TLP **12** (1–2): 189–218, 2012. © Cambridge University Press 2011
doi:10.1017/S1471068411000445 First published online 12 September 2011

The language features and architecture of B-Prolog

NENG-FA ZHOU

Department of Computer and Information Science,
Brooklyn College and Graduate Center,
The City University of New York, New York, USA
(*e-mail:* zhou@sci.brooklyn.cuny.edu)

submitted 5 October 2009; revised 1 March 2010; accepted 21 February 2011

Abstract

B-Prolog is a high-performance implementation of the standard Prolog language with several extensions including matching clauses, action rules for event handling, finite-domain constraint solving, arrays and hash tables, declarative loop constructs, and tabling. The B-Prolog system is based on the Tree-Oriented Abstract Machine (TOAM) architecture which differs from the Warren Abstract Machine (WAM) mainly in that (1) arguments are passed old fashionedly through the stack, (2) only one frame is used for each predicate call, and (3) instructions are provided for encoding matching trees. The most recent architecture, called TOAM Jr., departs further from the WAM in that it employs no registers for arguments or temporary variables, and provides variable-size instructions for encoding predicate calls. This paper gives an overview of the language features and a detailed description of the TOAM Jr. architecture, including architectural support for action rules and tabling.

KEYWORDS: Prolog, logic programming system

1 Introduction

Prior to the first release of B-Prolog in 1994, several prototypes had been developed that incorporated results from various experiments. The very first prototype was based on the Warren Abstract Machine (WAM) (Warren 1983) as implemented in SB-Prolog (Debray 1988). In the original WAM, the decision on which clauses to apply to a call is made solely on the basis of the type and sometimes the main functor of the first argument of the call. This may result in unnecessary creation of choice points and repeated execution of common unification operations among clauses in the predicate. The first experiment, inspired by the Rete algorithm used in production rule systems (Forgy 1982), aimed at improving the indexing scheme of the WAM. The results from that experiment included an intermediate language named *matching clauses* and a Prolog machine named TOAM (Tree-Oriented Abstract Machine) which provided instructions for encoding tries called matching trees (Zhou *et al.* 1990). Several other proposals had been made with the same objective (Van Roy *et al.* 1987; Hickey and Mudambi 1989; Kliger and Shapiro 1990), but

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

these proposed schemes had the drawback of possibly generating code of exponential size for certain programs.

The WAM was originally designed for both software and hardware implementations. In the WAM, arguments are passed through argument registers so that hardware registers can be exploited in native compilers and hardware implementations. In an emulator-based implementation, however, passing arguments through registers loses its advantage since registers are normally simulated. The second experiment, which took place during 1991–1994, was to have arguments passed old fashionedly through the stack as in DEC-10 Prolog (Warren 1977). The result from that experiment was NTOAM (Zhou 1994). In this machine, only one frame is used for each predicate call which stores a different set of information depending on the type of the predicate. This architecture was later refined and renamed to ATOAM (Zhou 1996a).

During the past 15 years since its first release, B-Prolog has undergone several major extensions and refinements. The first extension was to introduce a new type of frame, called a *suspension frame* for delayed calls (Zhou 1996b). In WAM-based systems, delayed calls are normally stored as terms on the heap (Carlsson 1987). The advantage of storing delayed calls on the stack rather than on the heap is that contest switching is light. It is unnecessary to allocate a frame when a delayed call wakes up and deallocate it when the delayed call suspends again. This advantage is especially important for programs where calls wake up and suspend frequently, such as constraint propagators (Zhou 2006).

A delay construct like freeze is too weak for implementing constraint solvers. New constructs, first delay clauses (Meier 1993; Zhou 1998) and then action rules (Zhou 2006), were introduced into B-Prolog. While these new constructs give significantly more modeling power, they required only minor changes to the architecture: for action rules, one extra slot was added into a suspension frame for holding events.

The action rule language serves well as a powerful and yet efficient intermediate language for compiling constraints over finite-domain variables. A constraint is compiled into propagators defined in action rules that maintain some sort of consistency for the constraint. The availability of fine-grained domain events facilitates programming AC-4-like propagation algorithms (Zhou *et al.* 2006). As propagators are stored on the stack as suspension frames, allocation of frames is not needed to activate propagators and hence context switching among propagators becomes faster.

Another major extension was tabling. Unlike OLDT (Tamaki and Sato 1986) and SLG (Chen and Warren 1996) which rely on suspension and resumption of subgoals to compute fixed points, the tabling mechanism, called *linear tabling* (Zhou *et al.* 2001, 2008), implemented in B-Prolog relies on iterative computation of topmost looping subgoals to compute fixed points. Linear tabling is simpler, easier to implement, and more space efficient than SLG, but a naive implementation may not be as fast due to the necessity of recomputation. Optimization techniques have been developed to make linear tabling competitive with SLG in time efficiency by significantly reducing the cost of recomputation (Zhou *et al.* 2008). For tabled predicates, a new type of frame was introduced into the architecture. Recently, the

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

tabling system has been modified to support table modes, which facilitate describing dynamic programming problems (Guo and Gupta 2008; Zhou *et al.* 2010). The PRISM system (Sato 2009) has been the main driving force for the design and implementation of the tabling system in B-Prolog.

In 2007, B-Prolog's abstract machine was replaced by a new one named TOAM Jr. (Zhou 2007). This switch improved the speed of B-Prolog by over 60% on the Aquarius benchmarks (Van Roy 1990). The old machine ATOAM, like the WAM, has a very fine-grained instruction set in the sense that roughly each symbol in the source program is mapped to one instruction. This fine granularity is a big obstacle to fast interpretation due to the high dispatching cost commonly seen in abstract machine emulators. The new machine TOAM Jr. uses no temporary registers at all and provides variable-size specialized instructions for encoding predicate calls. In WAM-based systems, similar efforts have also been made to specialize and merge instructions to reduce the cost of interpretation (Santos Costa 1999; Demoen and Nguyen 2000; Nässén *et al.* 2001; Morales *et al.* 2005).

The memory manager of B-Prolog has also been improved recently. B-Prolog employs an incremental copying garbage collector (Zhou 2000) based on the one proposed for the WAM by Older and Rummell (Older and Rummell 1992). Because of the existence of suspension frames on the stack, the garbage collector also reclaims space taken by unreachable stack frames. The memory manager automatically expands the stacks and data areas before they overflow, so applications can run with any initial setting for the spaces as long as the overall demand for memory can be met.

This paper overviews in Section 2 the language features of B-Prolog; gives in Section 3 a detailed description of TOAM Jr., the architecture B-Prolog has evolved into after nearly two decades; and summarizes in Section 4 the changes made to the memory architecture for supporting action rules and tabling. The reader is referred to Zhou (2006) for a detailed description of architectural support for action rules and to Zhou *et al.* (2006) for a detailed description of the extension of the architecture for tabling.

2 An overview of language features of B-Prolog

In addition to the standard Prolog language, B-Prolog offers several useful new features. In this section, we give an overview of them.

2.1 Matching clauses

A matching clause is a form of a clause where the determinacy and input/output unifications are denoted explicitly. The compiler translates matching clauses into matching trees and generates indices for all input arguments. The compilation of matching clauses is much simpler than that of normal Prolog clauses because no complex program analysis or specialization or dynamic indexing (Santos Costa *et al.* 2007) is necessary; also the generated code tends to be more compact and faster.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

The B-Prolog compiler and most of the library predicates are written in matching clauses.

A *determinate* matching clause takes the following form:

H, G => B

where H is an atomic formula, G and B are two sequences of atomic formulas. H is called the head, G the guard, and B the body of the clause. No call in G can bind variables in H and all calls in G must be in-line tests. In other words, the guard must be *flat*. For a call C, matching rather than unification is used to select a matching clause in its predicate. When applying the matching clause to C, the system rewrites C *determinately* into B. In other words, call C fails once call B fails.

A *nondeterminate* matching clause takes the following form:

H, G ?=> B

It differs from the determinate matching clause 'H, G => B' in that the rewriting from H into B is nondeterminate, i.e., the alternative clauses will be tried when B fails. In a predicate definition, determinate and nondeterminate matching clauses can coexist.

The following gives an example predicate in matching clauses that merges two sorted lists:

```
merge([],Ys,Zs) => Zs=Ys.
merge(Xs,[],Zs) => Zs=Xs.
merge([X|Xs],[Y|Ys],Zs),X<Y => Zs=[X|ZsT],merge(Xs,[Y|Ys],ZsT).
merge(Xs,[Y|Ys],Zs) => Zs=[Y|ZsT],merge(Xs,Ys,ZsT).
```

The cons `[Y|Ys]` occurs in both the head and the body of the third clause. To avoid reconstructing the term, we can rewrite the clause into the following:

```
merge([X|Xs],Ys,Zs),Ys=[Y|_],X<Y => Zs=[X|ZsT],merge(Xs,Ys,ZsT).
```

The call `Ys=[Y|_]` in the guard matches `Ys` against the pattern `[Y|_]`.

