A REVISED TEXTUAL TREE TRACE NOTATION FOR PROLOG

C. N. Taylor, J. B. H. du Boulay†, and M. J. Patel‡
Department of Computer Science, City University, London.
†School of Cognitive and Computing Sciences, University of Sussex, Brighton.
‡Applitech Research, Usmanpura, Ahmedabad, India.

Abstract This paper describes a "textual tree trace" (TTT) notation for representing the execution of Prolog programs. Compact, textual and non-linear, it provides detailed information about variable binding and execution history, and distinguishes several modes of goal failure. The revised form given here, yet to be tested empirically, is partly informed by Paul Mulholland's empirical comparisons of Prolog trace notations, in which an earlier version of the TTT notation was amongst those studied and criticised. The work presented here is an updated version of a previous workshop paper (Taylor, du Boulay, & Patel, 1994).

1 INTRODUCTION

Prolog is now a well-established language, with a wide range of applications. Its simple, uniform syntax and powerful inbuilt features of unification and backtracking often allow algorithms to be encoded more elegantly than in other, more conventional languages. However, these same features often present difficulties for novice Prolog programmers (Pain & Bundy, 1987; Taylor, 1988). Consequently, Prolog trace notations and tools have not only a debugging role, but also an important didactic role.

Prolog tracers vary considerably in their notations, interfaces, and the facets of Prolog execution which they display (for example, Byrd, 1980; Eisenstadt, 1984; Mellish, 1984; Eisenstadt & Brayshaw, 1988; Plummer, 1988; Horsfield, Bocca & Dahmen, 1990; Rajan, 1990). The development and now widespread availability of powerful graphical workstations and increasingly sophisticated graphical user interfaces and programming tools has provided a much wider range of possibilities to explore regarding the design of tracing tools. However, it should not be assumed that graphical modes of display are invariably superior to textual ones, particularly in the context of program tracing, where much of the information content involved is inherently textual. In practice, of course, graphical and textual approaches are not mutually exclusive, and can complement one other. Thus, although the notation described in this paper is composed entirely of text, its effectiveness in a tracing tool might well be enhanced by the use of graphical devices such as colouring, shading, flashing, and so on.

The work described here arose from a project entitled “Explanation Facilities for Prolog”, in which existing Prolog tracing tools and notations were investigated (see Patel, du Boulay & Taylor, 1991 & 1997) — particularly the standard “Spy” or “Byrd Box” textual tracer (Byrd, 1980; Clocksin & Mellish, 1981); the EPTB or “Extended Prolog Tracer for Beginners”, a prototype textual tracer giving very detailed information (Dichev & du Boulay, 1989); and the TPM or “Transparent Prolog Machine”, a graphical tree tracer developed at the Open University and available in several versions (Eisenstadt & Brayshaw, 1988; Eisenstadt, Brayshaw & Paine, 1991). During this project a new TTT (“textual tree trace”) notation and tracing tool were proposed, intended to combine some novel features with useful features from previous notations and tools. Only the notation — which uses a textual, non-linear, sideways tree format — is addressed here. Some issues relating to interface and tool design were discussed in the initial specification of notation and tool given in Taylor, du Boulay & Patel (1991).

Following the initial design, a prototype TTT meta-interpreter was implemented (in Prolog), and the notation continued to evolve. After a limited empirical study (Patel, du Boulay & Taylor, 1994) of its static features suggested that the initial notation was over-complex and potentially confusing, a simpler and more compact intermediate form was developed. Mulholland compared similar intermediate TTT notations with other Prolog trace notations in two fine-grained protocol-based studies (Mulholland 1994, 1995 & 1997). The first comparison involved novices and included initially also the Spy and PTP (both linear textual notations) and the TPM (a graphical tree notation), and subsequently the Plater notation (another linear textual notation, devised by Mulholland). The subjects used short traces in various notations, controlled via a uniform tracer interface, to deduce how a simple program being traced differed from a program text visible to them. The experimental protocols looked at the types of misunderstandings, information and strategies involved. Statistically significant differences were observed for some measures, with some notations better in some aspects but worse in others. Overall, Mulholland’s TTT performed better than the TPM and Spy, and slightly worse than the PTP, which in turn was outperformed by Mulholland’s Plater notation. In the second comparison, which involved experts, and a slightly improved TTT notation which showed intermediate bindings, the results were less clearcut, and the Prolog expertise of the subjects seem to dominate, allowing them to perform reasonably well using any of the notations concerned.

