Possible Extensions to the Byrd Box tracer aimed at Experts

Discussion paper for poster presentation at PPIG 1992

Kristina Höök, Annika Wærn and Helen Pain†

SICS, Knowledge Based System Laboratory, Box 1263, 164 28 Kista, Sweden
+46 8 752 15 00 (phone) +46 8 751 72 30 (fax)
kia@sics.se annika@sics.se

Abstract

We argue for the need of a study on how experienced users make use of the Prolog tracing facilities. We know that a lot of time is spent tracing programs during the programming development phase and that often the first attempt to find a bug fails. We divide Prolog bugs into conceptual bugs, related to the problem being solved rather than to the Prolog programming language, and mistakes, related mainly to the syntax of Prolog. We argue that most Prolog bugs arise from failed unifications, and that a lot of time is spent skipping through the trace facility. Experts also spend a lot of time writing their own debuggers in order to find their conceptual bugs, since they find the tracing facilities inadequate for this purpose.

Three changes to the Byrd Box trace are suggested in order to enhance the understanding of unification and help experts to find unification bugs quicker. The first change, is to use some techniques from abstract interpretation to make recursive calls condensed into a few lines in the trace. The second idea, is to allow for conditional skipping, where the expert can ask to see a goal before or after a certain goal has unified or failed (can also be seen in the TPM [Eisenstadt and Brayshaw 88]). The third idea, is to allow the expert to only see calls that affects a certain datastructure that is being built or de-composed. The last change to the tracer might possibly allow the expert to find some of her conceptual bugs as well.

† Dept. of AI, Univ. of Edinburgh, Scotland.

1 Introduction

In the area of understanding Prolog programming, as with many other areas, most of the studies have been concerned with novices rather than experts. Novices understanding, learning, misconceptions etc., have been studied without having a real norm for how experts perform the tasks put to the novices. Many of the studies have been concerned with the procedural nature of Prolog [Fung 87, Höök et al 90, Taylor 88], and how tracers should be designed to best convey the procedural information to novices based on the knowledge of the novices from these studies [Dichev and du Boulay 89, Eisenstadt and Brayshaw 88, Fung 89]. When tracers and tools aimed at experts are designed, for example [Shapiro 83, Ducassé 91], they are not based upon studies of experts and their needs, but rather upon what the author has experienced as important and/or what is theoretically possible.

There is thus a real need for studying expert Prolog programmers and distinguish the particular mistakes they make. Apart from giving us knowledge on what sort of tracers experts would need, a study of experts would also reveal what sort of strategies they employ and how these might be used when teaching novices to program and to debug programs.

In this paper will outline a study we think should be conducted, together with some preliminary results we have obtained from interviewing a small number of experts. (This study we believe **should** be made - we have not yet done it!) We divide the bugs into two main groups:

- *conceptual bugs*: that has to do with how the problem at hand should be solved rather than the factual Prolog programming
- *mistakes*: bugs that arise from misspellings of variables, changing a predicates arity and then forgetting to change all the places from where it is called, etc.

We can observe that no bugs arise from misunderstanding the interior workings of Prolog, and that is the good thing about experts: a tracer designed for them would not have to enhance the understanding of the backtracking mechanisms. They already know the language, they have no misconceptions of it. Their only problems are slips of memory or conceptual problems at the application level. Experts already know the language, they have no misconceptions of its inner workings. Their only problems are slips of memory (or fingers).

The bad thing about experts is that the interaction with the debugger must be powerful but not too requiring. If it is too demanding they will make their own debuggers, or use simple techniques like printing out messages at various points in the program execution. This puts high demands on any design of a debugger aimed at experts.

Therefore we shall propose three extensions to the standard Byrd Box tracers that we believe would be possible and interesting for further testing. Since they are extensions to the standard tracer, they stand a better chance to be used by the Prolog community than other previously proposed Prolog environments.

2 Background

A fair amount of work has been put into attacking the debugging problem of Prolog. We can see three main strategies of how to tackle debugging: *tracers, automatic analysis,* and *debugging languages*. Each is shortly described below.

Let us just start by describing the Byrd Box trace [Byrd 80] shortly, since that is used as a standard trace for Prolog, and it is also the one we have chosen as a basis for the work presented here.

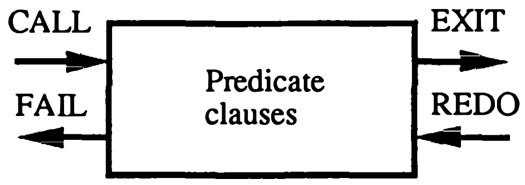

Figure 1. The Byrd Box.

