# Design and Implementation of the GNU Prolog System


**Daniel Diaz**
University of Paris 1
CRI, bureau C1407
90, rue de Tolbiac
75013 Paris, FRANCE
and INRIA-Rocquencourt
Daniel.Diaz@inria.fr

**Philippe Codognet**
University of Paris 6
LIP6, case 169
8, rue du Capitaine Scott
75015 Paris, FRANCE
and INRIA-Rocquencourt
Philippe.Codognet@lip6.fr



## Abstract

In this paper we describe the design and the implementation of the GNU Prolog system. This system draws on our previous experience of compiling Prolog to C in the `wamcc` system and of compiling finite domain constraints in the `clp(FD)` system. The compilation scheme has however been redesigned in order to overcome the drawbacks of compiling to C. In particular, GNU-Prolog is based on a low-level mini-assembly platform-independent language that makes it possible to avoid compiling C code, and thus drastically reduces compilation time. It also makes it possible to produce small stand-alone executable files as the result of the compilation process. Interestingly, GNU Prolog is now compliant to the ISO standard, includes several extensions (OS interface, sockets, global variables, etc) and integrates a powerful constraint solver over finite domains. The system is efficient and in terms of performance is comparable with commercial systems for both the Prolog and constraint aspects.


## 1 Introduction

GNU Prolog is a free Prolog compiler supported by the GNU organization (`http://www.gnu.org/software/prolog`). It is a complete system which includes: floating point numbers, streams, dynamic code, DCG, operating system interface, sockets, a Prolog debugger, a low-level WAM debugger, line editing facilities with completion on atoms, etc. GNU Prolog offers more than 300 Prolog built-in predicates and is compliant to the ISO standard for Prolog

[7] (GNU Prolog is today the only free Prolog system that is really compliant to this standard). There is also a powerful bidirectional interface between Prolog and C, featuring implicit Prolog $\leftrightarrow$ C type conversion, transparent I/O argument handling, non-deterministic C code, ISO error support, etc. Such an interface allows users to easily write their own extensions. GNU Prolog also includes an efficient constraint solver over finite domains (FD), similar to that of the `clp(FD)`, described in [5, 6]. The key feature of such a solver is the use of a simple (low-level) language that embeds the core propagation mechanism in order to define all (high-level) FD constraints. There are many advantages to this approach: constraints can be compiled, the user can define his own constraints (thanks to the definition language), the solver is open and extensible (as opposed to black-box solvers like the CHIP system by Cosytec). The solver does, however, already include many predefined constraints: arithmetic constraints, boolean constraints, symbolic constraints, reified constraints; there are more than 50 FD built-in constraints/predicates, and several predefined labeling heuristics.

The development of GNU Prolog started in January 1996 under the name of Calypso. Discussions with the GNU organization started in late 1998, and the first version was released in April 1999 as a GNU product. Two years later more than 25,000 copies have been downloaded from the INRIA ftp site (there are no statistics available for the main GNU ftp site nor for mirror sites) and GNU Prolog is now included in most commercial Linux distributions (Mandrake, Debian, etc).

GNU Prolog stems from our previous work on `wamcc` which was based on the idea of translating Prolog to C [4]. The novelty of the GNU Prolog compilation scheme is that it uses a mini-assembly (MA) language to avoid the intermediate step of compiling to C. This language has been specifically designed for GNU Prolog. The idea of the MA is then to have a machine-independent intermediate language in which Prolog is translated via the WAM (Warren Abstract Machine), the standard high-level abstract machine for Prolog compilation. The corresponding MA code is mapped to the assembly language of the target machine. The MA language is based on a reduced instruction set in order to simplify porting over various architectures. This new compilation process is from 5 to 10 times faster than `wamcc`+`gcc`.

The rest of this paper is organized as follows. Section 2 is devoted to the compilation scheme. Section 3 recalls the basic compilation from Prolog to the WAM. Section 4 introduces the MA language while Section 5 describes how this language can be mapped to a specific architecture. Section 6 is devoted to the link phase and Section 7 to memory management. Performance evaluation

of the Prolog engine is detailed in Section 8. The constraint solver and its performance are presented in Section 9. A short conclusion ends the paper.

# 2 General compilation scheme

## 2.1 Background

Traditionally, compilers for imperative, functional or logical languages decompose the compilation process into several steps and in particular use an abstract machine as an intermediate level between the high-level source code and the target low-level executable code. Indeed, since Pascal and the P-code, abstract machines have been highlighted as the backbone of the compilation process.

Logical languages are no exception and the compilation of Prolog to the WAM (Warren Abstract Machine [14]) is a de facto standard and well-known process. However, WAM code cannot be executed directly on mainstream computers and therefore requires some treatment in order to become executable. Classical techniques consist either in executing directly the WAM code with an emulator written in C (the original version of SICStus Prolog [2]) or assembler (Quintus Prolog) or directly compiling to native code (Prolog by BIM, latest version of SICStus Prolog, Aquarius Prolog [13]). Another approach consists in translating Prolog to C, a choice made to keep the implementation simple without penalizing too greatly its performance. The `wamcc` Prolog compiler was based on this approach, but also incorporated the idea of translating a WAM branching into a native code jump in order to reduce the overhead of calling a C function, see [4] for details. There is however a serious drawback to this approach, which is the size of the C file generated and the time taken to compile such a large program by standard C compilers (e.g. `gcc`). Indeed, a Prolog program produces a large number of WAM instructions (e.g. the 3,263 lines of `pl2wam`, the Prolog to WAM compiler, gives rise to 12,077 WAM instructions) and trying to inline each WAM instruction could lead to a very big C file that could not be handled by the C compiler. In order to cope with large Prolog source we decided, in `wamcc`, to translate most WAM instructions to a call to a C function which perfors the treatment. Obviously the execution is a little slower but the compilation is much faster (and the executable is smaller). However, even with this solution, the C compiler took too much time for large source files, especially in trying to optimize the code produced.

The main improvement to the GNU Prolog compilation scheme is to translate a

WAM file into a mini-assembly language which is machine-independent. The corresponding MA code is mapped to the assembly language of the target machine. In order to simplify the writing (i.e. porting) of such translators the instruction set of MA must be simple, in the spirit of the LLM3 abstract machine for Lisp [3], as opposed to the complex instruction set of the BAM designed for Aquarius Prolog [13]. Actually, the MA language is based on 11 instructions, most of which handle the control of Prolog and the invocation of C functions.

## 2.2 Overview

Native compilation is handled by `gplc`, the GNU Prolog command-line compiler which invokes several sub-compilers to produce an executable from a Prolog source. Classically the source file is used to obtain an object which is linked to the GNU Prolog libraries to produce an executable. To do so, the Prolog source is first compiled to obtain a WAM file. The WAM file is translated to an MA file. This file is then mapped to the assembly language of the target machine which next gives rise to an object. All objects are linked with the GNU Prolog libraries to provide an executable. The compiler also takes into account finite domain constraint definition files. It translates them to C and invokes the C compiler to obtain object files, as explained below. Figure 1 presents this compilation scheme.