Mulholland's work goes further than previous empirical work in its detail and in looking at both static and dynamic aspects of notations. However, his experiments are still very limited in scope. His main study with novices is based on very short traces of just one program, a task (trying to find how an invisible program being traced differs from a visible program text) untypical of the way tracers are actually used in practice, and one particular selection of comparison measures. Without a much wider range of comparison measures, traces, and tasks, his conclusions must be interpreted with some caution. For example, with large traces, one might expect a non-linear notation like the TTT to perform better than than any of the linear notations, which display information in a less compact and localised way, and so would require a lot more scrolling. This was not tested in Mulholland's experiment, in which traces were short enough to fit easily on one screen window, in any notation. Nonetheless, the latest version of the TTT notation, yet to be tested empirically, is partly informed by his experimental evidence.

Mulholland's main criticisms of the earlier TTT notation — particularly regarding its use by novices - include the following:

1. **Difficulty of tracking non-linear development.** The TTT notation is non-linear, so that changes often occur within a trace, rather than always at the end, as is the case with linear notations. Comments from subjects and timing misunderstandings suggested that novices found this harder to follow than linear development, at least initially. To help with this problem, Mulholland's TTT notation distinguished the most recently activated call from the other calls.

2. **Tree display method.** TTT's sideways tree format was described by Mulholland as perceptively less clear than a vertical tree format (although the basis for this opinion was not stated explicitly). It was suggested that increasing the indentation offset of subgoals relative to their parents (originally one character-width) would help to emphasise the tree structure.

3. **Clause-goal misunderstandings.** Some novices confused goals with clauses because the TTT notation does not have separate lines marking

clause entry.

4. **Insufficient visual emphasis of call status.** The status of calls was said to be not clear enough visually, because call status information was given only at the right-hand end of each line. It was recommended that some status information should be given at the left-hand end of each call line.

Regarding 1), non-linearity cannot be abandoned without destroying the whole character of the notation, but has compensating advantages in showing the structure of the computation more explicitly. Novices may indeed find it harder to understand at first than linear development, but possibly might find it more helpful in the long run, although there is insufficient evidence to establish whether or not this is the case. To aid comprehension of the non-linear development, the latest TTT notation marks every newly appeared line or line which has just changed, so that any changes between one stage and the next can be spotted more easily and quickly.

As far as 2) is concerned, a sideways tree actually has some advantages as far as the display of textual information is concerned. The vertical tree format used in the TPM's "long-distance view" leaves insufficient space between sibling nodes to display call arguments, and in some cases even the predicate names are truncated or not shown, with the result that such information has to be accessed by opening, closing and scrolling of subwindows, rather than being immediately visible, as it is in the TTT's sideways tree format. As for indentation, a parent-child indentation offset of one character-width was chosen as the default to keep the trace compact in the left-to-right dimension, but this could easily be left for the user to adjust on request.

The lack of separate lines marking clause entries, remarked on in 3), is a consequence of a general design aim of compactness (see section 2), the overall benefits of which hopefully outweigh the drawbacks, particularly when large traces are generated. For this reason, no revisions to the TTT notation have been made in connection with this problem. However, the resulting clause-goal confusions, observed in Mulholland's experiment with novices, might perhaps be ameliorated by some kind of separate commentary line, as proposed e.g. by Rajan (1990). Another simple remedy for reducing misunderstandings — as some of the novice subjects suggested, regarding trace notations in general — might be for the tracing tool to provide a symbol key or annotated example, which could be displayed on the screen by default, at least until a user was sufficiently familiar with the notation for this to be superfluous.

