PRoVing Termination of Prolog Programs

by

N. Francez\textsuperscript{1}, O. Grumberg\textsuperscript{1}, S. Katz\textsuperscript{1}, A. Pnueli\textsuperscript{2}

Technical Report \#369

May 1985

\textsuperscript{1} Computer Science Department, The Technion, Haifa 32000, Israel

\textsuperscript{2} Dept. of Applied Mathematics, Weizmann Institute, Rehovot, Israel
PROVING TERMINATION OF PROLOG PROGRAMS

by

Nissim Francez\textsuperscript{1}, Orna Grumberg\textsuperscript{1}, Shmuel Katz\textsuperscript{1} and Amir Pnueli\textsuperscript{2}

1) Computer Science dept., the Technion, Haifa 32000, ISRAEL
2) Dept. of Applied Mathematics, Weizmann Institute, Rehovot, ISRAEL

November 1984

Introduction

The results reported in this paper are motivated by two recent trends in computer science:

\textbf{Prolog:}

Since the gain in momentum by the 5th generation computer systems project, much attention is paid to Prolog [NGE]. It is by now the best known programming language based on the \textit{Logic programming} paradigm [KOA]. Still, most of the foundational issues are treated on the level of logic programming [Ave82, vEK78], and very little basic research has been conducted on the basis of Prolog itself (or its derivatives). In contrast, much research has been done on the less theoretical, implementation oriented and application oriented levels. In particular, in the area of program verification, Prolog programs are considered by many researchers as \textit{self specifying}, being non-procedural, and not needing elaborate correctness proofs. Even if this is taken to hold with respect to partial correctness (functional behavior), it certainly is \textit{not} so with respect to \textit{termination}.

\textbf{Termination proofs:}

Since the discovery of the fundamental connection between termination proofs of (deterministic) programs and \textit{well-founded} sets [FL67], much effort has been devoted to the extensions of this fundamental connection to richer classes of programming language constructs. In particular, the method has been extended to \textit{non-deterministic} programs. The main idea there is that a well-founded quantity, known also as the \textit{variant}, should decrease under every non-deterministic choice taken by the program. Recent research, however, in non-determinism and concurrency produced a more refined notion, that of \textit{relativized termination}, where it is required that only a \textit{certain kind} of infinite computations be absent (those that "count" for some semantic reason). A proliferation of proof rules of this kind is exhibited in proving \textit{fair termination} (i.e. termination under the assumption of fair scheduling) [GFMR81, LPS-81]. As it turns out, a similar phenomenon is present in the case of Prolog. Due to its \textit{leftmost-depth-first} search mode, certain infinite computation paths will \textit{never} be followed by a terminating program. This is a major difference between Prolog and general logic programming, that considers unrestricted nondeterminism in its proof (goal satisfaction) procedure.

The contribution of this paper is in the formulation of proof rules for proving termination of Prolog programs based on well-foundedness arguments. These rules are an adequate \textit{static} characterization of the differences between the flow of control in Prolog, with its dynamic interaction between recursion and backtracking, and between that of general logic programming, the abstract \textit{non-deterministic}
inference engine. Similar rules may be designed for variants of Prolog, based on different restrictions of the search procedure, and may serve as their abstract definition.

We present two kinds of rules, based on different views of the (operational) semantics. The first is based on a tree oriented semantics. It has a global flavour, viewing the whole program as one unit, having the full context at hand all the time.

The second is a more compositional rule, where some properties are proved for each procedure separately, and then used at the point of call. The usual "proof by assumption" paradigm for (mutual) recursion is adopted. This rule is based on stream semantics, similar to that of data-flow programs. The assertion language has to be enriched accordingly.

The paper assumes basic familiarity with Prolog, especially with the relationship it induces on the interaction between backtracking and recursion. This relationship is our primary concern here; we pay less attention to another central issue in Prolog, that of unification. To the best of our knowledge, no previous attempt has succeeded in a static, proof-theoretic, characterization of these two dynamic features. For a treatment of these issues on a semantic level, in the context of non-deterministic functional programs, see [FKP 77].

2. The sub-language considered:

As stated in the introduction, we are interested here in the effect of backtracking+recursion on termination. Thus, we restrict our attention to a sub-language of Prolog rich enough to serve as a tool for the study.

