The GNU Prolog System and its Implementation

Daniel Diaz
University of Paris 1
CRI, bureau C1407
90, rue de Tolbiac
75634 Paris Cedex 13, 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

We describe in this paper the GNU-Prolog system, a free system consisting of a Prolog compiler and a constraint solver on finite domains. GNU-Prolog is based on a low-level mini-assembly platform-independent language that makes it possible for efficient compilation time, and allows to produce small stand alone executable files as the result of the compilation process. Interestingly, the Prolog part is compliant to the ISO standard, and the constraint part includes several extensions, such as an efficient handling of reified constraints. The overall system is efficient and comparable in performances with commercial systems, both for the Prolog and constraint parts.

1. INTRODUCTION

GNU Prolog [1] is a free Prolog compiler supported by the GNU organization. It was released in April 1999 and more than 2500 copies have been downloaded up to now from the INRIA ftp mirror site [2]. It is built on previous systems developed at INRIA, namely `wamcc` [4] for Prolog and `clp(FD)` [5] for constraint solving on finite domains. However, the compilation scheme has been completely redesigned and the system extended.

The overall features of GNU Prolog can be summed up as follows. The Prolog system includes floating point numbers, streams, dynamic code, etc, and is compliant to the ISO standard for Prolog; it also integrates several extensions such as global variables, definite clause grammars (DCG), a sockets interface, an operating system interface, and a total of more than 300 Prolog built-in predicates. The system also include a Prolog debugger and a low-level WAM debugger, a line editing facility under the interactive interpreter with completion on atoms. Last but not least, there is a powerful bidirectional interface between Prolog and C. This features implicit Prolog $\leftrightarrow$ C type conversion, transparent I/O argument handling, non-deterministic C code, and ISO error support. Concerning performances, compiled predicates (native-code) are as fast as `wamcc` on average and consulted predicates (byte-code) are 5 times faster than `wamcc`. In the constraint solving part, there is no need for explicit declarations of FD variables, which are completely integrated into the Prolog environment, and fully compatible with Prolog variables and integers. 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 single (low-level) primitive to define all (high-level) FD constraints. There are many advantages in this approach: constraints can be compiled, the solver is open and extensible, and the performances are comparable to that of commercial solvers. The FD solver contains 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. Moreover new high-level constraints can be easily defined by the user and defined in terms of simple primitives.

The rest of this paper is organized as follows. Section 2 introduces the motivation for producing yet another Prolog and CLP system and provides some background. Section 3 describes the compilation scheme for Prolog and performance evaluation is detailed in Section 4. Then Section 5 describe the extension to handle constraint solving over finite domains, and a short conclusion ends the paper.

2. MOTIVATION AND BACKGROUND

Since Pascal and the P-code in the 70's , and more recently with Java and the JVM in the 90's abstract machines have been highlighted as the backbone of the compilation process. For Prolog, the Warren Abstract Machine (WAM) [14; 1] is a de facto standard and classical techniques consist in either executing directly the WAM code with an emulator written in C (original version of SICStus Prolog) or assembler (Quintus Prolog) or in directly compiling to native code (Prolog by BIM, latest version of SICStus Prolog, Aquarius Prolog). Another approach consists in translating Prolog to C. The idea is to compensate the lack of optimizations (required for simplicity) by the absence of emulation overhead phases (fetching and decoding) since, finally, the C compiler would produce native code. The `wamcc` Prolog compiler was based on this approach, but with also the idea of translat-

[1] `http://www.gnu.org/software/prolog`
[2] Statistics concerning the main GNU ftp site and other mirror sites are not available.

ing 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 big program by standard C compilers (e.g. GCC), especially in trying to optimize the code produced.

The novelty of the GNU Prolog compilation scheme is to translate a WAM file into a mini-assembly (MA) file. 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 the WAM is translated. 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 10 instructions, mainly to handle the control of Prolog and to call a C function.

3. PROLOG COMPILATION SCHEME

Classically the Prolog source file gives rise to an object which is linked to the GNU Prolog libraries to produce an executable. For that, the Prolog source is first compiled to obtain a WAM file. The WAM file is translated to a MA file. The MA file is then mapped to the assembly language of the target machine which next give rise to an object file. All object files are linked together 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 invoke the C compiler to obtain object files as explained later. Obviously all intermediate stages are hidden to the user who simply invokes the compiler on his Prolog file(s) (plus other files: foreign C file, FD files,etc) and obtains an executable. However, it is also possible to stop the compiler at any given stage. This can be useful, for instance, to see 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.

3.1 From Prolog to the WAM

The `pl2wam` sub-compiler accepts a Prolog source and produces a WAM file. This compiler is fully written in GNU Prolog and bootstrapped. It compiles each clause in several passes, as follows. First, the clause is simplified: control constructs like disjunctions, if-then or cut are rewritten (giving rise to auxiliary predicates if needed). Then, the clause is translated in a more practical internal format and variables are classified as permanent or temporary, permanent variables are assigned. Subsequently, the WAM code associated to the clause is generated. Finally, registers (temporary variables) are assigned and optimized. The code of each clause of a predicate is then grouped and the indexing code is generated, see [2] for details.

