Warren's Abstract Machine
A Java Implementation

Prolog Compiler and WAM Documentation

Stefan Büttcher
Sieglitzhofer Str. 19, 91054 Erlangen

<stefan@buettcher.org>

February 20, 2002

Contents

This document, as well as all Java sources belonging to my WAM implementation, can be found at http://stefan.buettcher.org/cs/wam/

1 Introduction and Overview

In 1983, David H. D. Warren presented an abstract architecture, well-suited to run Prolog programs on. According to Warren, the actual Prolog code is supposed to be transformed (or compiled) to an (abstract) meta-code, which then is to be executed on an abstract, virtual machine. Today, most publicly available Prolog compilers use the architecture suggested by Warren or at least a derivate of it.

In the course of the exercises accompanying the lecture "Artificial Intelligence I", taught by H. Stoyan at the FAU in winter term 2001/2002, I have developed a Java implementation of this WAM model, which is able to solve most of the tasks today's commercial Prolog implementations can cope with.

The main ideas during the development of my implementation of a WAM have been taken from Ait Kaci: *Warren's Abstract Machine - A Tutorial Reconstruction.* In contrast to Kaci, however, I decided not to design a too-fast architecture, but instead changed some operations for reasons of simplicity, e.g. my WAM does not make any use of the (global) X variable registers or the heap-based list and struc operations proposed by Kaci. Instead of optimizing the code by using the X's, it makes only use of the (local) Y registers. List and structure operations have been realized by two simple operations `unify_list` and `unify_struc`. I will explain this later.

Throughout this paper, I will assume that the reader has already gained some experience regarding the programming language Prolog, i.e. has written a few programs and has understood the idea behind unification and Prolog's backtracking.

2 Prolog/WAM Basics

A Prolog program is a logical program consisting of a sequence of *Horn Clauses.* One example of such a clause is

$$grandfather(X,\ Y)\ :-\ father(X,\ Z),\ father(Z,\ Y).$$

which is read "If X is the father of Z and Z is the father of Y, then X is the grandfather of Y." Another example is

$$member(X,\ [X]). \tag{1}$$

$$member(X,\ [Y|Z])\ :-\ member(X, Z). \tag{2}$$

which is read "X is a member of a list L if the list only consists of X (1) or if X is a member of the list that results from removing L's first element (2)." Obviously, this is a recursive statement.

From the compiler's point of view, WAM code is actual machine code, only that the machine is an abstract machine. So, a given Prolog program is ordinarily transformed to the (or a) corresponding WAM-code statement sequence.

male(john).

```
male:  try_me_else male2
get_constant john A1
proceed
```

male(william).

```
male2: retry_me_else male3
get_constant william A1
proceed
```

male(george).

```
male3: trust_me
get_constant george A1
proceed
```

Example 1: A Prolog program and the resulting WAM code.

The idea now becomes obvious: The WAM first tries to unify $A1$ (the 1st argument register) with the constant *john* by binding $A1$ to the constant. If successful, it proceeds. Otherwise, e.g. if $A1$ has already been bound to a different constant, backtracking is started and the WAM tries the second possibility, namely to bind $A1$ to *william*. If the third try is still unsuccessful, the WAM returns "fail" or tries backtracking in a higher layer. This behaviour is indicated by the commands *try_me_else*, *retry_me_else* and *trust_me*.

ancestor(X, Y) :- parent(X, Y), !.

```
ancestor:  try_me_else ancestor2
allocate
get_level Y1
call parent
cut Y1
deallocate
proceed
```

ancestor(X, Y) :- parent(Z, Y), ancestor(X, Z).

```
ancestor2:  trust_me
allocate
get_variable Y1 A1
get_variable Y2 A2
put_variable Y3 A1
put_value Y2 A2
call parent
put_value Y1 A1
put_value Y3 A2
call ancestor
deallocate
proceed
```

Example 2: A Prolog program and the resulting WAM code.

