Architecture of the Jinni 2000 Runtime System

Paul Tarau

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203
E-mail: tarau@cs.unt.edu

Abstract. We overview the Jinni 2000 continuation passing Java based Prolog system's runtime system and compilation technology.
Keywords: implementation of Prolog, WAM, BinWAM, continuation passing style compilation, data-representations for Prolog run-time systems.

1 Introduction

Jinni 2000 is a Java-emulated Prolog engine based on a program transformation introduced in [10]. It replaces the WAM by a simplified continuation passing logic engine [8] based on a mapping of full Prolog to binary logic programs (binarization). As conventional WAM's environments are discarded in favor of a heap-only run-time system heap garbage collection becomes instrumental both as a means to ensure ability to run large classes of Prolog programs and as a means to improve performance by reducing memory bandwidth.

2 The binarization transformation

We will start by reviewing the program transformation that allows compilation of logic programs towards a simplified WAM specialized for the execution of binary logic programs. We refer the reader to [10] for the original definition of this transformation.

Binary clauses have only one atom in the body (except for some inline 'builtin' operations like arithmetics) and therefore they need no 'return' after a call. A transformation introduced in [10] allows to faithfully represent logic programs with operationally equivalent binary programs.

To keep things simple we will describe our transformations in the case of definite programs. First, we need to modify the well-known description of SLD-resolution [5] to be closer to Prolog's operational semantics. We will follow here the notations of [11].

Let us define the *composition* operator $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument.

Let $\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n$ and $\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m$ be two clauses (suppose $n > 0, m \geq 0$). We define

$(\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n) \oplus (\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m) = (\mathtt{A}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m\mathtt{,A}_2\mathtt{,\ldots,A}_n)\theta$

with $\theta = \mathrm{mgu}(\mathtt{A}_1,\mathtt{B}_0)$. If the atoms $\mathtt{A}_1$ and $\mathtt{B}_0$ do not unify, the result of the composition is denoted as $\perp$. Furthermore, as usual, we consider $\mathtt{A}_0\mathtt{:-true,A}_2\mathtt{,\ldots,A}_n$ to be equivalent to $\mathtt{A}_0\mathtt{:-A}_2\mathtt{,\ldots,A}_n$, and for any clause `C`, $\perp \oplus \mathtt{C} = \mathtt{C} \oplus \perp = \perp$. We assume that at least one operand has been renamed to a variant with fresh variables.

This Prolog-like inference rule is called LD-resolution and it has the advantage of giving a more accurate description of Prolog's operational semantics than SLD-resolution.

Before defining the binarization transformation, we describe two auxiliary transformations.

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

The second transformation, inspired by [14], eliminates the metavariables by wrapping them in a `call/1` goal. E.g., the rule `and(X,Y):-X, Y` is transformed into `and(X,Y) :- call(X), call(Y).`

The transformation of [10] (*binarization*) adds continuations as extra arguments of atoms in a way that preserves also first argument indexing.

Let P be a definite program and *Cont* a new variable. Let T and $E = p(T_1, ..., T_n)$ be two expressions.[1] We denote by $\psi(E,T)$ the expression $p(T_1, ..., T_n, T)$. Starting with the clause