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 options makes it possible to study the basic Prolog to WAM compilation process.

3.2 From the WAM to the mini-assembly (MA)

The second stage translates a WAM file into a MA file. The idea of the MA language is to have a machine-independent intermediate language in which the WAM is translated. The design of MA comes from the study of the C code produced by `wamcc`. Indeed, in the `wamcc` system, most WAM instructions given rise to a call to a C function performing the treatment. The only exception was 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 then offers 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 (from which an object is produced).

In order to simplify the writing of translators of the MA to a given architecture (i.e. the mappers), the MA instruction set must be simple: it only contains 10 instructions that are described now:

`pl_jump` ***pl_label*:** give the control to the predicate whose corresponding symbol is ***pl_label***. This instruction corresponds to the WAM instruction `execute`.

`pl_call` ***pl_label*:** give the control to the predicate whose corresponding symbol is ***pl_label*** after initializing the continuation register `CP`. This instruction corresponds to the WAM instruction `call`.

`pl_ret`: give the control to the address stored in the continuation pointer `CP`. This instruction corresponds to the WAM instruction `proceed`.

`pl_fail`: give the control to the address stored in the last alternative (`ALT` cell of the last choice point). This instruction corresponds to the WAM instruction `fail`.

`jump` ***label*:** give the control to the symbol ***label***. This instruction is used when translating indexing WAM instructions to perform local control transfer (e.g. `try`, `retry` or `trust`). This instruction has been distinguished from `pl_jump` (even if both can be implemented in a same manner) since, on some machines, local jumps can be optimized.

`c_ret`: C return. This instruction is used at then end of the initialization function to give back the control to the caller.

`move` ***reg1*,*reg2*:** copy the WAM `X` or `Y` register ***reg1*** to the register ***reg2***.

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

`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. For instance, the WAM indexing instruction `switch_on_term` is implemented via a C function which returns the address of the selected code.

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

The MA declarations are presented now. The keyword `local` specifies 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 corresponding 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 can be declared as `initializer`.

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

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

3.3 From the mini-assembly to the assembly

The next stage of the compilation process consists in mapping the MA file to the assembly of the target machine. Since MA is based on a reduced instruction set, the writing of such translators is simplified. However, producing machine instructions is not an easy task. The first translator was written with the help of a C file produced by `wamcc`. Indeed, compiling this file to assembly with `gcc` gave us a first solution for the translation (since the MA instructions corresponds to a subset of that C code). We have then generalized this by defining a C file (independently from `wamcc`) whose compilation to assembly is a good starting point when porting GNU Prolog to a new architecture. Mappings from MA to target machines currently include SunOS/Sparc, Solaris/Sparc, Linux/ix86, Linux/PowerPC, Win95-98-NT/ix86, new ports are under development.

3.4 From objects to the executable

At link-time all objects are linked together with the Prolog libraries: Prolog built-in predicate library, FD built-in constraint/predicate library and run-time library. This last library contains in particular functions implementing WAM instructions. Linked objects come from: Prolog source, user C foreign code or FD constraint definition. 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, several objects can be grouped in a library (e.g. using `ar` under Unix). The Prolog and FD built-in libraries are created using this way. Defining a library allows the linker to only extract from it needed objects (i.e. containing referenced functions/data). For this reason, the GNU Prolog compiler can generate small executables by avoiding the inclusion of most unused built-in predicates.

4. PROLOG PERFORMANCE EVALUATION

We have tested the performances of GNU Prolog on a classical set of Prolog benchmarks (see table below). Compilation timings are rather good and we reached our initial goal since GNU Prolog compiles 5-10 times faster than `wamcc+gcc` while produced code is as fast as `wamcc`. The raw compilation speed is about 1000 lines per second on a Pentium 400 Mhz machine, recalling than the declarative and high-level aspect of Prolog makes it much more concise than C. The size of (stripped) objects is really small (less than 10 KBytes for many benchmarks) and shows that this approach really generates small code. The ability of GNU Prolog to produce small executables is an important feature that makes it possible to use them in many occasions (tools, web CGIs,...).
We have compared GNU Prolog with one commercial system: SICStus prolog and two academic systems: XSB-Prolog and SWI-Prolog. Table 1 presents execution times for those systems and the average speedup of GNU Prolog on a classical set of benchmarks (the `nand` program could not be compiled under XSB-Prolog). Timings are in seconds measured on a Pentium II 400 Mhz under Linux. In the heavyweight category, GNU Prolog is 1.2 times faster than SICtus emulated. To be fair let us mention that SICStus Prolog can compile to native code for some architectures (e.g. under SunOS/sparc but not yet under linux/ix86) and then it will be 2.5 times faster than GNU Prolog on those platforms. However, in the academic league, GNU Prolog is around 2.5 times faster than XSB-Prolog and more than 5 times faster than SWI-Prolog (without taking into account the `tak` benchmark).