A program \( P \) consists of a (finite) collection of procedures, denoted by \( p, q, r, \) etc. A procedure \( p \) consists of a (finite, positive) number of clauses. A compound clause has a head, the procedure's name and its formal parameters, and a body. For simplicity we assume that no two identical procedure names occur with different arities—common Prolog trick. The body of a compound clause consists of a finite sequence of subgoals. A subgoal names a procedure and supplies actual parameters. A simple clause (called also a fact) has a head only. As the scope rules of Prolog imply that variable names are local to clauses, we strengthen our assumptions somewhat in assuming that different procedures use different names, avoiding renaming rules or auxiliary indexing, problems orthogonal to the subject under focus here.

We assume here that the data domain is the integers, and avoid the treatment of arbitrary terms of full Prolog, which would require a notational extension only. Thus, unification in this context will only match numeric constants and variables. The usual numeric built-in procedures are allowed. Here numeric terms are always evaluated (to a constant) before unification.

We use the syntax of [CM 81] in this paper and refer the reader to it for further informal description of the semantics. Following (see figure 1) is an example program, for which a termination proof appears in the sequel. The order in which clauses are written (and that of subgoals within a goal) is crucial in the semantics of Prolog, while immaterial in the general Logic Programming paradigm.

Together with every program \( P \) is associated an initial goal \( g_0 = pr_0 (T (v_0)) \) and the task is to prove this goal using the program \( P \). By the semantics of Prolog, at each intermediate stage of a computation (proof), there is a current goal, denoted by \( g_c = pr_c (g_c) \) consisting of an ordered list of goals to be proved. The head of this list (when the list is not empty) is also referred to as the immediate goal (ig). In these goals pr is a procedure name (within \( P \)) and \( g \) a state, assigning values to all the formal parameters and local variables. In a given state, some variables may be uninstantiated, having no value at that state. Only a pair \( (P, g) \) is capable of termination. Actually, goals should have some unique labeling for identification, as multiple occurrences of the same goal may occur: We exclude their consideration to simplify the notation.
1) \( r(x) :- p(x-1,y), q(x-1,y). \)
2) \( p(0,0). \)
3) \( p(1,1). \)
4) \( p(u,v) :- p(u-1,v). \)
5) \( p(u,v) :- p(u+1,v). \)
6) \( q(0,0). \)
7) \( q(a,b) :- a > 0, q(a-1,b). \)

Figure 1: a provably B-terminating program

Finally, note that the sublanguage used excludes programs that may modify themselves by addition/deletion of rules (clauses), and features like cut, altering the flow of control.

3. B-termination

In this section we discuss in more detail the kind of termination properties exhibited by Prolog programs. It is referred to as B-termination. We refer to the \( i \)th (in order of appearance) clause as the \( i \)th direction in the (purely nondeterministic) execution of \( P \). We say that a direction \( i \) is enabled, denoted by \( E(i) \), at a given stage of a computation, if the head of the corresponding clause unifies with the head of the current goal. We use \( U(g_1, g_2) \) for the unification primitive (with its side effect). The set of all directions is denoted by \( D \). We use \( G(i) \) to denote the r.h.s. of clause number \( i \). We use the operator \( d \) to denote definiteness of variables: a variable is definite if and only if it is bound to a constant by the unification. In case two indefinite variables are equal, each binding to a constant of one of them binds the other as well.

The exposition that follows is inspired by the view of the execution of a program \( P \) as the following iterative nondeterministic program \( P^* \), in which mutual recursion is hidden by using an explicit stack of goals, the value of the state variable \( cg \) (current goal). This view mimics the operation of resolution, on which the execution of Prolog is actually based.

\[ P^* : = \bigcup_{i \in D} U(\text{head}(cg), g_i) \rightarrow cg := G(i) \cdot \text{tail}(cg)^i \]

Here the dot denotes list concatenation. Note that \( G(i) \) already reflects the side effects caused by the unification in the guard. This side effect is present also in \( \text{tail}(cg) \), as indicated by the superscript of the unified direction, \( i \). This notation is further elaborated later. The updating of the current goal introduces yet another side effect in causing all variables appearing in \( G(i) \) which are not affected by the unification to be indefinite. The program \( P^* \) terminates when the head of the current goal (i.e. the immediate goal) does not unify with any clause head. This may happen in case the current goal is the empty list, signaling success, or otherwise, signaling failure. A backtracking execution of \( P^* \) (rather uncommon in the usual context of guarded commands) induces a corresponding backtracking execution of \( P \). A standard nondeterministic execution of \( P \) induces the Logic Programming interpretation of \( P \).
We define an operational semantics for the language considered by associating with each program \( P \) and (initial) state \( \sigma_0 \) an execution tree \( T_{P, \sigma_0} \) representing all possible computations of \( P \) on \( \sigma_0 \). The construction is derived from the iterative description of \( P \) mentioned above. The nodes of the tree are labeled with states and its edges are labeled with directions.