Regarding 4), the latest TTT notation follows Mulholland's recommendation of giving more visual emphasis to differences in call status, by providing current status information at the left-hand end of each line, in addition to the more detailed information shown at the right-hand end of each line (see section 3 for details). One further change made to the notation is a method of showing variable bindings different to that used in the original TTT notation and in Mulholland's experiment with novices. This resembles Mulholland's later TTT notation, used with his expert subjects, in showing initial and intermediate bindings as well as call exit bindings, but improves on that further by displaying such bindings using a structured sideways tree format.

2 DESIGN PRINCIPLES

The TTT notation reflects the following design aims, which overlap with Rajan's (1990), although his concern interfaces as well as notations.

- **Localisation of information.** Information about a particular aspect of execution — e.g. a particular call or variable — should be localised, rather than being widely distributed across the trace, thereby reducing the amount of scrolling and visual scanning required to find such information.
- **Encoding of computational structure.** The overall structure of the computation should be encoded explicitly. An obvious way to do this is to adopt a tree format of some kind.
- **Correlation with source code.** The notation should facilitate correlation of trace output with the program being traced.
- **Avoidance of abstract symbolism.** Symbols should have self-evident meanings, as far as possible.
- **Explicit representation.** Information should be represented explicitly rather than implicitly, i.e. in a way that minimises the amount of inference required to extract the information. For example, the numbers of matching clauses should be shown explicitly.

- **Attention to variable bindings.** The trace should indicate the history of variable binding and unbinding, so that the construction and deconstruction of complex data-structures can be observed.
- **Cumulative notation.** At any stage, it should be possible to see the whole history of execution up to that stage, i.e. trace information should not be overwritten. Of course, one may wish to cut down on detail, but those details should be available for display.
- **Standard ASCII representation.** Traces should be constructed from the standard ASCII character set. This does not preclude the use of extra visual devices such as highlighting and colouring to aid comprehension, but it ensures that the basic notation can be displayed and printed easily on any kind of terminal or printer.

3 MAIN FEATURES

The main features of the notation are illustrated here by a summary of the symbols used (see Table 1), and by selected stages of a simple trace. The trace output is shown here with a constant level of detail; in a fully-developed TTT tracer, the level of detail would be controlled by both default restrictions and explicit user commands (for example, intermediate variable bindings would typically not be shown).

To illustrate the use of some of the symbols, consider the trace generated from the following program clauses (numerically labelled in the leftmost column).

1	`prefix([], L).`	[ ] *is a prefix of any* L
2	`prefix([H|T], [H|T1]):-`	[H|T] *is a prefix of* [H|T1]
	`prefix(T,T1).`	*if* T *is a prefix of* T1

Suppose the following query is evaluated against these clauses:

```
?- prefix(P, [a,b]), fail.
```

Comparison with the more familiar 'Spy' notation provides a useful perspective on the TTT notation. Spy tracers produce a simple linear trace, typically unindented, recording the events at each of four ports. Variables are notated using internal numbers, e.g. `_3`, which bear no relation to the variable names in the user's program. Figure 1 shows an intermediate stage

General call status notation	
?	Being tried or retried.
S	Succeeded.
F	Failed.
SF	Succeeded then failed on backtracking.
SS	Succeeded then succeeded again on backtracking.
Failure modes	
F	Default — failures resulting from subgoal failures, failures of system calls, and so on
!F	Cut failure — failure resulting from the action of the cut.
Fm	Match failure — a predicate with the same name and arity as the call exists, but none of its clauses match (or have previously matched) the call.
Fa	Arity failure — no predicate of the same name and arity as the call exists, but one of the same name and different arity does.
Fu	Undefined predicate failure — no predicate of the same name as the call exists, with or without the same arity as the call.
	Mulholland uses `Fm` differently, to mean that there are no matching clauses left, although some may have matched previously, before backtracking occurred. Thus the combination `SFm` sometimes appears in his TTT notation, but never in the TTT notation described here, which uses just `SF` in such cases, and in which `Fm`, `Fa` and `Fu` are never preceded by `S`.
Unification and binding	
/	In current binding sequence (e.g. `X/a` means `X` unified with `a`).
#	In old binding sequence, now undone (e.g. `X#a` means `X` formerly unified with `a`).
_5, _23	Numeric variable suffixes (added to variable names to distinguish different variables with the same name).
	Mulholland uses `=` and $\neq$ instead of `/` and `#`.
Call identifiers (for call number n in execution order, right-justified in a 5-character field, padded out with filler characters)	
?>>>n:	For calls currently being tried or retried (e.g. `?>>34, ?>>>9`).
S<<<n:	For calls returned successfully (e.g. `S<<<5, S<125`).
F###n:	For irretrievably failed calls (e.g. `F##23, F###8`).
	Mulholland's TTT uses the `>>>n:` prefix differently, to distinguish the most recently activated call, and — like the original TTT notation — uses a `***n:` prefix for all other calls.
Miscellaneous symbols	
|	Marks edge of block relating to a call.
*	Marks a newly appeared line or a line which has just changed.
;	Call disjunction.
(,)	Brackets for delimiting disjunctions.