2.2 Action rules

The lack of a facility for programming "active" subgoals that can be reactive to the environment has been considered one of the weaknesses of logic programming. To overcome this, B-Prolog provides action rules for programming agents (Zhou 2006). An agent is a subgoal that can be delayed and can later be activated by events. Each time an agent is activated, some action may be executed. Agents are a more general notion than delay constructs in early Prolog systems and processes in concurrent logic programming languages in the sense that agents can be responsive to various kinds of events including instantiation, domain, time, and user-defined events.

An action rule takes the following form:

$H, G, \{E\}$ => B

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

where H is a pattern for agents, G is a sequence of conditions on the agents, E is a set of patterns for events that can activate the agents, and B is a sequence of arbitrary Prolog goals (called *actions*) executed by the agents when they are activated. The sequence of actions B can succeed only once and hence can never leave choice points behind. The compiler replaces B with `once(`B`)` if it predicts that B may create choice points. When the event pattern E together with the enclosing braces is missing, an action rule degenerates into a determinate matching clause.

A set of built-in events is provided for programming constraint propagators and interactive graphical user interfaces. For example, `ins(`X`)` is an event that is posted when the variable X is instantiated. A user program can create and post its own events and define agents to handle them. A user-defined event takes the form of `event(`X`,`O`)` where X is a variable, called a *suspension variable*, that connects the event with its handling agents, and O is a Prolog term that contains the information to be transmitted to the agents. The built-in `post(`E`)` posts the event E. In the next subsection, we show the events provided for programming constraint propagators.

Consider the following examples:

```
echo(X),{event(X,Mes)}=>writeln(Mes).
ping(T),{time(T)} => writeln(ping).
```

The agent `echo(X)` echoes whatever message it receives. For example,

```
?-echo(X),post(event(X,hello)),post(event(X,world)).
```

outputs the message `hello` followed by `world`. The agent `ping(T)` responds to time events from the timer `T`. Each time it receives a time event, it prints the message `ping`. For example,

```
?-timer(T,1000),ping(T),repeat,fail.
```

creates a timer that posts a time event every second and creates an agent `ping(T)` to respond to the events. The repeat-fail loop makes the agent perpetual.

The action rule language has been found useful for programming coroutining such as `freeze`, implementing constraint propagators (Zhou 2006), and developing interactive graphical user interfaces (Zhou 2003). Action rules have been used by Schrijvers *et al.* (2006) as an intermediate language for compiling Constraint Handling Rules and by Zhou *et al.* (2011) to compile Answer Set Programs.

2.3 CLP(FD)

Like many Prolog-based finite-domain constraint solvers, B-Prolog's finite-domain solver was heavily influenced by the CHIP system (van Hentenryck 1989). The first full-fledged solver was released with B-Prolog version 2.1 in March 1997. That solver was implemented with delay clauses (Zhou 1998). During the past decade, the action rule language has been extended to support a rich class of domain events (`ins(`X`)`, `bound(`X`)`,`dom(`X`,`E`)`, and `dom_any(`X`,`E`)`) for programming constraint propagators (Zhou *et al.* 2006) and the system has been enriched with new domains (Boolean, trees, and finite sets), global constraints, and specialized fast constraint propagators.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

Recently, the two built-ins `in/2` and `notin/2` have been extended to allow positive and negative table (also called extensional) constraints (Zhou 2009).

The following program solves the SEND + MORE = MONEY puzzle. The call `Vars in 0..9` is a domain constraint, which narrows the domain of each of the variables in `Vars` down to the set of integers from 0 to 9. The call `alldifferent(Vars)` is a global constraint, which ensures that variables in the list `Vars` are pairwise different. The operator `#\=` is for inequality constraints and `#=` is for equality constraints. The call `labeling(Vars)` labels the variables in the given order with values that satisfy all the constraints. Another well-used built-in, named `labeling_ff`, labels the variables using the so called *first-fail principle* (van Hentenryck 1989).

```
sendmory(Vars):-
   Vars=[S,E,N,D,M,O,R,Y],
   Vars in 0..9,
   alldifferent(Vars),
   S #\= 0,
   M #\= 0,
   1000*S+100*E+10*N+D+1000*M+100*O+10*R+E #=
   10000*M+1000*O+100*N+10*E+Y,
   labeling(Vars).
```

All constraints are compiled into propagators defined in action rules. For example, the following predicate defines a propagator for maintaining arc consistency for the constraint `X+Y #= C`.

```
x_in_c_y_ac(X,Y,C),var(X),var(Y),
   {dom(Y,Ey)}
    =>
   Ex is C-Ey,
   domain_set_false(X,Ex).
x_in_c_y_ac(X,Y,C) => true.
```

Whenever an inner element `Ey` is excluded from the domain of `Y`, this propagator is triggered to exclude `Ex`, the counterpart of `Ey`, from the domain of `X`. For the constraint `X+Y #= C`, we need to generate two propagators, namely, `x_in_c_y_ac(X,Y,C)` and `x_in_c_y_ac(Y,X,C)`, to maintain the arc consistency. Note that in addition to these two propagators, we also need to generate propagators for maintaining interval consistency since `dom(Y,Ey)` only captures exclusions of inner elements, not bounds.

The `dom_any(`X,E`)` event, which captures the excluded value E from the domain X, facilitates implementing AC4-like algorithms (Mohr and Henderson 1986). Consider, for example, the channeling constraint `assignment(`Xs, Ys`)`, where Xs is a list of variables $[X_1,\ldots,X_n]$ (called *primal*) and Ys is another list of variables $[Y_1,\ldots,Y_n]$ (called *dual*), and the domain of each X_i and each Y_i $(i = 1,...,n)$ is $1..n$. The constraint is true iff $\forall_{i,j}(X_i = j \leftrightarrow Y_j = i)$ or equivalently $\forall_{i,j}(X_i \neq j \leftrightarrow Y_j \neq i)$. Clearly a straightforward encoding of the channeling constraint requires n^2 Boolean constraints. With the `dom_any` event, however, we can use only $2 \times n$ propagators

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

to implement the channeling constraint. Let `DualVarVector` be a vector created from the list of dual variables. For each primal variable `Xi` (with the index `I`), a propagator defined below is created to handle exclusions of values from the domain of `Xi`.

```
primal_dual(Xi,I,DualVarVector),var(Xi),
   {dom_any(Xi,J)}
   =>
   arg(J,DualVarVector,Yj),
   domain_set_false(Yj,I).
primal_dual(Xi,I,DualVarVector) => true.
```

Each time a value `J` is excluded from the domain of `Xi`, assume `Yj` is the `J`th variable in `DualVarVector`, then `I` must be excluded from the domain of `Yj`. We need to exchange primal and dual variables and create a propagator for each dual variable as well. Therefore, in total $2 \times n$ propagators are needed.

Thanks to the employment of action rules as the implementation language, the constraint solving part of B-Prolog is relatively small (3,800 lines of Prolog code and 4,500 lines of C code, including comments and empty lines) but its performance is very competitive with other CLP(FD) systems (Zhou 2006). Moreover, the action rule language is available to the programmer for implementing problem-specific propagators.

2.4 Arrays and the array subscript notation

A structure can be used as a one-dimensional array[1], and a multidimensional array can be represented as a structure of structures. To facilitate creating arrays, B-Prolog provides a built-in, called `new_array(`X`,`$Dims$`)`, where X must be an uninstantiated variable and *Dims* a list of positive integers that specifies the dimensions of the array. For example, the call `new_array(X,[10,20])` binds `X` to a two dimensional array whose first dimension has 10 elements and second dimension has 20 elements. All the array elements are initialized to be free variables.

The built-in predicate `arg/3` can be used to access array elements, but it requires a temporary variable to store the result, and a chain of calls to access an element of a multidimensional array. To facilitate accessing array elements, B-Prolog supports the array subscript notation X`[`I_1`,...,`I_n`]`, where X is a structure and each I_i is an integer expression. This common notation for accessing arrays is, however, not part of the standard Prolog syntax. To accommodate this notation, the parser is modified to insert a token `^` between a variable token and `[`. So, the notation X`[`I_1`,...,`I_n`]` is just a shorthand for X`^[`I_1`,...,`I_n`]`. This notation is interpreted as an array access when it occurs in an arithmetic expression, an arithmetic constraint, or as an argument of a call to `@=/2` [2].

[1] In B-Prolog, the maximum arity of a structure is 65535.

[2] X `@=` Y is the same as X `=` Y except that when an argument is an array access or a list comprehension (described later), it is evaluated before the unification.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

In any other context, it is treated as the term itself. The array subscript notation can also be used to access elements of lists. For example, the `nth/3` predicate can be defined as follows:

```
nth(I,L,E) :- E @= L[I].
```

Note that, for the array access notation `A[I]`, while it takes constant time to access the `I`th element if `A` is a structure, it takes O(`I`) time when `A` is a list.