Rules for constructing \( T_{P, \sigma_0} \):

1. The root is labeled \( \sigma_0 \).

2. Suppose some node is labeled \( \alpha \):
   - If \( \alpha \{ \text{cg} \} \neq \text{nil} \), then the node has a descendant node for each direction \( i \) s.t \( U(\text{head}(\alpha \{ \text{cg} \}), g_i) \) holds. The edge is labeled \( i \) and the descendant node \( \alpha' \), the state obtained after unification, as described
   - Otherwise (i.e \( \alpha \{ \text{cg} \} = \text{nil} \)), the node is a failing leaf.

3. When convenient, we abbreviate the tree to \( T_{P, \sigma} \) and depict in the node only the value of \( \text{cg} \). Note that for a B-terminating program the tree never degenerates to a single node (being both a root and a leaf).

Before passing to discuss termination, we formulate two axioms for the unification operation. This operation is special in serving as a guard with side effects. Both aspects have to be captured by the axioms. In the sequel, we apply these axioms implicitly whenever needed, without further mention. The axioms are shown in figures 2 and 3. They are partial correctness axioms, stating a precondition under which if unification successfully terminates, it yields the post condition.

We start the discussion of termination by considering a very simple example (figure 4). The collection of all possible executions of an initial goal, say \( g_0 = p(3) \), can be arranged in a tree \( T_{(p, p(3))} \) (see figure 5). At each node of this tree reside the current goal, and it has outgoing edges for all the clauses of the procedure specified by this goal that unify with its first subgoal. In case of fact, an extra edge, leading to a successful leaf, is added. Similarly, in case of failure due to nonunifiability of (the head of) the current goal, a failure leaf is added. Clearly, the tree \( T_{(p, p(3))} \) contains many infinite paths. However, due to Prolog's leftmost—depth-first traversal of this tree, no infinite path will ever be traversed due to the presence of a successful leaf, a left descendant of \( p(0) \), which, when encountered, terminates the computation.
For $1 \leq i \leq n$:

$$\{ (-d(\text{arg}_i) \land -d(\text{arg}_j)) \}$$

$$\cup (p(\text{arg}_1, \ldots, \text{arg}_n), p(\text{arg}_2, \ldots, \text{arg}_n))$$

$$\{ (-d(\text{arg}_i) \land -d(\text{arg}_j)) \land \text{arg}_i = \text{arg}_j \}$$

**Figure 3:** indefinite unification axiom

$P$: 1) $p(0)$.
2) $p(x) :- p(x-1)$.
3) $p(x) :- p(x+1)$.

**Figure 4:** simple example program $P$

![Diagram of tree T(P, P(3))]

**Figure 5:** the tree $T_{(P, P(3))}$

This example is oversimplified since $(P, p(3))$ terminates without ever backtracking. Once backtracking is also considered, one gets the following representation for a terminating program $(P, g_0)$, depicted as a tree $T_{(P, g_0)}$ in figure 6. In
Figure 6: the tree $T(P,g_0)$
(terminating program)

this tree, the path $\pi$ is the leftmost infinite path. To its left is a successful leaf $\sigma_1$ (in a double circle), and further to its left are $k \geq 0$ failing leaves $\sigma_j$, $1 \leq j \leq k$, where the procedure names in current goals of leaves are omitted for brevity.

Thus, $T(P,g_0)$ can be divided into two subtrees:

1) The tree $T(P,g_0)$ to the left (and not including) the leftmost infinite path $\pi$, which has to be finite and contain a successful (rightmost) leaf, and a finite number of failing leaves.

2) The tree $T(P,g_0)$ to the right of (and including) $\pi$, which may be infinite.

In the sequel, we omit references to the subscripts $(P,g_0)$ when clear from context.

For a nonterminating $(P,g_0)$, we get the dual picture as in figure 7. The tree is again divided as before. However, the leftmost successful leaf (if such is present) is to the right of $\pi$, the leftmost infinite path. In this case, Prolog's search mode will not terminate the execution, as it backtracks from all the failing leaves and "bumps into" $\pi$, on which it continues indefinitely.

There is yet a third possibility, that of the whole program terminating with a failure of the initial goal. This is an "uninteresting" case, as the tree $T(P,g_0)$ is finite, with all the leaves failing. Thus, our main concern is of statically distinguishing between the first two possibilities, the third being provable using the ordinary rule for non-deterministic termination (decreasing along every direction).