Obviously all intermediate stages are hidden to the user who simply invokes the compiler on his Prolog file(s) (plus other files: C foreign file, FD files, etc) and obtains an executable. However, it is also possible to stop the compiler at any given stage, which may be useful, for instance, to view the WAM code produced. Finally it is possible to give any kind of file to the compiler which will insert it in the compilation chain at the stage corresponding to its type. The type of a file is determined using the suffix of its file name. The following table presents all recognized types/suffixes:

| Suffix of the file | Type of the file | Handled by: |
|---|---|---|
| `.pl`, `.pro` | Prolog source file | `pl2wam` |
| `.wam` | WAM source file | `wam2ma` |
| `.ma` | Mini-assembly source file | `ma2asm` |
| `.s` | Assembly source file | assembler |
| `.c`, `.C`, `.c++`, `.cpp`,... | C or C++ source file | C compiler |
| `.fd` | FD constraint source file | `fd2c` |
| other file (`.o`, `.a`,...) | any other type (object, library,...) | linker |

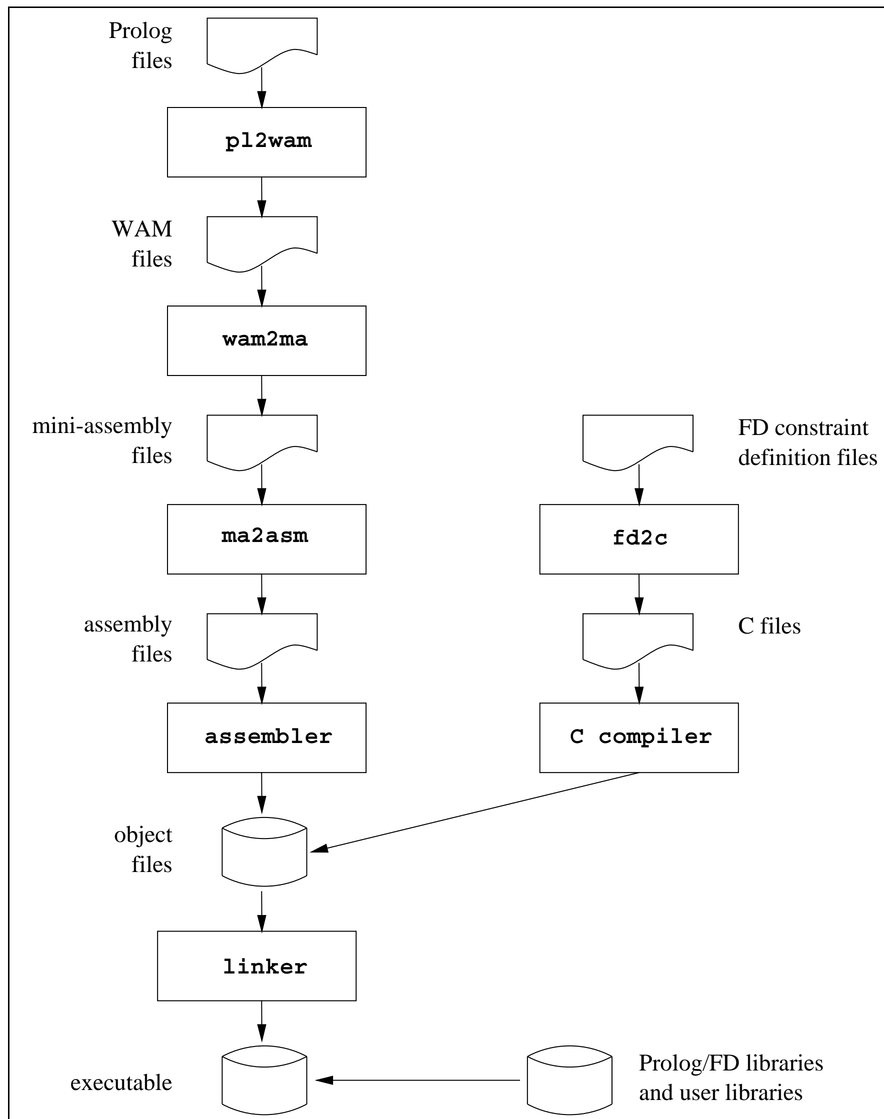


Figure 1: compilation scheme

# 3 From Prolog to the WAM

The `pl2wam` sub-compiler accepts a Prolog source and produces a WAM file. For more information about the WAM see [14, 1]. We will not describe in great detail this compiler as a lot of information is available in the literature.

This compiler is fully written in GNU Prolog (3,263 lines of Prolog code) and bootstrapped. It compiles each clause in several passes:

- the clause is simplified: control constructs like disjunctions, if-then or cut are rewritten (giving rise to auxiliary predicates if needed).
- the clause is translated into a more practical internal format. Variables are classified as being permanent or temporary. Permanent variables are assigned.
- the WAM code associated to the clause is generated.
- registers (temporary variables) are assigned and optimized.

The code of all the clauses of a predicate are grouped and the indexing code is generated. For further details the reader can refer to [2].

The WAM file produced by `pl2wam` is a text file complying to the Prolog syntax. This makes it possible to provide a WAM file (produced by another Prolog to WAM compiler using the appropriate syntax) to `gplc` but also to pass the WAM file to any emulator, e.g. a system written in Prolog.

This compiler benefits from many (well-known) optimizations: register optimization, unification reordering, inlining, last subterm optimization, etc. Most of these optimizations can be deactivated using command-line options. Deactivating all the options makes it possible to study the basic Prolog to WAM compilation process.

The following example shows the compilation of the clause `p(T,g(U),V):- q(a,T,V,U)` without any optimization:

```
% gplc -W --no-reorder --no-reg-opt t.pl
% cat t.wam
file_name('t.pl').                    % current file name: t.pl
```

```
predicate(p/3,1,static,private,user,[  % p/3 defined at line #1
    get_variable(x(5),0),              % p(T,
    get_structure(g/1,1),              %     g(
    unify_variable(x(4)),              %       U),
    get_variable(x(3),2),              %          V):-
    put_atom(a,0),                     % q(a,
    put_value(x(5),1),                 %     T,
    put_value(x(3),2),                 %       V,
    put_value(x(4),3),                 %         U).
    execute(q/4)]).
```

The WAM code associated to a predicate is given as a Prolog list. Some other information is present such as the file name (`t.pl`) and the line number (`1`) where `p/3` is defined (this information could be exploited by a graphical editor). The properties of `p/3` are also specified: `static` (as opposed to `dynamic`), `private` (as opposed to `public`) and `user` (to distinguish between user-defined and built-in predicates).

We present below the same example without unification reordering but with register optimization and keeping void instructions [1] in the WAM file:

```
% gplc -W --no-reorder --keep-void-inst t.pl
% cat t.wam
file_name('t.pl').

predicate(p/3,1,static,private,user,[
    get_variable(x(4),0),
    get_structure(g/1,1),
    unify_variable(x(3)),
    get_variable(x(2),2),
    put_atom(a,0),
    put_value(x(4),1),
    put_value(x(2),2),
    put_value(x(3),3),
    execute(q/4)]).
```

Now we show the same example fully optimized, with register optimization and unification reordering:

[1] A void instruction is a copy instruction where the source register is the the same as the destination.

```
% gplc -W t.pl
% cat t.wam
file_name('t.pl').

predicate(p/3,1,static,private,user,[
    get_structure(g/1,1),
    unify_variable(x(3)),
    put_value(x(0),1),
    put_atom(a,0),
    execute(q/4)]).
```

# 4 The mini-assembly language

## 4.1 Overview

The idea behind the mini-assembly (MA) language is to have a machine-independent intermediate language in which the WAM is translated. The design of the MA comes from the study of the C code produced by `wamcc`. Indeed, in the `wamcc` system, most WAM instructions gave rise to a call to a C function which carried out the treatment. The only exceptions were obviously instructions to manage the control of Prolog and some short instructions that were inlined. The MA language has been designed to avoid the use of the C stage and therefore provides instructions to handle the Prolog control, to call a C function and to test/use its returned value. The MA file is then mapped to the assembly of the target machine (see Section 5) from which an object is produced.