Table 1: Summary of TTT notation

```
*  ?>>>1: prefix(P, [a,b])  1SF 2S?
   |1     P#[]
   |2     P/[a|T_1]/[a]
    S<<<3: prefix(T_1, [b])  1S
    |1     T_1/[]
   F###2: fail  F
   F###4: fail  F
```

```
** (1) Call : prefix(_1, [a, b])?
** (1) Exit : prefix([], [a, b])?
** (2) Call : fail?
** (2) Fail : fail?
** (1) Redo : prefix([], [a, b])?
** (3) Call : prefix(_2, [b])?
** (3) Exit : prefix([], [b])?
** (1) Exit : prefix([a], [a, b])?
** (4) Call : fail?
** (4) Exit : fail?
** (1) Redo : prefix([a], [a, b])?
```

Figure 1: TTT trace (above) Spy trace (below)

of the TTT trace (above) and the corresponding unindented Spy trace (below). At this stage, the second call to the system predicate `fail` has failed, and the initial top-level call `prefix(P, [a,b])` is being requeried. Points to note:

- The proof tree is shown here in maximum detail to elucidate the notation. A fully developed TTT interface would provide both default and user controls on the amount of detail.

- Each call is represented by its own *call block* of one or more contiguous lines: for example, the top 3 lines of the trace relate to the first call. The depth of a call in the proof tree is encoded by its call block's indentation from the left-hand margin.

- Each call block begins with a *call line*, subdivided from left to right into: the *call identifier*; the *call term*; and the *call status field* (consisting of one subfield for each matching clause tried, or just one undivided field for system predicates). For example, the first line of the trace is subdivided as follows:

Call identifier	*Call term*	*Call status field*
`?>>>1:`	`prefix(P, [a,b])`	`1SF 2S?`
		(in two parts, for clauses 1 and 2)

The initial symbol of the call identifier, in this case `?`, indicates the call's current status.

- The non-linearity of the notation is illustrated by the insertion of call 3 between calls 1 and 2. Using different line prefixes to emphasise the current status of calls is a response to Mulholland's criticism of the original notation, in which status information was confined to the call status field, and call identifiers were padded out with the same filler character `*`, regardless of current call status. Differences in status are now visually much clearer.

- The call term is shown as instantiated when the call is first made. Actual variable names are used, with numerical suffixes to distinguish different variables with the same name, e.g. the variables `T_1` and `T_2` correspond to different invocations of clause 2 of `prefix`/2. Top-level variables (in this case, `P`) are left unsuffixed.

- The call status field provides more detailed status information than the call identifier, indicating not only the call's current status, but also its previous execution history, in a compact mnemonic notation. E.g. in the top line of the trace, `1SF 2S?` shows that clause 1 (of the predicate `prefix`) matched the call, succeeded once, and failed on backtracking; and subsequently clause 2 matched, succeeded initially, and is now being requeried after further backtracking.

- Any lines in a call block after the call line show variable bindings, annotated by the clause numbers to which they relate (unless they result from system predicates), for the variables unbound in the call term when it is called. In the unabbreviated notation, the bindings shown include not just initial and final bindings, but intermediate ones too, e.g. the line `|2        P/[a|T_1]/[a]` shows an active series of bindings for `P`, associated with clause 2 of `prefix`, i.e. `P` was instantiated first to `[a|T_1]`, and then to `[a]`. The line above, `|1        P#[]`, shows an earlier binding `[]` of `P`, associated with clause 1 and now completely undone by backtracking.