We call this kind of termination $B$-termination (for Backtracking termination) and denote it by

$$B \begin{bmatrix} \text{pre} \end{bmatrix} (g_0) \begin{bmatrix} \text{post} \end{bmatrix}$$

where $\text{pre}, \text{post}$ are the usual precondition and postcondition, respectively. The proposed proof-rule for $B$-termination is referred to as $BT$, while the usual rule for non-deterministic termination is $NDT$. 
The main idea behind the suggested rule \( BT \) is to identify, by means of decreasing well-founded variant the path \( \pi^* \) to the leftmost successful leaf, and showing that all the subtrees to its left are finite and have failing leaves. Thus, for each node on \( \pi^* \) the set of enabled directions is partitioned, depending on its variant value \( w \in W \), into three classes:

i. A direction \( \ell_w \) that causes a decrease in \( w \).

ii. The set \( F_w = \{ j \mid 1 \leq j \ll \ell_w, E(j) \} \), which leads to states to which \( NDT \) can be applied successfully.

iii. The set of all other directions \( \ell > \ell_w \), i.e., enabled, on which no restriction whatsoever is imposed.

Also, when a node with variant value \( w = 0 \) is reached, that state has to imply that the corresponding node is both a leaf (no enabled direction) and a successful one (the initial goal is satisfied and the current goal is empty).

4. the rule \( BT \) for \( \beta \)-termination

In this section we present the proof-rule \( BT \) for proving \( \beta \)-termination of a Prolog goal \((g_o)\) using well-founded sets. (The program \( P \) is implicit). For technical reasons we find it more convenient to use a parametrized invariant over the state \( \sigma \) and current goal \( cg \), denoted by \( pi(a, cg, w) \), whose additional parameter \( w \) ranges over a well-founded set \( W \), instead of using variant functions. We use \( 0 \) as a generic name for minimal elements in \( W \), we use the notation \([p] \cup g [q]\) to assert that if \( p \) holds and the current goal is updated by \( g \) after unification, then immediately afterwards \( g \) holds. The rule is presented in figure 8.

Explanations:

1). The clause INIT takes care that by invoking the initial goal the invariant \( pi \) is established by some \( w \in W \), given the global precondition \( pre \).
To prove \( \mathcal{P} \) (pre) \( (g_0) \) [post] (where \( g_0 = \text{pr}_0(s_0) \)):

Find:

1) a well-founded, partially-ordered, set \((\mathcal{W}, <)\)
2) a parametrized invariant \( pi(s, cg, w) \)
3) for each \( w > 0 \), a decreasing direction \( i_w \).

all satisfying:

(INIT)
\[ \text{pre}(s_0) \land cg = g_0 \rightarrow \exists w : pi(s_0, g_0, w) \]

(CONT)
\[ pi(s, cg, w) \land w > 0 \rightarrow E(i_w) \]

(TERM)
\[ pi(s, cg, 0) \rightarrow \forall j \in D : E(j) \land g_0 \land cg = () \land \text{post}(g) \]

(DEC)
\[ [pi(s, cg, w) \land w > 0] \land [\exists w' : w' < w \land pi(s', cg', w')] \]

(FIN)
\[ \vdash \text{NDT} [pi(s, cg, w) \land w > 0] \land [\exists w' : w' < w \land pi(s', cg', w')] \]

Figure 8: the rule \( \mathcal{B} \)

2) The clause CONT has a double role: it ensures that, as long as \( w > 0 \) holds, there is an enabled direction to take; furthermore, this direction is the decreasing direction (for that \( w \)), preventing vacuous choices of decreasing directions.

3) The clause TERM ensures the boundary condition: successful termination is reached once \( w = 0 \) holds. Success is expressed by implying the initial goal.

4) The clause DEC is the central clause, taking care that following a decreasing direction \( i_w \) indeed causes decrease in \( pi \) when the corresponding current goal \( G(i_w) \) is taken. Note that \( s' \) denotes the current state after the execution of the goal \( G(i_w') \).

5) The clause FIN takes care that for every enabled direction \( j \) smaller than \( i_w \), the rule \( \text{NDT} \) proves non-deterministic termination with a failing state.

As an application of the rules suggested above, we present a proof of

\[ \vdash_\mathcal{B} [C > 0] r(C) [\text{true}] \]

where the program \( P \) appeared in figure 1.

As the well-founded set we choose \((N, <)\), the natural numbers with their usual ordering. The parametrized invariant is shown in figure 9.

