TLP **12** (1–2): 97–126, 2012. © Cambridge University Press 2011
doi:10.1017/S1471068411000433 First published online 12 September 2011

The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines

PAUL TARAU

Department of Computer Science and Engineering,
University of North Texas, Denton, Texas 76203-6886, USA
(*e-mail:* `tarau@cs.unt.edu`)

submitted 10 October 2009; revised 28 February 2010; accepted 2 February 2011

Abstract

We describe the *BinProlog* system's compilation technology, runtime system and its extensions supporting first-class Logic Engines while providing a short history of its development, details of some of its newer re-implementations as well as an overview of the most important architectural choices involved in their design. With focus on its differences with conventional Warren Abstract Machine (WAM) implementations, we explain key details of *BinProlog*'s compilation technique, which replaces the WAM with a simplified *continuation passing* runtime system (the "BinWAM"), based on a mapping of full Prolog to *binary logic programs*. This is followed by a description of a *term compression* technique using a "tag-on-data" representation. Later derivatives, the Java-based *Jinni Prolog* compiler and the recently developed *Lean Prolog* system refine the *BinProlog* architecture with *first-class Logic Engines*, made generic through the use of an *Interactor* interface. An overview of their applications with focus on the ability to express at source level a wide variety of Prolog built-ins and extensions covers these newer developments.

KEYWORDS: Prolog, logic programming system, continuation passing style compilation, implementation of Prolog, first-class logic engines, data-representations for Prolog runtime systems

1 Introduction

At the time when we started work on the BinProlog compiler, around 1991, Warren Abstract Machine (WAM)-based implementations (Warren 1983; Aït-Kaci 1991) had reached already a significant level of maturity. The architectural changes occurring later can be seen mostly as extensions for constraint programming and runtime or compile-time optimizations.

BinProlog's design philosophy has been minimalistic from the very beginning. In the spirit of Occam's razor, while developing an implementation as an iterative process, this meant not just trying to optimize for speed and size, but also to actively look for opportunities to refactor and simplify.

The guiding principle, at each stage, was seeking answers to questions like:

- what can be removed from the WAM without risking significant, program independent, performance losses?
- what can be done to match, within small margins, performance gains resulting from new WAM optimizations (like read/write stream separation and instruction unfolding) while minimizing implementation complexity and code size?
- can one get away with uniform data representations (e.g. no special tags for lists) instead of extensive specialization, without major impact on performance?
- when designing new built-ins and extensions, can we use source-level transformations rather than changes to the emulator?

The first result of this design, BinProlog's *BinWAM* abstract machine has been originally implemented as a **C** emulator based on a program transformation introduced in Tarau and Boyer (1990).

While describing it, we assume familiarity with the WAM and focus on the differences between the two abstract machines. We refer to Aït-Kach (1991) for a tutorial description of the WAM, including its instruction set, runtime areas and compilation of unification and control structures.

The BinWAM replaces the WAM with a simplified continuation passing logic engine (Tarau 1991) based on a mapping of full Prolog to binary logic programs (binarization). Its key assumption is that as conventional WAM's environments are discarded in favor of a heap-only runtime system, heap garbage collection and efficient term representation become instrumental as means to ensure ability to run large classes of Prolog programs.

The second architectural novelty, present to some extent in the original BinProlog and a key element of its newer Java-based derivatives Jinni Prolog (Tarau 1999a; Tarau 2008b) and Lean Prolog (still under development), is the use of Interactors (and first-class Logic Engines, in particular) as a uniform mechanism for the source-level specification (and often actual implementation) of key built-ins and language extensions (Tarau 2000, 2008a; Tarau and Majumdar 2009).

We first explore various aspects of the compilation process and the runtime system. Next we discuss source-level specifications of key built-ins and extensions using first-class Logic Engines.

Sections 2 and 3 provide an overview BinProlog's key source-to-source transformation (*binarization*) and its use in compilation.

Section 4 introduces BinProlog's unusual "tag-on-data" term representation (Section 4.1) and studies its impact on term compression (Section 4.2).

Section 5 discusses optimizations of the runtime system like instruction compression and the implicit handling of read–write modes.

Section 6 introduces Logic Engines seen as implementations of a generic Interactor interface and describes their basic operations.

Section 7 applies Interactors to implement, at source level, some key Prolog built-ins, exceptions (Section 7.5) and higher order constructs (Section 7.6).

Section 8 applies the logic engine API to specify Prolog extensions ranging from dynamic database operations (Section 8.1) and backtracking if-then-else (Section 8.2) to predicates comparing alternative answers in Section 8.3.1 and mechanisms to encapsulate infinite streams in Section 8.3.2.

Section 9 gives a short historical account of BinProlog and its derivatives.

Section 10 discusses related work and Section 11 concludes the paper.

2 The binarization transformation

We start by reviewing the program transformation that allows compilation of logic programs towards a *simplified WAM specialized for the execution of binary programs* (called BinWAM from now on). Binary programs consist of facts and binary clauses that have only one atom in the body (except for some inline "built-in" operations like arithmetics) and therefore they need no "return" after a call. A transformation introduced in Tarau and Boyer (1990) allows the emulation of logic programs with operationally equivalent binary programs.

Before defining the *binarization* transformation, we describe two auxiliary transformations, commonly used by Prolog compilers.

The first transformation converts facts into rules by giving them the atom `true` as body. For example the fact `p` is transformed into the rule `p :- true`.

The second transformation eliminates *metavariables* (i.e. variables representing Prolog goals only known at runtime), by wrapping them in a `call/1` predicate, e.g. a clause like `and(X,Y):-X,Y` is transformed into `and(X,Y) :- call(X),call(Y)`.

The *binarization transformation* (first described in Tarau and Boyer 1990) adds continuations as the last argument of predicates in a way that preserves first argument indexing.

Let P be a definite program and $Cont$ a new variable. Let T and $E = p(T_1, \ldots, T_n)$ be two expressions (i.e. atoms or terms). We denote by $\psi(E, T)$ the expression $p(T_1, \ldots, T_n, T)$. Starting with the clause

`(C)` $\quad A :- B_1, B_2, \ldots, B_n.$

we construct the clause

`(C')` $\quad \psi(A, Cont) :- \psi(B_1, \psi(B_2, \ldots, \psi(B_n, Cont))).$

The set P' of all clauses `C'` obtained from the clauses of P is called the binarization of P.

The following example shows the result of this transformation on the well-known "naive reverse" program:

```
app([],Ys,Ys,Cont):-true(Cont).
app([A|Xs],Ys,[A|Zs],Cont):-app(Xs,Ys,Zs,Cont).

nrev([],[],Cont):-true(Cont).
nrev([X|Xs],Zs,Cont):-nrev(Xs,Ys,app(Ys,[X],Zs,Cont)).
```

Note that `true(Cont)` can be seen as a specialized version of Prolog's `call/1` that executes the goals stacked in the continuation variable `Cont`. Its semantics is expressed by the following clauses

```
true(app(X,Y,Z,Cont)):-app(X,Y,Z,Cont).
true(nrev(X,Y,Cont)):-nrev(X,Y,Cont).
true(true).
```

which together with the code for `nrev` and `app` run the binarized query

```
?- nrev([1,2,3],R,true).
```

in any Prolog, directly, returning `R=[3,2,1]`.

Prolog's inference rule (called LD-resolution) executes goals in the body of a clause left-to-right in a depth first order. LD-resolution describes Prolog's operational semantics more accurately than order-independent SLD-resolution (Lloyd 1987). The binarization transformation preserves a strong operational equivalence with the original program with respect to the LD-resolution rule, which is *reified* in the syntactical structure of the resulting program, where the order of the goals in the body becomes hardwired in the representation (Tarau and De Bosschere 1993b). This means that each resolution step of an LD-derivation on a definite program P can be mapped to an LD-resolution step of the binarized program P'. *More precisely, let* G *be an atomic goal and* $G' = \psi(G, true)$. *Then, the answers computed using LD-resolution obtained by querying* P *with* G *are the same as those obtained by querying* P' *with* G'. Note also that the concepts of SLD- and LD-resolution overlap in the case of binary programs.