The Byrd Box describes the control flow by using a box that represents the predicate that is invoced by a Prolog goal, figure 1. The invocation of the goal is the first port to that box, the so-called CALL-port. If the goal can be unified with one of the predicate clauses and its subgoals can be fulfilled, the control flow will exit through the EXIT-port. If none of the clauses unify, or their subgoals can not be fulfilled, the FAIL-port is used. If something fails, Prolog will try and go back to the previous goal that succeded and had other possible clauses. It will then enter through the REDO-port of the previous goal-box.

2.1 Tracers

The procedural side of Prolog, the flow of control, is usually presented by a tracer [Byrd 80, Plummer 88, Eisenstadt and Brayshaw 88]. The tracer will not find any bugs by itself, it will only provide insight into the control flow of the program.

It could be claimed that control flow in Prolog is simple, since there are only a few principles for how programs are executed and very few exceptions. As it turns out, the interaction between these simple principles when executing even fairly simple programs is hard to grasp [Taylor 88, Höök et al. 90, Fung 87, van Someren 87].

With novices it has been shown that the backtracking mechanism, especially together with recursive predicates is difficult to understand. Novices spend a lot of time learning how to control the backtracking, and experts often write completely deterministic programs to avoid the backtracking entirely. Unification is the other very important mechanism that makes Prolog into a powerful programming language. A lot of novices misconceptions has to do with unification.

These inherent difficulties in the language have been attacked by different solution approaches for Prolog debugging. We can see four main directions of this research: presenting backtracking information, presenting unification information, visual debugging/tracing, and faithfulness to source code.

The search mechanism in Prolog (including the backtracking order) can basically be presented in three different forms, either as an AND/OR-tree [Byrd 80], or in an AND-tree format [Rajan 86] or as an OR-tree [Höök et al 90]. The AND/OR-tree will give the fullest account of the search, but the disadvantage is that it can be hard to find the most recent choice-point in the tree<*which is where backtracking can occur*>.. The AND-tree is probably most suited for tracing deterministic programs, or succeeding goals, since it does not display the choices tried by the tracer at all. The OR-tree puts all its emphasis on the choice points, but it fails to give a compact and clear description of other parts of Prolog execution mechanisms. It does not keep the scope of a goal clear, and the goal stack (the AND-goals) still has to be displayed somehow (see [Höök et al 90] for a more thorough discussion of OR-trees).

Unification is another important aspect of Prologs control flow. Lately, a lot of attention has been given to how to display the unification in the trace. In what could be called Prologs standard-tracer, the Byrd Box tracer [Byrd 80], unification is not presented at all. Dichev and du Boulay has designed and implemented as special tracer for Prolog that gives extra attention to unification [Dichev and du Boulay 89]. Many of the more recent implementations of Prolog has included an extra line in the trace to make the unification more lucid. For instance, the Quintus Prolog has three extra ports to the Byrd Box, one of them named the Head-port which will show the clauses' heads that will be tried for unification.

With the improved graphical workstations, it has now become possible to display the AND/OR-trees in a graphical format. The graphical format could potentially give a better overview of the entire execution of a program. Examples of such graphical tracers are TPM [Eisenstadt and Brayshaw 88, 89], Vizzprol [Lazzeri 91], and the tracer designed by [Dewar and Cleary 86]. (Connected to these are visual logic programming languages like the parallel visual logic programming language designed by [Kahn and Saraswat 90].)

It is not obvious though, that these graphical tools will enhance the understanding of every Prolog program, both to novices and experts. In a recent study of Prolog novices by [Patel et al 91], it was found that certain problems are easier to solve with a textual tracer than with a graphical tracer (the TPM was used). They attribute this result to the fact that different representations of control flow will highlight different aspects of Prolog. TPM will highlight the relationship between different clauses in a Prolog program, while the textual tracer used (EBTB [Dichev and du Boulay 89]) emphasised the unification and provided a better understanding of the retry clauses.

There has been (to our knowledge) no study on expert behaviour with these visual tracers. It has been claimed [Eisenstadt and Brayshaw 88] that simply studying the shape of the execution tree would help an expert to locate bugs and to get an overview of the control flow, but this has not been proven. We believe that at current, there is no reason to assume that graphical tracers in general are better than new extensions to textual tracers.

The last issue that has been tackled in more recent tracers, concerns whether the tracer has a faithful relationship to the source code. In Vizzprool [Lazzeri 91] this aspect of tracing is emphasised. Vizzprol provides a graphical view of Prolog program. Execution is then superimposed on the graphical representation of the program. This means that the programmers variable names are not lost, as well as making it possible to recognise the code being executed and remember how it was meant to work.