Clearly, \( 0 \leq w \leq 2C + 3 \) is implied by \( pi \). Hence, we choose decreasing directions \( i_w \) for this range only. Figure 9 presents these values, as well as the sets \( F_w \).
\( p_i(x, y, u, v, a, b, cg, w) = \)
\( C > 0 \land \)
\( w = 2C + 1 \rightarrow [ cg = r(C) ] \land \)
\( w = 2C \rightarrow [ cg = p(x - 1, y) \cdot q(x - 1, y) \land x = C \land d(y) ] \land \)
\( w = C + u \rightarrow [ 0 < u < x \land u = y \land x = p(u - 1, v) \cdot q(x - 1, y) ] \land \)
\( -d(v) \land -d(y) \land (p(u - 1, v) \rightarrow p(x - 1, y)] \land \)
\( w = C \rightarrow [ p(x - 1, 0) \land cg = q(x - 1, y) \land x = C \land y = 0 ] \land \)
\( w = a \rightarrow [ 0 < a < x \land b = y = 0 \land p(x - 1, 0) \land x = C \land cg = (q(a - 1, 0)) \land \)
\( (q(a - 1, 0) \rightarrow q(x - 1, 0)] \land \)
\( w = 0 \rightarrow [ cg = () \land p(x - 1, 0) \land q(x - 1, 0) \land x = C ] \)

**Figure 9:** The parametrized invariant

for the NDT rule, as determined from the invariant chosen.

We now present some of the details involved in showing that the clauses of BT apply.

**INIT**

Clearly, \( C > 0 \) and \( cg = r(C) \) imply that \( p_i \) is satisfied by \( w = 2C + 1 \).

**CONT**

For \( w = 2C + 1 \) we have \( i_w = 1 \) and \( cg = (r(C)) \), and indeed \( U(r(C), r(x)) \) holds.

For \( w = 2C \) we have \( i_w = 4 \) and \( cg = p(x - 1, y) \cdot q(x - 1, y) \).

For \( C + 1 < w < 2C \) we have again \( i_w = 4 \) and \( cg = (p(u - 1, v) \cdot q(x - 1, y)) \). In both cases the current goal unifies with \( p(u, v) \).

\[
\begin{align*}
  w = 2C + 1 & \quad i_w = 1 & \phi \\
  C + 1 < w \leq 2C & \quad i_w = 4 & \text{if } w > C + 2 \text{ then } \phi \text{ else } \{3\} \\
  w = C + 1 & \quad i_w = 2 & \phi \\
  1 < w \leq C & \quad i_w = 7 & \phi \\
  w = 1 & \quad i_w = 6 & \phi
\end{align*}
\]

**Figure 10:** the decreasing directions and NDT sets
The rest of the cases are checked similarly.

**TERM**

\[ \pi(a, cg, 0) \] implies \( cg = () \) and hence termination. It also implies \( x = C \land p(x - 1, 0) \land q(x - 1, 0) \), which, by clause 1, implies the initial goal \( r(C) \) and, hence, success. The post condition \( true \) is trivially satisfied.

**DEC**

This is the interesting part of the proof, establishing the correctness of the "guesses" of the decreasing directions. We again do a case analysis according to the value of \( w \).

We first consider

\[ [\pi(a, cg, 2C + 1)] U: (p(x - 1, y), q(x - 1, y)) [\pi(a', cg', w')] \]

After unification, we have that \( cg' = (p(C - 1, y) \land q(C - 1, y) \land \text{true} \) and the invariant is reestablished with \( w' = 2C < w \).

The next case is

\[ [\pi(a, cg, w) \land w = 2C] U: p(u - 1, v) [\pi(a', cg', w')] \]

As the invariant initially holds, we obtain \( cg' = p(x - 1, y) \land q(x - 1, y) \). By the unification we have

\[ u = x - 1 \land v = y \land d(u) \land eg' = p(u - 1, v) \land q(x - 1, y) \].

By clause 4 (of the program) we also have that \( p(u - 1, v) \rightarrow p(x - 1, y) \). Hence, the invariant is reestablished with \( w' = 2C - 1 < w \).

Next, consider the case

\[ [\pi(a, cg, w) \land C + 1 < w < 2C] U: p(u - 1, v) [\pi(a', cg', w')] \]

As the invariant initially holds, we obtain

\[ w = C + u, 0 < u < x, cg = (p(u - 1, v) \land q(x - 1, y)), p(u - 1, v) \rightarrow p(x - 1, y) \] .

After the unification we have

\[ u' = u - 1, v' = v, cg' = (p(u' - 1, v') \land q(x - 1, y)) \].