3 Binarization-based compilation and runtime system

BinProlog's BinWAM virtual machine specializes the WAM to binary clauses, and therefore, it drops WAM's environments. Alternatively, assuming a two stack WAM implementation, the BinWAM can be seen as an OR-stack-only WAM. Independently, its simplifications of the indexing mechanism and a different "tag-on-data" representation are the most important differences with conventional WAM implementations. The latter also brings opportunities for a more compact heap representation that is discussed in Section 4.

Note also that continuations become explicit in the binary version of the program. We refer to Tarau and Dahl (1994) for a technique to access and manipulate them by modifying BinProlog's binarization preprocessor. This results in the ability to express constructs like a backtracking sensitive variant of catch/throw at source level. We focus in this section only on their uses in BinProlog's compiler and runtime system.

3.1 Metacalls as built-ins

The first step of our compilation process simply wraps metavariables inside a predicate `call/1`, and adds `true/0` as a body for facts, as most Prolog compilers do. The binarization transformation then adds a continuation as last arguments of each predicate and a new predicate `true/1` to deal with unit clauses. During this step, the arity of all predicates increases by `1` so that, for instance `call/1` becomes `call/2`.

Although we can add the special clause `true(true)`, and for each functor `f` occurring in the program, clauses like

```
true(f(...,Cont)):-f(...,Cont).
call(f(...),Cont):-f(...,Cont).
```

as an implementation of `true/1` and `call/2`, in practice it is simpler and more efficient to treat them as built-ins (Tarau 1991).

The built-in corresponding to `true/1` looks up the address of the predicate associated to `f(...,Cont)` and throws an exception if no such predicate is found. The built-in corresponding to `call/2` adds the extra argument `Cont` to `f(...)`, looks up whether a predicate definition is associated to `f(...,Cont)` and throws an exception if no definition is found. In both cases, when predicate definitions are found, the BinWAM fills up the argument registers and proceeds with the execution of the code of those predicates.

Note that the predicate look-ups are implemented efficiently by using hashing on a `<symbol, arity>` pair stored in one machine word. Moreover, they happen relatively infrequently. For the case of `call/2`-induced look-ups, as in ordinary Prolog compilation, they are generated only when metavariables are used. As calls to `true/1` only happen when execution reaches a "fact" in the original program, they also have a relatively little impact on performance, for typical recursion intensive programs.

3.2 Inline compilation of built-ins

Demoen and Mariën pointed out in Demoen and Marïen (1992) that a more implementation oriented view of binary programs can be very useful: a binary program is simply one that does not need an environment in the WAM. This view leads to inline code generation (rather than binarization) for built-ins occurring *immediately after the head*. For instance something like

```
a(X):-X>1,b(X),c(X).
```

is handled as:

```
a(X,Cont) :- inline_code_for(X>1),b(X,c(X,Cont)).
```

rather than

```
a(X,Cont) :- '>'(X,1,b(X,c(X,Cont))).
```

Inline expansion of built-ins contributes significantly to BinProlog's speed and supports the equivalent of WAM's last call optimization for frequently occurring linear recursive predicates containing such built-ins, as unnecessary construction of continuation terms on the heap is avoided for them.

3.3 Handling CUT

Like in the WAM, a special register `cutB` contains the choice point address up to where choice points need to be popped off the stack on backtracking. In clauses like

```
a(X):-X>1,!,b(X),c(X).
```

CUT can be handled inline (by a special instruction PUSH_CUT, generated when the compiler recognizes this case), after the built-in X>1, by trimming the choice point stack right away. On the other hand, in clauses like

```
a(X):-X>1,b(X),!,c(X).
```

a pair of instructions PUT_CUT and GET_CUT is needed. During the BinWAM's term creation, PUT_CUT saves to the heap the register cutB. This value of cutB is used by GET_CUT when the execution of the instruction sequence reaches it, to trim the choice point stack to the appropriate level.

3.4 Term construction in the absence of the AND-stack

The most important simplification in the BinWAM in comparison with the standard WAM is the absence of an AND-stack. Clearly, this is made possible by the fact that each binary clause has (at most) one goal in the body.

In procedural and call-by-value functional languages featuring only deterministic calls, it was a typical implementation choice to avoid repeated structure creation by using environment stacks containing only the variable bindings. The WAM (Warren 1983) follows this model based on the assumption that most logic programs are deterministic.

This is one of the key points where the execution models between the WAM and BinWAM differ. A careful analysis suggests that the choice between

- the standard WAM's late and repeated construction with variables of each goal in the body pushed on the AND stack
- the BinWAM's eager early construction on the heap (once) and reuse (by possibly trailing/untrailing variables)

favors different programming styles, with "AND-intensive", deterministic, possibly tail-recursive programs favoring the WAM while "OR-intensive", nondeterministic programs reusing structures through trailing/untrailing favoring the BinWAM. The following example illustrates the difference between the two execution models. In the clause

```
  p(X) :- q(X,Y), r(f(X,Y)).
```

binarized as

```
  p(X,C) :- q(X,Y,r(f(X,Y),C)).
```

the term f(X,Y) is created on the heap by the WAM as many times as the number of solutions of the predicate q. On the other hand, the BinWAM creates it only once and reuses it by undoing the bindings of variables X and Y (possibly trailed). This means that if q fails, the BinWAM's "speculative" term creation work is wasted. And it also means that if q is nondeterministic and has a large number of solutions, then the WAM's repeated term creation leads to a less efficient execution model.

P $\Rightarrow$	next clause address
H $\Rightarrow$	saved top of the heap
TR $\Rightarrow$	saved top of the trail
$A_{N+1} \Rightarrow$	continuation argument register
$A_N \Rightarrow$	saved argument register N
...	...
$A_1 \Rightarrow$	saved argument register 1

Fig. 1. A frame on BinProlog's OR-stack.

3.5 A minimalistic BinWAM instruction set

A minimalistic BinWAM instruction set (as shown for two simple **C** and `Java` implementations at `http://www.binnetcorp.com/OpenCode/free_prolog.html`) consists of the following subset of the WAM: `GET_STRUCTURE`, `UNIFY_VARIABLE`, `UNIFY_VALUE`, `EXECUTE`, `PROCEED`, `TRY_ME_ELSE`, `RETRY_ME_ELSE`, `TRUST_ME`, as well as the following instructions, which the reader will recognize as mild variations of their counterparts in the "vanilla" WAM instruction set (Aït-Kaci 1991).

- MOVE_REGISTER (simple register-to-register move)
- NONDET (sets up choice-point creation when needed)
- SWITCH (simple first-argument indexing)
- PUSH_CUT, PUT_CUT, GET_CUT (cut handling instructions for binary programs, along the lines of Demoen and Mariën 1992)

Note that specializations for CONSTANTs, LISTs as well as WRITE-mode variants of the GET and UNIFY instructions can be added as obvious optimizations.

3.6 The OR-stack

A simplified *OR-stack* having the layout shown in Figure 1 is used only for (one-level) *choice point creation* in nondeterministic predicates. No link pointers between frames are needed as the length of the frames can be derived from the arity of the predicate.

Given that variables kept on the local stack in conventional WAM are now located on the heap, the heap consumption of the program increases. It has been shown that, in some special cases, partial evaluation at source level can deal with the problem (Demoen 1992; Neumerkel 1992) but as a more practical solution, the impact of heap consumption has been alleviated in BinProlog by the use of an efficient copying garbage collector (Demoen *et al.* 1996).

3.7 A simplified clause selection and indexing mechanism

As the compiler works on a clause-by-clause basis, it is the responsibility of the loader (that is part of the runtime system) to index clauses and link the code. The runtime system uses a global $< key_1, key_2 > \rightarrow value$ hash table seen as an abstract *multipurpose dictionary*. This dictionary provides the following services:

- indexing compiled code, with key_1 as the functor of the predicate and key_2 as the functor of the first argument
- implementing multiple dynamic databases, with key_1 as the name of the database and key_2 the functor of a dynamic predicate
- supporting a user-level storage area (called "blackboard") containing global terms indexed by two keys

A 1-byte mark-field in the table is used to distinguish between *load-time* use when the *kernel* (including built-ins written in Prolog and the compiler itself) is loaded, and *runtime* use (when user programs are compiled and loaded) to protect against modifications to the kernel and for fast cleanup. Sharing of the global multipurpose dictionary, although somewhat slower than the small $key \rightarrow value$ hashing tables injected into the code-space of the standard WAM, keeps the implementation as simple as possible. Also, with data areas of fixed size (as in the original BinProlog implementation), one big dictionary provides overall better use of the available memory by sharing the hashing table for different purposes.

