Prolog and Compiling

Pranav
IMT2018020
IIIT Bangalore
Pranav.Kumar@iiitb.ac.in

Abhigna
IMT2018002
IIIT Bangalore
Abhigna.Banda@iiitb.ac.in

Udith Sai
IMT2018081
IIIT Bangalore
Udith.Sai@iiitb.ac.in

Abstract

We have used Prolog in the course purely as an interpreted language but it is possible to create standalone executables from Prolog source code. Our study aims to dive deep into the aspects of compilation in Prolog and the compiler architecture.

1. Introduction

What is compiled language and interpreted language? We know every program is a set of instructions, whether it's to add two numbers or send a request over the internet. Compilers and interpreters take human-readable code and convert it to computer-readable machine code. With a simple example we can understand the difference between the both. Suppose you have a hummus recipe that you want to make, but it's written in ancient Greek. There are two ways we can understand the recipe. The first is someone who directly translates the whole recipe to a readable language (suppose English). We can read the whole recipe and create the hummus. We can think of this as a Compiled version. The second option is having a friend who knows the ancient Greek and translates line by line while we prepare the hummus. We can think of this as an interpreted version where our friend is an interpreter.

Compiled Language

Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages. They also give the developer more control over hardware aspects, like memory management and CPU usage. Compiled languages need to be compiled manually and also the whole program should be compiled if there is any change in any step. Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Interpreted Language

Interpreters run through a program line by line and execute each command. If there is some change in any step then the interpreter takes care while running the program. Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time compilation, that gap is shrinking. Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

Advantages and Disadvantages

- **Advantages of Compiled Languages:** Programs that are compiled into native machine code tend to be faster than interpreted code. This is because the process of translating code at run time adds to the overhead, and can cause the program to be slower overall.
- **Disadvantages of compiled languages:** Additional time needed to complete the entire compilation step before testing. Platform dependence of the generated binary code.
- **Advantages of interpreted languages:** Interpreted languages tend to be more flexible, and often offer features like dynamic typing and smaller program size. Also,

because interpreters execute the source program code themselves, the code itself is platform independent.

- **Disadvantages of interpreted languages:** The most notable disadvantage is typical execution speed compared to compiled languages.

Most programming languages can have both compiled and interpreted implementations – the language itself is not necessarily compiled or interpreted. However, for simplicity's sake, they're typically referred to as such. Python, for example, can be executed as either a compiled program or as an interpreted language in interactive mode. On the other hand, most command line tools, CLIs, and shells can theoretically be classified as interpreted languages.

1.1 What about Prolog?

The Prolog language can be compiled and can be interpreted so the answer is both. Colmer Auer and his team built the first interpreter, and David Warren at the AI Department, University of Edinburgh, produced the first compiler. Typically, the trend is to develop and test basic programming constructs using the REPL (read-eval-print loop) and then move them into source code files that will be compiled to build up libraries. The libraries are then referenced from the REPL; rinse and repeat.

1.2 Interpretation in Prolog

Prolog is a logic programming language in the sense that its statements are interpreted as sentences of logic. LISP is a general-purpose programming language based on Church's lambda calculus. By the same token, Prolog is a programming language based on predicate

calculus. A Prolog program is a sequence of rules that can be viewed initially as parameter less procedures that call other procedures.

(rule) ::= (clause). | (unit clause).

(clause) ::= (head) :- (tail)

(head) ::= (literal)

(tail) ::= (literal) {,(literal)}

(unit clause) ::= (literal)

There are three ways of interpreting the semantics of Prolog rules and queries. The first is based on logic, in this particular case, on Boolean algebra. In a second interpretation of a Prolog rule, we assume that a (literal) is a goal to be satisfied. Finally, in a third interpretation we invoke the similarity between Prolog rules and context-free grammar rules. A Prolog program is associated with a context-free grammar in which a (literal) is a nonterminal and a (rule) corresponds to a grammar rule in which the (head) rewrites into the (tail); a unit clause is viewed as a grammar rule in which a nonterminal rewrite into the empty symbol t.