`allocate` and `deallocate` are used to create (or dispose, respectively) a local environment with local `Yn` variables.

More examples of Prolog programs transformed to WAM code will be shown in the section *Examples*.

3 The Prolog sub-language supported by this implementation

3.1 Syntactic elements

The syntax is shown in EBNF, terminal symbols are indicated by ′...′, non-terminal symbols are surrounded by <...> .

The initial symbol to start off with is <Program>.

- **<Program>** ::=
  <Clause> | <Clause> <Program>
- **<Clause>** ::=
  <Head> ’.’ | <Head> ’:-’ <Body> ’.’
- **<Head>** ::=
  <Predicate> | <Predicate> ’(’ <List> ’)’
- **<Body>** ::=
  <Condition> | <Condition> ’,’ <Body>
- **<Condition>** ::=
  <Predicate> | <Predicate> ’(’ <List> ’)’ | ’not’ <Predicate> |
  ’not’ <Predicate> ’(’ <List> ’)’ | <Variable> ’is’ <Expression> |
  <Element> <Comparator> <Element> | ’!’
- **<List>** ::=
  <Element> | <Element> ’,’ <List>
- **<Element>** ::=
  <Variable> | <Constant> | <Structure> | ’[’ <List> ’]’ |
  ’[’ <List> ’|’ <Variable> ’]’
- **<Structure>** ::=
  <Predicate> ’(’ <List> ’)’ | <Variable> ’(’ <List> ’)’
- **<Comparator>** ::=
  ’¯’ | ’! =’ | ’<’ | ’<=’ | ’=’ | ’>=’ | ’>’
- **<LowerAlpha>** ::=
  ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ |
  ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’
- **<UpperAlpha>** ::=
  ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’ |
  ’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’ | ’Y’ | ’Z’
- **<Figure>** ::=
  ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’
- **<Number>** ::=
  <Figure> <Figure>*
- **<AlphaNum>** ::=
  <LowerAlpha> | <UpperAlpha> | <Figure> | ’_’

- <**Predicate**> ::=
  <LowerAlpha> <AlphaNum>*
- <**Constant**> ::=
  <LowerAlpha> <AlphaNum>* | <Number> | '[]'
- <**Variable**> ::=
  <UpperAlpha> <AlphaNum>* | '_'

3.2 Built-in predicates

This is the list of built-in predicates with their respective arity and a description of their functionality.

- **assert**/1, **assertz**/1 Asserts a new fact at the end of the current database (program).

  Example. `assert(loves(onan, onan))`
- **atomic**/1 Succeeds if the variable references by the argument has been bound to an atomic entity, i.e. a simple constant, not a list and not a structure.
- **bound**/1 Succeeds if the variable referenced by the argument has been bound. If it has not been bound yet, execution fails.
- **call**/1 Used for dynamic calling where the name of the procedure to be called is unknown at development time.

  Example. `X = equal, call(X(Y, onan))` would result in $X = equal$, $Y = onan$, if `equal` has been defined before calling it.
- **consult**/1 Compiles a new program (indicated by the argument) and adds the resulting WAM code to the program currently in memory.
- **integer**/1 Succeeds if the argument is or references an integer value. Unbound variables or non-integers let the execution fail.
- **load**/1 Loads an already-compiled or hand-written WAM program into memory by adding it to the current program.
- **nl**/0, **newline**/0 Writes CRLF to `stdout`.
- **readln**/0 Reads a line from `stdin` and stores it in the argument variable.
- **retract**/1, **retractone**/1 Retracts one fact from the current database. Attention: The syntax is a little different from the usual one.

  Example. `assert(loves(onan, onan)), retractone(loves))` leaves the program unchanged.
- **retractall**/1 Retracts all clauses with the name given by the argument from the database.

  Example. `assert(loves(onan, onan)), loves(joker, virgin_mary), retractall(loves))`
- **write**/1 Writes the contents of the argument register to `stdout`.