Predicates are classified as *single-clause*, *deterministic* and *nondeterministic*. Only predicates having *all first-argument functors distinct* are detected as deterministic and indexed.

In contrast to the WAM's fairly elaborate indexing mechanism, indexing of deterministic predicates in the BinWAM is done by a unique SWITCH instruction.

If the first argument dereferences to a nonvariable, SWITCH either fails or finds the one-word address of the unique matching clause in the global hash-table, using the *predicate* and the *functor of the first argument* as a two-word key. Note that the basic difference with the WAM is the absence of intensive tag analysis. This is related also to our different low-level data representation that we discuss in Section 4.

A specialized JUMP-IF instruction deals with the frequent case of two clause deterministic predicates. To reduce the interpretation overhead, SWITCH and JUMP_IF are combined with the preceding EXECUTE and the following GET_STRUCTURE or GET_CONSTANT instruction, giving EXEC_SWITCH and EXEC_JUMP_IF. This not only avoids dereferencing the first argument twice, but also reduces unnecessary branching logic that breaks the processor's pipeline.

Note also that simplification of the indexing mechanism, in combination with smaller and unlinked choice points, helps making backtracking sometimes faster in the BinWAM than in conventional WAMs, as in the case of simple (but frequent!) predicates having only a few clauses.

However, as mentioned in Section 3.4, backtracking performance in the BinWAM also benefits from sharing structures occurring in the body of a clause in the OR-subtree it generates, instead of repeated creation as in conventional WAM. This property of binarized programs (see example in Section 3.4) was first pointed out in Demoen and Marïen (1992) as the typical case when binarized variants are faster than the original programs.

Our original assumption when simplifying the WAM's indexing instructions was that, for predicates having a more general distribution of first arguments, a source-to-source transformation, grouping similar arguments into new predicates, can be used.

Later in time, while noticing that often well-written Prolog code tends to be either "database type" (requiring multiple argument indexing) or "recursion intensive" (with small predicates having a few clauses, fitting well this simplified first argument indexing mechanism), it became clear that it makes sense to handle these two problems separately. As a result, we have kept this simplified indexing scheme (for "recursion intensive" compiled code) unchanged through the evolution of BinProlog and its derivatives. On the other hand, our newest implementation, *Lean Prolog* handles "database type" dynamic code efficiently using a very general multiargument indexing mechanism.

3.8 Binarization: Some infelicities

We have seen that binarization has helped building a simplified abstract machine that provides good performance with help from a few low-level optimizations. However, there are some "infelicities" that one has to face, somewhat similar to what any program transformation mechanism induces at runtime – and DCG grammars come to one's mind in the Prolog world.

For instance the execution order in the body is reified at compile time into a fixed structure. This means that things like dynamic reordering of the goal in a clause body or AND-parallel execution mechanisms become trickier. Also, inline compilation of constructs like if-then-else becomes more difficult – although one can argue that using a source-level technique, when available (e.g. by creating small new predicates) is an acceptable implementation in this case.

4 Data representation

We review here an unconventional data representation choice that turned out to also provide a surprising term-compression mechanism, which can be seen as a generalization of "CDR-coding" (Clark and Green 1977) used in LISP/Scheme systems.

4.1 Tag-on-pointer versus tag-on-data

When describing the data in a cell with a tag, we have basically two possibilities. We can put a tag in the pointer to the data or in the data cell itself.

The first possibility, probably most popular among WAM implementors, allows one to check the tag before deciding *if* and *how* it has to be processed. We choose the second possibility as in the presence of indexing, unifications are more often intended to succeed propagating bindings, rather than being used as a clause selection mechanism. This also justifies why we have not implemented the WAM's traditional `SWITCH_ON_TAG` instruction.

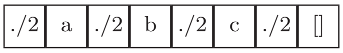

Fig. 2. Compressed list representation of [a,b,c].

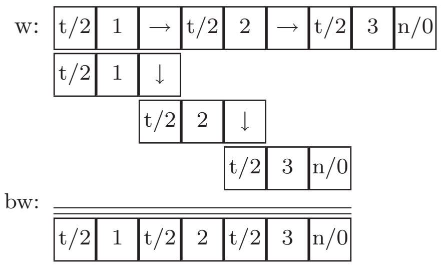

Fig. 3. Term compression. w: WAM, bw: BinWAM.

We found it very convenient to precompute a functor in the code-space as a word of the form `<arity,symbol-number,tag>`[1] and then simply compare it with objects on the heap or in registers. In contrast, in a conventional WAM, one compares the tags, finding out that they are almost always the same, then compares the functor-names and finally compares the arities – an unnecessary but costly if-logic. This is avoided with our *tag-on-data* representation, while also consuming as few tag bits as possible. Only 2 bits are used in BinProlog for tagging *variables*, *integers* and *functors/atoms*[2]. With this representation, a functor fits completely in one word.

As an interesting consequence, as we have found out later, when implementing a symbol garbage collector for a derivative of BinProlog, the "tag-on-data" representation makes scanning the heap for symbols (and updating them in place) a trivial operation.

4.2 Term compression

If a term has a last argument containing a functor, with our tag-on-data representation, we can avoid the extra pointer from the last argument to the functor cell and simply make them collapse. Obviously, the unification algorithm must take care of this case, but the space savings are important, especially in the case of lists, which become contiguous vectors with their Nth element directly addressable at offset `2*sizeof(term)*N+1` bytes from the beginning of the list, as shown in Figure 2.

The effect of this *last argument overlapping* on `t(1,t(2,t(3,n)))` is represented in Figure 3.

This representation also reduces the space consumption for lists and other "chained functors" to values similar or better than in the case of conventional WAMs. We refer to Tarau and Neumerkel (1993) for the details of the term-compression related optimizations of BinProlog.

[1] This technique is also used in various other Prologs, e.g. SICStus, Ciao.

[2] This representation limits arity and available symbol numbers – a problem that went away with the newer 64-bit versions of BinProlog.

5 Optimizing the runtime system

We give here an overview of the optimizations of the runtime system. Most of them are, at this point in time, "folklore" and shared with various other WAM-based Prolog implementations.

Instruction compression. It happens very often that a sequence of consecutive instructions share some WAM state information (Nässén *et al.* 2001). For example two consecutive unify instructions have the *same mode* as they correspond to arguments of the same structure. Moreover, due to our very simple instruction set, some instructions have only a few possible other instructions that can follow them. For example after an EXECUTE instruction, we can have a single, a deterministic or a nondeterministic clause. It makes sense to specialize the EXECUTE instruction with respect to what has to be done in each case. This gives, in the case of calls to deterministic predicates, the instructions EXEC_SWITCH and EXEC_JUMP_IF as mentioned in the section on indexing. On the other hand, some instructions are simply so small that just dispatching them can cost more than actually performing the associated WAM-step.

This in itself is a reason to compress two or more instructions taking less than a word in one instruction. This optimization has been part of WAM-based Prolog systems like Quintus, SICStus, Ciao as well. Also, having a small initial instruction set reduces the number of combined instructions needed to cover all cases. For example by compressing our UNIFY instructions and their WRITE-mode specializations, we get the new instructions:

```
UNIFY_VARIABLE_VARIABLE
WRITE_VARIABLE_VARIABLE
...
```

This gives, in the case of the binarized version of the recursive clause of append/3, the following code:

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

TRUST_ME_ELSE */4,     % keeps also the arity = 4
GET_STRUCTURE X1, ./2
UNIFY_VARIABLE_VARIABLE X5, A1
GET_STRUCTURE X3, ./2
UNIFY_VALUE_VARIABLE X5, A3
EXEC_JUMP_IF  append/4 % actually the address of append/4
```

The choice of candidates for instruction compression was based on low-level profiling (instruction frequencies) and possibility of sharing of common work by two successive instructions and frequencies of functors with various arities.

BinProlog also integrates the preceding GET_STRUCTURE instruction into the double UNIFY instructions and the preceding PUT_STRUCTURE into the double WRITE instructions. This gives another 16 instructions but it covers a large majority of uses of GET_STRUCTURE and PUT_STRUCTURE.

