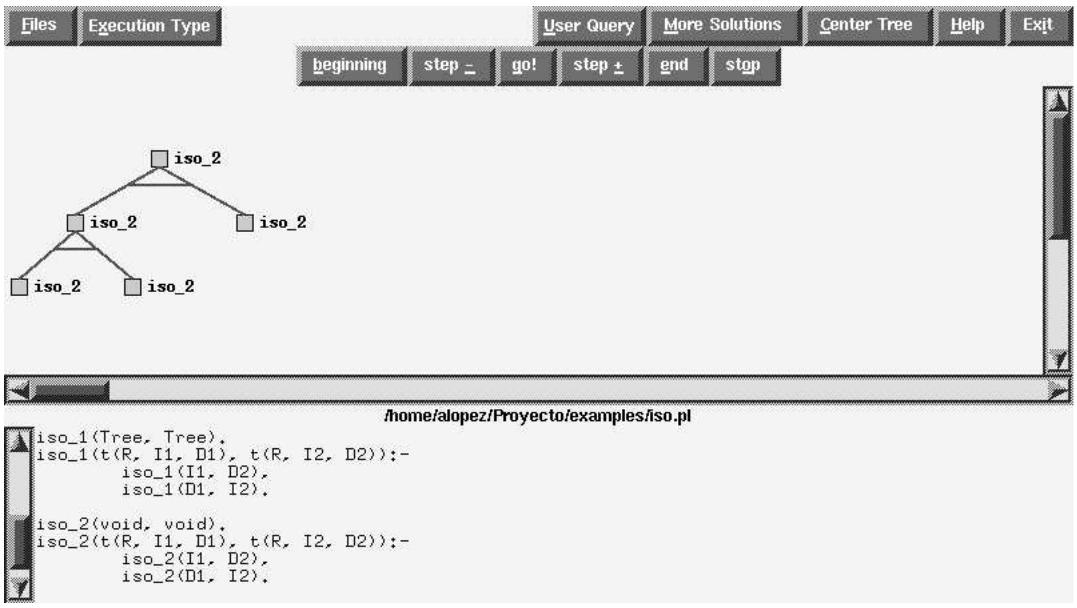


Figure 2: AND/OR Execution tree



```
iso_1(Tree, Tree).
iso_1(t(R, I1, D1), t(R, I2, D2)):-
        iso_1(I1, D2),
        iso_1(D1, I2).

iso_2(void, void).
iso_2(t(R, I1, D1), t(R, I2, D2)):-
        iso_2(I1, D2),
        iso_2(D1, I2).
```

Figure 3: The main window of the visualizer

2 Starting a Session and Making Queries

The buttons on top of the APT window allow interacting with files and setting general characteristics of the visualizer.

- *Files*: opens up a dialog window (Figure 4) which shows a list of files ending in '.pl' and subdirectories in the current directory. The window allows selecting a file and loading it. After that, the visualizer is ready to execute the program just loaded.

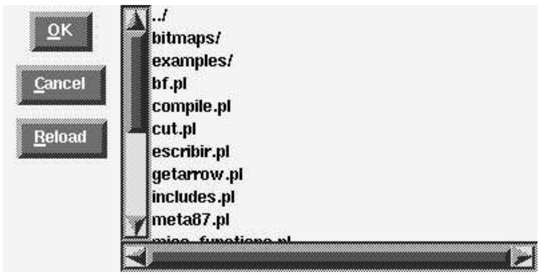


Figure 4: *Open File* dialog

  The buttons in the dialog box perform the following actions:

  - *OK*: accepts the file selected with the leftmost mouse button and reads it. Selecting a directory displays its contents. Clicking twice on a file or directory name performs the same action.
  - *Cancel*: Closes the dialog window, without loading any file or changing the state of the last file loaded.
  - *Reload*: Makes APT reload the last file read into memory. This is useful in case we had done any modification to it, and we do not want to look up the file in the directory.

  After a file is loaded, it appears in the built-in text editor (Section 6), where the user can modify and save it.