2.2 Automatic and Semi-automatic bug analysis

It is very difficult to do any kind of automatic bug analysis of Prolog. There are several reasons for this: Firstly, there is not much syntax in Prolog, almost all of a Prolog program are predicate-names, variable-names, etc. that are invented by the programmer. It is therefore hard to make automatic analysis only on the syntax of a program. Secondly, there is no typing of variables or predicates, and it is very hard to derive the types from the source code, since that means trying to figure out not only where a certain datastructure has been used, but also what constitutes different datastructures. The only predefined structure is the list-structure. Thirdly, it is very hard to detect the programmers plans from analysing the source code. There are certainly programming cliches of how to write, for instance, tail-recursive predicates, but even if we are able to recognize a tail-recursive program, that gives us no clue to whether the problem has been solved correctly or not.

One way of diagnosing bugs in novices programs, has been proposed by [Looi 88]. His tutoring system, APROPOS2, adopts an approach of matching the student Prolog program against a library of task algorithms for the task. Information about mode, types, predicate names, recursion types, programming techniques used, number of clauses for each predicate definition, clause orderings, and closeness of clause matching are used to select the best algorithm that fits and the best implementations of an algorithm. The system therefore has to have an extensive knowledge of each problem the student is asked to solve. Only then is it possible to make an automatic analysis at the declarative level.

Ehud Shapiro has proposed as semi-automatic way of detecting bugs [Shapiro 83]. By using an oracle to answer questions about what various subgoals of a program was supposed to return as output, his algorithm can narrow down the search of the bug. Unfortunately, we must rely on the user to be the oracle, which is why we call it semi-automatic. It is also the case that even when the bug has been narrowed down to a specific predicate, the user still has to understand why the predicate misbehaves and correct it. Note also that it sometimes can be very difficult for the programmer to detect that a high-level goal has given the wrong answer substitution, in particular if the program manipulates large data structures.

Shapiro's algorithmic debugging is often mentioned in the group of *declarative debuggers*. By a declarative debugger is understood a method that will find bugs at the conceptual problem level. There are other methods of finding mistakes bugs, and those we call *procedural debuggers*. O'Keefe has a number of methods, for instance his XREF program that will help and prevent bugs at load time [O'Keefe 81]. XREF will check that each variable is used at least twice within every clause, and it will also check that every predicate is called from some other part of the program with the correct number of arguments. Some Prolog implementations has this (or parts of this) facility built in [Carlsson and Widén 88].

2.3 Debugging languages

An interesting new approach can be found in Mireille Ducassé's work, [Ducassé and Emde 91, Ducassé 91]. She has attempted to design a language of debugging. The language can be used in two ways. Firstly, users can build their own debuggers. Secondly, the debugging primitive predicates she provide can be used dynamically while tracing or spying a program.

Ducassé work is very new, and has not yet been tested fully, but there seems to be an argument against a too complicated approach where it is required from the Prolog learner not only to learn Prolog but also to learn a whole debugging language. The primitives she proposes also requires extensive knowledge about Prologs execution mechanism in order to be understood. Still, as we shall see below, since experts spend a lot of their time writing meta-debugger, they might make use of this debugging language.

3 Talking to experts

We have talked to a few Prolog experts, and asked them to come up with their own subjective evaluation of what might give raise to a bug in their programs. We have also asked them how they know that the program contains a bug, and how they go about correcting it. Experts own subjective evaluation of what bugs they have, is of course not a good predicator of what bugs they actually encounter, but it might give us some clue to what tools experts would make use of. A more rigid study would probably reveal more bugs concerned with the control flow of Prolog than these exports would admit that they encounter.

The seven experts we talked to are all working at SICS[1] and have a long experience of Prolog[2]. Apart from asking them questions about what bugs they believe they encounter, we also asked them to send in loggings of debugging sessions. Sofar, we have only recieved a few such loggings, and no conclusions can be drawn from those loggings. Instead we shall say a few words about where these experts thought they had bugs and how they corrected them.

3.1 Conceptual bugs and mistakes

It turns out that all the experts claims to have bugs that arise from two different "areas". The first area is where they have a problem at the problem level, not to do with the Prolog program, but with the algorithm and representation of knowledge. We name these problems *conceptual bugs*. The conceptual problems can be comcern finding the correct algorithm, and when found, accounting for **all** the cases for which is should apply. Finding the algorithm involves both knowing how to solve the problem, but also (in most cases) how to represent it.

The second area is where the experts interiewed make mistakes due to memory slips, or to limited cognitive resources. They change the number of arguments of a predicate, but forget to change all the calls to it from other parts of the program, or they misspell a variable, etc. We call those bugs *mistakes*.