1. a :- b, c, d.
2. a :- e, f.
3. b :- f.
4. e.
5. f.
6. a :- f.

Here for example let us assume alphabets as literals. This is a simple example of a database. The rules will be stored sequentially in a database implemented as a one-dimensional array Rule [1, n] and contains pointers to a special type of linear list. Such a list is a record with two fields. the first storing a letter, and the second being either nil or a pointer to a linear list. Let the function cons be the constructor of a list element, and assume that its fields are accessible via the functions head and tail. The first rule and fifth rule defining a unit clause is stored in the database by

Rule[1] := cons (‘a’. cons (‘b’, cons (‘c’, cons (‘d’, nil)))).

Rule [5] := cons (‘e’. nil).

In Interpreter, we will create a knowledge base file (which is like database) where we store all the rules. Using the ‘consult’ command we access the knowledge base file in interpreter then we will be making queries in the interpreter. When a query is asked, the interpreter takes each clause in query and searches in the knowledge base in an order from top to bottom. So, there is even importance in order of rules in the knowledge base. When there is any change in the KB, then we need to consult the KB again using the interpreter.

2. Compilation in Prolog

For efficiency, the written Prolog code is compiled to abstract machine code. This abstract machine code is often influenced by the Warren Abstract machine Set (WAM). By compiling Prolog code to more low-level WAM code, the Prolog code is reasonably easy to translate to WAM instructions, as it can be more efficiently interpreted.

The compilation of Prolog to the WAM is the standard well known process. However, for the such compilation to be done WAM code cannot be implemented directly on computers and it requires certain modifications to be done to it. One approach would be to directly write the WAM code with an emulator attached to written in C.

The other approach would be to translate Prolog to C, to keep it simple and not stagger the performance greatly. The *wamcc* Prolog compiler uses this approach and it also added the functionality to translate a WAM branching into a native code jump in order to reduce the overhead by calling a C function. There was a downside to this approach. The size of the C file generated and the time taken to compile large files by C compilers such as (gcc) can affect the entire process. A Prolog program produces a huge number of WAM instructions and to inline each WAM instruction could lead to a very big C file that is too large to be handled by the C compiler.

One such compiler that helps us tackle these drawbacks is the **GNU Prolog Compiler** which we will dive deep into to in the further sections. The standout feature of the GNU Prolog compiler is that it can produce stand-alone executables.

3. The GNU Prolog Compiler

3.1 Introduction

GNU Prolog (also called **gprolog**) is a compiler developed by Daniel Diaz with an interactive debugging environment for Prolog available for Unix, Windows, Mac OS X and Linux.

The main improvement to the GNU Prolog compilation scheme is to translate a WAM file into a mini-assembly language which is machine-independent. With the GNU compiler the Prolog program is compiled to native code which then gives a machine dependent executable. This

is the usual native code compilation. The native-code does not provide ease to fully debug the code. GNU Prolog produces three types of code.

(i) interpreted code (ii) byte-code (iii) native-code

By default, the GNU Prolog compiler produces native code but from the command-line it can produce a file ready for **byte-code** loading.

GNU Prolog manages interpreted code using a Prolog interpreter written in Prolog.

3.2 Compilation Scheme

As previously discussed, the main design decision adopted in GNU Prolog was to use a simple WAM and to compensate for the lack of optimizations by producing native code, thereby avoiding the overhead of an emulator.

The compilation process is the key point of GNU Prolog.