- A `*` marks a line which has just appeared or just changed. Here only one line — the first — is so marked, but in general there may be several. This is helpful with a non-linear notation, in which changes may occur anywhere within the existing trace, not just at the end as is the case with linear notations.

The features just described reflect the previously stated design aims. "Call blocks", "call status fields" and "variable binding trees" all embody *localisation of information.* The explicit display of call arguments, clause numbers, 'actual' variable names and the use of a sideways top-down left-to-right tree structure all facilitate *correlation with the source code* corresponding to the trace. *Avoidance of abstract symbols* is illustrated by mnemonics (such as `F` for failure, `S` for success) and standard symbols (`/` to encode binding). Finally, the notation conforms to *standard ASCII representation*, and without sacrificing the other aims of *encoding of structure*, *explicit representation* of information, and a *cumulative notation*, it meets the important practical aim of *compactness.* In Figure 1, the TTT trace is only 7 lines compared to the Spy trace's 11 lines, even though it provides much more explicit information about matching clauses, variable bindings, and the structure of the computation. In some cases, a Spy trace will be shorter than the corresponding TTT trace, e.g. when there is no backtracking, and the calls contain on average 2 or more free variables at the time of calling. However, TTT traces are typically considerably shorter than those of any linear tracer — sometimes half the length, or less if most of the calls are fully instantiated when called.

4 DYNAMIC ASPECTS

In this section, selected stages of the trace for the `prefix` example illustrate the dynamic aspects of the TTT notation. If the query were being traced step-by-step, the tracer would stop at each stage, until prompted by the user to proceed.

Initial calling. When a call is first made, its call block contains only the call line, in which the call status field consists of a single `?` character. Any call currently being queried or requeried has a `?` at the rightmost end of its status field and a prefix of the form `?>>...` in its call identifier. The first stage of the trace illustrates this:

```
*  ?>>>1: prefix(P, [a,b])  ?
```

First top-level call

Clause head matching and resultant variable binding. When a clause head matches a call, its clause number is inserted into the call status field, to the left of the ?, and any resulting variable bindings are shown on separate lines (not only those of variables initially free in the call, but also 'knock-on effects' on other variables, as illustrated shortly). In the second line below, the 1 indicates that the sequence of bindings beginning on that line is associated with clause 1.

```
*  ?>>>1: prefix(P, [a,b])  1?
*  |1     P/[]
```

Head of clause 1 matches, and P *becomes bound to* [ ] *as a result.*

Success of a clause with no subgoals. When a clause succeeds, the ? immediately to the right of the corresponding clause number in the call status field is replaced by an S. If the clause has no subgoals, the S appears immediately after the stage in which the clause head matching is shown, as illustrated below.

```
*  S<<<1: prefix(P, [a,b])  1S
   |1     P/[]
```

Clause 1 succeeds immediately after matching, because it has no subgoals.

Calls to system predicates. The next stage shows a call to the system predicate `fail`, which has just failed, as indicated by a prefix of the form `F###` in its call identifier. It is easily identifiable as a call to a system predicate, because its status field (at the right-hand end of the call line) contains an F not preceded by any clause number. Similarly, a successful call to a system predicate would have a call status field with an S not preceded by any clause number.

```
   S<<<1: prefix(P, [a,b])  1S
   |1     P/[]
*  F###2: fail  F
```

System call fail *fails.*

Backtracking, clause retrying and variable unbinding. The next three stages illustrate backtracking, clause retrying and variable unbinding. From the second to third stages, the status field of call 1 changes from `1S?` to `1SF ?`, rather than simply to `1SF`, because there is another clause left to be tried (i.e. clause 2 of `prefix`/2) whose head also matches call 1. To represent the unbinding of `P` which accompanies the failure of clause 1, the `/` character between `P` and `[]` is replaced by a `#` character. The new binding of `[a|T_1]` for `P`, which results from the matching of clause 2 of `prefix/2` against call 1, is shown on a fresh line in the call block for call 1. The `2` on that line indicates that this binding is associated with clause 2, unlike the binding of `[]` — shown on the line above — which was associated with clause 1. Note that the binding for `P` is shown as `[a|T_1]`, not as `[a|T]`, although this is the first occurrence of `T`. Only variables mentioned in the top-level call are unsuffixed.

```
*  ?>>>1: prefix(P, [a,b])  1S?
   |1     P/[]
   F###2: fail  F
```