Connected with the conceptual bugs is the problem of optimizing code. The experts we talked to have substantial knowledge of how Prolog is implemented, and they try to use that knowledge to speed up their application. For instance, they try to use the fact that Prolog indexes on the first argument of a predicate, and so they spend some time trying to figure out what to have in the first argument that will differ between the clauses of a predicate. We speculate that in the process of changing the program to be more efficient, bugs sometimes arises. This concern to how Prolog is implemented might also interfere with their problem solving capacity, so that too much time is spent figuring out how to implement the program most efficient rather than how to solve the problem at hand. This requires much more study to be verified. In fact, it is also the case that some of these experts claim to write very declarative code, i.e. code were the search strategy of Prolog could be changed without changing the semantics of the program.

3.2 Expert debugging strategies

How are the bugs found by experts? Even more fundamental, how does a programmer know that the program has a bug in the first place?

[1] Swedish Institute of Computer Science.

[2] Some of them has implemented logic programming systems, others have made extensions to logic programming languages, and some have made applications in Prolog.

The second question might seem irrelevant, since all a programmer has to do is to run the program in order to check that it works; if it returns faulty output or no output there is a bug. Unfortunately, this strategy will only work for very simple programs that only account for a limited amount of input/output. For more advanced applications it might very well be the case that for some input the program behaves correctly, and only in a few cases it will render faulty input. Some of our experts therefore used an extensive test bed to be used to check the program whenever a change or extension to it had been made. In NL-applications this is sometimes refered to as the test corpus. In software development it is a known strategy.

Once it has been established that there is a bug, how do experts find it?

For the first bug class, the conceptual bugs, experts seem to adapt a strategy of designing their own customized debugger. It will provide (basically) what spy provides, but it will also enhance the understanding of the problem being solved rather than the interior working of Prolog. For example, in NL processing, customized debugger will help experts to debug the grammar rather than debugging the Prolog program that is used to implement the grammar.

Another example which one of our experts were working on, was an application for testing that a electronic circuit was working properly. To test this, he had a meta-interpreter that would take a configuration of a circuit as input and by using database information about the components properties together with rules on how the components should be connected and verify the circuit. His meta-debugger would write out information on how the meta-interpreter used the rules and DB information to step by step verify the circuit.

It could be claimed that using spy and portray would accomplish the same behaviour as these meta-debuggers does. We believe that this is only partly true, but sofar we have not analysed the meta-debuggers detailed enough to state exactly what is needed apart from the spy and portray functionalities. We can give a few examples though:

1) Experts do not want to see all the arguments of a predicate.

2) Experts may want to abstract away from the datastructure even more than is possible with portray. For instance, by giving new names to various parts of the structure and gather those in new structures which are then written out at several places of the execution.

3) Experts may not want to take away the possibility to see the whole datastructure at some point during the trace. It is very difficult to find the best balance between reducing the amount of information, and still allowing the user to inspect the whole datastructure whenever needed.

Other strategies are used for finding the second class of bugs, the mistakes. One strategy is simply to study the code and try and spot the bug. If that is not possible, the tracer is often used. One expert had the following strategy for how to use the trace in these cases. He would skip at the top level of the program until he has passed the bug. He would then start all over, but when getting close to the place where the bug has manisfested itself, he would go down one level in the trace, and start skipping at that level. Eventually he would reach the point where the bugs occurred. This strategy is like a divide-and-conquer algorithm.

In the SICSTUS-Prolog [Carlsson and Widén 88] it is possible to retry the latest call that you passed while skipping, and this facility was mentioned by several experts.

Once the bug is passed through too much skipping, you can go back to the latest goal on the stack and redo it as it was executed the first time. That way you can go deeper into that call without having to retrace the whole execution from start.

Experts do complain about knowing approximately where the bug is, but not being able to skip to that point with the available traceing and spy facilities. Instead, they have to collect the output from one part of the program, and save it somehow in order to be able to use that as a direct input to the part of the program that does not work. This problem becomes acute if the input to the buggy program is very big, and impossible to write by hand.

It should be pointed out that a bug might manifest itself at another point in the trace than where the problem really originated. If a variable gets a faulty instantiatin, the effect can be a faulty success or failure much later during execution. This problem points forward to one of the changes we would like to make to the Byrd Box trace described in section 4. In this case, experts do spend some time trying to figure out where the faulty instantiation pattern came from. By allowing the expert to specify that he wants to see where a certain part of a structure has come from, the real, original, bug might be found.