```
GET_UNIFY_VARIABLE_VARIABLE
...
PUT_WRITE_VARIABLE_VALUE
....
```

Reducing interpretation overhead on those critical, high-frequency instructions definitely contributes to the speed of our emulator. As a consequence, in the frequent case of structures of arity=2 (lists included), mode-related IF-logic is completely eliminated, with up to 50% speed improvements for simple predicates like append/3.

The following example shows the effect of this transformation:

```
a(X,Z):-b(X,Y),c(Y,Z). ⇒binary form⇒ a(X,Z,C):-b(X,Y,c(Y,Z,C)).

BinProlog BinWAM code, without compression

a/3:
PUT_STRUCTURE        X4<-c/3
WRITE_VARIABLE       X5
WRITE_VALUE          X2
WRITE_VALUE          X3
MOVE_REG             X2<-X5
MOVE_REG             X3<-X4
EXECUTE              b/3

BinProlog BinWAM code, with instruction compression

PUT_WRITE_VARIABLE_VALUE  X4<-c/3, X5,X2
WRITE_VALUE               X3
MOVE_REGx2                X2<-X5, X3<-X4
EXECUTE                   b/3
```

Note that instruction compression is usually applied inside a procedure. As BinProlog has a unique primitive EXECUTE instruction instead of standard WAM's CALL, ALLOCATE, DEALLOCATE, EXECUTE, PROCEED, we can afford to do instruction compression across procedure boundaries with very little increase in code size due to relatively few different ways to combine control instructions. Interprocedural instruction compression can be seen as a kind of "hand-crafted" *partial evaluation* at implementation language level, intended to optimize the main loop of the WAM-emulator. It has the same effect as *partial evaluation* at source level that also eliminates procedure calls. At the global level, knowledge about possible continuations can also remove the runtime effort of address look-up for metavariables in predicate positions and of useless trailing and dereferencing.

(Most of) the benefits of two-stream compilation for free. Let us point out here that in the case of GET_*_* instructions we have the benefits of separate READ and WRITE streams (for instance avoidance of mode checking) on some high-frequency instructions without actually incurring the compilation complexity and emulation overhead in generating them. As terms of depth 1 and functors of low arity dominate statistically Prolog programs, we can see that our instruction compression scheme

actually behaves as if two separate instruction streams were present, most of the time!

6 Logic engines as interactors

We now turn the page to a historically later architectural feature of BinProlog and its newer derivatives. While orthogonal to the BinWAM architecture, it shares the same philosophy: proceed with a fundamental system simplification on purely *esthetic grounds*, independently of short-term performance concerns, and hope that overall elegance will provide performance improvements for free, later[3].

BinProlog's Java-based re-implementation, *Jinni* has been mainly used in various applications (Tarau 1998, 1999a, 1999b, 2004a) as an *intelligent agent infrastructure*, by taking advantage of Prolog's knowledge processing capabilities in combination with a simple and easily extensible runtime kernel supporting a flexible reflexion mechanism (Tyagi and Tarau 2001). Naturally, this has suggested to investigate whether some basic agent-oriented language design ideas can be used for a refactoring of pure Prolog's interaction with the external world.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages (FIPA 1997) have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At the same time, it has been a long tradition of logic programming languages (Hermenegildo 1986; Lusk *et al.* 1993) to use multiple Logic Engines for supporting concurrent execution.

In this context, the Jinni Prolog agent programming framework (Tarau 2004b) and the recent versions of the BinProlog system (Tarau 2006) have been centered around logic engine constructs providing an API that supports reentrant instances of the language processor. This has naturally led to a view of Logic Engines as instances of a generalized family of iterators called *Fluents* (Tarau 2000), which have allowed the separation of the first-class language interpreters from the multithreading mechanism, while providing, at the same time, a very concise source-level reconstruction of Prolog's built-ins. Later, we have extended the original *Fluents* with a few new operations (Tarau and Majumdar 2009) supporting bidirectional, mixed-initiative exchanges between engines.

The resulting language constructs, which we have called *Interactors*, express coroutining, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source-level virtually all Prolog built-ins, including dynamic database operations.

In a wider programming language implementation context, a `yield` statement supports basic coroutining in newer object oriented languages like `Ruby` `C#` and

[3] Even in cases when such hopes do not materialize, indirect consequences of such architectural simplifications often lower software risks and bring increased system reliability while keeping implementation effort under control.

Python but it goes back as far as Conway (1963) and the *Coroutine Iterators* introduced in older languages like CLU (Liskov *et al.* 1981).

6.1 Logic Engines as answer generators

Our *Interactor API*, a unified interface to various stateful objects interacting with Prolog processors, has evolved progressively into a practical Prolog implementation framework starting with Tarau (2000) and continued with Tarau (2008a) and Tarau and Majumdar (2009). We summarize it here while instantiating the more general framework to focus on interoperation of Logic Engines. We refer to Tarau and Majumdar (2009) for the details of an emulation in terms of Horn Clause Logic of various engine operations.

An *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another Engine very much the same way a programmer controls Prolog's interactive top-level loop: launch a new goal, ask for a new answer, interpret it, react to it. A *Logic Engine* is an Engine running a Horn Clause Interpreter with LD-resolution (Tarau and Boyer 1993) on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas.

In our newer implementations, with all data areas dynamic, engines are lightweight and engine creation is fast and memory efficient[4] to the point where using them as building blocks for a significant number of built-ins and various language constructs is not always prohibitive in terms of performance.

6.2 Iterating over computed answers

Note that our Logic Engines are seen, in an object oriented-style, as implementing the *interface* `Interactor`. This supports a *uniform* interaction mechanism with a variety of objects ranging from Logic Engines to file/socket streams and iterators over external data structures.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine. The query

```
get(Interactor,AnswerInstance)
```

[4] The additional operation `load_engine(Interactor,AnswerPattern,Goal)` that clears data areas and initializes an engine with `AnswerPattern,Goal` has also been available as a further optimization, by providing a mechanism to reuse an existing engine.

tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then, backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the

```
stop(Interactor)
```

operation that might or might not reclaim resources held by the engine. In our later implementation *Lean Prolog*, we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected. While this API clearly refers to operations going beyond Horn Clause logic, it can be shown that a fairly high-level pure Prolog semantics can be given to them in a style somewhat similar to what one would do when writing a Prolog interpreter in Haskell, as shown in Section 4 of Tarau and Majumdar (2009).

So far, these operations provide a minimal API, powerful enough to switch tasks cooperatively between an engine and its "client"[5] and emulate key Prolog built-ins like `if-then-else` and `findall` (Tarau 2000), as well as higher order operations like *fold* and *best_of* (Tarau and Majumdar 2009). We give more details on emulations of these constructs in Section 7.

6.3 *A yield/return operation*

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

saves the state of the engine and transfers *control* and a *result* `Term` to its client. The client receives a copy of `Term` when using its `get/2` operation.

Note that an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/get` operations, an engine can provide a stream of *intermediate/final results* to itsclient,

[5] Another Prolog engine using and engine's services.

without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism simply by defining

```
throw(E):-return(exception(E)).
```

When combined with a `catch(Goal,Exception,OnException)`, on the client side, the client can decide, upon reading the exception with `get/2`, if it wants to handle it or to throw it to the next level.

6.4 Coroutining logic engines

Coroutining has been in use in Prolog systems mostly to implement constraint programming extensions. The typical mechanism involves *attributed variables* holding suspended goals that may be triggered by changes in the instantiation state of the variables. We discuss here a different form of coroutining, induced by the ability to switch back and forth between engines.

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's *client*[6] to *inject* new goals (executable data) to an arbitrary inner context of another engine. Two new primitives are needed:

```
to_engine(Engine,Data)
```

that is called by the client to send data to an Engine, and

```
from_engine(Data)
```

that is called by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

(1) the *client* creates and initializes a new *engine*
(2) the client triggers a new computation in the *engine*, parameterized as follows:
    (a) the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
    (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
    (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
(3) the *client* interprets the answer and proceeds with its next computation step
(4) the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

Using a metacall mechanism like `call/1`[7], one can implement a close equivalent of Ruby's `yield` statement as follows:

[6] Another engine that uses an engine's services.
[7] Which, interestingly enough, can itself be emulated in terms of engine operations (Tarau 2000) or directly through a source level transformation (Tarau and Boyer 1990).

