THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

A proposed prolog story

Citation for published version:
Bundy, A, Pain, H, Brna, P & Lynch, L 1986 'A proposed prolog story' DAI Research Paper No. 283.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Jun. 2019

A PROPOSED PROLOG STORY

Alan Bundy
Helen Pain
Paul Brna
Liam Lynch

D.A.I. RESEARCH PAPER NO. 283

Submitted to *The Journal of Logic Programming*, 1986.

Copyright (c) Alan Bundy, Helen Pain, Paul Brna and Liam Lynch, 1986

Department of Artificial Intelligence
University of Edinburgh

Abstract

A *Prolog Story* is an explanation of the workings of the Prolog interpreter or compiler[1] which a student programmer [2] can use to understand and predict the execution of a Prolog program. In previous papers [1,6] we examined the early Prolog teaching materials, [2,4], and extracted the Prolog stories they used. We concluded that there was: no complete story in use, no universal agreement on what stories to use, no consistent use of a single story, and no straightforward agreement between the stories in the teaching materials and the tracers, error messages, etc. provided in Prolog systems. We argued that this situation would be confusing for the novice programmer, especially those without previous computing experience and a weak scientific/mathematical background.

Since the publication of [1] an attempt has been made to meet some of these criticisms, e.g. by the second edition of 'Programming in Prolog' [2] and by various experimental trace packages, [4,7].

In this paper we list the topics that a complete Prolog story should cover. We describe techniques for covering these topics, drawing on the stories analysed in [1,6] and subsequent work. In particular, we address the question of how completeness can be attained without overwhelming the student with its complexity.

Keywords

Prolog Stories, Prolog, Logic Programming, Teaching Programming, Programming Environments.

---

[1] For the sake of simplicity below we will use the term 'interpreter' to refer to both the Prolog interpreter and compiler or any combination of them, except in section 3.1 where the distinction is vital.

[2] The victim of our studies can be variously thought of as a student, programmer or computer user. For the sake of consistency we use the term 'student' uniformly below.

A Proposed Prolog Story

by
Alan Bundy, Helen Pain, Paul Brna and Liam Lynch

1 Introduction

To understand the execution of a program, a model of the programming language is required. For Prolog it is necessary to explain such features as *backtracking, recursion* and *unification*. In order to do so, the model should include information about *resolution*, the *search space*, the flow of control through the search space and the Prolog *clauses*. We will call such a model, a *Prolog Story*. Our use of the word, 'story', is equivalent to du Boulay and O'Shea's 'virtual machine' in Logo, [5], and to Byrd's 'notional machine', [2].

A good Prolog story should have the following features:

- It should cover all the important aspects of Prolog behaviour, so that it can be safely used to predict the behaviour of Prolog programs.
- It should be simple to understand and use, even by people with no previous computing experience.
- It should illuminate the tricky aspects of Prolog behaviour such as pattern directed invocation, backtracking.
- It should be used universally by Prolog teachers, primers, trace messages, error messages, etc.

Our analysis in our previous papers [1,8] of Prolog books [3,6] revealed seven different stories: *OR Trees, AND/OR Trees, Byrd Boxes, Arrow Diagrams, Flow of Satisfaction, Full Traces and Partial Trees*. The first edition of 'Programming in Prolog' [3], particularly, tends to introduce new stories in an ad hoc way to explain features of Prolog, so that a student might not realise that they were all different views of the same interpreter, and hence might not develop the confidence to predict the behaviour of a previously unseen Prolog program.

In fact, none of the seven gives a complete explanation of the Prolog interpreter; each needs supplementing with one or more of the others. On the other hand, some of the stories explain the same aspect of Prolog and differ only superficially in the notation they use to do this. What is needed is a complete story that covers all the aspects of Prolog in a uniform and coherent manner. This is what we set out to do here.

2 What must a Prolog Story Cover?

Prolog is quite unlike classic imperative programming languages, such as Fortran, Pascal or Basic. It even differs significantly from functional languages, such as Lisp or Logo. For instance, it does not contain goto, global variables or assignment. The absence of these negative features, and the possibility of a declarative semantics, makes error-free Prolog easier to write and Prolog programs easier to understand.

On the other hand, Prolog still has some negative features, for instance, side-effecting evaluable predicates, such as assert/retract, cut, etc. Such 'impure' features take Prolog outside of pure logic programming, i.e. they cannot be explained by the declarative semantics. But even 'pure' Prolog has features which novice programmers find difficult to understand, such as recursion, backtracking, unification, pattern-directed control, functions as relations and scope of variables. Because of these features, some novice programmers find it particularly difficult, for instance, to write programs for building recursive data-structures, or to predict when a program will terminate and with what answer. We will concern ourselves mostly with these 'difficult to understand' features of 'pure' Prolog.

Some people have argued that the balance of negative features is against Prolog, especially as a first language for students without previous computing experience and with a weak scientific/mathematical background. Other authors, e.g. Kowalski [6], have argued that the possibility of a declarative semantics makes Prolog especially *suitable* for such students. Since most of the 'difficult to understand' features of 'pure' Prolog, e.g. unification, indirect control, are a necessary consequence of the declarative semantics, there is no way to have the best of both worlds.