Using clause 4 in the program with the above implication, we get

\[ p(u' - 1, v) \rightarrow p(C - 1, y) \]. Hence, the invariant is reestablished with

\[ w' = C + u' = C + u - 1 < w \].

Note that this holds also for \( w = C + 2 \), i.e., for the case \( cg = p(1, v) \) i.e. \( u - 1 = 1 \). This is the "real" guess of a successful continuation, leaving direction 3, which is also enabled, to \( NDT \) in clause \( FIN \).

As the last case discussed in detail, consider

\[ [\pi(a, cg, C + 1)] U: () [\pi(a', cg', w')] \]

The same precondition as before is obtained. After unification, we get

\[ u' = v' = 0, cg' = (q(C - 1, 0)) \]

(here an empty r.h.s of this direction is appended to the tail of the previous \( cg \)). By Modus Ponens we get \( p(x - 1, 0) \) and the invariant is reestablished by
\[ w' = C < w \]

We skip the details of the rest of the cases.

Finally,

FIN

As most of the sets \( F_w \) are empty, the only fact to be proved using NDT amounts to proving termination of \( q(C - 1, b) \), a simple exercise in termination of recursive procedures (as no backtracking is involved any more). We again skip the details.

This concludes the proof of the example using the rule \( BT \). We return to this example in the next section, presenting a more modular proof.

We now state the two basic theorems establishing the soundness and semantic completeness of the rule \( BT \). Their proofs rely directly on the observation made before about the form of the tree \( T_{P, g_0} \) for a \( B \)-terminating program.

**Theorem: (soundness of \( BT \))**

\[
\text{if } \models_{\text{BT}} \text{ pre } \to \text{ post } \Rightarrow \models \text{ pre } \to \text{ post }.
\]

**Proof:** If all clauses of \( BT \) are successfully applicable then clearly the tree must have the form shown in figure 6, i.e. the program indeed \( B \)-terminates.

**Theorem: (semantic completeness of \( BT \))**

\[
\text{If } \models \text{ pre } \to \text{ post } \Rightarrow \models_{\text{BT}} \text{ pre } \to \text{ post }.
\]

**Proof:** If the program fairly terminates for an initial state satisfying \( \text{ pre } \), the tree has the form shown in figure 6. We may choose \( W = \{1, \ldots, n\} \) naturally ordered, where \( n = \text{length}(\pi) \). For the invariant we define \( \pi_i(\sigma, cg, w) \) to hold if an occurrence of \( (\sigma, cg) \) occurs on \( \pi \) with distance \( w \). Also, \( \pi_i(\sigma, cg, n) \) holds for every \( (\sigma, cg) \) occurring on the tree not on \( \pi \).

From the fact that the tree is shaped as in figure 6 it clearly follows that all clauses of \( BT \) apply.

5. Compositional proofs of \( B \)-termination

In this section a compositional proof system for \( B \)-termination is presented, in which each procedure is treated separately, making use of assumptions about inner calls. This approach is similar to the way mutually recursive procedures are handled in the backtracking-free case.

The main difficulty of adding backtracking to this framework is the absence of context information in the isolated treatment of a single procedure. As a consequence, it is impossible to attribute success/failure to the various outcomes of a procedure, as a value that succeeds w.r.t that procedure may be rejected by a subsequent one. Thus, it is impossible to "judge" a procedure call by its first outcome.
This leads naturally to a different view of the semantics of a single procedure (outside of any context of its call). Instead of regarding a procedure as a (single) state-transfer, it is now regarded as a stream-transformer, in a sense similar to that of data flow programs. The idea is to characterize the procedure by the stream of values it produces (under the leftmost depth-first mode of execution) leaving the classification with respect to success to the receiver of the stream.

In the example program discussed above, the procedure \( p \), for a given call with a positive first argument, produces the stream \(<1,0>\), looping forever after producing its second output value. The procedure \( q \), for a call with a positive first argument, produces an one-element stream \(<0>\), terminating subsequently with a failure to produce any further results (for the given call). In the procedure \( r \) (called with a positive argument) the output stream of \( p \) is filtered through \( q \), resulting in an output stream for \( r \) consisting of the stream \(<0>\).

In the more general case, output streams may be infinite also, and some finite prefix will be consumed by the context. Consider for example the following procedure \( s(z) \), shown in figure 11. A call \( s(k) \) produces the ascending infinite stream \(<k,k+1,\ldots>\).