- *Execution Type*: This pops up a menu in which several parameters regarding the execution strategy can be set. The options available are:

  - *Depth First* (*default*). Executes a program with Horn clauses using a depth first, left to right strategy.
  - *Breadth First*: Allows executing a Horn clause program using a breadth-first strategy.
  - *Change Depth Search*: Modifies the maximum depth at which the visualizer stops executing the program and asks the user for confirmation to continue running the program. The default value for this depth level is 25.
  - *Show Grammar Information*: Enables showing the arguments of the standard DCGs. It is disabled by default.

- *User Query*: The user can issue a query in the window where the visualizer was launched from. Clicking on this button will pop up a menu with two entries:
  - *Query*: Causes a prompt to appear in the text window the application was launched from. A query can then be entered, which will be received by the visualizer and solved using the program loaded.
  - *Again*: The last query posted is repeated.
- *More Solutions*: The visualizer stops when the first solution is reached. Clicking on this window forces the visualizer to obtain more solutions to the query.
- *Center Tree*: Places the depiction of the search tree in its original position. Useful if it has been moved inside the canvas using the mouse (see Section 3).
- *Help*: Pops up a window with online help in hypertext format.
- *Exit*: Closes all the open windows, and quits the application.

3 Navigating Through the Execution

The visualizer shows the tree after a program query has been executed. The user can then navigate through that execution. This is done by using the second row of buttons in the main window. These buttons, and the actions they perform, are the following:

- *beginning*: Places the execution tree at the beginning of the execution (so that it can be replayed from the start).
- *end*: Places the execution tree at the end of the execution.
- *step* +: Goes one step forward in the execution, then stops and waits for more user commands. The tree is updated according to what happens in the execution.
- *step* −: Goes one step backwards in the execution, then stops and waits for more user commands. The tree is updated.
- *go!*: Goes forward in the execution automatically, starting at the point in which it was placed. The tree depiction is dynamically changed according to the events which happened in the execution.
- *stop*: Stops the animation (which can be resumed later).

Nodes in the tree are highlighted as the simulation proceeds. Different colors are used to show their state (not yet reached, running, succeeded, failed). The tree itself will eventually show only if there has been a success or a failure. The text of the clauses and the runtime substitutions can be seen in dedicated windows which are popped up by clicking the left mouse button on any tree node. These dedicated windows have their own mouse bindings (see Section 5).

4 Moving the Tree

If the tree is too large to fit in the window (which is often the case), slide bars can be used to navigate through it. Alternatively, the middle mouse button can be used to change the tree position on the canvas, clicking on any part of the tree and dragging. The tree can be restored back to its original position by using the *Center Tree* button.

5 Windows with Expanded Nodes

Any node in the tree can be *expanded* and viewed with greater detail by clicking on it with the leftmost mouse button. This makes a window similar to that in Figure 5 to show up.[1] This window presents detailed information relative to that node: the source text, the call made (with the runtime substitutions), and arrows showing the direction of the unification.

Expanded node windows allow moving back and forth in the *history* of the node. When a predicate is invoked, it receives the arguments with the (then current) runtime constraints. As the execution proceeds (both while inside the clause which corresponds to the node invocation, and after exit with success), the free variables in the call arguments can be further constrained. The different instantiation states, starting at the point of the node creation, can be seen in the expanded node window.

There is a box next to the source code, which gives information about the state of the node; see Figure 5 as a reference. A tick ($\surd$) means that the call has succeeded; a cross ($\times$) means that it has failed, and a quotation mark means that it has not finished yet. Under that symbol there is a number, which denotes the number of clauses in the called 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 tick mark; and the clause which is still under execution (if any, again) has a box with a small quotation mark in it.

Expanded node windows have special bindings for the mouse buttons.