- **writeln/1** Writes the contents of the argument register to `stdout` and starts a new line.

  Examples. `X = hello, writeln(X), writeln('Hello, world!')`

4 WAM code operations supported by this implementation

This implementation of Warren's Abstract Machine supports the following (WAM code) operations:

- **allocate** Allocates a new environment structure on the stack, consisting of a set of local (Y_i) variables and a copy of the current return address *continuation pointer*.

- **bigger** V_i V_j, **biggereq** V_i V_j, **smaller** V_i V_j, **smallereq** V_i V_j

- **call** *label* Sets the WAM's program counter (PC) to the line indicated by *label* or executes the corresponding internal predicate. Fails, if neither can be found.

- **create_variable** Q_i *varname* Is used for creating query variables that need to have a name.

  Example. The Prolog query `male(X).` would result in the WAM code

```
query$:  trust_me
create_variable Q0 X
put_value Q0 A0
call male
halt
```

- **cut** V_i Sets the last choicepoint, which will be used in case of backtracking, to the predecessor of that saved in variable V_i. If there is no predecessor, it sets the choicepoint to `null`.

- **get_constant** *constant* A_i Used in a clause's head in order to unify (i.e. bind) the argument variable A_i with the value *constant*.

- **get_level** V_i Stores the top element of the current choicepoint stack in variable V_i.

  Example. The Prolog code `faculty(X, 0) :- X < 0, !.` would result in the WAM code

```
faculty:  trust_me
allocate
get_variable Y0 A0
get_constant 0 A1
get_level Y2
put_constant 0 Y1
smaller Y0 Y1
cut Y2 deallocate
proceed
```

- **get_value** Y_i A_j Is used at subsequent occurrences of a local variable inside the clause's head, since A_j may not be simply copied into Y_i then but they must be unified.

  Example. The Prolog program `equal(X, X).` would result in the WAM code

  ```
  equal:  trust_me
  allocate
  get_variable Y0 A0
  get_value Y0 A1
  deallocate
  proceed
  ```

- **get_variable** Y_i A_j Stores the value of A_j in Y_i. This is used at the first occurrence of a local variable inside the clause's head.

- **halt**   Halts the execution.

- *is* V_i *operator* V_j V_k

- **noop**, **nop**   Does nothing.

- **proceed**   Sets the program counter (PC) to the value of the continuation pointer (return address, CP).

- **put_constant** *constant* A_i Binds the argument variable A_i to the value *constant*.

- **put_value** V_i A_j Copies the contents of variable V_i into A_j.

- **put_variable** V_i A_j Sets A_j's tag to *REF* (reference) and lets it point to Y_i.

- **trust_me**   This statement makes the WAM effectively do nothing. It does, however, indicate that this clause is the last one corresponding to a certain procedure name. This is necessary for asserting and retracting.

  Example. The Prolog program `green(apple).  green(grass).` would result in the WAM code

  ```
  green:  try_me_else green~2
  get_constant apple A0
  proceed
  green~2:  trust_me
  get_constant grass A0
  proceed
  ```

- **try_me_else** label, **retry_me_else** label   My WAM makes no difference between these two statements. The suggestions made by Kaci were only made for reasons of performance that seem quite senseless when applied to the kind of architecture I chose for implementing the WAM.

- **unify_list** V_i V_j V_k Here, V_i references the variable containing a list that shall be unified with a head V_j and a tail V_k. The effect of this operation can be explained best by an illustrative

  Example. The Prolog program `member(X, [X]).` would result in the WAM code

  ```
  member:  trust_me
  allocate
  get_variable Y0 A0
  ```

```
get_variable Y1 A1
put_constant [] Y2
unify_list Y3 Y0 Y2
unify_variable Y1 Y3
deallocate
proceed
```

- **unify_struc** V_i V_j V_k Is the structure-based pendant to `unify_list`. V_i is the structure, V_j its head (the part before the opening bracket) and V_k the tail (the part between the brackets).
- **unify_variable** V_i V_j Unifies the two variables referenced by V_i and V_j. In case of lists and structures, this can lead to recursive calls via `unify_list` and `unify_struc`.
- **deallocate** Frees the memory in use by the environment structure and restores the old continuation pointer (CP).