Since the 'difficulty in understanding' arises only when trying to teach the procedural semantics of Prolog, one solution to this problem is to teach only the declarative semantics. This might be a viable solution in a pure logic programming language, but an understanding of the impure aspects of Prolog requires the procedural semantics. For instance, the effect of cut, assert, write, etc. cannot be predicted without an understanding of the procedural semantics. This paper proposes a framework in which the procedural semantics of Prolog can be taught.

In this paper we adopt the working hypothesis that the disadvantages of Prolog can be largely overcome by the provision of a suitable Prolog story. We note that most of the 'difficult to understand' features of 'pure' Prolog do not occur in the stories of imperative and functional languages and have been inadequately explained in the early teaching materials. We must make a determined effort to do a thorough job before giving up. Making this determined effort is all the more important because the same basic problems of understanding pattern directed invocation, etc., arise in all rule-based systems, e.g. expert systems, and are likely to arise in many of the languages for programming the new, parallel-processing computers.

The information that must be conveyed by any complete Prolog story falls under the following headings:

- the *program database*, i.e. the *assertion* and *implication clauses* that go to make up the program;
- the *search space*, i.e. the *goal literals* either given by the user or generated by Prolog during the running of the program and the relationship between them and the program clauses;
- the *search strategy*, i.e. an indication of the order in which the goal literals are generated, and the order in which the goal literals and the program clauses are chosen for resolution; and
- the *resolution process*, i.e. the unification process and the formation of the new goal literals to replace the old.

From this the student should be readily able to extract such information as: which goal literals have been resolved away and which remain; which program clauses are still to be considered; whether success has been attained; what the current bindings of a variable are; why a particular program clause was or was not chosen for resolution with a literal; and what effect a cut would have on the search space.

In our previous papers [1,8] we showed that:

- the Byrd Box and the Arrow Diagram are alternative ways of illustrating the program database and its connection to the search space;
- the AND/OR Tree, the OR Tree, the Full Trace and the Partial Tree are alternative ways of illustrating the search space and sometimes the search strategy;
- and the Flow of Satisfaction is a way of illustrating the search strategy.

We need to take the best of these partial stories and combine them into an overall account of the Prolog interpreter, adding additional material as necessary.

We would like to make definitive suggestions as to how the different aspects of the story are to be presented, but this is made impossible because of the need to present them in different media. For instance, textbook presentations are limited to the static modes of diagrams and texts. Classroom presentations using blackboard or overhead projector are similarly limited, although a small amount of dynamic presentation is possible using overlays or alteration. More is available with the use of video tapes, but this medium is expensive and time-consuming to produce and inflexible in use. Terminal presentations depend crucially on the terminal type available: all types allow considerable dynamic presentation, but a bit-mapped display, or similar, is needed for graphics.

We will see the need for graphics in the display of the search space and search strategy, and for dynamics to describe changes over time, zoom in to focus on details or zoom out to get an overview, and to give the student control over what information is displayed. Our policy below is to assume initially that all media are available and to describe our ideal mode of presentation - and only then to discuss compromises with reality.

3 A Proposed Story

In this section we consider in detail the information to be presented by the Prolog story. This is organised according to the itemized list in the previous section:

- program database,
- search space,
- search strategy and
- resolution process

3.1 The Program Database

This is the simplest part of the story to represent. The program database can be displayed as a sequence of clauses in the standard syntax. In later sections we will consider augmenting this display with additional information, but for the moment we will restrict the discussion to the clauses alone. Typically one would want to see only a small part of the whole program, e.g. all the clauses defining one predicate, which we call the *procedure* for that predicate. This is an example of focusing on part of the story.

```
location(Person, Place):-
        room(Person, Place).
location(Person, Place):-
        visit(Person, Other),
        location(Other, Place).

room(alan, room19).
room(jane, room54).
room(betty, office).

visit(dave, alan).
visit(janet, betty).
visit(lincoln, dave).
```

Figure 1: The Program Database

An example program database is given in figure 1. In this example:

- 'location(Person, Place)' means that Person is located at Place,
- 'room(Person,Place)' means that Place is the room belonging to Person, and

- 'visit(Person1,Person2)' means Person1 is visiting Person2.

There is a slightly controversial issue when the program database is displayed on a computer, namely whether the display should reflect what the student typed in or what the computer understood.

- Displaying what the student typed in will assist him/her to understand it and hence to debug it. For instance, the student is more likely to understand his/her original variable names than some arbitrary renaming of the compiler's.
- On the other hand, displaying what the computer understood will assist the student in spotting errors which have resulted in a misunderstanding, e.g. if the clauses are reformatted then this may draw attention to a clause prematurely terminated by a misplaced full stop (see figure 2).

Clause with Misplaced Full Stop

```
location(Person, Place):-
        visit(Person, Other).
        location(Other, Place).
```

Reformatted Clause

```
location(Person, Place):-
        visit(Person, Other).

location(Other, Place).
```

Figure 2: Reformatting Can Assist Bug Spotting

To display what the student typed it is sufficient to display the original source-code file, ideally in an editor window so that errors can be readily corrected and the program incrementally compiled. In practice, it may be difficult or expensive to link this display to the search space, e.g. because the compiler has an internal, object-code representation of the program database, which it is hard to decompile back to the original source-code[3].