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

we construct the clause

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

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)).
```

These transformations preserve 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.

Note that each resolution step of an LD derivation on a definite program P can be mapped to an SLD-resolution step of the binarized program P'. Let G be an atomic goal and $G' = \psi(G, true)$. Then, computed answers obtained querying P with G are the same as those obtained by querying P' with G'.

Notice that the equivalence between the binary version and the original program can also be explained in terms of fold/unfold transformations as suggested by [7].

[1] Atom or term.

Clearly, continuations become explicit in the binary version of the program. We will devise a technique to access and manipulate them in an intuitive way, by modifying Jinni 2000's binarization preprocessor.

3 Binarization based compilation

3.1 Virtualisation of meta-predicates

Note that the first step of the transformation simply wraps metavariables inside a new predicate `call/1`. The second step adds continuations 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 clauses that describe how they work on the set of all the functors occurring in the program, in practice it is simpler and more efficient to treat them as built-ins [8]. Our current implementation actually performs their execution inline but still has to look up in a hash-table to transform the term to which a meta-variable points, to its corresponding predicate-entry. As this happens only when we reach a 'fact' in the original program, it has relatively little impact on performance. Note however that it can be done in part at compile time, by specializing the source program with respect to some known continuations.

3.2 Inline compilation of built-ins

Demoen and Mariën pointed out in [3] 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 allows inline code generation for built-ins occurring immediately after the head. Inline expansion of builtins contributes signifcantly to Jinni 2000's speed and allows last call optimisation for frequently occurring linear recursive predicates exactly as it happens in conventional WAMs.

3.3 Early term construction vs. late term construction

In procedural and functional languages featuring only deterministic calls it makes sense to avoid eager early structure creation. The WAM [15] follows this trend based on the argument that most logic programs are deterministic and therefore calls and structure creation in logic programming languages should follow this model. A more careful analysis suggests that the choice between

- late and repeated construction (standard WAMs with AND-stack)
- eager early construction (once) and reuse on demand as in BinWAM

will favor different programming styles. Let's note at this point that the WAM's (common sense) assumptions are subject to the following paradox:

- If a program is mostly deterministic then it will tend to fail only in the guards (shallow backtracking). In this case, when a predicate succeeds, all structures specified in the body of a selected clause will eventually get created. By postponing this, the WAM will be only as good as doing it eagerly upon entering the clause (as in BinWAM).
- If the program is mostly nondeterministic then late and repeated construction (WAM) is not better than eager early creation (BinWAM) which is done only once, because it implies more work on backtracking. While the BinWAM will only undo bindings to variables in the binarized body represented on the heap, a conventional WAM will repeatedly push/pop to its environment stack.

This explains in part why (against all obvious intuitions), a standard AND-stack based WAM is not necessarily faster than a properly implemented BinWAM [2].

3.4 A simplified run-time system

A simplified OR-stack having the layout shown in figure 1 is used only for (1-level) choice point creation in nondeterministic predicates.

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. Jinni 2000's OR-stack.

As a consequence, the heap consumption of the program goes up, although in some special cases, partial evaluation at source level can deal with the problem [2, 6], showing again that a heap-only approach is not necessarily worse. As automating these source-level transformations needs global compilation and possibly some help from programmer declared pragmas, we wanted to ensure good performance for our engine without counting on them.

[2] As the section about performance will show.

3.5 A simplified clause selection mechanism

As the compiler works on a clause-by-clause basis, it is the responsibility of the emulator to index clauses and link the code. It uses a global $< key, key > \rightarrow value$ hash table seen as an abstract multipurpose *dictionary*. A one byte mark-field is used to distinguish between load-time use and run-time use and for fast clean-up. We found that sharing of the global dictionary, although somewhat slower than the small $key \rightarrow value$ hashing tables injected into the code-space of the standard WAM, simplifies the implementation and makes it easier to switch to better indexing techniques in the future. The same table is used by the run-time system to get the addresses of meta-predicates, to perform first argument indexing and is offered to the user as entry point to a blackboard containing logically behaving global terms. This high level of code reuse contributes significantly to the small size and indirectly to the overall speed of Jinni 2000.

Predicates are classified as *single-clause*, *deterministic* and *nondeterministic*. At this time only predicates having all first-argument functors distinct are detected as deterministic. Although this is definitely a *RISCy* approach to indexing, we found that for predicates having a more general distribution of first-arguments, a source-to-source transformation can be used. In the future we plan to integrate typical uses of CUT and arithmetic tests in this indexing scheme and extended it with ML-style 'pattern matching' compilation that generates decision trees.

Indexing of deterministic predicates is done by a unique SWITCH instruction. If the first argument dereferences to a non-variable, SWITCH either fails or finds the 1-word address of the unique matching clause in the global hash-table, using the *predicate* and the *functor of the first argument* as a 2-word key. A specialized JUMP-IF instruction deals with the frequent case of 2 clause deterministic predicates. To reduce the interpretation overhead SWITCH and JUMP_IF are combined with the preceding EXECUTE and the following GET_STRUCTURE[3] instruction, giving EXEC_SWITCH and EXEC_JUMP_IF. This allows not only to avoid dereferencing the first argument twice, but also reduces branching that breaks the processor's pipeline. 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.

4 Data representation

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

When describing the data in a cell with a tag we have basically 2 possibilities. We can put a tag in the same cell as the address of the data or near the data 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.

[3] well, also GET_CONSTANT, although in this case it is not worth dealing with it separately

We choose the second possibility initially on purely *aesthetic* grounds while our engine had only 6 instructions. As we became aware that with good indexing unifications are more often intended to succeed and move data around than as a clause selection mechanism, WAM's *preemptive* tag checking lost some of its potential value. This also justifies why we have not implemented traditional WAMs `SWITCH_ON_TAG` instruction.

We found it very convenient to precompute a functor in the code-space as a word of the form `<arity,symbol-number,tag>` and then simply compare it with objects on the heap or in registers at 1-cycle cost instead of comparing the tags, finding out that they are almost always the same, then compare the functor-name and find out that they are also the same and finally compare the arities with a costly if-logic. Therefore we kept our unusual *tag-on-data* representation, that also has the advantage of consuming less tag bits. Up to now, we have only 3 different tags, implemented on 2 bits and still there's a 4-th tag available for future use.

A seemingly not so important (but potentially costly) point is related to the layout of the fields inside the word. We describe it as it helps to compare our implementation with WAMs having a less efficient low-level data representation. We choosed to put most frequently used data at the end of the word, the next frequently used one at the beginning of the word and finally the less frequently used in the middle. Even on byte addressable machines it is a good idea to fetch the whole data in a register and than get the lower bits with a *small* mask and the upper bits with a right shift in 1 cycle. It is less efficient to get the bits at left with a mask, as a huge mask needs two instructions on most RISCs to be loaded in a register.

With this representation a functor fits completely in one word:

arity	symbol-number	2-bit tag

By choosing TAG=0 for variables and having only 2-bit tags, every memory address (C pointer) looks like a logical variable. This gives a very low overhead and less error-prone integration of C code in the engine and it has, on architectures where indexed indirect addressing is not for free, a positive impact on performance.

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 N-th 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.

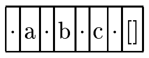

Fig. 2. List compression.

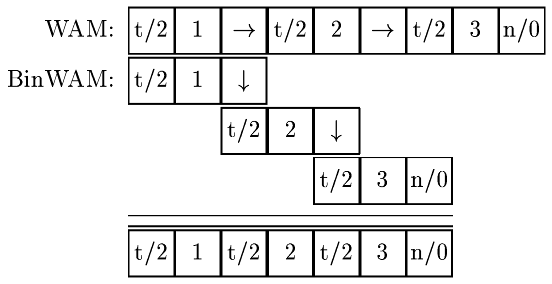

Fig. 3. Term compression.

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 [13] for the details of the term-compression related optimizations of Jinni 2000.

5 Optimizing the run-time system

5.1 Reducing the interpretation overhead

One can argue that the best way to reduce the interpretation overhead is by not doing it at all. However, most of the native code compilers we know of have a *compact-code* option that is actually emulated WAM. And being the most practical in term of compilation-time and code-size, this mode is used by the Prolog developer most of the time, except for the final product. On the other hand, some of the techniques that follow can be used to eliminate tests and jumps so they can be useful also in native code generation.

Instruction-compression and case-overlapping. It happens very often that a sequence of consecutive instructions share some WAM state information. 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. Again, having a small initial instruction set avoids combinatorial explosion in this case. For example, by compressing our UNIFY instructions and their WRITE-mode specializations, we get the following 8 new instructions:

```
UNIFY_VARIABLE_VARIABLE
WRITE_VARIABLE_VARIABLE
UNIFY_VALUE_VALUE
WRITE_VALUE_VALUE
UNIFY_VARIABLE_VALUE
WRITE_VARIABLE_VALUE
UNIFY_VALUE_VARIABLE
WRITE_VALUE_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_VAR_VAR X5, A1
GET_STRUCTURE X3, ./2
UNIFY_VAL_VAR X5, A3
EXEC_JUMP_IF  append/4 % actually the address of append/4
```

The choice of candidates for instruction compression is 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. This justifies the choice of instructions like UNIFY_VARIABLE_VARIABLE.

To emphasize some instruction compression possibilities not so obvious in the case of standard WAM let's show the Jinni 2000 versus SICStus code for the clause:

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

SICSTUS Prolog               Jinni 2000

clause(a/2/1,                a/3:
   [ifshallow                PUT_STRUCTURE    X4<-c/3
   ,neck(2)                  WRITE_VAR_VAL    X5,X2
   ,else                     WRITE_VALUE      X3
   ,endif                    MOVE_REG         X2<-X5
   ,allocate                 MOVE_REG         X3<-X4
   ,get_y_variable(1,1)      EXECUTE          b/3
   ,put_y_variable(0,1)
,init([])                       Jinni 2000 1.71
,call(b/2,2)
,put_y_unsafe_value(0,0)    PUT_WRITE_VAR_VAL  X4<-c/3, X5,X2
,put_y_value(1,1)           WRITE_VALUE        X3
,deallocate                 MOVE_REGx2         X2<-X5, X3<-X4
,execute(c/2)]).            EXECUTE            b/3
```