Re-evaluating clause 1 for call 1.

```
*  ?>>>1: prefix(P, [a,b])  1SF ?
*  |1     P#[]
   F###2: fail  F
```

Clause 1 cannot be resatisfied and so fails. P *becomes unbound.*

```
*  ?>>>1: prefix(P, [a,b])  1SF 2?
   |1     P#[]
*  |2     P/[a|T_1]
   F###2: fail  F
```

Head of clause 2 matches,

and a new binding for P *results.*

Calling of subgoals. The next stage shows the calling of the subgoal of clause 2 of `prefix/2`. Rather than being added to the end of the trace, as it would be in a linear notation, the subgoal's call line is inserted below the call block of its parent call, and immediately above the call line of the next sibling of the parent call. The indentation from the left-hand margin of the call line for the subgoal is one greater than the indentation of the call line for its parent call, thus encoding the subgoal's greater depth in the proof tree.

```
   ?>>>1: prefix(P, [a,b])  1SF 2?
   |1     P#[]
   |2     P/[a|T_1]
*   ?>>>3: prefix(T_1, [b])  ?
   F###2: fail  F
```

Subgoal of clause 2 is called.

Propagation of instantiation. The next stage illustrates how the 'knock-on' effects of variable instantiation are represented. Here, the binding of `T_1` to the value `[]` results in a further instantiation of `P` from `[a|T_1]` to `[a]`. Propagation of uninstantiation is represented in a similar way — see later.

```
   ?>>>1: prefix(P, [a,b])  1SF 2?
   |1     P#[]
*  |2     P/[a|T_1]/[a]
*   ?>>>3: prefix(T_1, [b])  1?
*   |1     T_1/[]
   F###2: fail  F
```

Head of clause 1 matches call 3, so T_1 *becomes bound, and* P *further instantiated.*

Success of a clause with subgoals. The next two stages show clause 1 succeeding for call 3; and then clause 2 succeeding for call 1 (because call 3 corresponds to the only subgoal of clause 2). Success of a call is indicated by `?` changing to `S` in its call status field, and further emphasised by a change in the prefix of its call identifier, from `?>>>` to `S<<<`.

```
   ?>>>1: prefix(P, [a,b])  1SF 2?
   |1     P#[]
   |2     P/[a|T_1]/[a]
*   S<<<3: prefix(T_1, [b])  1S
    |1     T_1/[]
   F###2: fail  F
```

Clause 1 succeeds for call 3 because it has no subgoals.

```
*  S<<<1: prefix(P, [a,b])  1SF 2S
   |1     P#[]
   |2     P/[a|T_1]/[a]
    S<<<3: prefix(T_1, [b])  1S
    |1     T_1/[]
   F###2: fail  F
```

Clause 2 succeeds for call 1 because there are no more of its subgoals to be satisfied.

Propagation of uninstantiation. A few stages later, a fresh call to `fail` (call 4) fails, and backtracking occurs. In the first stage below, clause 2 is requeried for call 1. In the second stage, clause 1 is requeried for call 3; but fails since it cannot be resatisfied, and `T_1` becomes unbound again, as shown in the third stage, which also shows the 'knock-on' effect of the partial uninstantiation of `P`, resulting from the unbinding of `T_1`.

```
*  ?>>>1: prefix(P, [a,b])  1SF 2S?
   |1     P#[]
   |2     P/[a|T_1]/[a]
    S<<<3: prefix(T_1, [b])  1S
    |1     T_1/[]
   F###2: fail  F
   F###4: fail  F
```

Attempting to resatisfy clause 2 for call 1.

```
   ?>>>1: prefix(P, [a,b])  1SF 2S?
   |1     P#[]
   |2     P/[a|T_1]/[a]
*   ?>>>3: prefix(T_1, [b])  1S?
    |1     T_1/[]
   F###2: fail  F
   F###4: fail  F
```

Attempting to resatisfy clause 1 for call 3.

```
   ?>>>1: prefix(P, [a,b])  1SF 2S?
   |1     P#[]
*  |2     P/[a|T_1]#[a]
*   ?>>>3: prefix(T_1, [b])  1SF ?
*   |1     T_1#[]
   F###2: fail  F
   F###4: fail  F
```

Clause 1 can't be resatisfied so it fails,
T_1 *becomes unbound,*
and the binding of P *reverts to* [a|T_1].