The best of both worlds may be had by making both versions of the program database available, e.g. by optional display of both source-code and decompiled object-code, or by providing environmental tools for reformatting clauses, spotting procedures that are used but never defined, etc. Many of these environmental tools already exist, e.g. the cross referencer XREF in NIP will spot undefined procedures.

[3] Although note that NIP now displays the original variable names on listing, so it is possible to overcome these problems.

3.2 The Search Space

The candidates for representing the search space considered in our original analysis [1] were: the AND/OR Tree, the OR Tree, the Full Trace and the Partial Tree. The Full Trace is an impoverished representation which reflects the limitations of a glass teletype rather than a recommended way of displaying the search space. The Partial Tree is a version of either the AND/OR or OR Tree [4] with the details of the goal literals omitted and only the names of the program clauses labelling the arcs. This is another example of focusing on only part of the story.

So the choice is between AND/OR and OR Trees. We have come to prefer AND/OR Trees, [8], because they enable finer discrimination of the information displayed, i.e. each node is labelled by only a single goal literal rather than several, thus focusing on a single goal is simpler to do.

An example AND/OR Tree, representing a complete search space, is shown in figure 3.

- The non-terminal nodes of the tree are labelled with goal literals.
- Some terminal nodes are labelled □ to indicate that the goal literal labelling the node above has been resolved away with no further subgoals being introduced.
- Some terminal nodes are labelled with a goal literal, which indicates that they would not resolve with any program clause.
- The links are collected into AND bundles by the small arcs. Each AND bundle represents the resolution of the goal literal above it with one of the program clauses.
- The name of this program clause labels the AND bundle. For instance, the label l1 refers to the first location clause in figure 1. Similarly, for l2, v1, etc.
- Each link in the AND bundle connects the goal literal being resolved away to one of the new goal literals inherited from the body of the program clause.
- Some AND bundles are also labelled by the unifier produced by the resolution.

AND/OR Trees offer great scope for focusing. It is possible to focus on only a small part of the tree, e.g. a node and its daughters connected by a single AND bundle. Alternatively, it is possible to focus only on certain information, e.g. to give the program clauses in full, but omit the goal literals and the unifiers. In the case of blackboard or textbook presentation, the choice will depend on the topic being taught. In the case of terminal presentation the choice can be left to the student, although a default may be offered. Ideally the student should be able to dynamically change the focus by zooming in on a particular node or zooming out to get an overview of the whole tree. In the latter

[4] It is not clear which from the description in [3].

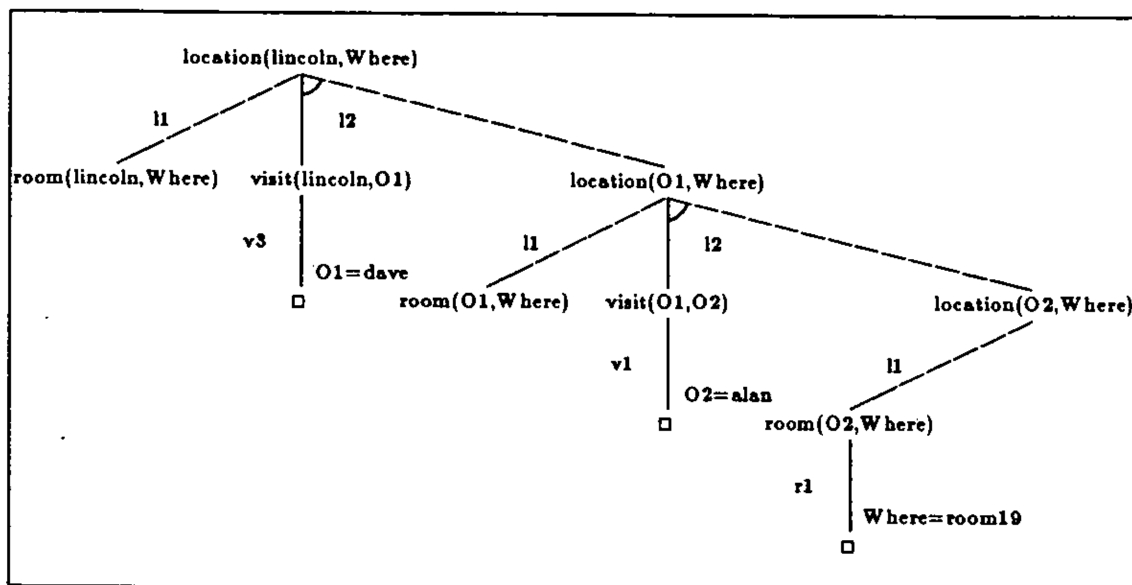

Figure 3: An AND/OR Tree

case it may be necessary to remove detail even to the extent of replacing whole subtrees with some abbreviation, e.g. a triangle.

An alternative, and more visually immediate, representation of the connection between the resolving clauses is given by an Arrow Diagram [3]. The program clauses are removed and replaced by arrows between the AND bundles in the AND/OR Tree and the appropriate clause in the program database[5]. An example is given in figure 4. To avoid overwhelming the student with detail it is usually necessary to focus on only a small part of the tree and database.