2.5 *Loops with* `foreach` *and list comprehension*

Prolog relies on recursion to describe loops. The lack of powerful loop constructs has arguably made Prolog less acceptable to beginners and less productive to experienced programmers because it is often tedious to define small auxiliary recursive predicates for loops. The emergence of constraint programming languages such as CLP(FD) has further revealed this weakness of Prolog as a modeling language. Inspired by ECL^{i}PS^e (Schimpf 2002) and functional languages, B-Prolog provides a construct, called `foreach`, for iterating over collections, and the list comprehension notation for constructing lists.

The `foreach` call has a very simple syntax and semantics. For example,

```
foreach(A in [a,b], I in 1..2, write((A,I))
```

outputs four tuples `(a,1)`, `(a,2)`, `(b,1)`, and `(b,2)`. The base `foreach` call has the form:

`foreach(`E_1 `in` D_1`, ..., `E_n `in` D_n`, `$LocalVars$`,`$Goal$`)`

where E_1 `in` D_1 is called an *iterator* (E_1 is called the *pattern* and D_i the *collection* of the iterator), *Goal* is a callable term, and *LocalVars* (optional) specifies a list of variables in *Goal* that are local to each iteration. The pattern of an iterator is normally a variable but it can be any term; the collection of an iterator is a list of terms and the notation $B_1..Step..B_2$ denotes the list of numbers B_1, B_1+Step, $B_1+2*Step$, ..., $B_1+k*Step$ where $B_1+k*Step$ is the last element that does not cross over B_2. The notation $L..U$ is a shorthand for $L..1..U$. The `foreach` call means that for each permutation of values $E_1 \in D_1$, ..., $E_n \in D_n$, the instance *Goal* is executed after local variables are renamed.

In general, a `foreach` call may also take as an argument a list of accumulators that can be used to accumulate values from each iteration. With accumulators, we can use `foreach` to describe recurrences for computing aggregates. Recurrences have to be read procedurally. For this reason, we adopt the list comprehension notation for constructing lists declaratively. A list comprehension takes the form:

`[`T `:` E_1 `in` D_1`, ..., `E_n `in` D_n`, `$LocalVars$`,`$Goal$`]`

where *LocalVars* (optional) specifies a list of local variables, *Goal* (optional) is a callable term. This construct means that for each permutation of values $E_1 \in D_1$, ..., $E_n \in D_n$, if the instance of *Goal* with renamed local variables is true, then T is added into the list. A list of this form is interpreted as a list comprehension if it occurs as an argument of a call to `'@='/2` or in arithmetic constraints.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

Calls to `foreach` and list comprehensions are translated into tail-recursive predicates. For example, the call `Xs @= [X : (X,_) in Ps]` is translated into

```
dummy(Ps, L, []), Xs @= L
```

where `dummy` is defined with matching clauses as follows:

```
dummy([],Xs,XsR) => Xs=XsR.
dummy([(X,_)|Ps],Xs,XsR) => Xs=[X|Xs1], dummy(Ps,Xs1,XsR).
dummy([_|Ps],Xs,XsR) => dummy(Ps,Xs,XsR).
dummy(Ps,_,_) => throw(illegal_argument_in_foreach(Ps)).
```

As can be seen in this example, there is little or no penalty to using these loop constructs compared with using recursion.

The loop constructs considerably enhance the modeling power of CLP(FD). The following gives two programs for the N-queens problem to illustrate different uses of the loop constructs. Here is the first program:

```
queens(N,Qs):-
    length(Qs,N),
    Qs in 1..N,
    foreach(I in 1..N-1, J in I+1..N,
            (Qs[I] #\= Qs[J],
             abs(Qs[I]-Qs[J]) #\= J-I)).
```

The call `queens(N,Qs)` creates a list `Qs` of `N` variables (one variable for each column), declares the domain of each of the variables to be `1..N`, and generates constraints to ensure that no two queens are placed in the same row or the same diagonal.

The following program models the problem with Boolean constraints.

```
bool_queens(N,Qs):-
    new_array(Qs,[N,N]),
    Vars @= [Qs[I,J] : I in 1..N, J in 1..N],
    Vars in 0..1,
    foreach(I in 1..N,     % one queen in each row
            sum([Qs[I,J] : J in 1..N]) #= 1),
    foreach(J in 1..N,     % one queen in each column
            sum([Qs[I,J] : I in 1..N]) #= 1),
    foreach(K in 1-N..N-1, % at most one queen in each left-down diag
            sum([Qs[I,J] : I in 1..N, J in 1..N, I-J=:=K]) #=< 1),
    foreach(K in 2..2*N,   % at most one queen in each left-up diag
            sum([Qs[I,J] : I in 1..N, J in 1..N, I+J=:=K]) #=< 1).
```

The call `Vars @= [Qs[I,J] : I in 1..N, J in 1..N]` extracts the variables from matrix `Qs` into list `Vars`. List comprehensions are used in aggregate constraints. For example, the constraint `sum([Qs[I,J] : J in 1..N]) #= 1` means that the sum of the `Ith` row of the matrix is equal to 1.

The `foreach` construct of B-Prolog is different from the loop constructs provided by ECL^{i}PS^e (Schimpf 2002). Syntactically, `foreach` in B-Prolog is a variable-length

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

call in which only one type of iterator, namely, *E* `in` *D*, is used for iteration, and an extra argument is used for accumulators if needed. In contrast, ECL^{i}PS^e provides a built-in, called `do/2`, and a base iterator, named `fromto/4`, from which six types of iterators are derived for describing various kinds of iteration and accumulation. In addition, in B-Prolog variables in a loop are assumed to be global unless they are declared local or occur in the patterns of the iterators (*global-by-default*). In contrast, in ECL^{i}PS^e variables are assumed to be local unless they are declared global (*local-by-default*). From the programmer's perspective, the necessity of declaring variables is a burden in both approaches and no approach is uniformly better than the other. Nevertheless, small loops tend to have fewer local variables than global ones, and for them global-by-default tends to impose less a burden than local-by-default. For example, while the two N-queens programs shown above contain no declaration of local variables, in ECL^{i}PS^e the variables `N` and `Qs` would have to be declared global. Large loop bodies, however, may require declaration of more local variables than global ones, but my personal opinion is that large loop bodies should be put in separate predicates for better readability. From the implementation perspective, ECL^{i}PS^e's local-by-default can be easily implemented by goal expansion while B-Prolog's global-by-default requires analysis of variable scopes. B-Prolog issues warnings for occurrences in loop goals of singleton variables including anonymous variables.

Semantically, B-Prolog's iterators are matching based while ECL^{i}PS^e's iterators are unification based. In B-Prolog, iterators never change collections unless the goal of the loop changes them explicitly. In contrast, in ECL^{i}PS^e variables in collections can be changed during iterations even if the goal does not touch on the variables. This implicit change of variables in collections may make loops less readable.

2.6 Tabling

Tabling has been found increasingly important not only for helping beginners write workable declarative programs but also for developing real-world applications such as natural language processing, model checking, and machine learning applications. B-Prolog implements a tabling mechanism, called linear tabling (Zhou *et al.* 2008), which is based on iterative computation of looping subgoals rather than suspension of them to compute the fixed points. The PRISM system (Sato and Kameya 2001), which heavily relies on tabling, has been the main driving force for the design and implementation of B-Prolog's tabling system.

The idea of tabling is to memorize the answers to tabled calls and use the answers to resolve subsequent variant calls. In B-Prolog, tabled predicates are declared explicitly by declarations in the following form:

```
:-table P1/N1,...,Pk/Nk.
```

For example, the following tabled predicate defines the transitive closure of a relation as given by `edge/2`.

```
:-table path/2.
```

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

```
path(X,Y):-edge(X,Y).
path(X,Y):-path(X,Z),edge(Z,Y).
```

With tabling, any query to the program is guaranteed to terminate as long as the term sizes are bounded.

By default, all the arguments of a tabled call are used in variant checking and all answers are tabled for a tabled predicate. B-Prolog supports table modes, which allow the system to use only input arguments in variant checking and table answers selectively. The table mode declaration

```
:-table p(M1,...,Mn):C.
```

directs the system on how to do tabling on `p/n`, where `C`, called a *cardinality limit*, is an integer which limits the number of answers to be tabled, and each `Mi` is a mode which can be `min`, `max`, `+` (input), or `-` (output). An argument with the mode `min` or `max`, called *optimized*, is assumed to be output. If the cardinality limit `C` is 1, it can be omitted with the preceding ':'. In the current implementation, only one argument can be optimized. Since an optimized argument is not required to be numeral and the built-in `@</2` is used to select answers with minimum or maximum values, multiple values can be optimized.