5 Class structure of this WAM implementation

- **WAM.java**
  - `public class WAM`
    This is the main class of the Abstract Machine. It contains the `main(String[])` method, the methods responsible for the WAM's code operations and all the built-in predicates. The `main` method constructs a new WAM object and then loops subsequent `wam.runQuery(String)` calls until the user quits the program.
  - `class WAM.Variable`
    The `Variable` class implements the WAM tagged variable type. WAM Variables can be of four different types:
    * `UNB`: The variable has been instantiated but not yet been bound to a value or another variable.
    * `REF`: The variable points to another variable, which may be bound or unbound.
    * `CON`: The variable has been bound to a constant.
    * `LIS`: The variable has been bound to a list structure.
    * `STR`: The variable has been bound to a term structure.

    Please note that in the current implementation, unbound variables are not represented by the `UNB` tag but instead by the `REF` tag with an internal reference pointing to itself (i.e. self-referencing = unbound).
  - `class WAM.ChoicePoint`
    An essential element in any WAM implementation is the ChoicePoint structure. If there are two different bodies belongig to the same procedure (e.g. `parent(anne, tom).` and `parent(john, tom).`, a choice point has to be created when executing the first possibility so that the WAM knows where to go next if the first one fails. When creating a new ChoicePoint instance, the current `arguments` vector, trail pointer, return address (continuation pointer) and local environment (see

WAM.Environment) are saved. Any following backtracking process will restore the data. Like Enviroments, ChoicePoint objects are organized in a stack-like fashion.

- class WAM.Environment
  Since there are local variables in most Prolog programs, we have to manage their organization. This implementation completely forgets about global (X) variables. Instead, only local (Y) variables are used. An Environment object stores the local variable vector and the current return address so that the return pointer will not be overridden by subsequent procedure calls. Environments are organized in a stack-like fashion, where the last element points to its predecessor and so forth.
- class WAM.Trail
  The Trail keeps track of any binding operations performed. This is necessary because in Prolog it is inevitable to backtrack. So, imagine the following situation: We have the Prolog program wears_brown(X) :- is_male(X), plays_piano(X). is_male(stefan). is_male(dominik). plays_piano(dominik). Then, we execute the query wears_brown(X)., which first will bind X to stefan and then recognize that stefan does not play piano. So, it needs to backtrack and remove the binding of the variable X to the constant value stefan. That is what the trail is good for.

- **Program.java**
  This file contains the public class Program, which essentially is a structure for storing a vector of WAM code statements. Additionally, it contains the TreeMap, mapping from label names to code lines, for faster jumping and some sugar.
- **Statement.java**
  This file contains the public class Statement, which represents a single line of WAM code. The Statement class is responsible for the translation of String-type WAM statements to a better readable form, e.g. integer constants for code operations, such as get_variable.
- **Compiler.java, PrologCompiler.java, QueryCompiler.java**
  These three files are responsible for compiling Prolog programs and runtime queries. There is nothing interesting about them. The Compiler class performs the actual parsing and code generation, using a syntax tree and built-in methods like procedure, body and list, while PrologCompiler and QueryCompiler do little more than calling the respective methods of a Compiler object.