A lot of information that the student might be interested in is only represented implicitly in the AND/OR Tree.

For instance, the proof of a goal is represented by a subtree of the AND/OR Tree called the *Proof Tree* in which there are no OR choices, i.e. each node has at most one AND bundle emerging from it. The Proof Tree is an example of an *AND Tree*. Figure 5 displays explicitly the Proof Tree which is only implicitly represented in figure 3. Note that all terminal nodes are labelled □, because any failed goal literals will not contribute to the proof.

Since the Proof Tree is a subtree of the AND/OR Tree it can also be displayed by highlighting, e.g. colour, boldface, etc. Similarly, the subtree which will be pruned by a

[5]In [3] the arrow is between a goal clause from an OR Tree and the program clause. We have modified this to fit in with our proposed story based on an AND/OR Tree.

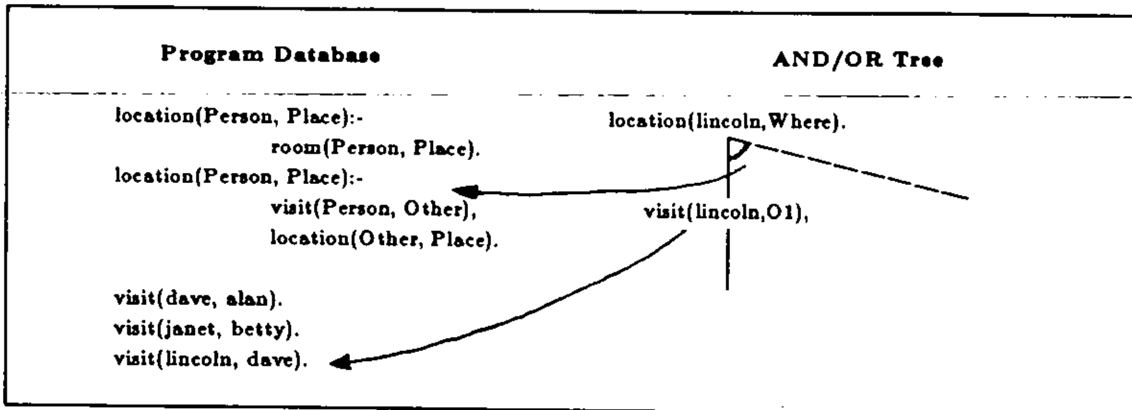

Figure 4: An Arrow Diagram

cut might be highlighted, or the pruning process might be illustrated by a pair of before and after snapshots.

The binding of a particular variable may be implicitly given by an indefinite number of unifiers. For instance, a variable, List, may be bound to [a|Tl1] in one resolution. Subsequently, Tl1 may be bound to [b|Tl2] and Tl2 to []. Thus List now has the binding [a,b], but this information is implicit in three unifiers. Only if the variable in question is in the original goal clause will its binding ever be given explicitly - and then only at the end of the procedure call. A student might be interested in the binding of some other variable and/or a partial binding of a variable during a procedure call. Variable bindings can be represented in the same form as unifiers, e.g. List=[a,b], and can optionally be used as labels on AND bundles, with highlighting to distinguish them from the unifiers. At a computer terminal the binding might be available at student request.

3.3 The Search Strategy

Given the AND/OR Tree representation of the search space and the database of clauses, the search strategy can be represented by indicating the order in which the nodes of the tree and the clauses of the database are visited. The Prolog interpreter searches the AND/OR Tree depth first. It is usual to arrange the tree so that the search proceeds from left to right. There is an attempt to unify the goal literal labelling each node with the head of each of the clauses in the procedure for that predicate. [6] This attempted unification takes place in a top to bottom order.

A stage in the growth of the AND/OR Tree can be represented by an incomplete

[6] Clever indexing is sometimes used to reduce the number of clauses on which unification is attempted.