The system uses only input arguments in variant checking, disregarding all output arguments. After an answer is produced, the system tables it unconditionally if the cardinality limit is not yet reached. When the cardinality limit has been reached, however, the system tables the answer only if it is better than some existing answer in terms of the argument with the `min` or `max` mode. In this case, the new answer replaces the worst answer in the table.

Mode-directed tabling in B-Prolog was motivated by the need to scale up the PRISM system(Sato and Kameya 2001; Sato 2009; Zhou *et al.* 2010) for handling large data sets. For a given set of possibly incomplete observed data, PRISM collects all explanations for these data using tabling and estimates the probability distributions by conducting EM learning (Dempster *et al.* 1977) on these explanations. For many real-world applications, the set of explanations may be too large to be completely collected even in compressed form. Mode-directed tabling allows for collecting a subset of explanations.

Mode-directed tabling is in general very useful for declarative description of dynamic programming problems (Guo and Gupta 2008). For example, the following program encodes Dijkstra's algorithm for finding a path with the minimum weight between a pair of nodes.

```
:-table sp(+,+,-,min).
sp(X,Y,[(X,Y)],W) :-
    edge(X,Y,W).
sp(X,Y,[(X,Z)|Path],W) :-
    edge(X,Z,W1),
    sp(Z,Y,Path,W2),
    W is W1+W2.
```

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

The table mode states that only one path with the minimum weight is tabled for each pair of nodes.

2.7 Other extensions and features

The JIPL interface with Java: This interface was designed and implemented by Nobukuni Kino, originally for his K-Prolog system, and was ported to B-Prolog. This bi-directional interface makes it possible for Java applications to use Prolog features such as search and constraint solving, and for Prolog applications to use Java resources such as networking, GUI, database, and concurrent programming packages.

PRISM (Sato 2009): This is an extension of Prolog that integrates logic programming, probabilistic reasoning, and EM learning. It allows for the description of independent probabilistic choices and their logical consequences in general logic programs. PRISM supports parameter learning. For a given set of possibly incomplete observed data, PRISM can estimate the probability distributions to best explain the data. This power is suitable for applications such as learning parameters of stochastic grammars, training stochastic models for gene sequence analysis, game record analysis, user modeling, and obtaining probabilistic information for tuning systems performance. PRISM offers incomparable flexibility compared with specific statistical model such as Hidden Markov Models (HMMs), Probabilistic Context Free Grammars (PCFGs), and discrete Bayesian networks. PRISM is a product of the PRISM team at Tokyo Institute of Technology led by Taisuke Sato.

CGLIB (Zhou 2003): This is a constraint-based high-level graphics library developed for B-Prolog. It supports over 20 types of basic graphical objects and provides a set of constraints including nonoverlap, grid, table, and tree constraints that facilitates the specification of layouts of objects. The constraint solver of B-Prolog serves as a general-purpose and efficient layout manager, which is significantly more flexible than the special-purpose layout managers used in Java. The library uses action rules available in B-Prolog for creating agents and programming interactions among agents or between agents and users. CGLIB is supported in the Windows version only.

Logtalk (Moura 2009): This is an extension of Prolog developed by Paulo Moura that supports object-oriented programming. It runs with several Prolog systems. Thanks to Paulo Moura's effort, Logtalk has been made to run with B-Prolog seamlessly. Logtalk can be used as a module system on top of B-Prolog.

The LP/MIP interface: B-Prolog provides an interface to LP/MIP (linear programming and mixed integer programming) packages such as GLPK and CPLEX. With the declarative loop constructs, B-Prolog can serve as a powerful modeling language for LP/MIP problems.

3 The TOAM Jr. Prolog machine

We assume that all the clauses of a given Prolog program have been translated into matching clauses where input and output unifications are separated and the

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

determinacy is denoted explicitly. This form of matching clauses is called `canonical form`. Consider, for example, the `append` predicate in Prolog:

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

This program is translated equivalently into the following matching clauses with no assumption on modes of arguments:

```
append(Xs,Ys,Zs),var(Xs) =>
   append_aux(Xs,Ys,Zs).
append([],Ys,Zs) =>
   Ys=Zs.
append([X|Xs],Ys,Zs) =>
   Zs=[X|Zs1],
   append(Xs,Ys,Zs1).
```

```
append_aux(Xs,Ys,Zs) ?=>
   Xs=[],
   Ys=Zs.
append_aux(Xs,Ys,Zs) =>
   Xs=[X|Xs1],
   Zs=[X|Zs1],
   append(Xs1,Ys,Zs1).
```

The B-Prolog compiler does not infer modes but makes use of modes supplied by the programmer to generate more compact canonical-form programs. For example, with the mode declaration

```
:-mode append(+,+,-).
```

The append predicate is translated into the following canonical form:

```
append([],Ys,Zs) =>
   Ys=Zs.
append([X|Xs],Ys,Zs) =>
   Zs=[X|Zs1],
   append(Xs,Ys,Zs1).
```

The compiler does not check modes at compile time or generate code for verifying modes at run time.

3.1 The memory architecture

Except for changes made to accommodate event handling and garbage collection as to be detailed below, the memory architecture is the same as the ATOAM (Zhou 1996a) employed in early versions of B-Prolog.

3.1.1 Code and data areas

TOAM Jr. uses all the stacks and data areas used by the WAM (see Figure 1). There is a data area called *code area* that contains, besides instructions compiled from programs, a symbol table that stores information about the atom, function, and predicate symbols in the programs. There is one record for each symbol in the table which stores such information as the *name*, *arity*, *type*, and *entry point* if the symbol is defined.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

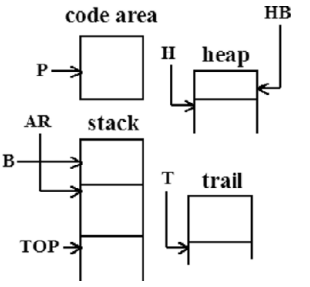


Fig. 1. The memory architecture of TOAM Jr.

The *stack* stores frames associated with predicate calls. Predicate call arguments are passed through the stack and only one frame is used for each predicate call. Each time a predicate is called, a frame is placed on top of the stack unless the frame currently at the top can be reused. Frames for different types of predicates have different structures. For standard Prolog, a frame is either *determinate* or *nondeterminate*. A nondeterminate frame is also called a *choice point*.

The *heap* stores terms created during execution.

The *trail* stack stores updates that must be undone upon backtracking. The use of a trail stack to support backtracking is the major difference between Prolog machines and machines for other languages such as Pascal, Lisp, and Java.

3.1.2 Term representation

Terms are represented in the same way as in the WAM (Warren 1983). A term is represented by a word containing a value and a tag. The tag distinguishes the type of the term. It may be `REF` denoting a reference, `INT` denoting an integer, `ATM` denoting an atom, `STR` denoting a structure, or `LST` denoting a cons. A float is treated as a special structure in our implementation. Another tag is used for suspension variables including domain variables.

3.1.3 Registers

The following registers are used to represent the machine status (see Figure 1):

`P:`	Current program pointer
`TOP:`	Top of the stack
`AR:`	Current frame pointer
`H:`	Top of the heap
`T:`	Top of the trail stack
`B:`	Latest choice point
`HB:`	`H` slot of the latest choice point, i.e., `B->H`

The `HB` register, which also exists in the WAM, is an alias for `B->H`. It is used in checking whether or not a variable needs to be trailed. When a free variable is

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

bound, if it is a heap variable older than HB or a stack variable older than B, then it is trailed.

3.1.4 Stack frame structures

Frames for different types of predicates have different structures. A determinate frame has the following structure:

A1..An:	Arguments
AR:	Parent frame pointer
CP:	Continuation program pointer
BTM:	Bottom of the frame
TOP:	Top of the frame
Y1..Ym:	Local variables

Where BTM points to the bottom of the frame, i.e., the slot for the first argument A1, and TOP points to the top of the frame, i.e., the slot just next to that for the last local variable Ym. The BTM slot was not in the original ATOAM (Zhou 1996a). This slot was introduced to support garbage collection and event-driven action rules which require a new type of frames called *suspension frames* (Zhou 2006). The AR register points to the AR slot of the current frame. Arguments and local variables are accessed through offsets with respect to the AR slot.

It is the caller's job to place the arguments and fill in the AR and CP slots. The callee fills in the BTM and TOP slots.

A choice point contains, besides the slots in a determinate frame, four slots located between the TOP slot and local variables:

CPF:	Backtracking program pointer
H:	Top of the heap
T:	Top of the trail
B:	Previous choice point

The CPF slot stores the program pointer to continue with when the current branch fails. The slot H points to the top of the heap and T points to the top of the trail stack when the frame was allocated. When a variable is bound, it must be trailed if it is older than B or HB. When execution backtracks to the latest choice point, the bound variables trailed on the trail stack between T and B->T are set back to free, the machine status registers H and T are restored, and the program pointer P is set to be B->CPF.