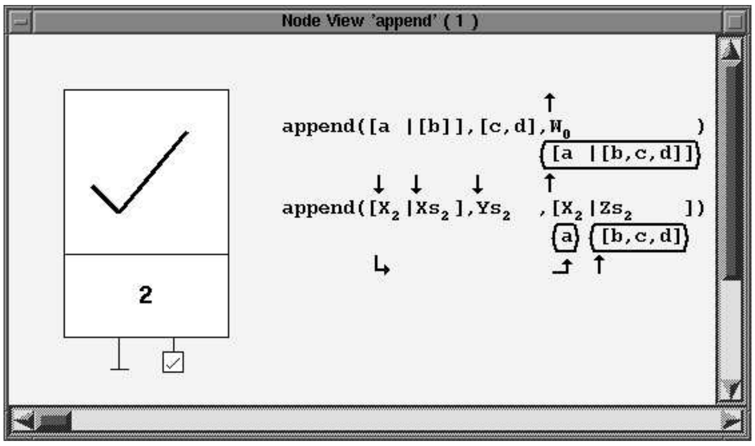
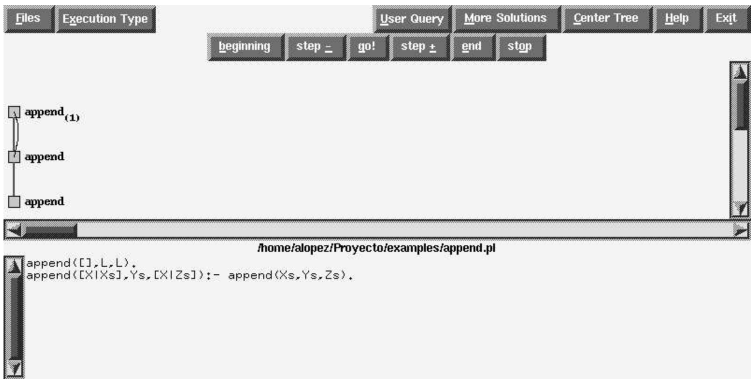

Node View 'append' (1)
2
append([a |[b]],[c,d],W0)
[a |[b,c,d]]
append([X2 |Xs2],Ys2 ,[X2 |Zs2])
a
[b,c,d]

Figure 5: Expanded node belonging to the execution of the goal `append([a,b], [c,d], W)`

[1]In the case of supporting other constraint systems, different depictions may be used — see [CH97] for details.

- Each argument in the call which is further constrained during the execution is enclosed in a rectangle with rounded corners. Clicking on the border of this rectangle will locate in the search tree the place where this argument (either variable or non-variable) was created. This is done as follows:

    1. A line is drawn in the execution tree connecting the node we are looking at and the node in which the variable/constraint was created. This allows immediately seeing where variable instantiations were generated. An example appears in Figure 6, which shows the origin of the instantiation of variable Zs_2 which appears in Figure 5.



