

The binary WAM, a simplified Prolog engine

Ulrich Neumerkel

Institut für Computersprachen

Technische Universität Wien

ulrich@mips.complang.tuwien.ac.at

I Existing Abstract Machines for sequential Prolog

II Binary Prolog and the binary WAM

III Source-to-source optimizations for binary Prolog

I

Existing Abstract Machines for sequential Prolog

Abstract Machine defines framework:

- basic interfaces
- instruction set for intermediate code

starting point for optimizations in native code compilation

1972	Prolog 0	Roussel	Algol-W	str. copying, dif
1973	Prolog I	Battani, Meloni	Fortran	str. sharing, 200 LIPS
1977	PLM	Warren	DEC-10	str sharing
1979	—” —	—” —		last call optimization
1982	ZIP	Bowen, Clocksin, Mellish		str. copying
1983	WAM	Warren		str. copying
1986	VAM _{2P}	Krall		str. sharing + copying
1990	BINWAM	Tarau		“WAM-RISC”

Evaluation criteria for Abstract Machines

- Simplicity

- small instruction set
- small implicit state
- simple meta-interpreter for essential architecture
- compact and reasonably efficient emulator

- Efficiency

Programs comparable to procedural programs should run with similar efficiency.

- Level of optimizations (source; intermediate code; machine code)

low level = few optimizations

best: source-to-source level optimization

Compiled versions with (complex) abstract interpretation can improve machine, but inherent problems remain.

Data areas, all machines very similar

AND-control:

local/environment stack

global/copy stack, heap

OR-control:

choice point stack

trail

Traditionally, choice-stack combined with environment stack.

Instruction Formats

Machine	yr.	Operands		Decoding	Implicit operands	control transfer position	instr. removal
		Head	Goal				
PLM	77	2	1	h [g]	none	prefix	no
ZIP	83	1	1	g, h	arg-stack	postfix	yes
WAM	83	2	2	g, h	none	postfix	yes
VAM _{2P}	86	1	1	h+g	none	prefix	no
VAM _{1P}	86	0	2	g	none	prefix	yes

Interface between predicates

1. Determinate interface

PLM: reference to initialized goal.

ZIP: arguments on stack, all arguments initialized.

WAM: register interface, all arguments initialized.

VAM: simultaneous reading of goal and head.

Environment stack **interface between head and body.**

$p(s(1)) \leftarrow$

... .

$\leftarrow \dots, p(s(X)), \dots .$

does not allocate a structure

2. Nondeterminate interface

PLM, ZIP, VAM: choice points of constant size

WAM: choice points contains additionally copy of argument registers, optimization for shallow backtracking required

Format of intermediate code

Important for native code compilation

PLM: only heads can be compiled, goals remain data

ZIP: linear sequence of code, compilable, but many modes

WAM: very easy to compile

full compilation of head-unification doubles code (Demoen-Mariën-Meier 1989)

VAM: compact for intermediary code, but quadratic code for compilation (VAM_{1P})

Handling of terms

- tagging and type tests: minimized in compiled code by abstract interpretation
- single assignment, difficult to overcome (compile time garbage collection, reference counting)

Treatment of logical variables

Often only used to pass parameters.

Parameters should be implemented as in a procedural language:
no trail-checking, no useless initializations, no useless dereferencing

1 Head-variables: parameter passing from head to first goal.

$$\begin{array}{l} p(\dots, HV) \leftarrow \\ \quad q(\dots, HV), \\ \quad \dots \end{array}$$

PLM, ZIP, VAM_{2P}: copy variable from stack to stack
larger space requirements

WAM: no operation, or move variable from register to register
no additional space requirements

Treatment of logical variables

2 Existential, internal variables: parameter passing from one goal to the next.

$$\begin{aligned} p(\dots) \leftarrow \\ & q(\dots, V), \\ & r(\dots, V), \\ & \dots \end{aligned}$$

PLM: V initialized after head p .

- lots of trail-checking/trailing

ZIP, WAM: V initialized before calling q . Implies trail-checking in the head.

Improvement for WAM by Joachim Beer: extra data type uninitialized variable.

VAM: V initialized while unifying goal q with head q . No trailing/trail checking, even if q is nondeterminate.