The original ATOAM presented in (Zhou 1996a) had another type of frame, called *non-flat*, for determinate predicates that have non-flat or deep guards. This frame was abandoned since it is difficult for the compiler to extract non-flat guards to take advantage of this feature.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

Control:
- `allocate_det`(i_1,i_2)
- `allocate_nondet`(i_1,i_2)
- `return`
- `fork`(l)
- `cut`
- `fail`

Branch:
- `jmpn_constant`(y, l_{var}, l_{fail}, a)
- `jmpn_struct`($y, l_{var}, l_{fail}, f/n, y_1, \ldots, y_n$)
- `switch_on_cons`($y, l_{nil}, l_{var}, l_{fail}, y_1, y_2$)
- `hash`($y, i, (val_1, l_1), \ldots, (val_i, l_i), l_{var}, l_{fail}$)

Unify:
- `unify_constant`(y, a)
- `unify_value`(y_1, y_2)
- `unify_struct`($y, f/n, z_1, \ldots, z_n$)
- `unify_list`($y, i, z_1, \ldots, z_i, z_{i+1}$)

Move:
- `move_struct`($y, f/n, z_1, \ldots, z_n$)
- `move_list`($y, i, z_1, \ldots, z_i, z_{i+1}$)

Call:
- `call`($p/n, z_1, \ldots, z_n$)
- `last_call`($i, p/n, z_1, \ldots, z_n$)

Fig. 2. The TOAM Jr. base instruction set.

3.1.5 *Assertions*

The following assertions always hold during execution:

(1) No heap cell can reference a stack slot.
(2) No older stack slot can reference a younger stack slot and no older heap variable can reference a younger heap variable.
(3) No slot in a frame can reference another slot in the same frame.

Assertions 1 and 2 are also enforced by the WAM. The third assertion is needed to make dereferencing the arguments of a last call unnecessary when the current frame is reused. To enforce this assertion, when two terms being unified are stack variables, the unification procedure *globalizes* them by creating a new heap variable and letting both stack variables reference it.

3.2 The base instruction set

Figure 2 gives TOAM Jr.'s base instruction set. An instruction with operands is denoted as a Prolog structure whose functor denotes the name and whose arguments denote the operands; an instruction with no operand is denoted as an atom. An operand is either a frame slot y, an integer literal i, a label l, a constant a, a functor f/n, a predicate symbol p/n, or a *tagged* operand z. If an instruction carries two or more operands of the same type, subscripts are used to differentiate them.

In the examples below, an operand y occurs as `y(I)` where `I` is the offset w.r.t. the `AR` slot of the current frame[3], and a tagged operand z occurs as one of the following:

- $v(i)$ denotes an uninitialized frame slot with offset i. This is for the first occurrence of a variable. A singleton variable, also called a dummy variable, is denoted as $v(0)$.

[3] Arguments have positive offsets and local variables have negative offsets.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

- $u(i)$ denotes an initialized frame slot with offset i. This is for subsequent occurrences of a variable.
- $c(a)$ denotes a constant a.

To distinguish among these three cases, a tagged operand carries a *code tag*. So if an operand is $v(i)$, it occurs in the instruction as the integer i with a tag; if it is $u(i)$, it occurs as the integer i with a different tag; and if it is $c(a)$, it occurs as the WAM representation of a, i.e., an `INT` tagged integer or an `ATM` tagged atom.

3.2.1 Control instructions

The first instruction in the compiled code of a predicate is an allocate instruction which takes two operands: the arity and the size of the frame, not counting the arguments. By the time an allocate instruction is executed, the arguments of the current call should have been placed on top of the stack, and the `AR` and `CP` slots should have been set by the caller. An allocate instruction is responsible for fixing the size of the current frame and saving status registers if necessary. In the actual implementation, an allocate instruction also handles events and interrupt signals if there are any. For the sake of simplicity, these operations are not included in the definition. Nevertheless, it is assumed that any predicate can be interrupted and preempted by event handlers. Therefore, a run-time test is needed in `last_call` to determine if the current frame can be deallocated or reused.

- The `allocate_det` instruction starts the code of a determinate predicate. It sets the `BTM` and `TOP` slots and updates the `TOP` register.
- The `allocate_nondet` instruction starts the code of a nondeterminate predicate. In addition to fixing the size of the frame, it also saves the contents of the status registers into the frame.
- The `return` instruction returns control to the caller, and deallocates the frame if the current frame is the topmost one that is not pointed to by the `B` register. In ATOAM, when the current frame is deallocated, the top of the stack is set to the top of the parent frame or the latest choice point, whichever is younger. With event handling, however, this becomes unsafe because the chain of active frames, forming a spaghetti stack, are not in chronological order (Zhou 2006). For this reason, the top of the stack is set to the bottom of the current frame after it is deallocated.
- The `fork` instruction sets the `CPF` slot of the current frame.
- The `cut` instruction discards the alternative branches of this frame and all the choice points that are younger.
- The `fail` instruction lets execution backtrack to the latest choice point.

Example

The following shows a canonical-form program and its compiled code:

```
%    p ?=> true.
%    p => true.
```

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

```
p/0: allocate_nondet(0,8)
     fork(l1)
     return
 l1: cut
     return
```

Since the predicate is nondeterminate and there is no local variable, the allocated frame contains eight slots reserved for saving the machine status.

3.2.2 Branch instructions

Unification calls in the guards of clauses in a predicate are encoded as *branch* instructions. Each branch instruction takes a label l_{fail} to go to on failure of the test and also a label l_{var} to go to when the tested operand is a variable. The `jmpn_struct` instruction fetches the arguments of the tested structure into designated frame slots when the test is successful. The `switch_on_cons` instruction moves control to the next instruction if the tested operand is a cons and to l_{nil} if it is an empty list. When the tested operand is a cons, the instruction also fetches the head and tail of the cons into the designated frame slots. The `hash` instruction determines the address of the next instruction based on the tested operand and a hash table.

The following shows an example.

```
%    p(F),F=f(A),A=a => true.
p/1: allocate_det(1,4)
     jmpn_struct(y(1),l_fail,l_fail,f/1,y(1))
     jmpn_constant(y(1),l_fail,l_fail,a)
     return
```

The operand `l_fail` is the address of a `fail` instruction. Notice that the argument slot with offset 1 allocated to the variable `F` is reused for `A`. None of the branch instructions carries tagged operands.

3.2.3 Unify instructions

Recall that in canonical-form Prolog every unification call in the bodies takes the form $V = T$ where V is a variable and T is either a variable, a constant, a list with no compound elements, or a compound term with no compound arguments. A unify instruction encodes a unification call where neither V nor T is a first-occurrence variable in the clause. For each type of T, there is a type of unify instruction. The `unify_constant` instruction is used if T is a constant; `unify_value` is used if T is a variable; `unify_list` is used if T is a list, and `unify_struct` is used if T is a structure. The `unify_list`$(y, i, z_1, \ldots, z_i, z_{i+1})$ encodes the list $[z_1, \ldots, z_i | z_{i+1}]$. The `unify_struct` and `unify_list` instructions have variable lengths and the operands for list elements or structure arguments are all tagged. In a `unify_struct` instruction, the number of tagged operands is determined by the functor f/n; and in a `unify_list` instruction the number is given as a separate operand.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

A unify instruction for $V = T$ unifies the term referenced by V with T if V is not free. In WAM's terminology, the unification is said to be in `read` mode in this case. If V is a free variable, the instruction builds the term T and binds V to the term. This mode is called `write` in the WAM. Since a unification is encoded as only one instruction, there is no need to use a register for the mode.

Special care must be taken to ensure that no heap cell references a stack slot. The `unify_list` and `unify_struct` instructions must dereference a tagged operand if the operand is not a first-occurrence variable and globalize it if the dereferenced term is a stack variable.

The following shows an example.

```
%    p(F) => F=f(L),L=[X,X,a].
p/1: allocate_det(1,4)
     unify_struct(y(1),f/1,v(1))
     unify_list(y(1),3,v(1),u(1),c(a),c([]))
     return
```

The argument slot with offset 1 allocated to the variable `F` is reused for `L` and later also for `X`. Since `L` is a first-occurrence variable, it is encoded as the tagged operand `v(1)`. The variable `X` occurs twice in `L=[X,X,a]`. The first occurrence is encoded as `v(1)` and the second one is encoded as `u(1)`. The tagged operand `c(a)` encodes the constant element `a` and the operand `c([])` encodes the empty tail of the list.

3.2.4 Move instructions

A move instruction is used to encode a unification $V = T$ where V is a first-occurrence variable in the clause. T is assumed to be a compound term. If T is a constant or a variable, the unification can be performed at compile time by substituting all occurrences of V for T in the clause. For this reason, only `move_struct` and `move_list` instructions are needed.