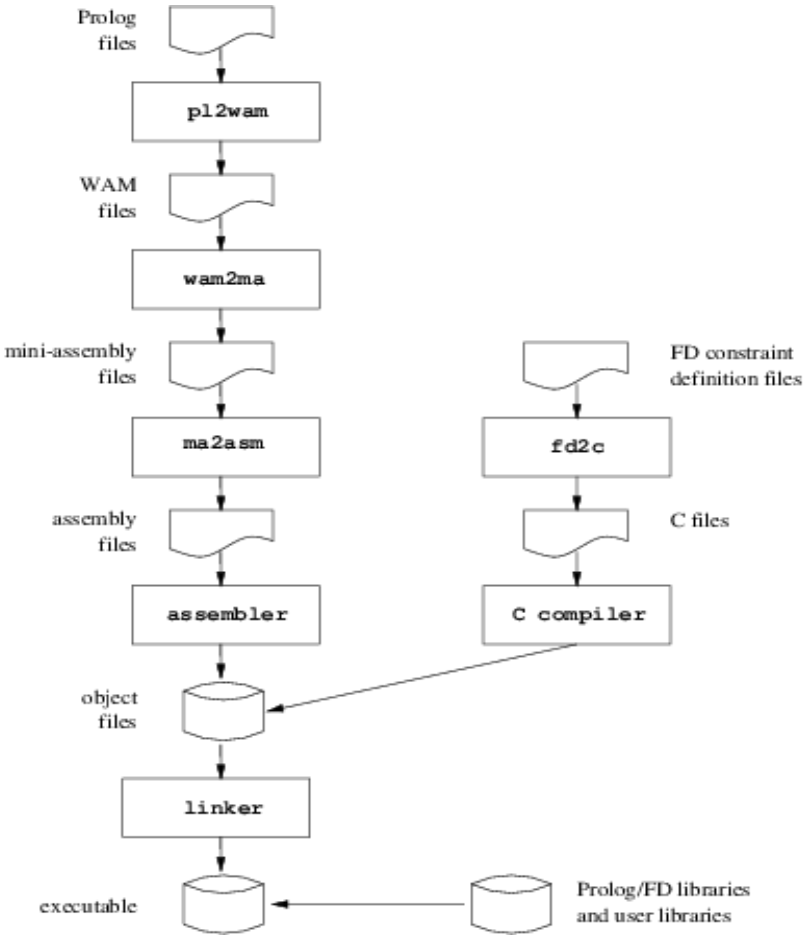
In the earlier discussed compiler *wamcc* , the native code was produced via C. A Prolog file was translated into a WAM file, which itself was translated to C and finally to object code by the C compiler. The drawback, as mentioned, was the time needed to compile the file. GNU Prolog helps in sidestepping this issue by not compiling to C but instead directly generates assembly code.

This direct translation from the WAM to assembly code takes a lot of significant effort. This issue is simplified by defining an intermediate language called **Mini-Assembly (MA).**

MA is a machine-independent assembly language which is well suited to be the target of a WAM translator. The compiler is split into several passes, with distinct executables for which

the respective intermediate representations (IRs) must be materialized as plain text or file streams.

The flowchart below describes the compilation scheme in the GNU Prolog.



pl2wam *compiles a Prolog source filw into WAM code.*

wam2ma converts *the WAM code to the MA language.*

ma2asm *translates the abstract MA code to architecture specific machine instructions.*

fd2c compiles *FD constraint definitions into C functions which perform constraint propagation at runtime.*

Compilation in Prolog is done in several stages.

- The Prolog file is compiled to get a WAM file.
- This WAM file is translated to a mini-assembly language (machine independent language.)
- This file is then mapped to the assembly language of the target machine and the mapping is in the form of assembly files.
- Now this assembly file is assembled to give an object file.
- The object file is linked to GNU Prolog libraries using a linker to give an executable file.
- The compiler also takes into account Finite domain constraint definition files.
- It translates them to C files.
- It invokes the C compiler to obtain object files.

WAM cannot be executed directly on mainstream computers and require some treatment to become executables.