In order to simplify the writing of the mappers responsible for translating MA code, the MA instruction set must be simple: it only contains 11 instructions.

## 4.2 The MA instruction set

Here is a description of each MA instruction:

`pl_jump` ***`pl_label`*** **:** branch the execution to the predicate whose corresponding symbol is ***`pl_label`*** . This symbol is an identifier whose construction

is explained in Section 4.3. This instruction corresponds to the WAM instruction `execute`.

`pl_call` ***`pl_label`*** **:** branch the execution to the predicate whose corresponding symbol is ***`pl_label`*** after initializing the continuation register `CP` to the address of the very next instruction. This instruction corresponds to the WAM instruction `call`.

`pl_ret`**:** branch the execution to the address given by the continuation pointer `CP`. This instruction corresponds to the WAM instruction `proceed`.

`pl_fail`**:** branch the execution to the address given by the last alternative (`ALT` cell of the last choice point pointed by the WAM `B` register). This instruction corresponds to the WAM instruction `fail`.

`jump` ***`label`*** **:** branch the execution to the symbol ***`label`***, which could be any internal label. This instruction is used when translating WAM indexing instructions (e.g. `try`, `retry` or `trust`) to perform local control transfer, i.e. branching within the same predicate. This instruction has been distinguished from `pl_jump` (even if both can be implemented/translated in a similar manner) since, on some machines, local jumps can be optimized.

`call_c` ***`fct_name`*** `(`***`arg`*** `,...)`**:** call the C function ***`fct_name`*** passing the arguments ***`arg`***,... Each argument can be an integer, a floating point number (C `double`), a string, the address of a label, the address or the content of a memory location, the address or the content of a WAM `X` or `Y` register. This instruction is used to translate most WAM instructions.

`fail_ret`**:** perform a Prolog fail (like `pl_fail`) if the value returned by the previous C function call is 0. This instruction is used after a C function call returning a boolean to indicate its issue (e.g. functions performing the unification).

`jump_ret`**:** branch the execution to the address returned by the previous C function call. This instruction makes it possible to use C functions to determine where to transfer the control to. For instance, the WAM indexing instruction `switch_on_term` is implemented via a C function accepting several addresses and returning the address of the selected code.

`move_ret` ***`target`*** **:** copy the value returned by the previous C function call in ***`target`*** which can be either a memory location or a WAM `X` or `Y` register.

`c_ret`**:** C return. This instruction is used at the end of the initialization function (see below) to give the control back to the caller.

`move` *`reg1`*`,`*`reg2`*`:` copy the content of the WAM `X` or `Y` register *`reg1`* in the register *`reg2`*.

It is worth noticing the minimality of the language which is based on a very restricted instruction set. Note however the presence of the `move` instruction to perform a copy of WAM X or Y registers. We could instead invoke a C function to perform such a move (using `call_c`). However, those moves between registers are rather frequent and the invocation of a C function would be costly in terms of execution time. Thus a trade off must be found between the minimality of the instruction set and the performance. Obviously, it would be possible to extend this instruction set (e.g. adding arithmetic instructions) but this would make the writing of the mappers much more complicated.

In addition to the above instructions, the MA language also includes several declarations. In the following, the keyword `local` is used for a local symbol (only visible in the current object) while `global` allows other object to see that symbol.

`pl_code local/global` *`pl_label`*`:` define a Prolog predicate whose associated symbol is *`pl_label`*. For the moment all predicates are `global` (i.e. visible by all other Prolog objects). But `local` will be used when implementing a module system.

`c_code local/global/initializer` *`label`*`:` define a function that can be called by a C function. The use of `initializer` ensures that this function will be executed first, when the Prolog engine is started. Only one function per file may be declared as `initializer`.

`long local/global` *`ident`* `=` *`value`*`:` allocate the space for a `long` variable whose name is *`ident`* and initialize it with the integer *`value`*. The initialization is optional (i.e. the `=` *`value`* part may be omitted).

`long local/global` *`ident`*`(`*`Size`*`):` allocate the space for an array of *`Size`* `long`s whose name is *`ident`*.

The *WAM* $\rightarrow$ *MA* translation can be performed in linear time w.r.t. the size of the WAM file (the translation is performed on the fly while the WAM file is being read).

## 4.3 Associating an identifier to a predicate name

Since the MA language is later mapped to the assembly of the target machine only classical identifiers can be used (a letter followed by letters, digits or the underscore character). In particular, it is necessary to associate such an identifier to each predicate (referenced as *pl_label* in Section 4.2). Since the syntax of identifiers is more restrictive than the syntax of Prolog atoms (which can include any character using quotes) GNU Prolog uses a hexadecimal representation where each predicate name is translated into a symbol beginning with an `X` followed by the hexadecimal notation of the code of each character of the name followed by an underscore and the arity. For instance `append/3` is coded by the symbol `X617070656E64_3` (`61` is the hexadecimal representation of the code of `a`, `70` is associated to `p`, etc). The linker is then responsible for resolving external references (e.g. call to built-in predicates or to user predicates defined in another object). The output of the linker is filtered by GNU Prolog to decode eventual hexadecimal notations in the case of errors (e.g. an undefined predicate, multiple definitions for a predicate).

## 4.4 An example

Here we present the MA code associated to the simple clause `p(T,g(U),V):-q(a,T,V,U)`. Associated WAM instructions are shown as comments.

```
% gplc -M --comment t.pl
% cat t.ma
pl_code global X70_3                          ; define predicate p/3
   call_c   Get_Structure_Tagged(fn(0),X(1))  ; get_structure(g/1,1)
   fail_ret
   call_c   Unify_Variable()                  ; unify_variable(x(3))
   move_ret X(3)
   move     X(0),X(1)                         ; put_value(x(0),1)
   call_c   Put_Atom_Tagged(ta(0))            ; put_atom(a,0)
   move_ret X(0)
   pl_jump  X71_4                             ; execute(q/4)

                                              ; table definitions
long local at(2)                              ;   2 atoms
long local ta(1)                              ;   1 tagged atom
long local fn(1)                              ;   1 functor/arity

c_code initializer Object_Initializer         ; object initializer
```

```
call_c   Create_Atom("t.pl")               ; atom #0 is 't.pl'
move_ret at(0)
call_c   Create_Atom("p")                  ; atom #1 is 'p'
move_ret at(1)
call_c   Create_Atom_Tagged("a")           ; tagged atom #0 is 'a'
move_ret ta(0)
call_c   Create_Functor_Arity_Tagged("g",1); func/arity #0 is g/1
move_ret fn(0)
call_c   Create_Pred(at(1),3,at(0),1,1,&X70_3)
c_ret                                      ; record predicate p/3
```

It is easy to see that most WAM instructions give rise to a C function call (e.g. `call_c Get_Structure_Tagged()`). Calls to functions that can fail (unification) are followed by a `fail_ret` that performs a Prolog fail if the returned value is 0. Note the presence of the MA instruction `move` to perform a copy of WAM registers (associated to the WAM instruction `put_value(x(0),1)`).