4 Three enhancements worth testing?

We have adopted the viewpoint that as a first step towards better debugging environments for experts, we should think about ways to expand the available standard-tracers rather than building any new tools. It might be the case that this extension of the standard-tracer could be built using the debugging language developed by Ducassé. We have not investiaged this possibility yet. The changes we propose here will probably not cover all the extensions necessary to provide an expert with all the tools he needs. The extensions are directed foremost towards providing better support for viewing the unification and the data flow in Prolog. As such they will mostly cover the mistake bugs rather than the conceptual bugs.

Below, we have not explained how these extensions would be implemented. A post-mortem trace is needed, i.e. we first run the whole program, and then we chose which parts of it to display. The same approach is made in the TPM as well as in Opium.

4.1 Types + Abstract Interpretation

The first extension to the Byrd Box trace is to use some techniques from abstract interpretation to make recursive calls condensed into a few lines in the trace. A lot of the skipping done in the standard-tracer, has to do with skipping through a recursive predicate. Often, it would be enough to follow the recursion for the first one or two turns, and then skip all the way to a failure or the base case. If we could, in addition to this kind of skipping, already from the start make the types of the arguments to the recursive predicate available to the user, she would be able to jump more easily to the right place in the recusion in order to find the faulty behaviour or instantiation.

Abstract interpretation is a technique for global analysis of programs. Used over the unification part of the SLD-resolution it allows us to use abstract substitutions instead of running the program with the concrete datatypes. The program is executed with an abstract interpreter able to manipulate such abstract substitutions. We can use these abstract substitutions to give the user an idea of what sort of datastructure her program is generating.

Usually in abstract interpretation, the datatypes are given by the user, which gives us a typed Prolog. In our case, we would be very interested in deriving the types from the source code. We have not investigated how this would be done, or even if it is possible.

4.2 Conditional skipping

The second idea is to allow for conditional skipping, where the expert can ask to see a goal before or after a certain goal has unified or failed.

In [Eisenstadt and Brayshaw 89, Eisenstadt and Brayshaw 88] "selective highlighting" of the TPM-trace is described. This is basically what we mean by conditional skipping. It gives a possibility to ask questions like "where did variable X get instantiated to such-and-such" or "where in the program does foo get called by bar", which are answered by the TPM system through highlighting the place(s) in the graphical AND/OR-tree. The following constraints on what you want to highlight are available:

- the name and arguments of the goal,
- the parent goal name and its argument
- constraints expressed as Prolog goals, on what should hold at the time,
- if you want to see all instances, or only one, and in the latter case, should it be the first or latest, a success or fail, before or after all the conditions are fulfilled.

In the last case, where the user has asked to see one instance of a goal where the some conditions are fulfilled, the tracer is actually wind back to that point in the execution history, and the user is allowed to step back and forth, inspect unification patterns, etc., from that point.

To do the same kind of conditional skipping in the Byrd Box trace, would mean to allow for the user to specify all the above mentioned conditions, as well as conditions related to the ports of the Byrd Box. This means allowing the user also to see exit and redo ports. An enhanced Byrd Box trace should also contain various unify-ports, which could also be used in these conditions.

```
q_sort([], L, L).
q_sort([X|List], S0, S) :-
   split(X, List, Small, Big),
   q_sort(Small, S0, [X|S1]),
   q_sort(Big, S1, S).

split(_X, [], [], []).
split(X, [Y|Rest], [Y|Small], Big) :-
   X > Y,
   split(X,Rest, Small, Big).
split(X, [Y|Rest], [Y|Small], Big) :-
   X =< Y,
   split(X,Rest, Small, Big).
```

Figure 2. The faulty q_sort program.

We furthermore expect our Byrd Box tracer to be extended with an enumeration with which clause (1st, 2d, ...) is being tried. Questions like: "when does the second clause of the foo-predicate successfully unify with this goal" could then be answered. It should also be the case that we can specify following an argument irrespective of what predicate is being called.

Let us look at an example where we do want to know what happens to one of the arguments of a list that is being sorted by the q_sort program displayed in figure 2. This example is taken from [Ducassé 91], and the program contains a bug.