Fresh intermediate bindings. In the next stage, clause 2 matches call 3, resulting in a new binding `[b|T_2]` for `T_1`, and a new intermediate binding `[a,b|T_2]` for `P`, which is shown on a fresh line, with the same indentation as the now unbound value `[a]`. The structured display of bindings in a 'sideways tree' encodes the fact that both `[a]` and `[a,b|T_2]` are 'children' of `[a|T_1]`. This method of showing bindings is an improvement on the method used in some earlier versions of the TTT notation (including Mulholland's) in which no intermediate bindings were shown, and top-level variables were shown in several calls when they unified with clause variables, rather than just been shown at the top-level, as here.

```
   ?>>>1: prefix(P, [a,b])  1SF 2S?
   |1     P#[]
   |2     P/[a|T_1]#[a]
*  |               /[a,b|T_2]
*   ?>>>3: prefix(T_1, [b])  1SF 2?
    |1     T_1#[]
*   |2     T_1/[b|T_2]
   F###2: fail  F
   F###4: fail  F
```

Head of clause 2 matches call 3,

and a new binding of T_1 *results,*
bringing about a new binding of P *also.*

Repeated success of a clause. Several stages later, clause 2 has succeeded for call 3, and consequently, clause 2 succeeds again for call 1, indicated by a second `S` after the 2 in the status field of call 1. At this stage, the bindings of `P` associated with each success of call 1 (once using clause 1, and twice using clause 2) can be read off as the 'leaves' of the two 'sideways binding trees' for P. Thus in the tree for clause 1, there is only one leaf (the binding []); whilst in the tree for clause 2, there are two, i.e. [a] and [a,b]. The chain of bindings leading to a particular binding can be read off by following a path to that binding from the root of the binding tree which contains it: for example, the chain [a|T_1], [a,b|T_2] leads to the binding [a,b] of P.

```
*  S<<<1: prefix(P, [a,b])  1SF 2SS
   |1     P#[]
   |2     P/[a|T_1]#[a]
   |               /[a,b|T_2]/[a,b]
    S<<<3: prefix(T_1, [b])  1SF 2S
    |1     T_1#[]
    |2     T_1/[b|T_2]/[b]
     S<<<5: prefix(T_2, [])  1S
     |1     T_2/[]
   F###2: fail  F
   F###4: fail  F
```

Clause 2 succeeds for the 2nd time for call 1.

The TTT notation has some other minor features, not illustrated here. These include special notations for clause numbering in database-changing programs (those involving assert, retract, etc.), and for disjunctive calls.

5 CONCLUSION

A "textual tree trace" (TTT) notation has been described, in which the execution of a Prolog goal is represented by a 'sideways tree', growing rightward and downward from a root displayed at the top left-hand margin. This form of tree facilitates correlation of the trace with the program clauses involved in its generation — particularly if the latter are displayed with the subgoals of a clause uniformly indented with respect to the clause head. Like some previous notations, the TTT notation shows clause head matching events, distinguishes several modes of failure, and shows 'actual names' of variables as they appear in the program being traced (distinct variables with the same name are distinguished by adding numerical suffixes). The characteristic features of the notation include compactness, localisation of information pertaining to each goal, non-linear expansion of the trace and a detailed view of variable binding and unbinding. The revised form described here has yet to be empirically tested, but takes some account of Mulholland's

(1997) empirically-based criticisms of an earlier form of the notation. The main improvements made are a clearer display of call status information, explicit marking of lines where changes have just occurred, and a different and more structured way of showing variable bindings, which includes intermediate bindings. Overall, the TTT notation illustrates an approach to Prolog tracing which combines the immediately visible display of key textual information about goals and data structures, with the explicit representation of computational structure usually associated with graphical formats.