**Stream-extended assertion language**

As a result of the above discussion, we have to extend the assertion language to be able to deal with assertions about streams. We use Greek letters to refer to assertions from an underlying assertion language, asserting facts about single elements. Finite concatenations of such formulae are interpreted over streams in the natural way: a prefix of the appropriate length is such that its respective elements satisfy the respective assertions from the concatenation. For example, the assertion \( \varphi \eta \) is satisfied by streams the length of which is at least 2, the first element satisfying \( \varphi \) and the second element satisfying \( \eta \). A frequently occurring form of stream assertions is that of \( \varphi^* \). It is satisfied by any stream having a (finite) prefix, each of which elements satisfies \( \varphi \). The assertion \( \omega \) is the assertion satisfied by the empty stream only. Its main use is in truncating a stream, claiming something about its exact form, without any suffixes. For example, the assertion \( \varphi_1 \varphi_2 \omega \) is satisfied only by streams of length 2. It is not satisfied by a diverging computation that fails to produce any output value in a finite time.

The main assertions made about a procedure \( p \) is the following:

\[ [\varphi \eta] p [\varphi^* \omega] \]

The validity of such an assertion is defined as follows:

*If the procedure \( p \) is presented with an input stream that has a prefix satisfying the precondition, then it will produce an output stream having a prefix satisfying the postcondition.*

1) \( s(z) \).

2) \( s(z) : -s(z+1) \).

*Figure 11: a program producing an infinite output stream*
We deliberately distinguish between the last state assertion within a stream assertion from its predecessors, as it will represent an assertion about the (intentionally) successful element, all its predecessors being failing elements. This intention is formalized by the proof rules suggested for deriving such stream correctness assertions about procedures.

The proof system to follow, for the deduction of assertions of the above form, is called CBT (for Compositional BT).

The first rule CONJ describes how to compose assertions about conjuncts into assertions about conjunctions. It is the analogue of the more usual rules of sequential compositions. This rule asserts the existence of an appropriate intermediate stream between two conjuncts.

The next rule COM describes how to combine assertions about "failing" prefixes and assertions about a "successful" element.

The next two rules S and F describe how to obtain assertions about the "successful" and "failing" elements by inspecting the internals of the procedure. Here again the backtracking is considered and similarly to the previous section, decreasing directions are guessed, this time for each procedure independently. In these two rules, and in their recursive analogs, we use the notation \( \varphi \), for a direction \( t \), to denote the assertion obtained from \( \varphi \) after the side effect of unification and updating the current goal. We first ignore the (mutual) recursion and add it at the end by considering proof from assumptions.

The rule F deals with the failing case and is the equivalent of NDT in the previous section.

The second rule S deals with success. Here the real guessing occurs.

---

**Figure 12: the rule CONJ**

\[
\begin{align*}
[\varphi \cdot \mu] p & [\eta \cdot \phi] \\
[\eta \cdot \phi] q & [\psi \cdot \nu] \\
\hline
[\varphi \cdot \mu] (p, q) & [\psi \cdot \nu]
\end{align*}
\]

---

**Figure 13: the rule COM**

\[
\begin{align*}
[\varphi] p & [\psi \cdot \mu] \\
[\eta] p & [\psi \cdot \phi] \\
\hline
[\varphi \cdot \eta] p & [\psi \cdot \phi]
\end{align*}
\]
\[ \forall i: 1 \leq i \leq n \land E(i); [\varphi(i) G(i)] \psi \alpha \]

Figure 14: the rule \( F \)

\[ [\varphi] p [\psi \alpha] \]

\[ [\varphi] p [\psi \alpha] \]

Figure 15: the rule \( S \)

Finally, to deal with recursion, all the assertions are considered parametrized by a parameter ranging over a well-founded set, as before. The recursion rules \( S\text{-REC} \) and \( F\text{-REC} \) state that assumptions about calls with a smaller parameter are permitted to be used when dealing with the directions in the procedure's body. The first uses as an assumption an assertion of the form of the conclusion of \( S \), while the second does the same for \( F \). For both rules, note that the selection of the decreasing direction \( j_n \) depends on the rank \( n \). It can be made state dependent instead.

\[ \forall \alpha: \exists k, k < j; [\varphi_k] G(k) [\psi \alpha], \quad [\varphi_k] G(k) [\psi \alpha] \quad [\exists n: \alpha(n)] p [\psi \alpha] \]

Figure 16: the rule \( S\text{-REC} \)
F-REC

\[ \exists n: \varphi(n) \] p [ \psi^* ]