If we trace the goal q_sort with three elements, as below figure 3, in a full trace, we end up with a trace of 36 lines. We might then come up with the idea that we want to follow the number 3 in the initial list to be sorted, and only see exits from predicates where the 3 is involved. That trace is shown in figure 4.

```
 ?- q_sort([1,3,2],L,[]).
   1  1  Call: q_sort([1,3,2],_112,[]) ?
   2  2  Call: split(1,[3,2],_283,_284) ?
   3  3  Call: prolog:1>3 ?
   3  3  Fail: prolog:1>3 ?
   3  3  Call: prolog:1=<3 ?
   3  3  Exit: prolog:1=<3 ?
   4  3  Call: split(1,[2],_406,_284) ?
   5  4  Call: prolog:1>2 ?
   5  4  Fail: prolog:1>2 ?
   5  4  Call: prolog:1=<2 ?
   5  4  Exit: prolog:1=<2 ?
   6  4  Call: split(1,[],_747,_284) ?
   6  4  Exit: split(1,[],[],[]) ?
   4  3  Exit: split(1,[2],[2],[]) ?
   2  2  Exit: split(1,[3,2],[3,2],[]) ?
   7  2  Call: q_sort([3,2],_112,[1|_293]) ?
   8  3  Call: split(3,[2],_1439,_1440) ?
   9  4  Call: prolog:3>2 ?
   9  4  Exit: prolog:3>2 ?
   10  4  Call: split(3,[],_1561,_1440) ?
   10  4  Exit: split(3,[],[],[]) ?
   8  3  Exit: split(3,[2],[2],[]) ?
   11  3  Call: q_sort([2],_112,[3|_1449]) ?
   12  4  Call: split(2,[],_2170,_2171) ?
   12  4  Exit: split(2,[],[],[]) ?
   13  4  Call: q_sort([],_112,[2|_2180]) ?
   13  4  Exit: q_sort([],[2|_2180],[2|_2180]) ?
   14  4  Call: q_sort([],_2180,[3|_1449]) ?
   14  4  Exit: q_sort([],[3|_1449],[3|_1449]) ?
   11  3  Exit: q_sort([2],[2,3|_1449],[3|_1449]) ?
   15  3  Call: q_sort([],_1449,[1|_293]) ?
   15  3  Exit: q_sort([],[1|_293],[1|_293]) ?
   7  2  Exit: q_sort([3,2],[2,3,1|_293],[1|_293]) ?
   16  2  Call: q_sort([],_293,[]) ?
   16  2  Exit: q_sort([],[],[]) ?
   1  1  Exit: q_sort([1,3,2],[2,3,1],[]) ?

L = [2,3,1] ?
```

Figure 3. The full trace of the goal q_sort.

```
| ?- q_sort([1,3,2],L,[]).
  3  3  Exit: prolog:1=<3 ?
  2  2  Exit: split(1,[3,2],[3,2],[]) ?
  9  4  Exit: prolog:3>2 ?
  10  4  Exit: split(3,[],[],[]) ?
  8  3  Exit: split(3,[2],[2],[]) ?
  14  4  Exit: q_sort([],[3|_1449],[3|_1449]) ?
  11  3  Exit: q_sort([2],[2,3|_1449],[3|_1449]) ?
  7  2  Exit: q_sort([3,2],[2,3,1|_293],[1|_293]) ?
  1  1  Exit: q_sort([1,3,2],[2,3,1],[]) ?

L = [2,3,1] ?
```

Figure 4. A restricted trace with only exits where 3 is involved.

By only reading the first few lines of this second, restricted trace, we find that indeed number 3 has been found to be bigger than 1, but still it has ended up in the "small-list" of the split-predicate. We can then localize the bugs to the second split-clause, where big elements by mistake are place in the small-list. The correct split-definition is shown in figure 5.

```
split(_X, [], [], []).
split(X, [Y|Rest], [Y|Small], Big) :-
   X > Y,
   split(X,Rest, Small, Big).
split(X, [Y|Rest], Small, [Y|Big]) :-
   X =< Y,
   split(X,Rest, Small, Big).
```

Figure 5. Correct split-predicate.

4.3 Following a datastructure

The third idea, is to allow the expert to only see ports of the Byrd Box that affects a certain datastructure (or parts of it) that is being built or decomposed. The experts we talked to, all had datastructures that were either giant or very hard to interpret. What their meta-debuggers did, was to surpress either details of how the Prolog program worked on this datastructure or details of the datastructure. A facilitity that would enable them to see what is happening to one or several datastructures in their program, might help them overcome this difficulty.

One way to do this would be for the user to mark (click-on or by some other means) the datastructure they find interesting. Either it could be marked in the source code, or in the trace. They would then specify if they want to follow it backwards, to see where its consituents came from, or if the wanted to follow it forwards to see where parts got instantiated or taken from the structure. Ideally, it would be nice to provide an interface that could display the datastructure in another window, a *data-window*, where it during the execution would be successively built or decomposed. This interface would also provide possibilities to surpress parts of the datastructure that are irrelevant in the same manner as the in-built predicate *portray* (available in most Prolog implementations). In another window, the *trace-window*, the trace would be shown. The user would then decide to either see a full trace or a trace which only shows the ports where the datastructure is being affected.