Treatment of logical variables

3 Last call optimization (TRO) and existential variables.

$p(\dots) \leftarrow$
 $\dots,$
 $q(\dots, V),$
 $r(\dots, V).$

WAM: unsafe variables: Mostly, V is bound when calling r . Otherwise, V is allocated (saved) on the heap.

PLM, ZIP: copying

VAM_{2P}: last call optimization after unifying goal and head, very tricky

Strengths of AMs:

PLM: structure sharing, still used in ATP

WAM: good, as long as registers can be used

- argument registers make variable passing costly
- registers lost after proceed (facts)

VAM: handles variables often as VARs in procedural languages

Missing optimizations

- efficient handling of variables
- flexible calling conventions
- interprocedural state
- leaf procedures

II

Binary Prolog and the binary WAM

Binary Prolog

- subset of full Prolog, only one goal in clauses
- AND-control compiled within terms (continuations)
- cuts implemented with additional parameters
- convenient intermediary language

$p(X, X).$	$p(X, X, \text{Cont}) \leftarrow$
	$\text{Cont}.$
$p(X, Y) \leftarrow$	$p(X, Y, \text{Cont}) \leftarrow$
$q(Y, Z),$	$q(Y, Z, r(Z, X, \text{Cont})).$
$r(Z, X).$	

Binary WAM

Subset of WAM without environments. Similar: Mali, Prolog by BIM
BinProlog by Paul Tarau (Version 2.07)

- C-emulator in 4500 LOC
- 123 instructions. SICStus: $556 = 266 + 266 + 24$
- most builtins inline instructions
- slightly faster than SICStus.
- larger heap consumption

Interesting for compilation:

long sequences of unconditional instructions = single basic block

1. unify instructions
2. builtins
3. **put instructions, create continuations** (basic block)
4. execute

Orthogonal data structures

Implementation of pointers

Classical approach: three different pointer tags

1. Reference for variables and sharing
2. Pointer to structure
3. Pointer to list as “optimization”

BinProlog: only a single pointer tag, no list optimization

1. Reference for variables, sharing, structures, no pointer tag

simplifies implementation:

smaller case analysis

simpler indexing

dereferencing for structures implicit

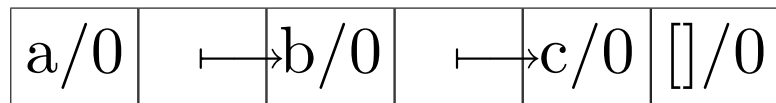
increases memory consumption?

Last argument overlapping

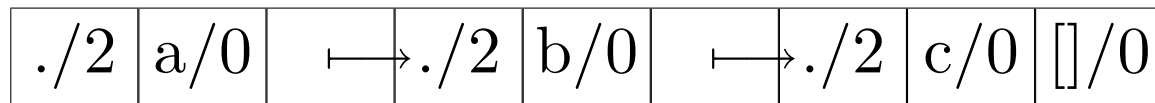
Collapsing references to structures in the last argument

Representation of $[a,b,c]$, n elements:

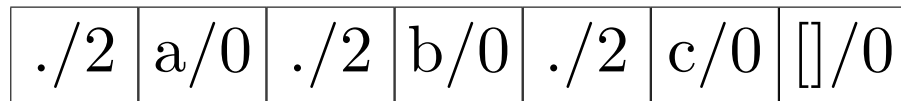
Classical encoding: $2n$ cells



Naïve encoding: $3n$ cells



Last argument overlapping: $2n + 1$ cells



Representation of $s(s(s(0)))$, s^n :

Classical approach = naïve encoding: $2n$ cells

Last argument overlapping: $n + 1$ cells

Minimal adaptations for last argument overlapping:

- write-mode for get_structure instruction:

```
get_structure An:
  deref(An);
  if(VAR(An))
    { trail(An);
      if (An + 1 == H)
        H = An;
      *H++ = functor;
      ...
    }
```

- copy_term/2 Cheney-like copying, combination of depth-first (for last argument) and breadth-first.
- garbage collector
- code-generation for put-structure instructions:
instead of bottom up (from leaves to root) now top down for last arguments

Impacts of last argument overlaps:

- fewer pointers in terms
- fewer dereferencing
- fewer dependencies for writing/reading functors
- compact continuations
- cyclic unification simpler to implement

III

Source-to-source optimizations for binary Prolog

- argument reordering to minimize register moves
- minimizing continuations with auxiliary predicates
- definition of new predicates for sequences of built-ins
- minimizing size of choice points