```
ask_engine(Engine,(Answer:-Goal), Result):-
  to_engine(Engine,(Answer:-Goal)),
  get(Engine,Result).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

The predicate `ask_engine/3` sends a query (possibly built at runtime) to an engine, which, in turn, executes it and returns a result with an `engine_yield` operation. The query is typically a goal or a pattern of the form `AnswerPattern:-Goal` in which case the engine interprets it as a request to instantiate `AnswerPattern` by executing `Goal` before returning the answer instance.

As the following example shows, this allows the client to use, from outside, the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1=>S2),sum_loop(S2).

inc_test(R1,R2):-new_engine(_,sum_loop(0),E),
   ask_engine(E,(S1=>S2:-S2 is S1+2),R1),
   ask_engine(E,(S1=>S2:-S2 is S1+5),R2).

?- inc_test(R1,R2).
R1=the(0=>2), R2=the(2=>7).
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed. The goal can then access the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

7 Source level extensions through new definitions

To give a glimpse of the expressiveness of the resulting Horn Clause + Engines language, first described in Tarau (2000) we specify a number of built-in predicates known as "impossible to emulate" in Horn Clause Prolog (except by significantly lowering the level of abstraction and implementing something close to the virtual machine itself).

7.1 *Negation, first_solution/3, if_then_else/3*

These constructs are implemented simply by discarding all but the first solution produced by an engine. The predicate `first_solution` (usable to implement `once/1`), returns `the(X)` or the atom `no` as first solution of goal `G`:

```
first_solution(X,G,Answer):-
  new_engine(X,G,E),
  get(E,R),stop(E),
  Answer=R.

not(G):-first_solution(_,G,no).
```

The same applies to an emulation of Prolog's if-then-else construct, shown here as the predicate `if_then_else/3`, which, if `Cond` succeeds, calls `Then`, keeping the bindings produced by `Cond` and otherwise calls `Else` after undoing the bindings of the call to `Cond`.

```
if_then_else(Cond,Then,Else):-
  new_engine(Cond,Cond,E),
  get(E,Answer), stop(E),
  select_then_else(Answer,Cond,Then,Else,Goal),
  Goal.

select_then_else(the(Cond),Cond,Then,_Else,Then).
select_then_else(no,_,_,_Then,Else,Else).
```

Note that these operations require the use of `CUT` in typical Prolog library implementations. While in the presence of engines, one can control the generation of multiple answers directly and only use the CUT when more complex control constructs are required (like in the case of embedded disjunctions), given the efficient WAM-level implementation of `CUT` and the frequent use of Prolog's if-then-else construct, emulations of these built-ins can be seen mostly as an executable specification of their faster low-level counterparts.

7.2 Reflective meta-interpreters

A simple Horn Clause+Engines meta-interpreter **metacall/1** just *reflects* backtracking through **element_of/2** over deterministic engine operations.

```
metacall(Goal):-
  new_engine(Goal,Goal,E),
  element_of(E,Goal).

element_of(E,X):-get(E,the(A)),select_from(E,A,X).

select_from(_,A,A).
select_from(E,_,X):-element_of(E,X).
```

We can see `metacall/1` as an operation that fuses two orthogonal language features provided by an engine: *computing an answer of a Goal*, and *advancing to the next answer*, through the source level operations `element_of/2` and `select_from/3` which "borrow" the ability to backtrack from the underlying interpreter. The existence of the simple meta-interpreter defined by `metacall/1` indicates that first-class engines lift the expressiveness of Horn Clause logic significantly.

7.3 All-solution predicates

All-solution predicates like `findall/3` can be obtained by collecting answers through recursion. The (simplified) code consists of `findall/3` that creates an engine and `collect_all_answers/3` that recurses while new answers are available.

```
findall(X,G,Xs):-
  new_engine(X,G,E),
  get(E,Answer),
  collect_all_answers(Answer,E,Xs).

collect_all_answers(no,_,[]).
collect_all_answers(the(X),E,[X|Xs]):-get(E,Answer),
  collect_all_answers(Answer,E,Xs).
```

Note that after the auxiliary engine created for `findall/3` is discarded, heap space is needed *only to hold the computed answers*, as it is also the case with the conventional implementation of `findall`. Note also that the implementation handles embedded uses of `findall` naturally and that no low-level built-ins are needed.

7.4 Term copying and instantiation state detection

As standardizing variables in the returned answer is part of the semantics of `get/2`, term copying is just computing a first solution to `true/0`. Implementing `var/1` uses the fact that only free variables can have copies unifiable with two distinct constants.

```
copy_term(X,CX):-first_solution(X,true,the(CX)).

var(X):-copy_term(X,a),copy_term(X,b).
```

The previous definitions have shown that the resulting language subsumes (through user provided definitions) constructs like negation as failure, if-then-else, once, `copy_term`, `findall` – this suggests calling this layer *Kernel Prolog*. As Kernel Prolog contains negation as failure, following Deransart *et al.* (1996) we can, in principle, use it for an executable specification of full Prolog.

It is important to note here that the engine-based implementation serves in some cases just as a proof of expressiveness and that, in practice, operations like `var/1` for which even a small overhead is unacceptable are implemented directly as built-ins. Nevertheless, the engine-based source-level definitions provide in all cases a reference implementation usable as a specification for testing purposes.

7.5 Implementing exceptions

While it is possible to implement an exception mechanism at source level as shown in Tarau and Dahl (1994), through a continuation passing program transformation (binarization), one can use engines for the same purpose. By returning a new answer pattern as indication of an exception, a simple and efficient implementation of exceptions is obtained.

We have actually chosen this implementation scenario in the BinProlog compiler, which also provides a **return/1** operation to exit an engine's emulator loop with an arbitrary answer pattern, possibly before the end of a successful derivation. The (somewhat simplified) code is as follows:

```
throw(E):-return(exception(E)).

catch(Goal,Exception,OnException):-
  new_engine(answer(Goal),Goal,Engine),
  element_of(Engine,Answer),
  do_catch(Answer,Goal,Exception,OnException,Engine).

do_catch(exception(E),_,Exception,OnException,Engine):-
  (E=Exception→
    OnException % call action if matching
  ; throw(E)    % throw again otherwise
  ), stop(Engine).
do_catch(the(Goal),Goal,_,_,_).
```

The `throw/1` operation returns a special exception pattern, while the `catch/3` operation stops the engine, calls a handler on matching exceptions or rethrows nonmatching ones to the next layer. If engines are lightweight, the cost of using them for exception handling is acceptable performance-wise, most of the time. However, it is also possible to reuse an engine (using `load_engine/3`) – for instance in an inner loop, to define a handler for all exceptions that can occur, rather than wrapping up each call into a new engine with a catch.

7.6 Interactors and higher order constructs

As a glimpse at the expressiveness of the Interactor API, we implement, in the tradition of higher order functional programming, a *fold* operation. The predicate `efoldl` can be seen as a generalization of `findall` connecting results produced by independent branches of a backtracking Prolog engine by applying to them a closure `F` using `call/4`:

```
efoldl(Engine,F,R1,R2):-get(Engine,X),efoldl_cont(X,Engine,F,R1,R2).

efoldl_cont(no,_Engine,_F,R,R).
efoldl_cont(the(X),Engine,F,R1,R2):-call(F,R1,X,R),efoldl(Engine,F,R,R2).
```

Classic functional programming idioms like *reverse as fold* are then implemented simply as:

```
reverse(Xs,Ys):-
  new_engine(X,member(X,Xs),E),
  efoldl(E,reverse_cons,[],Ys).

reverse_cons(Y,X,[X|Y]).
```

Note also the automatic *deforestation* effect (Wadler 1990) of this programming style – no intermediate list structures need to be built, if one wants to aggregate the values retrieved from an arbitrary generator engine with an operation like sum or product.

8 Extending the Prolog kernel using interactors

We review here a few typical extensions of the Prolog kernel showing that using first-class Logic Engines results in a compact and portable architecture that is built almost entirely *at source level.*

8.1 Emulating dynamic databases with interactors

The gain in expressiveness coming directly from the view of Logic Engines as iterative answer generators (i.e. Fluents (Tarau 2000)) is significant. The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

The key idea for implementing dynamic database operations with interactors is to use a logic engine's state in an infinite recursive loop.