According to the encoding presented in Section 4.3, the symbol `X70_3` is associated to `p/3` (and `X71_4` to `q/4`) .

It might be worthwhile here looking at how atoms are managed. Classically all atoms are stored in a hash-table. As GNU Prolog is not restricted to a single source file, but allows for separate linking, the hash-table must be built at run-time. That is why the `Object_Initializer` function is first invoked and is responsible for updating the atom table with the atoms required by the object. The `Create_Atom` function adds an atom to the hash table (if not already present) and returns its associated hash value (key) which is stored in a local array (`at(`*atom_number*`)`). The `Create_Atom_Tagged` function is similar but returns a tagged WAM word instead of only the key. All the atoms used in WAM instructions (e.g. `put_atom`) are thus tagged only once by the initializer function and stored in a local array (`ta(`*atom_number*`)`). Similarly, all <functor/arity> pairs used by WAM instructions (e.g. `get_structure`) are computed once and stored in a local array (`fn(`*f_n_number*`)`). The initialization function also updates the predicate table with predicates defined in the current object. The `Create_Pred` function updates this table with the name and the arity of the predicate, the file name and the line number where it is defined, a mask encoding its properties (public/private, static/dynamic, user-defined/built-in) and the address of the code. It is worth noticing that this table is not needed for native execution since all static references are resolved by the linker. However, it is useful to implement some built-in predicates like `current_predicate/1` and, more importantly, to implement meta-call (e.g. `call/1`) since it is necessary to find the address of the code associated to a

predicate (given as a Prolog term). The way in which the initializer function is automatically invoked at run-time is explained below in Section 6.

# 5 Mapping the mini-assembly to a target machine

The next stage of the compilation process consists in mapping the MA file to the assembly language of the target machine. Since MA is based on a reduced instruction set, the writing of such mappers is simplified. However, producing machine instructions is not an easy task. The first mapper was written with the help of a C file produced by `wamcc`. Indeed, compiling this file to assembly with `gcc` gave us a starting point for the translation (since the MA instructions correspond to a subset of that C code). We then generalized this approach by defining a C file (now independently from `wamcc`). Each portion of this C code corresponds to an MA instruction and the study of the assembly code produced by `gcc` is a good starting point. This gives preliminary information about register conventions, C calling conventions, etc. However, to further optimize the assembly code it is necessary to refer to the technical documentation of the processor together with the ABI (Application Binary Interface) used by the operating system.

Here is the relevant portion of the linux/ix86 assembly code corresponding to the definition of `p/3` (the associated MA code is shown as comments):

```
% gplc -S --comment t.pl
% cat t.s
.text
fail:
   jmp    *-4(%ebp)        # fail

   .globl X70_3
X70_3:
   movl   fn+0,%eax        # call_c Get_Structure_Tagged(fn(0),X(1))
   movl   reg_bank+4,%edx
   call   Get_Structure_Tagged
   testl  %eax,%eax        # fail_ret
   je     fail
   call   Unify_Variable   # call_c   Unify_Variable()
   movl   %eax,reg_bank+12 # move_ret X(3)
```

```
   movl   reg_bank+0,%eax  # move     X(0),X(1)
   movl   %eax,reg_bank+4
   movl   ta+0,%eax        # call_c   Put_Atom_Tagged(ta(0))
   call   Put_Atom_Tagged
   movl   %eax,reg_bank+0  # move_ret X(0)
   jmp    X71_4            # pl_jump  X71_4

.data
   .local at               # long local at(2)
   .comm  at,8,4
   .local ta               # long local ta 1
   .comm  ta,4,4
   .local fn               # long local fn 1
   .comm  fn,4,4
```

Here again, a crucial point is that the mapping *MA* $\rightarrow$ *assembly* is executed in linear time w.r.t. the size of the MA file (the translation is done on the fly while the MA file is being read). Obviously the translation into the assembly language of a given machine makes several optimizations possibles. For instance the ix86 mapper uses 2 registers: `ebx` stores the value of `TR` (the trail top WAM register) and `ebp` stores the value of the `B` WAM register (last choice point). Other registers are stored in a global array (`reg_bank`) which consists of 256 `X` registers (WAM arguments) followed by the rest of the WAM registers (`H`, `CP`, ...)[2]. More generally, it is possible to use machine registers if it is ensured that they are saved and restored by the functions using them (the ABI gives this information).

Note the definition of a `fail` label which performs a WAM `fail`. The associated code branches to the value of the `ALT` cell of that choice-point (stored at the offset -1(*4 bytes) from `B`). Since `B` is kept in a machine register this can be achieved with only one instruction.

Let us take a closer look at how a C function is invoked. Firstly, all arguments are first loaded and then the function is called. Under ix86, the arguments are passed into the stack (as usual for CISC processors). The classical way to call a C function is then to use `push` instructions to initialize arguments, to call the function and, after the return of the function, either to use several `pop` instructions or to perform a stack pointer adjustment (adding a positive number to it). One possible optimization is to use a *fastcall calling convention* consisting in passing up to three arguments into registers (`eax`, `edx` and `ecx` in

[2]On RISC machines the base of this bank is stored in a global register since access to a global variable requires 2 cycles while using a global register makes it possible to access any WAM register in only 1 cycle.

that order). In this way, only those functions which require more than three arguments will use the stack to store the rest of the arguments. In order to avoid the problem of stack adjustments at the end of the C function, GNU Prolog does not push the arguments but instead reserves, at the start of the Prolog engine, enough space in the stack and then copies arguments on that space using a `move` instruction (with a positive offset relative to `esp`) instead of a `push` instruction. This scheme is adopted mostly for reasons of simplicity and has hardly any impact on performance.

Further machine-specific optimizations could be achieved, e.g. short branching detection, but this would require a multi-pass mapper.

# 6 Linking

At link-time all objects are linked with the GNU Prolog libraries: Prolog built-in predicate library, FD built-in constraint/predicate library and run-time library. In particular this last library includes all the functions which implement WAM instructions (e.g. `Put_Atom_Tagged()`). Linked objects come from: Prolog sources, user-written C foreign code or FD constraint definitions. This stage resolves external symbols (e.g. a call to a predicate defined in another module).

Since a Prolog source gives rise to a classical object file, several objects can be grouped in a library (e.g. using `ar` under Unix). The Prolog and FD built-in libraries are created in this way (and the user can also define his own libraries). Defining a library allows the linker to extract only the object files that are necessary (i.e. those containing referenced functions/data). For this reason, GNU Prolog can generate small executables by avoiding the inclusion of most unused built-in predicates. To optimally reduce this size, the linker should exclude *all* (rather than *most*) unused predicates. To do this we should define one built-in predicate per Prolog file (similar to what is done for the standard C library) since the object is the unit of inclusion of the linker (i.e. when a symbol is referenced the whole object where this symbol is found is linked). For the moment built-in predicates are grouped according to theme, for instance, a program using `write/1` will give rise to an executable which also contains the code of `writeq/1`, `display/1`, etc. In the future we will define only one predicate per file.