Our extension to the standard-trace would to start with, not include this data-window, but only a facility to abstract away from the full trace to a trace where only ports affecting the datastructure we are interested in are left. Let us look at one example trace where we have followed a datastructure backwards to see how is is being construed. In figure 6 we find the program (that does not contain a bug).

```
sentence(sent(Np,Vp)) --> np(Np), vp(Vp).
np(np(Det,Noun)) --> det(Det), noun(Noun).
vp(vp(Verb)) --> verb(Verb).

det(det(the)) --> [the].
noun(noun(man)) --> [man].
verb(verb(eats)) --> [eats].
```

Figure 6. A simple NL-parsing program, from [Clocksin and Mellish 81].

A full trace of a call to the program with the sentence “The man eats” is found in figure 7. Two things are happening in this program. One structure is being built (the grammatical form of the input sentence), another one is being decomposed (the original sentence). Assume that we would like to know how the structure representating the grammatical form is being built. Figure 8 shows a trace which emphasises that aspect of the program for the NP-part of the grammatical form. In the trace we have also surpressed the extra arguments added by the DCG-formalism.

```
1 1  Call: sentence(_70,[the,man,eats],[]) ?
2 2  Call: np(_278,[the,man,eats],_286) ?
3 3  Call: det(_399,[the,man,eats],_407) ?
4 4  Call: prolog:'C'([the,man,eats],the,_407) ?
4 4  Exit: prolog:'C'([the,man,eats],the,[man,eats]) ?
3 3  Exit: det(det(the),[the,man,eats],[man,eats]) ?
5 3  Call: noun(_400,[man,eats],_286) ?
6 4  Call: prolog:'C'([man,eats],man,_286) ?
6 4  Exit: prolog:'C'([man,eats],man,[eats]) ?
5 3  Exit: noun(noun(man),[man,eats],[eats]) ?
2 2  Exit: np(np(det(the),noun(man)),[the,man,eats],[eats]) ?
7 2  Call: vp(_279,[eats],[]) ?
8 3  Call: verb(_1452,[eats],[]) ?
9 4  Call: prolog:'C'([eats],eats,[]) ?
9 4  Exit: prolog:'C'([eats],eats,[]) ?
8 3  Exit: verb(verb(eats),[eats],[]) ?
7 2  Exit: vp(vp(verb(eats)),[eats],[]) ?
1 1  Exit: sentence(sent(np(det(the),noun(man)),vp(verb(eats))),
[the,man,eats], []) ?

X = sent(np(det(the),noun(man)),vp(verb(eats))) ?
```

Figure 7. A full trace of the sentence program.

```
3 3  Exit: det(det(the)) ?
5 3  Exit: noun(noun(man)) ?
2 2  Exit: np(np(det(the),noun(man))) ?
1 1  Exit: sentence(sent(np(det(the),noun(man)),vp(verb(eats)))) ?
```

Figure 8. A restricted trace of the sentence program.

The facilitiy to follow a datastructure (or part of it) might be used both to find the mistake bugs, but also (to some extent) to find the conceptual bugs. In many cases, the problem being solved is hidden in the datstructures being sent around. This conclusion does not hold for programs that are meta-intepreters where you want to see various rules being applied and also when they are applied. But even in those cases, it is interesting to see what happens to the data when the rules are applied.

One of our experts used Prolog to develop Natural Language (NL) systems. For a NL system we believe that much would be gained if the sentence being parsed as well as the logical structure being built to represent the semantics of the sentence, could both be displayed dynamically while being built. In the corresponding trace, we might also chose only to see calls to predicates that actually do apply to the strucutre and where things actually do change in the logical form.

Most important about the approach of having a trace- and a data-window, is that it would emphasis the possibility to deal with very large datastructures. In many Prolog environments the data is often not displayed in any satisfactory way, especially not when it grows to be big. Since this approach would display the data alone and make it possible to chose which parts of the data that are to be expanded and which are surpressed in the representation, it would reduce the amount of information to a more managable amount. It would also be the case that the data would not be printed out repeatadly over and over as it is in the standard trace. Instead the instantations made with variables in the structure would replace the variables, and so the user would see the "final" structure all the time, rather than parts of it being processed at points in the exectution.

5 Conclusions

We have argued for the need of studing expert Prolog programmers. Through talking to a few experts, we have been able to outline a few ideas of how the recognise bugs and their strategies for correcting them. This requires further study.

Furthermore, we have argued that we should aim at extending the available standard tracer, and we have proposed three possible extensions.