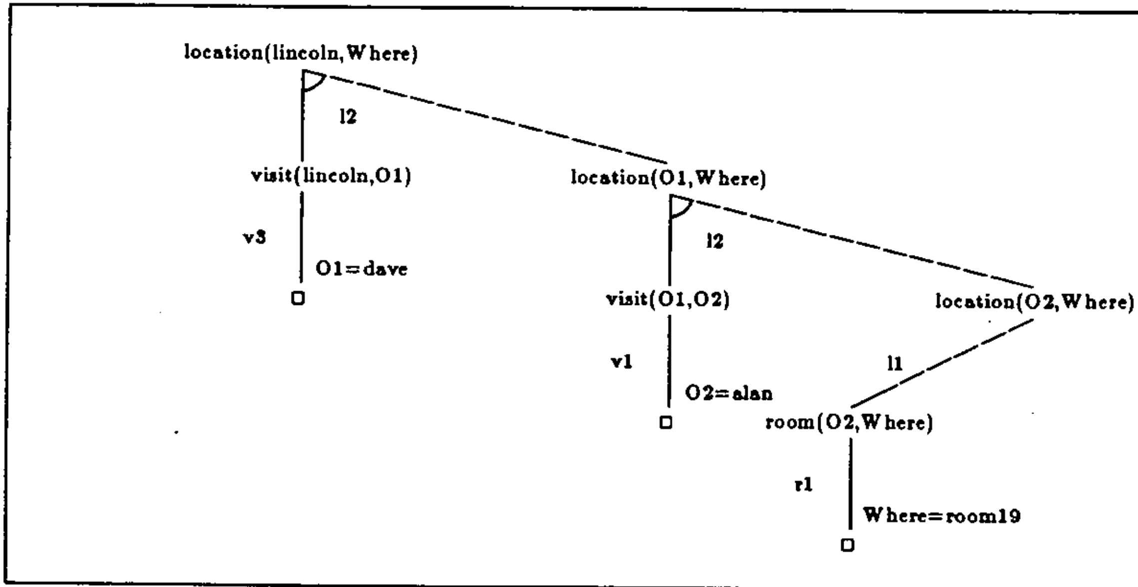

Figure 5: A Proof Tree

search tree, called an *Execution Tree* [7] (see, e.g. figure 6). The goal literal which is currently being unified against the program clauses is called the *current goal literal*. It is highlighted by underlining in figure 6. If a terminal node to the right of the current goal literal is labelled by a goal literal, it does not indicate that it cannot be resolved with anything. Rather, it indicates that no attempt at resolution has yet been made.

The order in which nodes are visited can be represented by a sequence of Execution Trees at different stages of development, e.g. before and after a particular AND bundle and its associated goal literals are added by a successful resolution, before and after a cut prunes part of the tree, etc. This technique is best suited to a naturally dynamic, graphic medium, such as a bit-mapped terminal or video tape. In a textbook or blackboard it is time-consuming to produce and overwhelming to the student, unless used sparingly.

Alternatively, the order in which the nodes of the tree are visited can be indicated by an arrow from the root of the tree to the current node which passes through all the visited nodes in order of visiting. This is essentially the representation used in the Flow of Satisfaction, [3], where the arrow is shown connecting the node labels, but all other details of the AND/OR Tree are omitted (including the links!). An example is given in figure 7.

It is also useful to show incomplete Proof Trees, since the student may want to know what parts of the search tree are still germane to the current proof, what goal literals remain to be established, etc. An example of an incomplete Proof Tree is given in

[7] Omitting part of the AND/OR Tree in this way is another example of focusing.

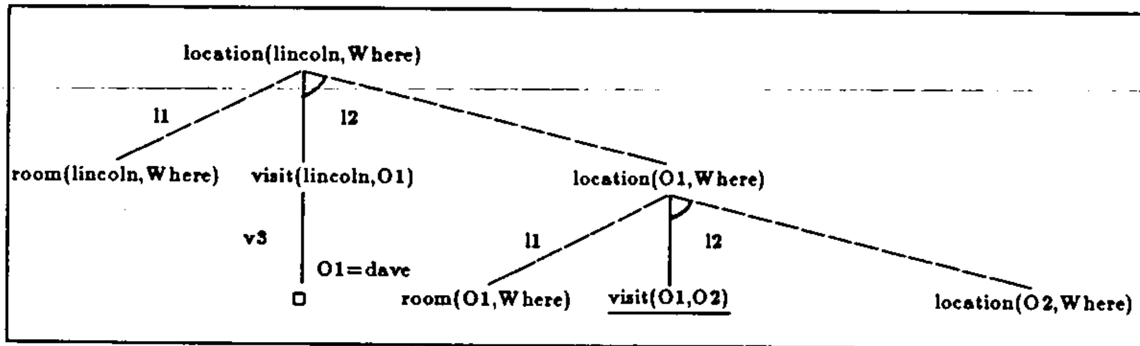

Figure 6: An Execution Tree

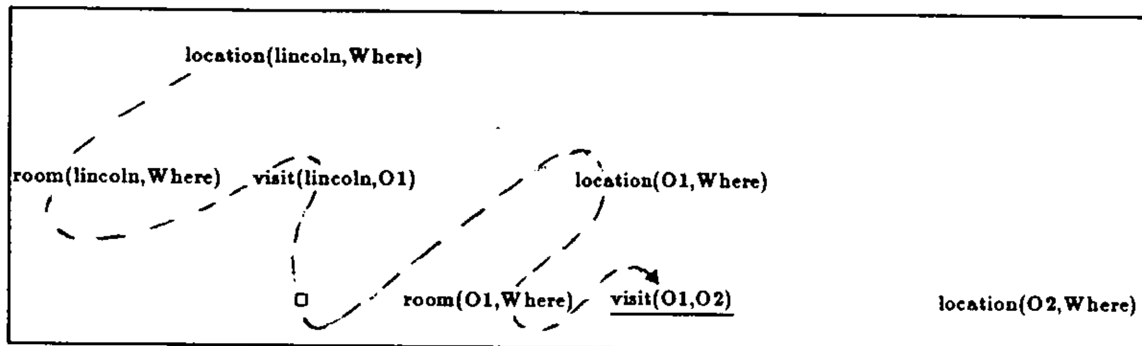

Figure 7: Flow of Satisfaction

figure 8. The current goal literal is underlined. The outstanding goal literals are those labelling the terminal nodes of this tree, namely visit(O1,O2) and location(O2,Where). The Wilk/Bowen tracer, [7], provides an option of printing this list of outstanding goal literals.