Figure 17: the rule F-REC

We now present part of the details in showing

\[ \vdash_{\text{CBT}} [C > 0] r(x) [\text{true}] \]

We want to use the rule S (for the only direction existing in \( r \)). To this end, we have to prove

\[ [x = C > 0 \land d(y)] p(x-1, y).q(x-1, y) [\text{true}] \]

where \( \varphi = C > 0 \) and \( \varphi^1 = x = C > 0 \land d(y) \) (after unification by \( U: G(1) \)). In order to show \( 2 \), we want to use the rule CONJ. Hence, we have to show

\[ [u \geq 0 \land d(y) \land x > 0] p(u, v) \]

and

\[ [(a \geq 0 \land b = 1) \land a > 0 \land b = 0] q(a, b) [\text{true}] \]

The parameter substitution used is \( x - 1 \) for \( u \) and \( a \) and \( y \) for \( v \) and \( b \). Note that \( x \) can be substituted for as it is not subject to change in \( p \) and \( q \). We omit the proof of \( 3 \) and show that of \( 4 \). We want to use the rule COM. Hence, we have to show

\[ [a \geq 0 \land b = 1] q(a, b) [\text{true}] \]

and

\[ [a \geq 0 \land b = 0] q(a, b) [\text{true}] \]

Again the proof of \( 6 \) is omitted and we show \( 5 \). We want to use the rule F-REC and have, therefore, to chose a parametrized invariant. We take

\[ \varphi(n) = n = a \land a \geq 0 \land b = 1 \]

and use the assumption

\[ [m < n \land \varphi(m)] q(a, b) [\text{true}] \]

As direction one of \( q \) is not unifiable, we show the following for direction 2 in \( q \):

\[ [\varphi^2(n)] a > 0, q(a-1, b) [\text{true}] \]

We distinguish between two cases:

a) \( a = 0 \), and therefore \( a > 0 \) fails, yielding \( \text{fl} \).

b) \( a > 0 \), in which case \( 8 \) reduces to

\[ [\varphi^2(n) \land a > 0] q(a-1, b) [\text{true}] \]

which is obtained by the parameter substitution \( a - 1 \) for \( a \) and \( n - 1 \) for \( m \) in the assumption 7.

This finishes the example.
In order to prove the soundness and semantic completeness theorems of the CBT system, a formal definition of the stream semantics is needed. This will be provided in a fuller version of the paper. Several attempts at such a stream oriented semantics exist (e.g. [JM 83]), however none of them exactly fits the language and proof rules considered here.

Given an appropriate stream semantics, the assertions needed for the completeness proof in order to apply the recursion rules are the following (informally expressed):

For $S$–$REC$:

**Precondition:**

\[ \varphi(\sigma, l, n) \overset{\text{def}}{=} \text{The partial } \sigma\text{-computation of length } n \text{ yields a stream of length } l. \]

**Postconditions:**

1) \( \psi(\sigma, \sigma', n) \overset{\text{def}}{=} \sigma' \text{ is the } i'\text{th element in the output stream of the } \sigma\text{-computation for some } i < l. \)

2) \( \psi(\sigma, \sigma', l) \overset{\text{def}}{=} \sigma' \text{ is the } l'\text{th element of the output stream of the } \sigma\text{-computation.} \)

In these assertions $l$ is an auxiliary variable reflecting the context dependency of failure/success of the part of the output stream relevant to a given recursive call.
For F–REC:

**Precondition:**

\[ \varphi(\sigma, n) \overset{\text{def}}{=} \text{The full } \sigma\text{-computation is finite and of length } n. \]

**Postcondition:**

\[ \psi(\sigma, \sigma') \overset{\text{def}}{=} \sigma' \text{ is an element of the output stream of the full } \sigma\text{-computation} \]

Recall that the other postcondition is \[. \]

The full details of the completeness proof will appear in the fuller version of the paper.

§ Conclusions

The paper presented two kinds of proof rules to prove \( B \)-termination, the kind of termination obtained by the combination of backtracking and recursion as displayed by Prolog. The first kind of rule takes into account the context of the whole program and is based on a tree oriented operational semantics. The second kind is more compositional, dealing with separate procedures in a context independent way. It is based on a stream oriented semantics.

We would like to stress that such proof rules may be successfully used for defining other search strategies than the actual one used by Prolog. An interesting question is to define in this way a probabilistic backtracking search strategy.

**Acknowledgement:** The first author was partially supported by the fund for the promotion of research, the Technion.
References:


[NGC] New Generation Computing, Springer; (the existence of a journal devoted to the subject is taken as evidence for interest in that subject).