3.2.5 Call instructions

A `call` instruction encodes a non-last call in the body of a clause.

```
call(p/n, z_1, ..., z_n){
    for each z_i (i = 1, ..., n) do
        *TOP-- = value of z_i
    parent_ar = AR;
    AR = TOP;
    AR->AR = parent_ar;
    AR->CP = P;
    P = entrypoint(p/n);
}
```

After passing the arguments to the callee's frame, the instruction also sets the `AR` and `CP` slots of the frame, and lets the `AR` register point to the frame.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

The value of each tagged operand z_i is computed as follows. If it is $v(k)$, then the value is the address of the frame slot with offset k (it is initialized to be a free variable) unless when k is 0, in which case the value is the content of the TOP register. If it is $u(k)$, then the value is the content of the frame slot with offset k. Otherwise, the value is z_i itself, which is a tagged constant.

A last_call instruction encodes the last call in the body of a determinate clause or a clause in a nondeterminate predicate that contains cuts. For a nondeterminate clause in a nondeterminate predicate that does not contain cuts, the last call is encoded as a call instruction followed by a return instruction. Unlike the call instruction which always allocates a new frame for the callee, the last_call instruction reuses the current frame if it is a determinate frame or a choice point frame whose alternatives have been cut off. The last_call instruction takes an integer, called *layout bit vector*, which tells what arguments are misplaced and hence need to be rearranged into proper slots in the callee's frame when the current frame is reused. There is a bit for each argument and the argument needs to be rearranged if its bit is 1 [4].

```
last_call(layout,p/n,z1,...,zn){
   if (AR->TOP==TOP && B!=AR){ /* reuse */
       for each argument zi(i = 1,...,n) do
           if (zi is tagged u and its layout bit is 1)
               copy zi to a temporary frame;
       move AR->AR and AR->CP if necessary;
       arg_ptr = AR->BTM+1;
       for each argument zi(i = 1,...,n) do
           if (zi's layout bit is 1)
               *(arg_ptr-i) = the value zi;
       AR = AR+(AR->BTM)-n;
       P = entrypoint(p/n);
   } else
       call(p/n,z1,...,zn);
}
```

The following steps are taken to reuse the current frame: First, all the misplaced arguments that are tagged u are copied out to a temporary frame. Because of the enforcement of assertion 3, it is unnecessary to fully dereference stack slots, but free variables in the frame must be globalized since otherwise unrelated arguments may be wrongly aliased. Constants and first-occurrence variables in the arguments are not touched in this step. Second, if the arity of the current frame is different from the arity of the last call, the AR and CP slots are moved. Third, all misplaced arguments are moved into the frame for the callee. For u-tagged arguments, the values in the temporary frame are used instead of the old ones because the old

[4] In the actual implementation, an integer with 28 bits is used for a layout vector. If the last call has more than 28 arguments, then the last-call optimization is abandoned.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

values may have been overwritten by other values. Finally, the AR register is set to be AR+(AR->BTM)-n.

For example,

```
% p(X,Y,Z) => S=f(X,Y),q(S),r(Z,Y,X,W).
p/3: allocate_det(3,5)
     move_struct(y(-1),f/2,u(3),u(2))  % S=f(X,Y)
     call(q/1,u(-1))                   % q(S)
     last_call(0b1011,r/4,u(1),u(2),u(3),v(0))
```

The binary literal '0b1101' is the layout bit vector for the last call which indicates that all the arguments except for the second one (Y) are misplaced. The variable W is a singleton variable in the clause and is encoded as v(0).

3.3 Storage allocation

Each variable is allocated a frame slot and is accessed through the offset of the slot. All singleton variables have offset 0. When an operand is tagged *v*, the offset must be tested. If the offset is 0, then it is known to be a singleton variable[5].

Frame slots allocated to variables are reclaimed as early as possible such that they can be reused for other variables. A variable is said to be *inactive* if it is not accessible in either forward or backward execution. The storage allocated to a variable can be reclaimed immediately after the call in which the variable becomes inactive. Because of the existence of nondeterminate predicates, a variable may still be active even after its last occurrence. For example, consider the clause

```
a(U) => b(U,V),c(V,W),d(W).
```

The slot allocated to U can be reused after b(U,V) since the clause is determinate, but if b/2 is nondeterminate the slot allocated to V cannot be reused even after c(V,W).

3.4 Instruction specialization and merging

Instruction specialization and merging are two well-known important techniques used in abstract machine implementations. The omission of registers can make these techniques more effective. In this section, we discuss how some base instructions are specialized and what instructions are merged.

3.4.1 Instruction specialization

The variable length instructions that take tagged operands are targets for specialization. A variable length instruction is more expensive to interpret than a fixed length instruction since the emulator needs to fetch the number of operands and iterate

[5] A variable with offset 0 is never stored in the current frame. Recall that the slot with offset 0 stores the pointer to the parent frame.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

through the operands using a loop statement. A tagged operand is more expensive to interpret than an untagged one because its interpretation involves the following overhead: (1) testing the tag; (2) untagging the operand if it is a variable tagged u or v; and (3) testing if the offset is 0 if the operand is an uninitialized variable tagged v. For an instruction of length up to n, $\Sigma_{i=1}^{n} 3^i$ specialized instructions can be created (recall that there are three different operand types, namely, $v(i)$, $u(i)$, and $c(a)$). Obviously, reckless introduction of specialized instructions will result in explosion of the emulator size and even performance degradation depending on the platform.

A specialized instruction carries the number and the types of its operands in its opcode. An instruction, named `unify_cons`(y, z_1, z_2), is introduced to replace `unify_list` that has two operands. The `unify_cons` instruction is further specialized so no operand is tagged.

Specialized instructions are introduced for `unify_struct` that has up to two arguments so no operand is tagged. Any `unify_struct` instruction that has more than two arguments is translated to a specialized instruction for the first two arguments followed by `unify_arg` instructions. In this way, no operand is tagged.

For the `call` instruction, specialized instructions in the form of `call_`k`_u` ($k = 1, \ldots, 9$) are introduced which carry k initialized variables as operands in addition to the predicate symbol. Specialized instructions are also introduced for often-occurring call patterns such as `u`, `v`, and `uv`.

Specialized versions of the `last_call` instruction are introduced that carry indices of misplaced arguments explicitly as operands. In general, a specialized instruction for a last call takes the form `last_call_k`$(i_1, \ldots, i_k, p/n, z_1, \ldots, z_n)$ where the integers $i_1, \ldots, i_k$ are indices of misplaced arguments that need to be rearranged. The currently implemented abstract machine has three specialized instructions ($k = 0, 1, 2$). Further specialized instructions are used to encode tail-recursive calls.

3.4.2 *Instruction merging*

The dispatching cost is considered one of the biggest sources of overhead in abstract machine emulators. Even with fast dispatching techniques such as threaded code, the overhead cannot be neglected. A widely used technique in abstract machine implementations for reducing the overhead is called instruction merging, which amounts to combining several instructions into one. Although our instructions have large granularity, there are still opportunities for merging instructions.

It is often the case that a `switch_on_cons` or `fork` instruction is followed by a unify instruction and it is also often the case that a unify instruction is followed by a `cut` instruction. So it makes sense to introduce merged instructions for these cases. We also introduce merged unify instructions for combining unify instructions and `return`. In addition, `cut` and `fail` are merged as well as `cut` and `return`.

When merging two instructions, we do not just combine the routines for the original instructions to create the routine for the merged instruction. Sometimes the merged instruction can be interpreted more efficiently. For example, consider the merged instruction `fork_unify_constant`(l, y, a), which combines `fork`(l) and

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

`unify_constant(`y, a`)`. The alternative program pointer `CPF` is set to be l if the unification succeeds. If the unification fails, however, execution can simply jump to l because the machine status has not changed since the creation of the frame. So the merged instruction not only saves the setting of `CPF` but also replaces expensive backtracking with cheap jumping.

The same idea can be applied to merged instructions of unify and cut. Consider the merged instruction `unify_constant_cut(`y, a`)`. If y is a free variable, then `cut` can be performed before y is bound to a. In this way, unnecessary trailing of y can be avoided.

3.5 Discussion

Compiling a high-level language into an abstract or virtual machine has become a popular implementation method, which has traditionally been adopted by compilers for Lisp and Prolog, and recently made popular by implementations of Java and Microsoft .NET. One of the biggest issues in designing an abstract machine concerns whether to have arguments passed through registers or stack frames. Stack-based abstract machines are more common than register-based machines as exemplified by the Java Virtual Machine and Microsoft Intermediate Language.

One of the biggest advantages of passing arguments through stack frames over through registers is that instructions for procedure calls need not take destinations of arguments explicitly as operands. This leads to more compact bytecode and less interpretation overhead as well. For historical reasons, most Prolog systems are based on the WAM, which is a register machine, except for B-Prolog which is based on a stack machine called ATOAM. Even ATOAM retains registers for temporary variables.