The order in which the program clauses are unified with a goal literal can be represented, similarly to the Flow of Satisfaction, by an arrow from the first clause of each procedure to the current clause (see figure 9). Note that there will be an arrow for each goal literal with the same predicate as the procedure.

Alternatively, the order could be represented by a sequence of snapshots of the Arrow Diagram, showing successive positions of the arrow from the goal literal to the program clauses. Again this technique is best suited to a naturally dynamic, graphic medium.

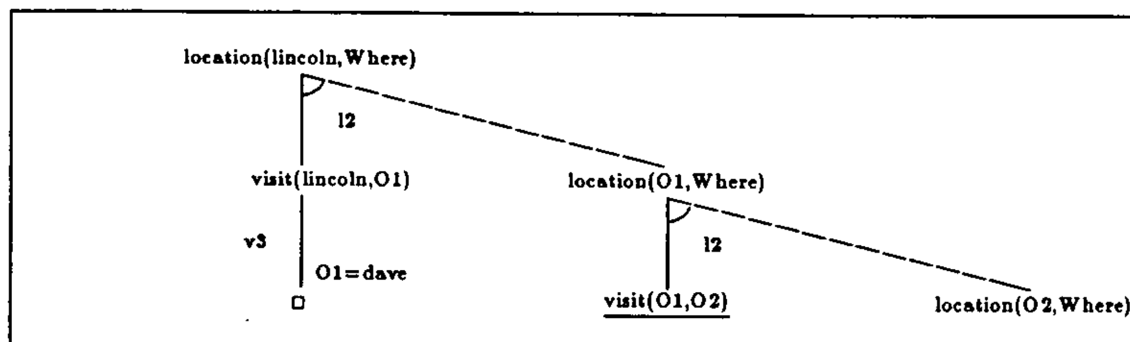

Figure 8: An Incomplete Proof Tree

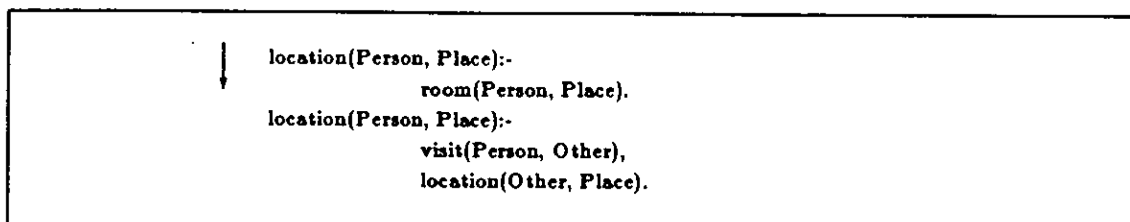

Figure 9: Order of Attempted Unification

3.4 The Resolution Process

The resolution process consists of three main parts: the unification of the goal literal with the head of the program clause, the formation of the new goal literals from the body of the program clause and the unifier, and the standardization-apart of the variables.

The formation of a resolvant is usually illustrated (c.f. [6] p 150) by listing the two resolving clauses and their resolvant and underlining the resolving literals. Figure 10 adapts this illustration to our problem of illustrating the formation of the new goal literals from an old goal literal and a program clause.

- The first row contains the program clause with the head underlined,
- the second row contains the underlined goal literal,
- the third row contains the new goal literals, and
- the fourth row contains the unifier.
- The first column contains the two resolving literals, and
- the subsequent columns contain the program clause body literals and the corresponding new goal literals.

- The lines emphasize this vertical correspondence.

We call this a *Resolution Table.*

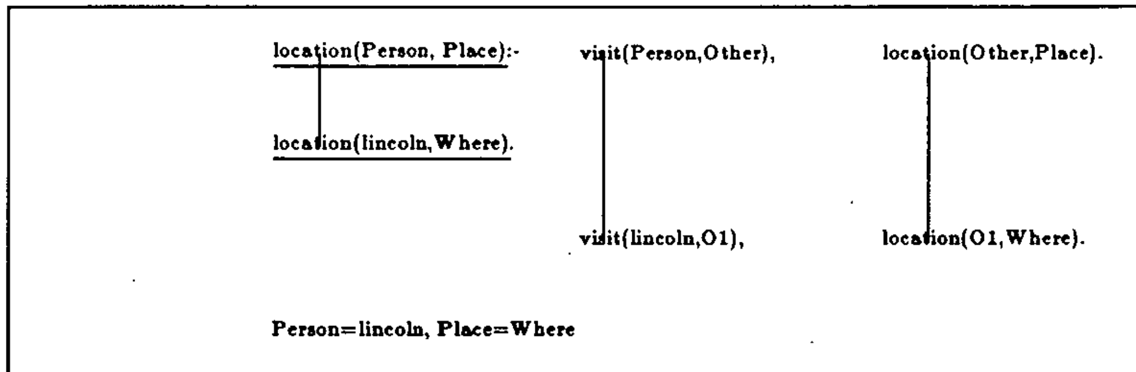

Figure 10: A Resolution Table

The Resolution Table can be readily extended to illustrate the unification process by using vertical lines to represent the correspondence between terms and by giving a sequence of Resolution Table snapshots showing successive stages during unification. One such snapshot is given in figure 11. Note that the partial unifier is applied to the new goal literals, but not the program clause. Vertical lines are drawn between those terms which have been unified so far. This technique is easily extended to complex terms consisting of functions with arguments.