Clearly, combinatorial explosion due to the elaborate case analysis (safe vs. unsafe, x-variable vs. y-variable) makes the job of advanced instruction compression tedious in the case of standard WAM [4]. Moreover, in the case of an emulated engine just decoding the `init`, `allocate`, `deallocate` and `call` instructions costs more than Jinni 2000's simple `PUT_STRUCTURE` and `WRITE_VAR_VAL` and their straightforward IF-less work on copying 3 heap cells from the registers.

Jinni 2000 also integrates the preceding GET_STRUCTURE instruction into the double UNIFY instructions and the preceding PUT_STRUCTURE into the double WRITE instructions (starting from version 1.71). This gives another 16 instructions but it covers a large majority of uses of GET_STRUCTURE and PUT_STRUCTURE. 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. The impact of this optimization can be seen clearly on the **NREV** benchmark (we refer to the section on performance evaluation).

Moreover, in the case of native code, reordering of Jinni 2000's `PUT_STRUCTURE` + `WRITE` groups can be very useful in slot-filling and more intricate super-scalar processing related instruction scheduling. On the other hand, the presence of environments in conventional WAM limits those reordering optimizations to one chunk. A very straightforward compilation to C [12] and the possibility of optimized 'burst-mode' structure creation in PUT instructions are a direct consequence of binarization and would be harder to apply to AND-stack based traditional WAMs, which exhibit much less uniform instruction patterns.

Other Prologs also do instruction compression, and it is not unusual to hear about engines having 1000 instructions or more. Therefore this optimization is quite common. However, we found out that simplifying the unification instructions of the BinWAM allows for very 'general-purpose' instruction compression. Conventional WAMs often limit this kind of optimization to lists. One can find out about the impact of this by changing the list-constructor of the NREV benchmark. Overall, in a simplified engine instruction compression can be made more 'abstract' and therefore with fewer compressed instructions we can hit a statistically more relevant part of the code. In Jinni 2000, for instance, arithmetic expressions or programs manipulating binary trees will benefit from our compression strategy while this may not be the case with conventional WAMs,

[4] Current implementations of SICStus also do extensive instruction folding. They do need however a much larger set of folded instructions - about 256. This makes emulated code in SICStus 2.1_9 almost as large as native code and about 3-times larger than emulated Jinni 2000 code.

unless they duplicate their (already) baroque list-instruction optimizations for arbitrary structures.