In Section 4.4 we have mentioned the role of the initializer function (called

`Object_Initializer` in our example). It is worth explaining how this function is invoked. Indeed, the Prolog engine must be able to dynamically find at run-time all the objects selected by the linker and execute their initializer function. The solution retained in GNU Prolog consists in marking all object files with magic numbers together with the address of the initializer function. At run-time, a pertinent portion of the data segment is scanned to detect each linked object (thanks to the magic numbers) and invoke its initializer.

Another solution would consist in storing, for each object, the address of its initializer function in a specific data section (also called segment), as the linker will group all the data defined within the same section. At run-time, this section has to be scanned and each initializer invoked. This approach is used under Win32 and will also be adopted for architectures supporting user-defined sections (e.g. those using ELF executable format).

# 7 Memory management

Here we just recall that the WAM memory management consists in using three stacks: the Local Stack for control blocks and local variable, the Heap for data structures, and the Trail for storing bindings to undo upon backtracking (GNU Prolog also manages a stack for constraint solving).

It is mandatory to control the growth of stacks and to alert the user in case of overflow. This is usually done by incorporating software tests either at each memory allocation (potentially several times for each clause for the Heap) or at each clause entry (checking all stacks) or thanks to new WAM-like instructions. This control might however be costly, all the more because basically current machine architectures allow for hardware tests. Indeed machines use virtual memory, meaning that the user does not have to bother about physical addresses and real memory size, and provide, logically if not physically, very large linear memory (e.g. 4 GBytes on 32 bits architectures). When some data must be accessed, the memory manager detects if the memory *page* to which it actually belongs is physically present in memory or not (*page fault*). In the latter case, the memory manager loads it in memory after swapping another page on disk if necessary. Interestingly, the memory manager raises an *exception signal* when a page fault refers to an un-allocated (i.e. free) page. The idea is therefore to have such a signal raised in case of stack overflow. To ensure this we only have to free (i.e. to give back) each page following a stack (see figure 2). When an attempt to read/write in this page occurs the signal

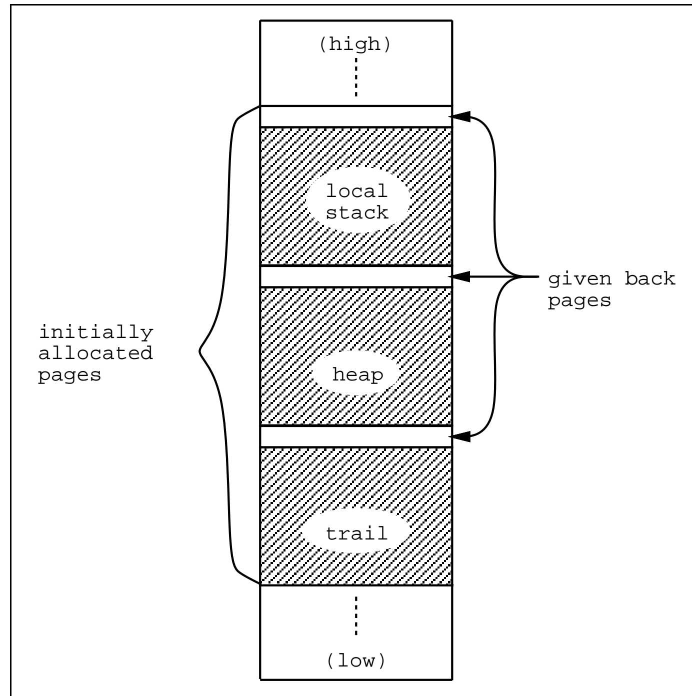


Figure 2: memory allocation

triggered is caught by a C function (*handler*) responsible of diagnosing the overflow (checking top stack pointers) and of generating the adequate error message.

The easiest way to implement this scheme under Unix is to use the `mmap` function which makes it possible to map a file to part of the memory. All the pages of this part are initially marked “swapped” on the corresponding page of the file. Then, reading and writing on this file are done by simply reading and writing the memory. There is usually a special device (`/dev/zero`) which returns zero on initial reading and on which writing is not reflected. This device is therefore well suited to our stacks since only memory operations are performed. Thanks to the `munmap` function, each page following a stack is given back to the memory manager.

Under windows we use the Win32 function `VirtualAlloc` to allocate the initial space and `VirtualProtect` to give back a page to the memory manager (thereby ensuring it will not be allocated by another request).

In the future we plan to reserve much more space between two consecutive stacks in order to handle dynamic stack growths. Indeed, when a stack overflow is detected one (or more) page(s) is allocated and the execution is resumed.

We did not implement this solution initially since, under Unix, this is exactly what the memory manager does when a page fault occurs (corresponding to a valid page). Hence, the user can define a very big initial space for each stack (via environment variables) but only the portion that is actually will be physically allocated. However, this is not true under Win32 (where we have to manage the growth explicitly).

Another important issue is the integration of a Garbage Collector. Our preference is for a simple (but efficient) collector in order to ensure easy maintenance and allow for possible extensions.

# 8 Tagging Scheme

The current version of GNU Prolog uses a tagging scheme which optimizes `Tag` / `UnTag` operations on the `REF` tag. Indeed, these operations are executed very often in the WAM (e.g. by the `Deref` procedure). This is done using 0 as the tag for `REF` and using low bits of a word to encode the tag. Since all address in the WAM are word-aligned, the 2 (resp. 3) least significant bits of the address are 0 on 32-bit (resp. 64-bit) machines. The tagging scheme used by GNU Prolog depends on the number of tags (data types) and on the target architecture. The current version uses 7 tags and thus requires three bits to encode them. The following table details the tagging scheme used on 32 and 64-bit machines.

| Tag | Type | on 32 bits | on 64 bits |
|---|---|---|---|
| `REF` | reference | 0 *address* 00 | *address* 000 |
| `LST` | list | 0 *address* 01 | *address* 001 |
| `STC` | structure | 0 *address* 10 | *address* 010 |
| `ATM` | atom | 0 *hash key* 11 | *hash key* 011 |
| `FLT` | float | 1 *address* 00 | *address* 100 |
| `FDV` | FD variable | 1 *address* 01 | *address* 101 |
| `INT` | integer | 1 *value* 11 | *value* 111 |

Obviously `Tag` and `UnTag` are void operations when the tag is `REF`.

For other tags whose values are addresses (`LST`, `STC`, `FLT` and `FDV`), the `Tag` macro simply adds the tag mask. Here an addition is better than a bitwise OR [3] since it allows the C compiler to optimize expressions like `Tag(LST,`

[3]but obviously semantically equivalent since affected bits in the address are 0-bits.

H+1) producing $H+4+1$ ($= H+5$ !) [4]. Using a bitwise OR we would obtain two instructions: one to compute $H+4$ and another one for the bitwise OR. The UnTag macro resets tag bits to 0 with one instruction (a bitwise AND).

For the ATM tag, the value is a hash key, i.e. a positive integer. The Tag macro generates two instructions: a logical left shift and the addition of the tag mask. This is possible since a hash key is a small positive integer, and thus we can be sure that the three most significant bits are zero. The UnTag operation simply consists in a logical right shift.