Acknowledgements

The authors thank Paul Brna and Pablo Romero for useful comments on a draft of this paper. The initial work on TTT was supported by Joint Research Council Grant no. SPG8825737. The TTT prototype was implemented in POPLOG.

REFERENCES

Byrd, L. (1980). Understanding the control flow of Prolog programs. In S-A Tarnlund (Ed)., *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary.

Clocksin, W.F. & Mellish, C.S. (1981). *Programming in Prolog.* Berlin: Springer-Verlag.

Dichev, C. & du Boulay, B. (1989). An enhanced trace tool for Prolog. *Cognitive Science Research Paper No.138*, University of Sussex.

Eisenstadt, M. (1984). A Powerful PROLOG Trace Package. *ECAI-84: Advances in Artificial Intelligence*, T. O'Shea (Ed.) Amsterdam: Elsevier Science Publishers.

Eisenstadt, M. & Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, **5**(4), 277–342.

Eisenstadt, M., Brayshaw, M. & Paine, J. (1991). *The Transparent Prolog Machine: Visualizing Logic Programs.* Oxford: Intellect Books.

Horsfield, T., Bocca, J. & Dahmen, M. (1990). MegaLog User Guide.

Mellish, C. (1984) Teach * Tracer, *POPLOG programming environment*, University of Sussex.

Mulholland, P. (1994). The effect of graphical and textual visualisation on the comprehension of Prolog execution by novices: an empirical analysis.

In *6th Workshop of the Psychology of Programming Interest Group*, 18–26, Open University.

Mulholland, P. (1995). A framework for describing and evaluating software visualisation systems: a case-study in Prolog. Ph.D. Thesis, Knowledge Media Institute, Open University.

Mulholland, P. (1997) Using a fine-grained comparative evaluation technique to understand and design software visualisation tools. In Wiedenbeck, S & Scholtz, J. (Eds.), *Empirical Studies of Programmers: Seventh Workshop*, New York, ACM Press.

Pain, H. & Bundy, A. (1987). What stories should we tell novice Prolog programmers? In Hawley, R. (Ed.), *Artificial Intelligence Programming Environments*, Chichester: Ellis Horwood.

Patel, M.J., du Boulay, J.B.H. & Taylor, C. (1991). Effect of format on information and problem solving. *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, Chicago, 852–856.

Patel, M.J., du Boulay, J.B.H. & Taylor, C. (1994). Textual Tree (Prolog) Tracer: An Experimental Evaluation. In D. Gilmore, R. Winder & F. Detienne (Eds.), *User-Centred Requirements for Software Engineering Environments*, Springer-Verlag, 127–141.

Patel, M.J., du Boulay, J.B.H., & Taylor, C. (1997). Evaluation of Contrasting Prolog Trace Output Formats, *International Journal of Human-Computer Studies*, **47**, 289–322.

Plummer, D. (1988). Coda: an extended debugger for Prolog. *Logic Programming: Proceedings of the 5th International Conference and Symposium*, 496–511, Cambridge, MA: MIT Press.

Rajan, T. (1990). Principles for the design of dynamic tracing environments for novice programmers. *Instructional Science*, **19** (4/5), 377–406.

Taylor, C., du Boulay, J.B.H. & Patel, M.J. (1991). Outline proposal for a Prolog "Textual Tree Tracer" (TTT). *Cognitive Science Research Paper No.177*, University of Sussex.

Taylor, C., du Boulay, J.B.H. & Patel, M.J. (1994). Textual tree trace notation for Prolog: an overview. In R. M. Bottino, P. Forcheri & M. T. Molfino (Eds.), *International Conference on Logic Programming ICLP'94: Post-Conference Workshop on Logic Programming and Education*, Santa Margherita Ligure.

Taylor, J. A. (1988). Programming in Prolog: an in-depth study of the problems for beginners learning to program in Prolog. D.Phil thesis, School of Cognitive and Computing Sciences, University of Sussex.