6 Acknowledgement

We are grateful to the experts who took their time to try and find out what bugs they usually battle with. Thanks also to Sverker Jansson and Ben du Boulay for discussions about the rough ideas contained in here.

7 References

[Byrd 80] "Understanding the Control Flow of Prolog Programs", L. Byrd, In S-Å. Tärnlund (ed.) Proc. of the Logic Progr. Workshop, Debrecen, Hungary, 1980.

[Carlsson and Widén 88] "SICStus Prolog User's Manual", M. Carlsson, J. Widén, SICS Research Report, R88007C, 1988.

[Clocksin and Mellish 81] "Programming in Prolog", W.F. Clocksin and C.S. Mellish, Springer-Verlag, Berlin Heidelberg, 1981.

[Dewar and Cleary 86] "Graphical Display of Complex Information within a Prolog debugger" A.D. Dewar, J.G. Cleary, Int. J. of Man Machine Studies, 25, pp. 503 - 521, 1986.

[Dichev and du Boulay 89] "An enhanced trace tool for Prolog", C. Dichev, J.B.H. du Boulay, in Proc. of the Third International Conf. Children in the information age, 149-163, Sofia, Bulgaria, 1989.

[Ducassé and Emde 91] "OPIUM: A Debugging Environment for Prolog Development and Debugging Research", Mireille Ducassé and Anna-Maria Emde, ACM Software Engineering Notes, vol 16, no 1, pp. 54 - 59, Jan. 1991.

[Ducassé 91] "Abstract Views of prolog Executions in Opium", Mireille Ducassé, ICLP'91, San Diego, MIT-press, 1991.

[Eisenstadt and Brayshaw 88] "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming", M. Eisenstadt and M. Brayshaw, J. of Logic Progr., 5(4), pp. 277-342, 1988.

[Eisenstadt and Brayshaw 89] "An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers", M. Eisenstadt and M. Brayshaw, In E. Soloway and J. Spohrer (Eds.), "Understanding the novice programmer", Hillsdale N. J.: Lawrenece Erlbaum Associates, 1989.

[Fung 87] "Novice's Predictions of Prolog's Control Flow: A Report on an Empirical Study", P. Fung, CITE Report No. 35, Open University, UK, 1987.

[Fung 89] "An Application of Formal Semantics to Student Modeling: an investigation in the domain of traching Prolog" P. Fung, Unpublished Ph.D. thesis, CITE Ph.D. thesis No. 5, 1989.

[Hook et al. 90] "Redo 'Try Once and Pass': the influence of complexity and graphical notation on novices' understanding of Prolog", Instructional Science 19 (4-5): 337 - 360, 1990.

[Kahn and Saraswat 90] "Complete Vizualisations of Concurrent Programs and their Executions", K. A. Kahn and V. A. Saraswat, In proc. of the workshop on Visual Languages, IEEE, oct. 1990.

[Lazzeri 91] "VIZZPROL: A Tool for Vizualizing Prolog Programs" Santos Gerardo Lazzeri, Preconf. workshop of the ICLP'91, Paris, 1991.

[Looi 88] "APROPOS2: A Program Analyser for a Prolog Intelligent Teaching System", C. K. Looi, Proc. of the International Conf. on Intelligent Tutoring Systems, Montreal, 1988.

[O'Keefe 81] "Mode Error Analysis in Interpreted Code: a Prolog debugging aid", R. A. O'Keefe, Avail. from the Artifical Intelligence Applications Inst., Edinburgh Univ., 1981.

[Patel et al. 91] "Effect of format on information and problem solving", Mukesh J Patel, Benedict du Boulay, and Chris Taylor, Proc. Thirteenth Annual Conf. of COG SCI91, Chicago, 1991.

[Plummer 88] "Coda: An Extended Debugger for Prolog", Dave Plummer, in R. Kowalski and K. Bowen (eds.) "Logic Programming: Proceedings of the Fifth International Conference and Symposium", pp. 496 - 511, MIT Press, 1988.

[Shapiro 83] "Algorithmic Program Debugging" E. Y. Shapiro, MIT Press, 1983.

[Taylor 88] "Programming in Prolog: An In-Depth Study of Problems for Beginners Learning to Program in Prolog", J. Taylor, Unpublished Ph.D. thesis, CSRP 111, School of Cogn. and Soc. Sc., University of Sussex, 1988.

[van Someren 87] "What's Wrong? Understanding Beginners' Problems with Prolog", M. van Someren, Draft Research Paper, Dept. of Social Sc. Informatics, and Dept. of Experimental Psych., Univ. of Amsterdam, 1987.