First, a simple difference-list based infinite server loop is built:

```
queue_server:-queue_server(Xs,Xs).

queue_server(Hs1,Ts1):-
  from_engine(Q),server_task(Q,Hs1,Ts1,Hs2,Ts2,A),return(A),
  queue_server(Hs2,Ts2).
```

Next, we provide the queue operations, needed to maintain the state of the database. To keep the code simple, we only focus in this section on operations resulting in additions at the end of the database.

```
server_task(add_element(X),Xs,[X|Ys],Xs,Ys,yes).
server_task(queue,Xs,Ys,Xs,Ys,Xs-Ys).
server_task(delete_element(X),Xs,Ys,NewXs,Ys,YesNo):-
  server_task_delete(X,Xs,NewXs,YesNo).
```

Then, we implement the auxiliary predicates supporting various queue operations.

```
server_task_delete(X,Xs,NewXs,YesNo):-
  select_nonvar(X,Xs,NewXs),!,
  YesNo=yes(X).
server_task_delete(_,Xs,Xs,no).

select_nonvar(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
select_nonvar(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],
  select_nonvar(X,Xs,Ys).
```

Next, we put it all together, as a dynamic database API.

We can create a new engine server providing Prolog database operations:

```
new_edb(Engine):-new_engine(done,queue_server,Engine).
```

We can add new clauses to the database

```
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).
```

and we can return fresh instances of asserted clauses

```
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).
```

or remove them from the database

```
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
  ask_engine(Engine,delete_element(Clause),the(yes(Clause))).
```

Finally, the database can be discarded by stopping the engine that hosts it:

```
edb_delete(Engine):-stop(Engine).
```

Externally implemented dynamic databases can also be made visible as Interactors and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases can be provided. This simplifies adding module, object or agent layers at source level. By combining database and communication Interactors, support for mobile code and autonomous agents can be built as shown in Tarau and Dahl (2001). Encapsulating external stateful objects like file systems, external database or Web service interfaces as Interactors can provide a uniform interfacing mechanism and reduce programmer learning curves in Prolog applications.

A note on practicality is needed here. While indexing can be added at source level by using hashing on various arguments, the relative performance compared to compiled code, of this emulated database is 2–3 orders of magnitude slower. Therefore, in our various Prolog systems, we have used this more as an executable specification rather than the default implementation of the database.

8.2 Refining control: A backtracking if-then-else

Various Prolog implementations also provide a variant of `if-then-else` (called `*->/3` in SWI-Prolog and `if/3` in SICStus-Prolog) that either backtracks over multiple answers of its guard `Cond` (and calls its `Then` branch for each) or it switches to the `Else` branch if no such answers of `Cond` are found. With the same API, we can implement it at source level as follows:

```
if_any(Cond,Then,Else):-
  new_engine(Cond,Cond,Engine),
  get(Engine,Answer),
  select_then_or_else(Answer,Engine,Cond,Then,Else).

select_then_or_else(no,_,_,_,Else):-Else.
select_then_or_else(the(BoundCond),Engine,Cond,Then,_):-
  backtrack_over_then(BoundCond,Engine,Cond,Then).

backtrack_over_then(Cond,_,Cond,Then):-Then.
backtrack_over_then(_,Engine,Cond,Then):-
  get(Engine,the(NewBoundCond)),
  backtrack_over_then(NewBoundCond,Engine,Cond,Then).
```

8.3 Simplifying algorithms: Interactors and combinatorial generation

Various combinatorial generation algorithms have elegant backtracking implementations. However, it is notoriously difficult (or inelegant, through the use of ad-hoc side effects) to compare answers generated by different OR-branches of Prolog's search tree.

8.3.1 Comparing alternative answers

Optimization problems, selecting the "best" among answers produced on alternative branches can easily be expressed as follows:

- running the generator in a separate logic engine
- collecting and comparing the answers in a client controlling the engine

The second step can actually be automated, provided that the comparison criterion is given as a predicate

```
compare_answers(Comparator,First,Second,Best)
```

to be applied to the engine with an `efold` operation:

```
best_of(Answer,Comparator,Generator):-
  new_engine(Answer,Generator,E),
  efoldl(E,compare_answers(Comparator),no,Best),
  Answer=Best.

compare_answers(Comparator,A1,A2,Best):-
  ( A1\==no,call(Comparator,A1,A2)->Best=A1
  ; Best=A2
  ).

?-best_of(X,>,member(X,[2,1,4,3])).
X=4
```

Note that in the call to `compare_answers`, the closure `compare_answers(Comparator)` gets the extra arguments `A1` and `A2` out of which, depending on the comparison, `Best` is selected at each step of `efoldl`.

8.3.2 Encapsulating infinite computation streams

An infinite stream of natural numbers is implemented as:

```
loop(N):-return(N),N1 is N+1,loop(N1).
```

The following example shows a simple space efficient generator for the infinite stream of prime numbers:

```
prime(P):-prime_engine(E),element_of(E,P).

prime_engine(E):-new_engine(_,new_prime(1),E).

new_prime(N):- N1 is N+1,
  (test_prime(N1) -> true ; return(N1)),
  new_prime(N1).

test_prime(N):-M is integer(sqrt(N)),between(2,M,D),N mod D =:=0
```

Note that the program has been wrapped, using the `element_of` predicate to provide one answer at a time through backtracking. Alternatively, a forward recursing client can use the `get(Engine)` operation to extract primes one at a time from the stream.

9 A short history of BinProlog and its derivatives

The first iteration of BinProlog goes back to around 1990. Along the years, it has pioneered some interesting architectural choices while adopting a number of new (at the time) implementation ideas from others. From 1999 on, we have also released a Java port of BinProlog called Jinni Prolog, using essentially the same runtime system and compiler as BinProlog and resulting in some new developments happening either on the Java or C side. Some of BinProlog's features are interesting to mention mostly for historical reasons – as they either became part of various Prolog systems, when genuinely practical, or, on the contrary, have turned out to have only limited, program-specific benefits. Among features for which BinProlog has been either a pioneer or an early adopter in the world of Prolog implementations that have not been covered in this paper are:

- an efficient implementation of `findall` using a heap splitting technique resulting in a single copy operation (Tarau 1992)
- a multithreading API using native threads under explicit programmer control (around 1992–1993)
- a blackboard architecture using Linda coordination between threads (Tarau and De Bosschere 1993a; De Bosschere and Tarau 1996)
- backtrackable global variables (around 1993)
- a mechanism for "partial compilation" to C (Tarau *et al.* 1994, 1996)
- using continuations to implement Prolog extensions, including catch/throw (Tarau and Dahl 1994)
- cyclic terms (originating in Prolog III) and subterm-sharing implemented using a space efficient value trailing mechanism (around 1993)
- memoing of goal-independent answer substitutions for deterministic calls (Tarau and De Bosschere 1993b; Tarau *et al.* 1997)
- a DCG variant using backtrackable state updates (Dahl *et al.* 1997)
- on the fly compilation of dynamic code, based on runtime call/update statistics (around 1994–1995) (a technique similar to the *HotSpot* compilation now popular in Java VMs)
- segment preserving copying GC (Demoen *et al.* 1996)
- assumption grammars – a mechanism extending Prolog grammar with hypothetical reasoning (Dahl *et al.* 1997)
- strong mobility of code and data by transporting live continuations between Prolog processes (Tarau and Dahl 1998, 2001)
- Prolog-based shared virtual worlds supporting simple natural language interactions (Tarau *et al.* 1999)

Elements of the BinProlog continuation passing implementation model have been successfully reused in a few different Prolog systems:

- jProlog (`http://www.cs.kuleuven.be/~bmd/PrologInJava/`) written in Java, mostly by Bart Demoen with some help from Paul Tarau in 1997, used a Prolog to Java translator with binarization as a source to source transformation
- Jinni Prolog, written in Java by Paul Tarau (`http://www.binnetcorp.com/Jinni`) actively developed since 1998, first as continuation passing interpreter and later as a BinWAM compiler
- A Java port of Free Prolog (a variant of BinProlog 2.0) by Peter Wilson `http://www.binnetcorp.com/OpenCode/free_prolog.html` (around 1999) with additions and fixes by Paul Tarau
- Kernel Prolog, a continuation passing interpreter written in Java by Paul Tarau (`http://www.binnetcorp.com/OpenCode/kernelprolog.html`) around 1999
- PrologCafe (`http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/`), a fairly complete Prolog system derived from jProlog implemented by Mutsunori Banbara and Naoyuki Tamura
- `P#` derived from PrologCafe, written in `C#` by Jon Cook (`http://homepages.inf.ed.ac.uk/jcook/`)
- Carl Friedrich Bolz's Python-based Prolog interpreter and JIT compiler, using AND+OR-continuations directly, without a program transformation (Bolz *et al.* 2010)
- Lean Prolog – a new first-class Logic Engines based lightweight Prolog system using two identical C and Java-based BinWAM runtime systems to balance performance and flexibility implemented by Paul Tarau (work in progress, started in 2008)