Program	GNU Prolog 1.0.5	Sicstus Prolog 3.7.1	XSB Prolog 1.8.1	SWI Prolog 3.2.8
`boyer`	0.332	0.324	0.889	1.424
`browse`	0.430	0.424	0.837	1.274
`cal`	0.030	0.074	0.146	0.328
`chat_parser`	0.080	0.092	0.254	0.252
`crypt`	0.006	0.006	0.004	0.034
`ham`	0.304	0.340	0.634	0.770
`meta_qsort`	0.006	0.004	0.014	0.034
`nand`	0.018	0.018	?.???	0.072
`nrev`	0.044	0.036	0.096	0.206
`poly_10`	0.028	0.024	0.062	0.104
`queens (16)`	0.238	0.416	0.802	2.374
`queens_n (10)`	1.148	1.262	0.002	4.288
`reducer`	0.022	0.026	0.066	0.094
`sdda`	0.002	0.002	0.002	0.034
`sendmore`	0.026	0.046	0.088	0.182
`tak`	0.038	0.072	0.164	30.510
`zebra`	0.026	0.020	0.044	0.060
GNU Prolog speedup		1.2	2.4	5.7

Table 1: GNU Prolog versus other Prolog systems

5. CONSTRAINT SOLVING

Constraint Programming is a widely successful extension of Logic Programming, which has proved to have a significant impact for 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.

5.1 The FD Constraint System

The FD constraint system is a general purpose constraint framework for solving discrete constraint satisfaction problems (CSPs). It has been originally proposed by Pascal Van Hentenryck in a concurrent constraint setting [12], an efficient implementation in the `clp(FD)` system is described in [5; 6]. FD is based on a single *primitive constraint* by which complex constraints are defined, so for example constraints such as $X = Y$ or $X \leq 2Y$ are *defined* by 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 X to belong 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`)`, that represents the current domain of Y;
- `min(`Y`)`, the minimal value of the current domain of Y;
- `max(`Y`)`, the maximal value of the current domain of Y.

When an X `in` r constraint uses an indexical 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 equation 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:

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

Observe that this translation has also 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 (that is, decreases), the domain of X gets reduced. If instead 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, one can express in the constraint definition (that is, 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, full or partial look-ahead, depending on the use of `dom` or `min/max` indexical terms.

5.2 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. Indeed, the basic X `in` r primivite does not offer a way to define reified constraints (except via a C user function) and does not allow the user to control the propagation triggers. The need of symbolic constraints like `element/3` also enhanced the need of handling list of variables at the primitive level. Due to restricted space in this paper we simply present examples of constraint definitions with this language.

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 */
}
```

The head defines, in a C-like syntax, 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.

Let us consider a more complex example and define so-called reified constraints by using the facility offered by the language to delay the activation of an X `in` r constraint. 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 */
}
```

The constraint definition language is compiled to C by the `fd2c` sub-compiler. Each constraint gives rise to a C function returning a boolean depending on the issue 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 to the corresponding C function. For instance, to define the previous constraint one would declare a predicate `'x=c <=> b'/3` as follows :

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

The C source file obtained from an FD definition file is submitted to the C compiler to obtain an object which is then included by the linker.

5.3 Benchmarking FD constraints

The performances of the FD constraint solver of GNU Prolog is the same as `clp(FD)` [6], that is, 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 the one of SICStus Prolog. As the syntax and the set of predefined constraints is 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 the not their expressive power) we selected benchmarks with a similar formulation in both systems for which we can expect both solvers to perform the same computations. Table 2 presents execution times for both solvers and the speedup for GNU Prolog. On average GNU Prolog is around 4 times faster than SICStus Prolog.

Program	GNU Prolog 1.0.5	Sicstus Prolog 3.7.1	Speedup
`crypta`	0.008	0.012	1.5
`eq10`	0.006	0.020	3.3
`eq20`	0.010	0.030	3.0
`donald`	0.210	0.820	3.9
`alpha`	0.450	2.880	6.4
`alpha ff`	0.010	0.030	3.0
`queens 16`	0.050	0.270	5.4
`cars` all	0.015	0.060	4.0

Table 2: GNU Prolog FD solver versus SICStus FD solver

6. 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 with also many extensions very useful in practice (global variables, OS interface, sockets,etc). The finite domain constraint part of GNU Prolog contains all classical arithmetic and symbolic constraints, and integrates also an efficient treatment of reified constraint and boolean constraints. GNU Prolog produces native binaries and the executable files produced are stand alone. The size of those executable files can be quite small since GNU Prolog can avoid to link the code of most unused built-in predicates. The performances of GNU Prolog are close to commercial systems, both in the Prolog and the Constraint parts.

7. 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.