The case of the INT tag is a bit more complex. Since an integer may be negative, we cannot be sure that the three most significant bits are 0. Therefore the Tag macro requires three instructions, two to shift the value left, resetting high bits to zero, and one to add the tag mask. This can be optimized if the high bit of the tag is 1 (which is the case here) and therefore results in only two instructions: a logical left shift and a bitwise OR with the tag mask. The UnTag macro requires two instructions to perform a sign-bit propagation: a logical left shift (to bring back the sign bit) followed by an arithmetic right shift (which propagates the sign bit).

When developing wamcc, we tried to optimize operations on integers (using 0 for the INT tag). For reasons of simplicity, the first versions of GNU Prolog were developed in this perspective. The resulting gain obtained by the current tagging scheme is around 20 % while the slowdown on integer operations is marginal.

# 9 Prolog performance evaluation

## 9.1 Compilation

Table 1 presents the performance of the GNU Prolog compilation scheme on a classical set of benchmarks, times are in seconds and sizes in KBytes measured on a PentiumIII 733 Mhz with 192 MBytes of memory running Linux (Mandrake 7.1). We have also added the GNU Prolog pl2wam sub-compiler since it is a more representative example. For each program, the table gives: the number of lines of the Prolog source program [5], the compilation time needed to

[4] $H+4$ corresponds to H+1 on a 32-bit machine, 1 is the tag mask for LST.

[5] neither blank lines nor comments are counted.

| Program | lines | object time | object size | executable time | executable size |
|---|---|---|---|---|---|
| boyer | 416 | 0.25 | 38 | 0.35 | 174 |
| browse | 144 | 0.10 | 11 | 0.20 | 147 |
| cal | 168 | 0.10 | 11 | 0.22 | 145 |
| chat_parser | 957 | 0.70 | 101 | 0.83 | 236 |
| ham | 101 | 0.05 | 9 | 0.20 | 144 |
| nrev | 109 | 0.06 | 5 | 0.20 | 139 |
| poly_10 | 138 | 0.07 | 9 | 0.23 | 147 |
| queens | 112 | 0.05 | 5 | 0.17 | 138 |
| queensn | 89 | 0.06 | 4 | 0.20 | 139 |
| reducer | 373 | 0.22 | 29 | 0.37 | 165 |
| sendmore | 104 | 0.06 | 7 | 0.21 | 141 |
| tak | 78 | 0.03 | 3 | 0.17 | 137 |
| zebra | 94 | 0.07 | 6 | 0.18 | 140 |
| pl2wam | 3263 | 1.60 | 251 | 1.83 | 518 |

Table 1: Compilation evaluation (PIII-733 Mhz / Linux)

produce the object, the size of the object code, the total compilation time (including the link) and the final executable size. Sizes are those of the stripped objects/executables, i.e. they do not include the symbol table but they do include all of the data and code segments.

The size of (stripped) objects shows that this approach really generates small code (less than 10 KBytes for many benchmarks). The size of the whole executable shows the interest of excluding most of the unused built-in predicates. Indeed, when all built-in predicates (Prolog+FD) are present the size is at least 600 KBytes (this is the size of the GNU Prolog interactive top-level). Let us recall that we can even further reduce this size with a little reorganization of the GNU Prolog libraries (see Section 6). An important feature of GNU Prolog is its ability to produce small stand-alone executables which can be used in many applications (tools, web, CGIs, etc). Other Prolog systems cannot produce such stand-alone executables since they always require the presence of the emulator at run-time (500 KBytes to 2 MBytes).

Compilation times are rather good and we have achieved our initial goal since GNU Prolog compiles 5-10 times faster than `wamcc`+`gcc`. Obviously this factor is not constant and the gain is more effective on large programs (and thus it is difficult to give an upper bound to the speedup factor). This is due to

| Program | lines | gnu | wamcc | speedup |
|---|---|---|---|---|
| cal | 168 | 0.22 | 0.74 | 3.36 |
| boyer | 416 | 0.35 | 1.89 | 5.50 |
| chat_parser | 957 | 0.83 | 5.77 | 6.91 |
| pl2wam | 3263 | 1.60 | 14.73 | 9.20 |

Table 2: Compilation speed (in secs on PIII-733 / Linux)

the fact that the translation from the WAM to an object is done in linear time (each translation only needs one pass) while a C compiler may require a quadratic time (and even worse) for its optimizations. Table 2 illustrates this comparison of compilation times for both systems on some representative benchmarks (since wamcc is not able to compile pl2wam due to a restricted built-in library we only took into account, for both systems, the time needed to generate the object file).

## 9.2 Benchmarking Prolog

| Program × 10 iter. | gnu 1.2.5 | yap 4.3.12 | wamcc 2.21 | sicstus 3.8.5 | ciao 1.6 | bin 8.00 | xsb 2.2 | swi 3.4.2 |
|---|---|---|---|---|---|---|---|---|
| boyer | 1.44 | 1.23 | 1.87 | 1.58 | 1.56 | 4.01 | 3.76 | 7.53 |
| browse | 1.85 | 1.12 | 2.00 | 2.04 | 2.15 | 5.20 | 3.50 | 7.13 |
| cal | 0.18 | 0.31 | 0.20 | 0.46 | 0.45 | 0.54 | 0.72 | 2.12 |
| chat_parser | 0.31 | 0.28 | 0.50 | 0.41 | 0.41 | 0.67 | 0.55 | 0.82 |
| ham | 1.27 | 1.00 | 2.08 | 1.62 | 1.66 | 2.47 | 2.57 | 4.07 |
| nrev | 0.17 | 0.08 | 0.26 | 0.14 | 0.25 | 0.00 | 0.43 | 1.04 |
| poly_10 | 0.10 | 0.08 | 0.14 | 0.12 | 0.13 | 0.22 | 0.24 | 0.46 |
| queens | 1.03 | 0.79 | 1.05 | 1.76 | 1.99 | 2.69 | 3.02 | 13.44 |
| queensn | 4.58 | 3.46 | 6.69 | 5.69 | 6.80 | 8.75 | 9.52 | 22.84 |
| reducer | 0.09 | 0.07 | 0.12 | 0.09 | 0.10 | 0.18 | 0.23 | 0.28 |
| sendmore | 0.10 | 0.10 | 0.12 | 0.19 | 0.24 | 0.44 | 0.32 | 0.84 |
| tak | 0.25 | 0.25 | 0.31 | 0.37 | 0.40 | 0.86 | 0.68 | 137.65 |
| zebra | 0.09 | 0.07 | 0.16 | 0.12 | 0.15 | 0.19 | 0.15 | 0.21 |
| Speedup | | ↓ 1.26 | 1.36 | 1.38 | 1.56 | 2.61 | 2.49 | 5.77 |

Table 3: GNU Prolog versus other systems (in secs on PIII-733 / Linux)

In this section we compare GNU Prolog with two commercially available systems: SICStus Prolog emulated [6] and BinProlog (evaluation version) and five

[6] with default installation options of the binary version provided by SICS.

academic systems: Yap Prolog, `wamcc`, CIAO, XSB-Prolog and SWI-Prolog. Table 3 presents execution times for these systems and the average speedup (or slowdown when preceded by a ↓ sign) of GNU Prolog. For each benchmark, the execution time is the total time needed for 10 iterations. Times are measured on a PentiumIII 733 Mhz with 192 MBytes of memory running Linux.