10 Related work

Most modern Prolog implementations are centered around the WAM (Warren 1983; Aït-Kaci 1991), which has stood amazingly well the test of time. In this sense, BinProlog's BinWAM is no exception although its overall "rate of mutations" with respect to the original WAM is probably comparable to systems like Neng-Fa Zhou's TOAM or TOAM-Jr (Zhou *et al.* 1990; Zhou 2007) or Jan Wielemaker's SWI-Prolog (Wielemaker 2003) and definitely higher, if various extensions are factored out, than the basic architecture of systems like GNU-Prolog (Diaz and Codognet 2001), SICStus Prolog (Carlsson *et al.*), Ciao (Carro and Hermenegildo 1999), YAP (da Silva and Costa 2006) or XSB (Swift and Warren 1994). We refer to Van Roy (1994) and Demoen and Nguyen (2000) for extensive comparisons of compilation techniques and abstract machines for various logic programming systems.

Techniques for adding built-ins to binary Prolog are first discussed in Demoen and Mariën (1992), where an implementation-oriented view of binary programs that a binary program is simply one that does not need an environment in the WAM is advocated. Their paper also describes a technique for implementing Prolog's CUT in a binary Prolog compiler. Extensions to BinProlog's AND-continuation passing transformation to also cover OR-continuations are described in Lindgren (1994).

Multiple Logic Engines have been present in one form or another in various parallel implementation of logic programming languages (Ueda 1985; Shapiro 1989). Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel (Hermenegildo 1986) and OR-parallel (Lusk *et al.* 1993) execution models are worth mentioning.

However, with the exception of this author's papers on this topic (Tarau 1999a, 1999c, 2000, 2008a, 2011; Tarau and Dahl 2001; Tarau and Majumdar 2009), we have not found an extensive use of first-class Logic Engines as a mechanism to enhance language expressiveness, independently of their use for parallel programming, with maybe the exception of Casas *et al.* (2007) where such an API is discussed for parallel symbolic languages in general. In combination with multithreading (Tarau 2011), our own engine-based API bears similarities with various other Prolog systems, notably (Carro and Hermenegildo 1999; Wielemaker 2003) while focusing on uncoupling "concurrency for performance" and "concurrency for expressiveness".

The use of a garbage collected, infinitely looping recursive program to encapsulate state goes back to early work in logic programming and it is likely to be common in implementing various server programs. However, an infinitely recursive pure Horn Clause program is an "information sink" that does not communicate with the outside world on its own. The minimal API (`to_engine/2` and `from_engine/1`) described in this paper provides interoperation with such programs, in a generic way.

11 Conclusion

At the time of writing, BinProlog has been around for almost 20 years. As mostly a single-implementor system, BinProlog has not kept up with systems that have benefited from a larger implementation effort in terms of optimizations and extensions like constraints or tabling, partly also because our research interests have diverged towards areas as diverse as natural language processing, logic synthesis or computational mathematics. On the other hand, re-implementations in Java and a number of experimental features make it still relevant as a due member of the unusually rich and colorful family of Prolog systems.

Our own re-implementations of BinProlog's virtual machine have been extended with first-class Logic Engines that can be used to build on top of pure Prolog a practical Prolog system, including dynamic database operations, entirely at source level. In a broader sense, interactors can be seen as a starting point for rethinking fundamental programming language constructs like Iterators and Coroutining in terms of language constructs inspired by *performatives* in agent-oriented programming. Along these lines, we are currently building a new BinWAM-based implementation, *Lean Prolog*, which combines a minimal WAM kernel with an almost entirely source-level *interactor*-based implementation of Prolog's built-ins and libraries. We believe that under this new incarnation some of BinProlog's architectural choices are likely to have an interesting impact on the design and implementation of future logic programming languages.

Acknowledgements

We are thankful to María García de la Banda, Bart Demoen, Manuel Hermenegildo, Joachim Schimpf and D.S. Warren and to the anonymous reviewers for their careful reading and thoughtful comments on earlier drafts of this paper.

Special thanks go to Koen De Bosschere, Bart Demoen, Geert Engels, Ulrich Neumerkel, Satyam Tyagi and Peter Wilson for their contribution to the implementation of BinProlog and Jinni Prolog components.

Richard O'Keefe's public domain Prolog parser and writer have been instrumental in turning BinProlog into a self-contained Prolog system quickly.

Fruitful discussions with Hassan Aït-Kaci, Patrice Boizumault, Michel Boyer, Mats Carlsson, Jacques Cohen, Veronica Dahl, Bart Demoen, Thomas Lindgren, Arun Majumdar, Olivier Ridoux, Kostis Sagonas, Sten-Åke Tärnlund, D. S. Warren, Neng-Fa Zhou and comments from a large number of BinProlog users helped improving its design and implementation.

References

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, USA. ISBN: 0-262-51058-8 978-0-262-51058-5.

Bolz, C. F., Leuschel, M. and Schneider, D. 2010. Towards a Jitting VM for Prolog execution. In *Proc. of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10).* ACM, New York, NY, USA, 99–108.

Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H. and Sjoland, T. 2009. SICStus Prolog user's manual. SICS, Kista, Sweden.

Carro, M. and Hermenegildo, M. V. 1999. Concurrency in Prolog using threads and a shared database. In *Proc. of the International Conference on Logic Programming (ICLP)*, 320–334.

Casas, A., Carro, M. and Hermenegildo, M. 2007. Towards a high-level implementation of flexible parallelism primitives for symbolic languages. In *Proc. of the 2007 International Workshop on Parallel Symbolic Computation.* ACM, New York, NY, USA, 93–94.

Clark, D. W. and Green, C. C. 1977. An empirical study of list structure in Lisp. *Communications of the ACM 20*, 78–87.

Conway, M. E. 1963. Design of a separable transition-diagram compiler. *Communications of the ACM 6,* 7, 396–408.

Dahl, V., Tarau, P. and Li, R. 1997. Assumption grammars for processing natural language. In *Proc. of the 14th International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, USA, 256–270.

da Silva, A. F. and Costa, V. S. 2006. The design and implementation of the yap compiler: An optimizing compiler for logic programming languages. In *Proc. of the International Conference on Logic Programming (ICLP)*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, Berlin, Heidelberg, 461–462.

De Bosschere, K. and Tarau, P. January 1996. Blackboard-based extensions in Prolog. *Software—Practice and Experience 26,* 1, 49–69.

Demoen, B. 1992. On the transformation of a prolog program to a more efficient binary program. In *Proc. of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR)*, 242–252.

Demoen, B., Engels, G. and Tarau, P. 1996. Segment preserving copying garbage collection for WAM based Prolog. In *Proc. of the 1996 ACM Symposium on Applied Computing*. ACM Press, Philadelphia, PA, USA, 380–386.

Demoen, B. and Mariën, A. 1992. Implementation of Prolog as binary definite programs. In *Proc. of the First Russian Conference on Logic Programming (RCLP)*, A. Voronkov, Ed. Lecture Notes in Artificial Intelligence, vol. 592. Springer-Verlag, Berlin, Heidelberg, 165–176.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *Proc. of the First International Conference on Computational Logic (CL '00)*. Springer-Verlag, London, UK, 1240–1254.

Deransart, P., Ed-Dbali, A. and Cervoni, L. 1996. *Prolog: The Standard*, Springer-Verlag, Berlin. ISBN: 3-540-59304-7.

Diaz, D. and Codognet, P. 2001. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming 2001,* 6, 1–29.

FIPA. 1997. FIPA 97 specification part 2: Agent communication language. Version 2.0.

Hermenegildo, M. V. 1986. An abstract machine for restricted and-parallel execution of logic programs. In *Proc. of the Third International Conference on Logic Programming*. Springer-Verlag, New York, NY, USA, 25–39.

Lindgren, T. 1994. A continuation-passing style for prolog. In *Proc. of the 1994 International Symposium on Logic programming (ILPS '94)*. MIT Press, Cambridge, MA, USA, 603–617.