6 Example Prolog programs with corresponding WAM code

In order to make it a little bit easier for you to understand the relationship between a Prolog program and the resulting WAM code, I am presenting some compilation examples. Besides: When starting to write my own WAM, I looked on the web for some sample WAM code, but I really couldn't find too much information on that topic. That is another reason for including this section.

```
male(john).                  male:  try_me_else male2
                             get_constant john A0
                             proceed

male(thomas).                male2:  retry_me_else male3
                             get_constant thomas A0
                             proceed

male(william).               male3:  retry_me_else male4
                             get_constant william A0
                             proceed

male(james).                 male4:  trust_me
                             get_constant james A0
                             proceed

female(X) :-                 female:  trust_me
  not(male(X)).              allocate
                             get_variable Y0 A0
                             put_constant male Y1
                             put_constant [] Y2
                             unify_list Y3 Y0 Y2
                             unify_struc Y4 Y1 Y3
                             put_value Y4 A0
                             call not
                             deallocate
                             proceed

not(Call) :-                 not:  try_me_else not2
  call(Call), !, fail.       allocate
                             get_variable Y0 A0
                             get_level Y1
                             put_value Y0 A0
                             call call
                             cut Y1
                             call fail
                             deallocate
                             proceed

not(Call).                   not2:  trust_me
                             proceed

equal(X, X).                 equal:  trust_me
                             allocate
                             get_variable Y0 A0
                             get_value Y0 A1
                             deallocate
                             proceed

father(william, thomas).     father:  try_me_else father2
                             get_constant william A0
                             get_constant thomas A1
                             proceed
```

```
father(william, sue).        father2:  retry_me_else father3
                             get_constant william A0
                             get_constant sue A1
                             proceed

father(john, william).       father3:  retry_me_else father4
                             get_constant john A0
                             get_constant william A1
                             proceed

father(james, anne).         father4:  trust_me
                             get_constant james A0
                             get_constant anne A1
                             proceed

mother(anne, thomas).        mother:  try_me_else mother2
                             get_constant anne A0
                             get_constant thomas A1
                             proceed

mother(anne, sue).           mother2:  retry_me_else mother3
                             get_constant anne A0
                             get_constant sue A1
                             proceed

mother(jeanne, william).     mother3:  retry_me_else mother4
                             get_constant jeanne A0
                             get_constant william A1
                             proceed

mother(denise, anne).        mother4:  trust_me
                             get_constant denise A0
                             get_constant anne A1
                             proceed

parent(X, Y) :-              parent:  try_me_else parent2
  father(X, Y).              allocate
                             get_variable Y0 A0
                             get_variable Y1 A1
                             put_value Y0 A0
                             put_value Y1 A1
                             call father
                             deallocate
                             proceed

parent(X, Y) :-              parent2:  trust_me
  mother(X, Y).              allocate
                             get_variable Y0 A0
                             get_variable Y1 A1
                             put_value Y0 A0
                             put_value Y1 A1
                             call mother
                             deallocate
                             proceed
```

```
grandparent(X, Y) :-
  parent(X, Z), parent(Z, Y).
```

```
grandparent:  trust_me
allocate
get_variable Y0 A0
get_variable Y1 A1
put_value Y0 A0
put_value Y2 A1
call parent
put_value Y2 A0
put_value Y1 A1
call parent
deallocate
proceed
```

```
grandfather(X, Y) :-
  grandparent(X, Y), male(X).
```

```
grandfather:  trust_me
allocate
get_variable Y0 A0
get_variable Y1 A1
call grandparent
put_value Y0 A0
call male
deallocate
proceed
```

```
grandmother(X, Y) :-
  grandparent(X, Y), female(X).
```

```
grandmother:  trust_me
allocate
get_variable Y0 A0
get_variable Y1 A1
call grandparent
put_value Y0 A0
call female
deallocate
proceed
```

```
brother(X, Y) :-
  male(X), parent(Z, X),
  parent(Z, Y), not(equal(X, Y)).
```

```
brother:  trust_me
allocate
get_variable Y0 A0
get_variable Y1 A1
call male
put_value Y2 A0
put_value Y0 A1
call parent
put_value Y2 A0
put_value Y1 A1
call parent
put_constant equal Y3
put_constant [] Y4
unify_list Y5 Y1 Y4
unify_list Y6 Y0 Y5
unify_struc Y7 Y3 Y6
put_value Y7 A0
call not
deallocate
proceed
```