For Prolog, a register machine such as the WAM does have its merits even when registers are normally simulated. First, no frame needs to be created for determinate binary programs. Second, registers are represented as global variables in C and the addresses of the variables can be computed at load time rather than run time. Third, in some implementations a register never references a stack slot, and hence when building a compound term on the heap the emulator needs not dereference a component if it is stored in a register. In a highly specialized abstract machine such as the one adopted in Quintus Prolog (according to Nässén *et al.* 2001), the registers an instruction manipulates can be encoded as part of the opcode rather than taken explicitly as operands. In this way, if the emulator is implemented in an assembly language to which hardware registers are directly available, abstract machine registers can be mapped to native registers.

Nevertheless, using registers has more cons than pros for Prolog emulators. First, as mentioned above, instructions for procedure calls have to carry destination registers as operands which results in less compact code. Second, long-lived data stored in registers have to be saved in stack frames and loaded later when they are used. In Prolog, variables shared by multiple chunks[6] or multiple clauses are

[6] A chunk consists of a non-inline call preceded by inline calls. The head of a clause belongs to the chunk of the first non-inline call in the body.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

long-lived. Third, the information in a frame cannot be easily reused by the last call if the clause of the frame contains multiple chunks. Extra efforts are needed to reuse frames for such clauses (Demoen and Nguyen 2008a; Meier 1991). Finally, registers make it more expensive to interpret tagged operands and harder to combine instructions because two more operand types, namely, uninitialized and initialized register variables, have to be considered. An alternative approach to facilitating instruction merging is to store all data in registers, as done in the BinWAM (Tarau and Boyer 1990). Nevertheless, this approach causes overhead on nonbinary clauses because of the necessity to create continuations as first-class terms.

Another design issue of abstract machines concerns the granularity of instructions. The WAM has a fine-grained instruction set in the sense that an instruction roughly encodes a symbol in the source program. The ATOAM follows the WAM as far as granularity of instructions is concerned. There are Prolog machines that provide even more fine-grained instructions such as explicit dereference instructions (Van Roy 1994). A fine-grained instruction set opens up more operations for optimization in a native compiler, but hinders fast interpretation due to a high dispatching cost. Instruction specialization and merging are two widely used techniques in abstract machine emulators for reducing the cost of interpretation (Santos Costa 1999; Demoen and Nguyen 2000; Nässén *et al.* 2001). In terms of granularity of instructions, TOAM Jr. resides between the WAM and a Prolog interpreter (Maier and Warren 1988) where terms are interpreted without being flattened. The use of coarse-grained instructions reduces the code size and the number of executed instructions for programs, leading to a reduced dispatching cost.

Nevertheless, the interpretation of a variable number of tagged operands imposes certain overhead. After terms are flattened and registers are omitted, the number of possible operand types is reduced to three (constants, uninitialized variables, and initialized variables). Because of the existence of only three operand types, the cost of interpreting tagged operands is smaller than the dispatching cost. Also, the specialization of most frequently executed instructions makes interpretation of tagged operands unnecessary.

4 Architectural support for the extensions

This TOAM architecture has been extended to support action rules (Zhou 2006) and tabling (Zhou *et al.* 2008). This section overviews the changes to the memory architecture.

4.1 Architectural support for action rules

When a predicate defined by action rules is invoked by a call, a *suspension frame* is pushed onto the stack, which contains, besides the slots in a determinate frame, the following four slots:

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

STATE:	State of the frame
EVENT:	Activating event
REEP:	Re-entrance program pointer
PREV:	Previous suspension frame

The STATE slot indicates the current state of the frame, which can be *start*, *sleep*, *woken*, or *end*. The suspension frame enters the *start* state immediately after it is created and remains in it until the call is suspended for the first time or it is ended because no action rule is applicable. Normally, a suspension frame transits to the *end* state through the *sleep* and *woken* states, but it can transit to the *end* state directly if its call is never suspended. The EVENT slot stores the most recent event that activated the call. The REEP slot stores the program pointer to continue when the call is activated. The PREV slot stores the pointer to the previous suspension frame.

Consider, for example, the following predicate:

```
x_in_c_y_ac(X,Y,C),var(X),var(Y),
   {dom(Y,Ey)}
    =>
   Ex is C-Ey,
   domain_set_false(X,Ex).
x_in_c_y_ac(X,Y,C) => true.
```

When a call `x_in_c_y_ac(X,Y,C)` is executed, a suspension frame is created for it. The conditions `var(X)` and `var(Y)` are tested. If both are true, the frame transits from *start* to *sleep*. After an event `dom(Y,Ey)` is posted, the call is activated and the state of its frame is changed from *sleep* to *woken*. All woken frames are connected into a chain of active frames by the AR slots. When the woken call `x_in_c_y_ac(X,Y,C)` is executed, the conditions `var(X)` and `var(Y)` are tested, and if they are true the body of the action rule is executed. If the body succeeds, the state of the suspension frame is changed from *woken* to *sleep*. If either `var(X)` or `var(Y)` fails, the action rule is no longer applicable and the alternative commitment rule is tried. The state of the suspension frame is changed from *woken* to *end* before the commitment rule is applied.

Events are checked at the entry and exit points of each predicate. If the queue of events is not empty, those frames that are watching the events are added into the active chain and the current predicate is interrupted. Actually, as the frame of the interrupted predicate is connected in the active chain after all the woken frames, no special action is needed to resume its execution once all the activated calls complete their execution.

At a checkpoint for events, there may be multiple events posted that are all watched by a suspended call. If this is the case, then the frame must be copied and one copy is added into the active chain for each event. For example, if at a checkpoint two events `dom(Y,Ey1)` and `dom(Y,Ey2)` are on the event queue, the call `x_in_c_y_ac(X,Y,C)` needs to be activated twice, one for each event. For this, the

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

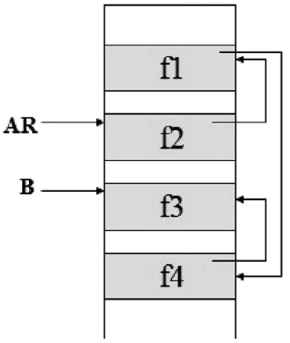


Fig. 3. A nonchronological active frame chain.

frame is copied, and the copy and the original frame are connected to the active chain, each holding one of the events in the `EVENT` slot.

With suspension frames on the stack, the active chain is no longer chronological. Figure 3 illustrates such a situation. The frames `f1` and `f2` are suspension frames, `f3` is the latest choice point frame, and `f4` is a determinate frame. The execution of `f4` was interrupted by an event that woke up `f1` and `f2`. The snapshot depicts the moment immediately after the two woken frames were added into the active chain and `f2` became the current active frame.

Placing delayed calls as suspension frames on the stack makes context switching light. It is unnecessary to allocate a frame when a delayed call wakes up and deallocate it when the delayed call suspends again. Nevertheless, the nonchronologicality of the active chain on the stack requires run-time testing to determine if the current frame can be deallocated or reused. Moreover, unreachable frames on the stack need to be garbage collected (Zhou 2000). A different scheme has been proposed which stores the WAM environments for delayed calls on the heap (Demoen and Nguyen 2008b), but this scheme also complicates memory management.

4.2 Architectural support for tabling

B-Prolog implements a tabling mechanism, called linear tabling (Zhou *et al.* 2008), which relies on iterative evaluation of top-most looping subgoals to compute fixed points. This is in contrast to the SLG mechanism implemented in XSB (Sagonas and Swift 1998), which relies on suspension and resumption of subgoals to compute fixed points.

A new data area, called *table area*, is introduced for memorizing tabled subgoals and their answers. The data structures used for the subgoal table and answer tables are orthogonal to the tabling mechanism. They can be hash tables as implemented in B-Prolog (Zhou *et al.* 2008), tries as implemented in XSB (Ramakrishnan *et al.* 1998), or some other data structures. For each tabled subgoal and its variants, there is an entry in the subgoal table, which stores a pointer to the copy of the subgoal in the table, a pointer to the answer table for the subgoal, a pointer to the strongly connected component (SCC) to which the subgoal belongs, a word that indicates the state of the subgoal (e.g., whether the subgoal is complete, whether the subgoal

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

is a looping one, and whether the answer table has been updated during the current round of evaluation). The answers in the answer table constitute a chain with a dummy answer sitting in the front. In this way, answers can be retrieved one by one through backtracking.

The frame, called a *tabled frame*, for a subgoal of tabled predicate contains the following two slots in addition to those slots stored in a choice point frame:

`SubgoalTable:`	Pointer to the subgoal table entry
`CurrentAnswer:`	Pointer to the current answer that has been consumed

The `SubgoalTable` points to the subgoal table entry, and the `CurrentAnswer` points to the current answer that has been consumed. The next unconsumed answer can be reached from this reference.