Liskov, B., Atkinson, R. R., Bloom, T., Moss, J. E. B., Schaffert, C., Scheifler, R. and Snyder, A. 1981. *CLU Reference Manual.* Lecture Notes in Computer Science, vol. 114. Springer, Berlin, Heidelberg.

Lloyd, J. 1987. *Foundations of Logic Programming (Symbolic Computation/Artificial Intelligence)*, 2nd ed. Springer-Verlag, Berlin. .

Lusk, E., Mudambi, S., Gmbh, E. and Overbeek, R. 1993. Applications of the aurora parallel Prolog system to computational molecular biology. In *Proc. of the Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems (JICSLP '92),* Washington DC. MIT Press.

Nässén, H., Carlsson, M. and Sagonas, K. 2001. Instruction merging and specialization in the sicstus prolog virtual machine. In *Proc. of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '01)*. ACM, New York, NY, USA, 49–60.

Neumerkel, U. 1992. *Specialization of Prolog Programs with Partially Static Goals and Binarization.* PhD Thesis, Technische Universität Wien.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Computing Serveys 21,* 3, 413–510.

Swift, T. and Warren, D. S. 1994. An abstract machine for SLG resolution: Definite programs. In *Proc. of the 1994 International Symposium on Logic Programming*, M. Bruynooghe, Ed. MIT Press, Massachusetts Institute of Technology, 633–652.

Tarau, P. 1991. A simplified abstract machine for the execution of binary metaprograms. In *Proc. of the Logic Programming Conference '91*. ICOT, Tokyo, 119–128.

Tarau, P. 1992. Ecological memory management in a continuation passing Prolog engine. In *Proc. of the International Workshop on Memory Management(IWMM '92)*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, vol. 637. Springer, Berlin, Heidelberg, 344–356.

Tarau, P. 1998. Towards inference and computation mobility: The Jinni experiment. In *Proc. of the European Workshop on Logics in Artificial Intelligence (JELIA '98)*, J. Dix and U. Furbach, Eds. Lecture Notes in Artificial Intelligence, vol. 1489. Springer, Dagstuhl, Germany, 385–390. Invited talk.

Tarau, P. 1999a. Inference and computation mobility with jinni. In *The Logic Programming Paradigm: A 25 Year Perspective*, K. Apt, V. Marek and M. Truszczynski, Eds. Springer, Berlin, Heidelberg, 33–48. ISBN 3-540-65463-1.

Tarau, P. 1999b. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proc. of the Fourth International Conference on the Practical Application of Intelligent Agents and Multi-Agents*, London, UK, 109–123.

Tarau, P. 1999c. Multi-engine horn clause Prolog. In *Proc. of the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Las Cruces, NM, USA, G. Gupta and E. Pontelli, Eds.

Tarau, P. 2000. Fluents: A refactoring of Prolog for uniform reflection and interoperation with external objects. In *Proc. of the First International Conference on Computational Logic (CL '00)*, J. Lloyd, Ed. Lecture Notes in Computer Science, vol. 1861. Springer-Verlag, London, UK.

Tarau, P. 2004a. Agent oriented logic programming constructs in jinni 2004. In *Proc. of the 20th International Conference on Logic Programming (ICLP '04)*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, Saint-Malo, France, 477–478.

Tarau, P. 2004b. Orthogonal language constructs for agent oriented logic programming. In *Proc. of the Fourth Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS '04)*, M. Carro and J. F. Morales, Eds. Springer, Saint-Malo, France.

Tarau, P. 2006. *BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide*. Technical Report. BinNet Corp.

Tarau, P. 2008a. Logic engines as interactors. In *Proc. of the 24th International Conference on Logic Programming*, M. Garcia de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science. Springer, Udine, Italy, 703–707.

Tarau, P. 2008b. The Jinni Prolog Compiler: A fast and flexible Prolog-in-Java [online]. Accessed 2008. URL: http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html

Tarau, P. 2011. Concurrent programming constructs in multi-engine prolog. In *Proc. of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*. ACM, New York, NY, USA.

Tarau, P. and Boyer, M. 1990. Elementary logic programs. In *Proc. of the Programming Language Implementation and Logic Programming*, P. Deransart and J. Maluszyński, Eds. Lecture Notes in Computer Science, vol. 456. Springer, Berlin, Heidelberg, 159–173.

Tarau, P. and Boyer, M. 1993. Nonstandard answers of elementary logic programs. In *Constructing Logic Programs*, J. Jacquet, Ed. J. Wiley, Hoboken, NJ, 279–300.

Tarau, P. and Dahl, V. 1994. Logic programming and logic grammars with first-order continuations. In *Proc. of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR '94),* Lecture Notes in Computer Science. Springer, Pisa.

Tarau, P. and Dahl, V. 1998. Mobile threads through first order continuations. In *Proc. of the APPAI-GULP-PRODE '98*, Coruna, Spain.

Tarau, P. and Dahl, V. May 2001. High-level networking with mobile code and first order AND-continuations. *Theory and Practice of Logic Programming 1,* 3, 359–380. Cambridge University Press.

Tarau, P. and De Bosschere, K. 1993a. Blackboard based logic programming in BinProlog. In *Proc. of the Fifth University of New Brunswick Artificial Intelligence Symposium*, L. Goldfarb, Ed. Fredericton, NB, 137–147.

Tarau, P. and De Bosschere, K. 1993b. Memoing with abstract answers and Delphi lemmas. In *Logic Program Synthesis and Transformation*, Y. Deville, Ed. Springer-Verlag, Louvain-la-Neuve, 196–209.

Tarau, P., De Bosschere, K., Dahl, V. and Rochefort, S. March 1999. LogiMOO: An extensible multi-user virtual world with natural language control. *Journal of Logic Programming 38,* 3, 331–353.

Tarau, P., De Bosschere, K. and Demoen, B. November 1996. Partial translation: Towards a portable and efficient Prolog implementation technology. *Journal of Logic Programming 29,* 1–3, 65–83.

Tarau, P., De Bosschere, K. and Demoen, B. February 1997. On Delphi lemmas and other memoing techniques for deterministic logic programs. *Journal of Logic Programming 30,* 2, 145–163.

Tarau, P., Demoen, B. and De Bosschere, K. 1994. The power of partial translation: An experiment with the C-ification of binary Prolog. In *Proc. of the First COMPULOG-NOE Area Meeting on Parallelism and Implementation Technology*, Madrid/Spain, M. García de la Banda, J. and M. Hermenegildo, Eds, Bordeau, France, 3–17.

Tarau, P. and Majumdar, A. 2009. Interoperating logic engines. In *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL '09)*. Lecture Notes in Computer Science, vol. 5418. Springer, Savannah, Georgia, 137–151.

Tarau, P. and Neumerkel, U. November 1993. *Compact Representation of Terms and Instructions in the BinWAM*. Technical Report 93-3, Department of d'Informatique, Université de Moncton. Available by ftp from clement.info.umoncton.ca

Tyagi, S. and Tarau, P. 2001. A most specific method finding algorithm for reflection based dynamic Prolog-to-Java interfaces. In *Proc. of the International Symposium on Practical Aspects of Declarative Languages (PADL '01)*, I. Ramakrishan and G. Gupta, Eds. Springer-Verlag, Las Vegas, NV, USA.

Ueda, K. 1985. Guarded horn clauses. In *Proc. of the Fourth Conference on Logic Programming, Tokyo, Japan, July 1-3, 1985*, E. Wada, Ed. Lecture Notes in Computer Science, vol. 221. Springer, 168–179.

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

Wadler, P. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73,* 2, 231–248.

Warren, D. H. D. October 1983. An abstract Prolog instruction set. Technical Note 309, SRI International.

Wielemaker, J. 2003. Native preemptive threads in SWI-Prolog. In *Proc. of the International Conference on Logic Programming (ICLP)*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, 331–345.

Zhou, N.-F. 2007. A register-free abstract prolog machine with jumbo instructions. In *Proc. of the International Conference on Logic Programming (ICLP)*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, Berlin, Heidelberg, 455–457.

Zhou, N.-F., Takagi, T. and Ushijima, K. 1990. *A matching tree oriented abstract machine for Prolog*. Logic Programming. MIT Press, Cambridge, MA, USA, 159–173. ISBN: 0-262-73090-1.