An other point is that instruction compression is usually applied inside a procedure. As Jinni 2000 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. Inter-procedural instruction compression can be seen as a kind of 'hand-crafted' *partial evaluation* at Java level, intended to optimize the main loop of the WAM-emulator. This can be seen as a special case of the *call forwarding* technique used in the implementation of `jc` [4, 1]. It has the same effect as *partial evaluation* at source level which also eliminates procedure calls. At the global level, knowledge about possible continuations can also remove the run-time effort of address look-up for meta-predicates and useless trailing and dereferencing.

Case-overlapping is a well-known technique that saves code-size in the emulator. Note that we can share within the main `switch` statement of the emulator a `case` when an instruction is a specialization of its predecessor, based on the well known property of the `case` statement in C, that in the absence of a `break;` or `continue;` control flows from a `case` label to the next. It is a 0-cost operation. The following example, from our emulator, shows how we can share the code between EXEC_SWITCH and EXEC.

```
case EXEC_SWITCH:
  .........
case SWITCH:
  .........
break;
```

Note that instructions like EXEC_SWITCH or EXEC_JUMP_IF have actually a global compilation flavor as they exploit knowledge about the procedure which is called. Due to the very simple initial instruction set of the Jinni 2000 engine these WAM-level code transformations are performed in C at load-time with no visible penalty on compilation time.

Finally, we can often apply both instruction compression and case-overlapping to further reduce the space requirements. As compressed WRITE-instructions are still just special cases of corresponding compressed UNIFY-instructions we have:

```
case UNIFY_VAR_VAL:
  .........
case WRITE_VAR_VAL:
  .........
break;
```

The benefits of two-stream sequences for free. Notice that for 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 Conclusion

The simplicity Jinni 2000's basic instruction set due to its specialization to binary programs allowed us to apply low level optimizations not easily available on standard WAM. Based on virtualization of demo-predicates at WAM-level [9] that largely eliminate the overhead of metaprogramming introduced by binarization, the Jinni 2000 engine has proven itself as a viable simpler alternative to standard WAM. This is especially true in a Java-emulated WAM's where simplification of the instruction set has a very positive impact on limiting the interpretation overhead.

References

1. K. De Bosschere, S. Debray, D. Gudeman, and S. Kannan. Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 409–420, Portland/USA, Jan. 1994. ACM.
2. B. Demoen. On the Transformation of a Prolog program to a more efficient Binary program. Technical Report 130, K.U.Leuven, Dec. 1990.
3. B. Demoen and A. Mariën. Implementation of Prolog as binary definite Programs. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 165–176, Berlin, Heidelberg, 1992. Springer-Verlag.
4. D. Gudeman, K. De Bosschere, and S. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 399–413, Washington, Nov. 1992. MIT press.
5. J. Lloyd. *Foundations of Logic Programming.* Symbolic computation — Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
6. U. Neumerkel. *Specialization of Prolog Programs with Partially Static Goals and Binarization.* Phd thesis, Technische Universität Wien, 1992.
7. M. Proietti. On the definition of binarization in terms of fold/unfold., June 1994. Personal Communication.
8. P. Tarau. A Simplified Abstract Machine for the Execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.
9. P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.

10. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.
11. P. Tarau and K. De Bosschere. Memoing with Abstract Answers and Delphi Lemmas. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Springer-Verlag, pages 196–209, Louvain-la-Neuve, July 1993.
12. P. Tarau, B. Demoen, and K. De Bosschere. The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog. In K. George, J. Carrol, E. Deaton, D. Oppenheim, and J. Hightower, editors, *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 152–176, Nashville, Feb. 1995. ACM Press.
13. P. Tarau and U. Neumerkel. Compact Representation of Terms and Instructions in the BinWAM. Technical Report 93-3, Dept. d'Informatique, Université de Moncton, Nov. 1993. available by ftp from clement.info.umoncton.ca.
14. D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.
15. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.