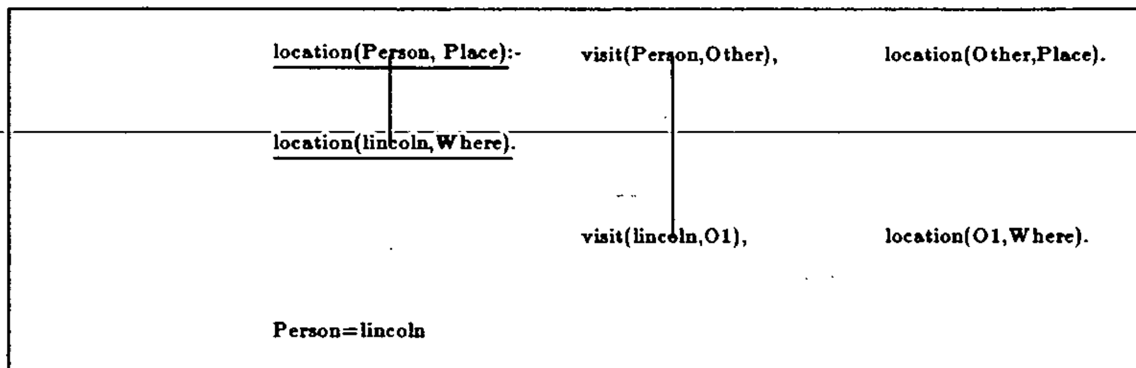

Figure 11: Snapshot of a Resolution Table

In the penultimate snapshot the unification is complete, but it remains to standardize the variables apart. This is done in the last snapshot (e.g. figure 10) by renaming any unbound variables, e.g. 'Other' to 'O1'.

Resolution Tables can also be used to illustrate failed resolutions. In this case the last snapshot will illustrate a failure of unification. A highlighted vertical line can be used to indicate the particular terms that could not be unified (see figure 12). The SODA tracer [4] is able to give similar information about both successful and unsuccessful unification attempts.

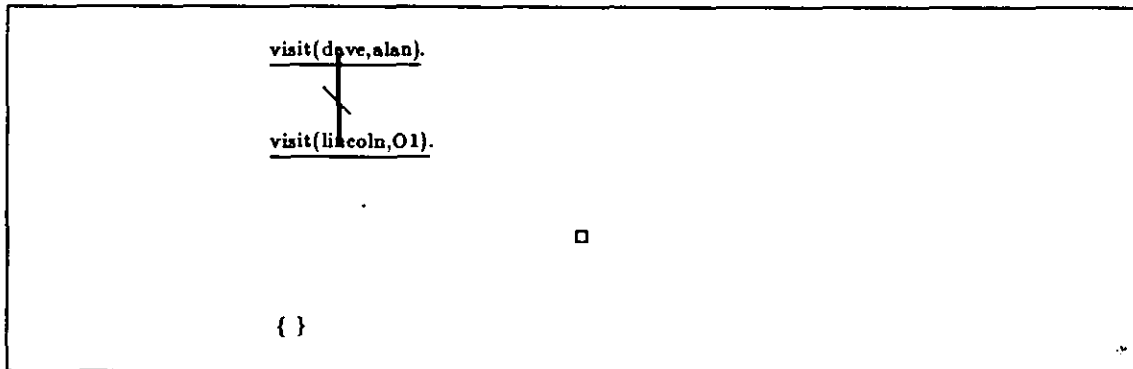

Figure 12: A Failed Resolution

3.5 Overview of Whole Story

In the sections above we have discussed each of the individual parts of the Prolog story we are proposing. In this section we put it all together. Figure 13 shows the Program Database, with the order of unification indicated, plus the AND/OR Execution Tree. The interconnections between these two are illustrated by the arrow diagram and the resolution table. To avoid cluttering the figure we have omitted much of the detail.

It is useful to present this overview in textbook and classroom to show how the various aspects of the story fit together. Except for trivial cases the amount of detail can be overwhelming unless this is only sketched in as in figure 13. On a bit-mapped terminal this overview can be presented with the aid of windows, allowing the student to focus on a particular window and zoom in for more detail. The arrows connecting the windows may be technically difficult to display, so some other way of relating the information in them may be required.

Note that multiple windows can be used. For instance, one window might be used for each procedure in the Program Database. Different resolution processes might be displayed simultaneously in separate Resolution Table windows. Different areas of the AND/OR Tree might be displayed in different windows with differing amounts of detail.