To summarize, GNU Prolog is slightly slower than Yap Prolog, which is the fastest emulated Prolog system. On the other hand, GNU Prolog is 1.4 times faster than SICStus emulated and `wamcc`, 1.6 times faster than CIAO, around 2.5 times faster than BinProlog and XSB-Prolog and more than 5 times faster than SWI-Prolog (without taking into account the `tak` benchmark). To be fair we should point out that we did not have enough time to carry out an exhaustive comparison with all the Prolog systems and their variants. For instance, SICStus Prolog can compile to native code for some architectures (e.g. under SunOS/sparc but not yet under Linux/ix86) and then it should be twice as fast as GNU Prolog on those platforms. Let us also note that BinProlog can also partly compile to C.

It is worth noticing that the above benchmarks, classical in the Logic Programming community, are small programs that are not really representative of real-life applications, and no longer well suited to assessing the performance of efficient Prolog systems running on fast hardware. Most of the times in the above table are under a second although they represent ten executions of each benchmark. Since there is no general agreement on a more up-to-date set of benchmarks, we took the most obvious one: `pl2wam` (the GNU Prolog to WAM compiler compiling itself) which is more than 3,000 lines long. Table 4 shows the execution times for the Prolog systems in which we were able to execute this benchmark with minor adaptations.

| | gnu 1.2.5 | yap 4.3.12 | sicstus 3.8.5 | ciao 1.6 | swi 3.4.2 |
|---|---|---|---|---|---|
| `pl2wam` | 0.77 | 1.25 | 3.02 | 3.44 | 2.38 |
| Speedup | | 1.62 | 3.92 | 4.47 | 3.09 |

Table 4: The `pl2wam` benchmark (in secs on PIII-733 / Linux)

This benchmark spends around 35 % of the total time performing Prolog term input/output. Therefore SICStus and CIAO, which are based on the classical Warren/O'Keefe's IO term library (written in Prolog), show poor performance: around 4 times slower than GNU Prolog. SWI-Prolog uses a C implementation of `read/write` predicates and is only 3 times slower than GNU Prolog. Finally Yap Prolog, which is nevertheless also based on a C library, is surprisingly 1.6

times slower than GNU Prolog. Even if a single benchmark could obviously not be representative of all applications, it should be recalled that compilation is the primary task of any Prolog system and its performance is thus interesting to observe.

Taken as a whole, this performance evaluation shows that a Prolog system based on a simple WAM engine can nevertheless have a good level of efficiency with this MA-based native compilation scheme. Obviously further improvements could be achieved by integrating all well-known WAM optimizations. Another interesting issue consists in inlining C calls to the run-time library. A straightforward first step is to replace some `call_c` MA instructions by the assembly code of the function involved. Preliminary results show a 20-30 % speedup, but better performance could be achieved, for instance by taking argument loading into account.

# 10 Constraint solving

Constraint Programming is a widely successful extension of Logic Programming, which has had a significant impact on a variety of industrial applications, see [9]. It is thus natural to include a constraint solving extension to any modern Prolog-based system.

GNU Prolog compiles finite domain constraints in the same way as its predecessor `clp(FD)`, described in [5, 6]. It is based on the so-called "RISC approach" which consists in translating at compile-time all complex user-constraints (e.g. disequations, linear equations or inequations) into simple, primitive constraints (the FD constraint system) at a lower level which really embeds the propagation mechanism for constraint solving. Let us first present the basic ideas of the FD constraint system and then detail the extensions to this framework implemented in GNU Prolog.

## 10.1 The FD constraint system

The FD constraint system is a general purpose constraint framework for solving discrete constraint satisfaction problems (CSPs). It was originally proposed by Pascal Van Hentenryck in a concurrent constraint setting [12], and an efficient implementation in the `clp(FD)` system is described in [5, 6]. FD is based on a

single *primitive constraint* through which complex constraints are defined, so for example constraints such as $X = Y$ or $X \leq 2Y$ are *defined* by means of FD constraints, instead of being built into the theory. Each constraint is thought of as a set of propagation rules describing how the domain of each variable is related to the domain of other variables, i.e. rules for describing node and arc consistency propagation (see for instance [10] for more details on CSPs and consistency algorithms).

A *constraint* is a formula of the form $X$ `in` $r$ where $X$ is a variable and $r$ is a range. A *range* in FD is a (non empty) finite set of natural numbers. Intuitively, a constraint $X$ `in` $r$ enforces that $X$ belongs to the range denoted by $r$. Such a range can be not only a *constant range* (e.g. 1..10) but also an *indexical range* when it contains one or more of the following:

- `dom(`$Y$`)` which represents the whole current domain of $Y$;
- `min(`$Y$`)` which represents the minimal value of the current domain of $Y$;
- `max(`$Y$`)` which represents the maximal value of the current domain of $Y$.

Obviously when $Y$ is instantiated, all three indexicals evaluate to this value. When an $X$ `in` $r$ constraint uses an indexical term depending on another variable $Y$ it becomes *store-sensitive* and must be checked each time the domain of $Y$ is updated. This is how consistency checking and domain reduction is achieved.

Complex constraints such as linear equations or inequations, as well as symbolic constraints can be defined in terms of the FD constraint system, see [6]. For instance, the constraint $X \leq Y$, is translated as follows:

$$\mathrm{X} \leq \mathrm{Y} \qquad \equiv \qquad \mathrm{X\ in\ 0..max(Y)} \quad \wedge \quad \mathrm{Y\ in\ min(X)..}\infty$$

Notice that this translation also has an operational flavor, and specifies, for a given n-ary constraint, how a variable domain has to be updated in terms of the other variable. For example, in the FD constraint `X in 0..max(Y)`, whenever the largest value of the domain of Y changes (i.e. decreases), the domain of X is reduced. If, on the other hand, the domain of Y changes but its largest value remains the same, then the domain of X does not change. One can therefore consider those primitive `X in r` constraints as a low-level language in which the propagation scheme has to be expressed. Indeed, it is possible to express in the constraint definition (i.e. the translation of a high-level user constraint

into a set of primitive constraints) the propagation scheme chosen to solve the constraint, such as forward-checking, or full or partial look-ahead, depending on the use of `dom` or `min/max` indexical terms.

## 10.2 Finite Domain constraints in GNU Prolog

In GNU Prolog we have designed a specific language to define FD constraints in a flexible and powerful way. The basic $X$ `in` $r$ primitive does not offer a way of defining reified constraints (except via a C user function) and does not allow the user to control the propagation triggers. The need for symbolic constraints like `element/3` also highlighted the need to handle lists of variables at the primitive level. The GNU Prolog constraint definition language has then been designed to allow the user to define complex constraints and not only basic arithmetic constraints. This language is compiled to C by the `fd2c` sub-compiler. The C source file obtained is submitted to the C compiler to obtain an object which is then included by the linker as shown in Figure 1. We present the main features of this language by means of some examples of constraint definitions.