When a tabled predicate is invoked by a subgoal, a tabled frame is pushed onto the stack. The subgoal table is looked up to see if a variant of the subgoal exists. If so, the `SubgoalTable` slot is set to point to the entry and `CurrentAnswer` is set to point to the first answer in the answer table (recall that the first answer is a dummy). If the state of the entry is complete, the subgoal only consumes existing answers one by one through backtracking. If the state of the entry is not complete, the subgoal is resolved using clauses if it appears for the first time and using existing answers if it has occurred before in the current round of evaluation. If no variant of the subgoal exists in the subgoal table, then an entry is allocated and the subgoal is resolved using clauses.

After all clauses are tried on a tabled subgoal, a test is performed to see if the subgoal is complete. A subgoal is complete if it has never occurred in a loop, or it is a top-most looping subgoal and none of the subgoals in its SCC has obtained any new answer during the current round of evaluation. The execution of a top-most looping subgoal is iterated until it becomes complete. When a top-most looping subgoal becomes complete, all the subgoals in its SCC become complete as well.

As can be seen, the change to the architecture is minimal for supporting linear tabling. Unlike in the implementations of SLG, no effort is needed to preserve states of tabled subgoals and the garbage collector is kept untouched in linear tabling. Linear tabling is more space efficient than SLG since no stack frames are frozen for tabled subgoals. Nevertheless, linear tabling without optimization could be computationally more expensive than SLG due to the necessity of recomputation (Zhou *et al.* 2008).

5 Final remarks

This paper has surveyed the language features of B-Prolog and given a detailed description of TOAM Jr. with architectural support for action rules and tabling. B-Prolog has strengths and weaknesses. The competitive Prolog engine, the cutting-edge CLP(FD) system, and the efficient tabling system are clear advantages of B-Prolog. With them, B-Prolog serves well the core application domains such as constraint solving and dynamic programming. We will further strengthen B-Prolog

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

as a tool for these applications. Future work includes parallelizing action rules for better performance in constraint solving and improving the tabling system to enhance the scalability of B-Prolog for large-scale machine-learning applications.

The shortcomings of B-Prolog are also obvious. The lack of certain functionalities such as a module system, native interfaces with database and networking libraries, and support of unicode increasingly hinders the adoption of B-Prolog in many other application domains. Additions of these new features are also part of the future work.

Acknowledgements

Very early experiments were conducted while the author was a PhD student at Kyushu University during 1988–1991. The first working system and the versions up to 4.0 were built while the author was with Kyushu Institute of Technology during 1991–1999. Most recent improvements and enhancements have been conducted at Brooklyn College of the City University of New York. The B-Prolog system is indebted to many people in the logic programming community. I wish to express my gratitude to Taisuke Sato and Yoshitaka Kameya for their support, encouragement, and propelling. Their PRISM system has been a strong driving force for recent improvements in the tabling system and memory management. The countless feedbacks from the PRISM team greatly helped enhance the robustness of the system. Special thanks are also due to Bart Demoen for his intensive scrutiny of both the design and the implementation of B-Prolog, Yi-Dong Shen for his cooperation on linear tabling, and Paulo Moura and Ulrich Neumerkel for helping make the core part of B-Prolog more compatible with the ISO standard. Thanks also go to the anonymous referees and the editors, Maria García de la Banda and Bart Demoen, for their detailed comments and guidances on the presentation. B-Prolog-related projects have received numerous grants from various funding organizations, most recently from AIST, CISDD, PSC CUNY, and NSF.

References

Carlsson, M. 1987. Freeze, indexing, and other implementation issues in the WAM. In *Proceedings of the International Conference on Logic Programming (ICLP)*. MIT Press, Cambridge, USA, 40–58.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 1, 20–74.

Debray, S. K. 1988. *The SB-Prolog System, Version 3.0*. SUNY Stony Brook.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *Proceedings of the International Conference on Computational Logic (CL)*. LNAI, vol. 1861. 1240–1254.

Demoen, B. and Nguyen, P.-L. 2008a. Environment reuse in the WAM. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 698–702.

Demoen, B. and Nguyen, P.-L. 2008b. Two WAM implementations of action rules. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 621–635.

Dempster, A. P., Laird, N. M. and Rubin, D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Proceedings of the Royal Statistical Society*, vol. 39, no. 1, 1–38.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19*, 17–37.

Guo, H.-F. and Gupta, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Software Practice and Experience 38*, 1, 75–94.

Hickey, T. J. and Mudambi, S. 1989. Global compilation of Prolog. *Journal of Logic Programming 7*, 3, 193–230.

Kliger, S. and Shapiro, E. Y. 1990. From decision trees to decision graphs. In *Proceedings of the North American Conference on Logic Programming (NACLP)*. 97–116.

Maier, D. and Warren, D. S. 1988. *Computing with Logic: Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company, San Francisco, California, USA.

Meier, M. 1991. Recursion versus iteration in Prolog. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 157–169.

Meier, M. 1993. Better late than never. In *ICLP-Workshop on Implementation of Logic Programming Systems*. 151–165.

Mohr, R. and Henderson, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence 28*, 225–233.

Morales, J. F., Carro, M., Puebla, G. and Hermenegildo, M. V. 2005. A generator of efficient abstract machine implementations and its application to emulator minimization. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 21–36.

Moura, P. 2009. From plain Prolog to Logtalk objects: Effective code encapsulation and reuse. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 23.

Nässén, H., Carlsson, M. and Sagonas, K. F. 2001. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP)*. 49–60.

Older, W. J. and Rummell, J. A. 1992. An incremental garbage collector for WAM-based Prolog. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*. 369–383.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T. and Warren, D. 1998. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38*, 31–54.

Sagonas, K. and Swift, T. 1998. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems 20*, 3, 586–634.

Santos Costa, V. 1999. Optimizing bytecode emulation for Prolog. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP)*. LNCS, vol. 1702. 261–277.

Santos Costa, V., Sagonas, K. F. and Lopes, R. 2007. Demand-driven indexing of Prolog clauses. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 395–409.

Sato, T. 2009. Generative modeling by PRISM. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 24–35.

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15*, 391–454.

Schimpf, J. 2002. Logical loops. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 224–238.

Schrijvers, T., Zhou, N.-F. and Demoen, B. 2006. Translating constraint handling rules into action rules. In *Proceedings of the Third Workshop on Constraint Handling Rules*. 141–155.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 84–98.

Tarau, P. and Boyer, M. 1990. Elementary logic programs. In *Proceedings of Programming Language Implementation and Logic Programming*. 159–173.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press

van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press.

Van Roy, P. 1990. Can logic programming execute as fast as imperative programming? PhD thesis UCB/CSD-90-600, EECS Department, University of California, Berkeley, CA.

Van Roy, P. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19*, 20, 385–441.

Van Roy, P., Demoen, B. and Willems, Y. D. 1987. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *TAPSOFT,* vol. 2. 111–125.

Warren, D. H. D. 1977. Implementing Prolog-compiling predicate logic programs. Research report, 39–40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, UK.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Technical note 309, SRI International, Menlo Park, California.

Zhou, N.-F. 1994. On the scheme of passing arguments in stack frames for Prolog. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 159–174.

Zhou, N.-F. 1996a. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems 18*, 6, 752–779.

Zhou, N.-F. 1996b. A novel implementation method of delay. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*. 97–111.

Zhou, N.-F. 1998. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*. 70–84.

Zhou, N.-F. 2000. Garbage collection in B-Prolog. In *First Workshop on Memory Management in Logic Programming Implementations*. 16–25.

Zhou, N.-F. 2003. CGLIB—a constraint-based graphics library. *Software Practice and Experience 33*, 13, 1199–1216.

Zhou, N.-F. 2006. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP) 6*, 5, 483–508.

Zhou, N.-F. 2007. A register-free abstract Prolog machine with jumbo instructions. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 455–457.

Zhou, N.-F. 2009. Encoding table constraints in CLP(FD) based on pair-wise AC. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 402–416.

Zhou, N.-F., Kameya, Y., and Sato, T. 2010. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*. 213–218.

Zhou, N.-F., Sato, T. and Shen, Y.-D. 2008. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming (TPLP) 8,* 1, 81–109.

Zhou, N.-F., Shen, Y.-D. and You, J. 2011. Compiling answer set programs into event-driven action rules. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 376–381.

Zhou, N.-F., Shen, Y.-D., Yuan, L. and You, J. 2001. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming 1*, 1–15.

Zhou, N.-F., Takagi, T. and Ushijima, K. 1990. A matching tree oriented abstract machine for Prolog. In *Proceedings of the International Conference on Logic Programming (ICLP)*. 159–173.

Zhou, N.-F., Wallace, M. and Stuckey, P. J. 2006. The `dom` event and its use in implementing constraint propagators. Technical report TR-2006013, CUNY Computer Science, The City University of New York, New York.

https://doi.org/10.1017/S1471068411000445 Published online by Cambridge University Press