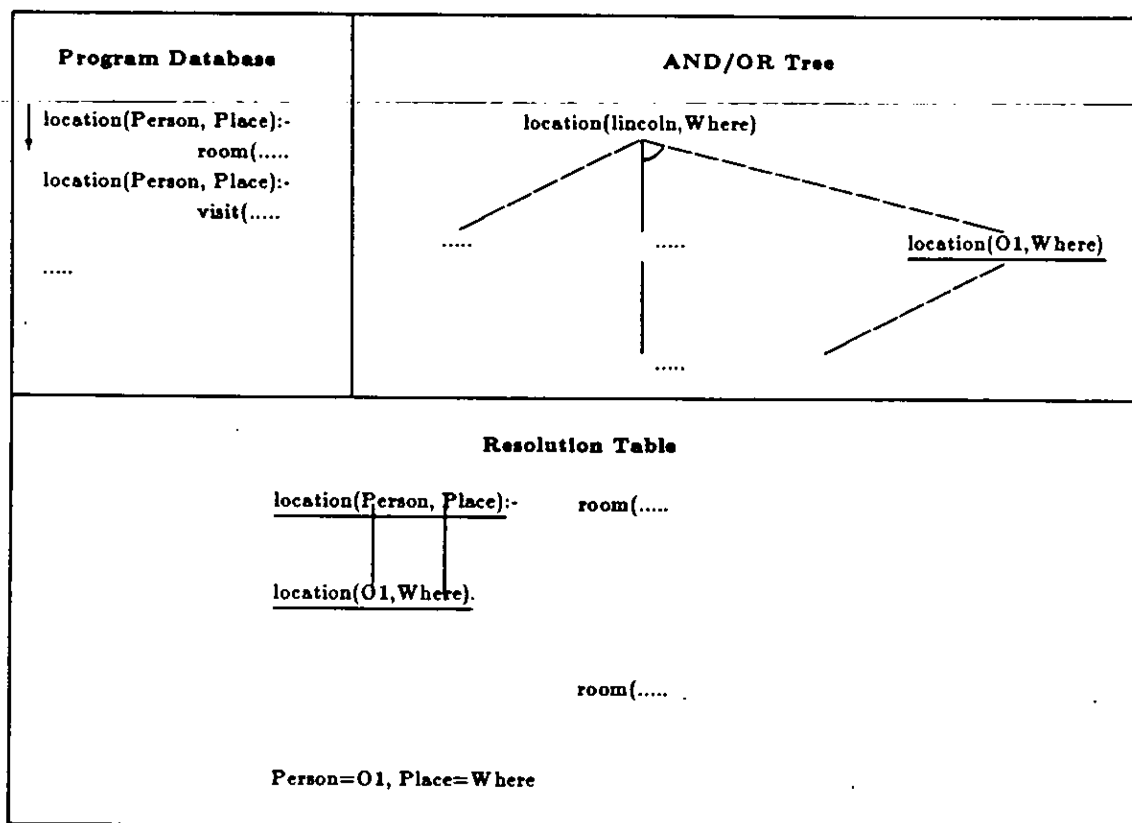

Figure 13: An Overview of the Story

4 Conclusion

In this paper we have discussed the information that a Prolog story must cover and suggested ways to represent this information in a number of media. We have suggested representations for the program database, the search space, the search strategy and the resolution process. These representations contain both text and diagrams and have both static and dynamic aspects. Versions of them can and should be used in lectures, textbooks and terminal sessions. It is vital that students receive a consistent account of the workings of the Prolog interpreter from textbooks, lectures, error messages, tracer and other environmental tools. Error messages which refer to a Prolog story that the student has never been taught are less than helpful.

The complete story about all but the most trivial Prolog procedure call contains a large amount of information. This could easily overwhelm the student. Hence, it is vital to focus on only the information germane to the point at issue and to omit the rest. This can and should be done in such a way that the student is aware of how the current partial story fits into the larger picture. Otherwise, the student may get the false impression

that different stories are needed to explain different aspects of Prolog and, hence, lack confidence in his/her ability to predict the outcome of a Prolog procedure call.

Despite our attempt at completeness, it is virtually certain that we have omitted some aspect of Prolog that should have been incorporated in the story. However, we have provided a framework into which additional information, further detail, etc., can and should be fitted.

Acknowledgements

This research was supported by ESRC grant, number H00230012. We thank the other members of the Mathematical Reasoning Group and the Programming Support Group at Edinburgh for numerous conversations and useful feedback.

References

[1] A. Bundy. *What stories should we tell Prolog Students?* Working Paper 156, Dept. of Artificial Intelligence, Edinburgh, 1984.

[2] L. Byrd. Understanding the control flow of prolog programs. In S. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–38, 1980. Available from Edinburgh as Research Paper 151.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

[4] Plummer D. *SODA: Screen Oriented Debugging Aid.* Blue Book Note 260, Dept. of Artificial Intelligence, Edinburgh, 1985.

[5] B du Boulay and T. O'Shea. Seeing the works: a strategy for teaching interactive programming. In *Proceedings of the Workshop on Computing Skills and Adaptive Systems,* Liverpool, March 1978. also available as DAI working paper no. 28.

[6] R. Kowalski. *Logic for Problem Solving. Artificial Intelligence Series,* North Holland, 1979.

[7] Wilk P. and Bowen D. *An extended AND/OR Tree debugger for Prolog.* Program support group note 49, Artificial Intelligence Applications Institute, Edinburgh, 1985.

[8] H. Pain and Bundy A. What stories should we tell novice prolog programmers. In Hawley R., editor, *Artificial Intelligence Programming Environments Book,* 1985. Also available as DAI research paper 269.