Let us define a constraint $X + C = Y$ ($X$ and $Y$ are FD variables, $C$ is an integer):

```
x_plus_c_eq_y(fdv X,int C,fdv Y)
{
 start X in min(Y) - C .. max(Y) - C         /* X = Y - C */
 start Y in min(X) + C .. max(X) + C         /* Y = X + C */
}
```

A constraint is defined in a C-like syntax. The head declares the name of the constraint (`x_plus_c_eq_y`) and for each argument its type (`fdv`, `int`) and its name. The keyword `start` activates an $X$ `in` $r$ primitive. The first states that the bounds of $X$ must be between $min(Y) - C$ and $max(Y) - C$. Similarly, the second indicates how to update $Y$ from $X$.

Here is a more complex example to define $min(X, A) = Z$ (where $A$ and $Z$ are FD variables and $A$ an integer):

```
min_x_a_eq_z(fdv X,int A,fdv Z)
{
 start (c1) Z in Min(min(X),A)..max_integer  /* Z >= min(X,A) */
```

```
 start (c2) Z in 0 .. max(X)                /* Z <= X */
 start (c3) X in min(Z) .. max_integer
 start      Z in 0 .. A                     /* Z <= A */

 wait_switch
        case A>max(Z)                       /* case : A != Z */
                stop c1
                stop c2
                stop c3
                start Z in min(X) .. max(X)  /* Z = X */
                start X in min(Z) .. max(Z)
}
```

The first $X$ `in` $r$ constraint uses a C macro `Min` to compute the minimum between $min(X)$ and $A$. The keyword `max_integer` represents the greatest integer that an FD variable can take. Note the use of the `wait_switch` instruction to enforce $X = Z$ (and to stop the constraints `c1`, `c2`, `c3`) as soon as the case $A \neq Z$ is detected.

This facility offered by the language to delay the activation of an $X$ `in` $r$ constraint makes it possible to define reified constraints. The basic idea of reified constraints is to consider the truth values of constraints as first-class objects which are given the form ("reified") of boolean values. This allows the user to make assumptions about the issues of constraints in a given store in order to define other constraints. The following example illustrates how to define $X = C \Leftrightarrow B$ where $X$ is an FD variable, $C$ an integer and $B$ a boolean variable (i.e. an FD variable whose domain is 0..1) which captures the truth value of the constraint $X = C$. The definition below waits until either the truth of $X = C$ or the value of $B$ is known:

```
truth_x_eq_c(fdv X,int C,fdv B)
{
 wait_switch
        case max(B)==0                      /* case : B = 0  */
                start X in ~{ C }           /* X != C */

        case min(B)==1                      /* case : B = 1  */
                start X in { C }            /* X = C */

        case min(X)>C || max(X)<C           /* case : X != C */
                start B in { 0 }            /* B = 0 */
```

```
        case min(X)==C && max(X)==C          /* case : X = C  */
                start B in { 1 }             /* B = 1 */
}
```

Each constraint gives rise to a C function returning a boolean depending on the outcome of the addition of the constraint to the store. The link between Prolog and a constraint is done by a specific Prolog predicate `fd_tell/1` which is in fact compiled to a `call_c` to the corresponding C function followed by a `fail_ret`. For instance, to define the previous constraint one would declare a predicate `'x=c <=> b'/3` in the following way:

```
'x=c <=> b'(X,C,B) :-
          fd_tell(truth_x_eq_c(X,C,B)).
```

Obviously, GNU Prolog offers a wide variety of high-level constraints in the built-in library. However, low-level definitions of constraints as illustrated above are open to the expert programmer who needs to customize or enrich the constraint solver for some practical application.

## 10.3 Benchmarking FD constraints

The performance of the FD constraint solver of GNU Prolog is the same as `clp(FD)` [6], in other words, equivalent to the Ilog_Solver commercial C++ system from ILOG and on average twice as fast as CHIP, a commercial constraint logic programming system from Cosytec, on a similar subset of constraints.

In this section we compare the FD constraint solver with that of SICStus Prolog. As the syntax and the set of predefined constraints are not the same in both systems we use some examples provided with SICStus. Since we are interested here in comparing the raw performance of the implementation of the solvers (and not their expressive power) we selected benchmarks with a similar formulation in both systems for which we could expect both solvers to perform the same computations. Table 5 presents execution times for both solvers and the speedup for GNU Prolog. Times are measured on a PentiumII 400 Mhz with 128 MBytes of memory running Linux (RedHat 5.1). On average GNU Prolog is around 4 times faster than SICStus Prolog.

| Program | gnu 1.2.5 | sicstus 3.7.1 | speedup |
|---|---|---|---|
| crypta | 0.008 | 0.012 | 1.50 |
| eq10 | 0.006 | 0.020 | 3.33 |
| eq20 | 0.010 | 0.030 | 3.00 |
| donald | 0.210 | 0.820 | 3.90 |
| alpha | 0.450 | 2.880 | 6.40 |
| alpha ff | 0.010 | 0.030 | 3.00 |
| queens 16 | 0.050 | 0.270 | 5.40 |
| cars all | 0.015 | 0.060 | 4.00 |

Table 5: GNU Prolog FD solver versus SICStus FD solver

# 11 Conclusion

GNU Prolog is a free Prolog compiler with constraint solving over finite domains. The Prolog part of GNU Prolog conforms to the ISO standard for Prolog and also provides many extensions which are very useful in practice (global variables, OS interface, sockets, etc), while the finite domain constraint part contains all classical arithmetic and symbolic constraints, and also integrates an efficient treatment of reified constraints and boolean constraints. The compilation scheme of GNU Prolog uses a restricted mini-assembly language which drastically speeds up compilation times and thus combines the advantages of fast compilation (like emulator-based systems) and native compilation. Indeed, GNU Prolog produces native binaries which are stand-alone. The size of these executable files can be quite small since GNU Prolog can avoid linking the code of most unused built-in predicates. It could thus be worth investigating the use of the MA language as a back-end for other logic or functional languages. The performance of GNU Prolog is close to that of commercial systems, both for the Prolog part and the Constraints part and several times faster than other popular free systems. Current work is focused on improving these performances. A promising direction consists in inlining the assembly code of some GNU Prolog runtime library functions (associated to WAM instructions) instead of emitting C function calls. Preliminary experiments have given encouraging results.

# References


[1] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming Series, MIT Press, 1991.

[2] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine.* PhD dissertation, SICS, Sweden, 1990.

[3] J. Chailloux. La machine LLM3. Technical Report RT-055, INRIA, 1985.

[4] P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*, Tokyo, Japan, MIT Press, 1995.

[5] P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(FD)`. In *Proc. ICLP'93, 10th International Conference on Logic Programming.* Budapest, Hungary, MIT Press, 1993.

[6] P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, Vol. 27, No. 3, June 1996.

[7] Information technology - Programming languages - Prolog - Part 1: General Core. ISO/IEC 13211-1, 1995.

[8] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[9] V. Saraswat, P. Van Hentenryck, P. Codognet et al. Constraint Programming. *ACM Computing Surveys*, vol. 28, no. 4, Dec. 1996.

[10] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[11] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, 1989.

[12] P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). In *Constraint Programming: Basics and Trends*, A. Podelski (Ed.), LNCS 910, Springer Verlag 1995. First version: Research Report, Brown University, Jan. 1992.

[13] P. Van Roy and A. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer, pp 54-67, 1992.

[14] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.