/home/alopez/Proyecto/examples/append.pl

```
append([],L,L).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

Figure 6: A line linking variable Zs_2 in the selected node with the origin of its instantiation.

2. A new window appears, which corresponds to the node where the instantiation was originated. Figure 7 shows the window corresponding to the origin of the variable mentioned in the previous point. This window can be closed by clicking with the left mouse button anywhere in it.

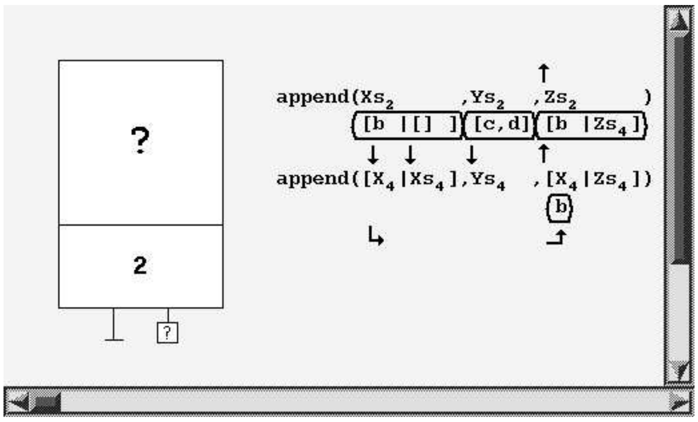


Figure 7: The node in which variable Zs_2 was instantiated

- Clicking the middle mouse button anywhere in a expanded node window will present a menu with the following entries:

  - *Show me in the Tree*: This is used to identify in the tree the node corresponding to an expanded window node. A unique identifier is added to the window title and written beside the predicate name in the corresponding node in the tree. This is useful if we have several expanded node windows open at a time.
  - *Node History*: This entry pops up a sub-menu which allows navigating forwards and backwards in the history of the node.
  - *Redraw Box*: Places the node at the point in its history where it was when first opened.
  - *Close Box*: closes the window.

- When the node corresponding to the window appears with a shadow (which means that there has been previous invocations, redone on backtracking), clicking with the leftmost button on that shadow causes the information about other invocations to appear. This actually means switching to a different universe existent in previous invocations.

6 The Text Editor

Files open from the visualizer are loaded into the text editor built-in into the visualizer. The file can then be edited and saved. The full path name of the file is shown at the top of the text editor window. This file editor offer a full range of editing commands, most of them Emacs-compatible.

- <Any-Key>, inserts characters
- <LeftmouseButton>, sets the insert point and clears the selection.
- <Control-LeftmouseButton>, sets the insert point without affecting the selection.
- <LeftmouseButton Motion>, sweeps out a selection from the insert point.
- <Double LeftmouseButton>, selects the word under the mouse.
- <Triple LeftmouseButton>, selects the line under the mouse.
- <Shift-LeftmouseButton>, adjusts the end of selection closest to the mouse.
- <Shift-LeftmouseButton Motion>, continues to adjust the selection.
- <MiddlemouseButton>, pastes the selection.
- <MiddlemouseButton Motion>, scrolls the window.
- <Key-Left> or <Control-b>, moves the cursor left one character. Also clears the selection.
- <Shift-Left>, moves the cursor and extends the selection.
- <Control-Left>, moves the cursor by words, also clears the selection.
- <Control-Shift-Left>, moves the cursor by words, extends the selection.

- <Key-Right> or <Control-f>, the same as Left operations. All Right operations are analogous to Left operations.
- <Meta-b> or <Meta-f>, same as Control-Left and Control-Right.
- <Key-Up> or <Control-p>, moves the cursor up one line. Clears the selection.
- <Shift-Up>, moves the cursor up one line. Extends the selection.
- <Control-Up>, moves the cursor up by paragraphs, which are a group of lines separated by a blank line.
- <Control-Shift-Up>, moves the cursor up by paragraph, extends the selection.
- <Key-Down> or <Control-n>, the same as the Up bindings.
- <Next> or <Prior>, moves the cursor by a screenful. Clears the selection.
- <Shift-Next> or <Shift-Prior>, moves the cursor by a screenful. Extends the selection.
- <Home> or <Control-a>, moves the cursor to line start, clears the selection.
- <Shift-Home>, moves the cursor to line start, extends the selection.
- <End> or <Control-e>, moves the cursor to line end, clears the selection.
- <Shift-End>, moves the cursor to line end, extends the selection.
- <Control-Home> or <Meta-less>, moves the cursor to the beginning of text, clears the selection.
- <Control-End> or <Meta-greater>, moves the cursor to end text, clears the selection.
- <Select> or <Control-space>, sets the selection anchor to the position of the cursor.
- <Shift-Select> or <Control-Shift-space>, adjusts the selection to the position of the cursor.
- <Control-slash>, selects everything in the text widget.
- <Control-backslash>, clears the selection.
- <Delete>, deletes the selection, if any. Otherwise deletes the character to the right of the cursor.
- <Backspace> or <Control-h>, deletes the selection if any. Otherwise deletes the character to the left of the cursor.
- <Control-d>, deletes the character to the right of the cursor.
- <Meta-d>, deletes the word to the right of the cursor.
- <Control-k>, deletes from cursor to the end of the line. If you are at the end of the line, delete the newline character.
- <Control-o>, inserts a newline but does not advance the cursor.

- <Control-w>, deletes the word to left of the cursor.
- <Control-x>, deletes the selection if any.
- <Control-t>, transposes the characters on either side of the cursor.
- <Control-x Control-s>, saves the current file. It makes a copy of the old file before the save is done.

Bibliography

[CH97] M. Carro and M. Hermenegildo. Some design issues in contraint program visualization and abstraction. Technical Report CLIP1/97.1, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, September 1997. Also as deliverable of the ESPRIT project DISCIPL.

[EB88] 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.