Initially WAM code was directly executed with an emulator (A device **emulator** is a program or device that enables a computer system to behave like another device. An **emulator** essentially allows one computer system (aka "the host") to imitate the functions of another (aka "the guest"). written in C. They also directly compiled in to native code. They also translated Prolog to C (eg: wamcc Prolog Compiler) but this approach had a big disadvantage. The code produced to C was very large and took a lot of time for compilation. This is because when Prolog is converted to WAM, a lot of instructions are generated in WAM that ultimately convert to C.

3.2.1 Compiling Prolog to WAM

The pl2wam sub-compiler takes a Prolog file and produces a WAM file. This is written in Prolog. It produces the text files with a representation of WAM programs. The WAM file produced in a text file format makes it possible to pass through any emulator. The main advantage of using this sub-compiler is that it does register optimization, unification reordering, in lining and last sub term optimization.

3.2.2 WAM to Mini-Assembly

Mini-Assembly language is the machine independent language. Wam2ma sub-compiler translates WAM file to MA file. The MA language has been designed to avoid the use of the C stage. The MA instruction set is simple with only 11 instructions. We could use C functions for certain operations instead of using MA instructions. But there is a trade-off between minimality of the instruction set and the performance. The translation can be done in linear time proportional to the size of the WAM file. The MA uses only classical identifiers and each identifier is associated with a predicate. These predicates are represented in hexadecimal form.

3.2.3 MA to assembly language

The ma2asm sub-compiler maps the MA file to the assembly language. This mapping is easy as the number of instructions set in MA is very less. Initially mapper was written with the help of a C file produced by wamcc. Now the approach is independent of wamcc and each MA instruction corresponds to the C code. Further optimization oof assembly code is done with the help of Application Binary Interface used by OS. The translation can be done in linear time proportional to the size of the MA file. This translation makes several optimizations like short branching detection possible.

3.2.4 Assembly to Object files

The assembler is used to convert an assembly file into an object file. The assembly optimizer and the compiler are used to convert, respectively, a linear assembly file and a C file into an object file.

3.2.4 Finite Domain Constraint System in Prolog

Constraint Programming is a widely successful extension of Logic Programming, which has had a significant impact on a variety of industrial applications. It is thus natural to include a constraint solving extension to any modern Prolog-based system. This system helps in translating at compile-time all complex user-constraints (e.g., Dis equations, linear equations or inequations) into simple, primitive constraints at a lower level which really embeds the propagation mechanism for constraint solving. The FD constraint system is a general-purpose constraint framework for solving discrete constraint satisfaction problems (CSPs). FD is based on a single primitive constraint through which complex constraints are defined. A constraint is a formula of the form X in r where X is a variable and r is 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. 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. For example, X≤Y is a complex constraint.

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

Here when there is a change in value of Y, there is a change in the range of X. Similarly, when there is a change in the value of X, there is a change in the range of Y. One can therefore consider those primitive X in r constraints as a low-level language in which the

propagation scheme has to be expressed. In GNU Prolog there is 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 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.

3.2.5 Linking

In this stage, all the object files are linked with the GNU Prolog libraries. The Prolog libraries include built-in library, FD built-in constraint/predicate library and runtime library. Several objects can be grouped in a library. The concept of library makes it possible to extract only the object files that are necessary. For this reason, GNU Prolog can generate small executables by avoiding the inclusion of most unused built-in predicates. This helps in reducing the size of final executables. For the runtime library, the Prolog engine must be able to dynamically find the objects selected by linker at runtime and must be able to execute their initializer function. One approach is to mark the object files with magic numbers together with the address of the initializer function. At runtime data segment is scanned and invokes relevant initializers. The other approach is for each object the address of the initializer function is stored in a specific data segment and the linker will group all the data defined in the same section. At runtime, the section is scanned and initializer is invoked.

4. Conclusion

We have presented the most significant aspects of GNU Prolog for which the key issues were simplicity and extensibility without sacrificing performance. The implemented native-code approach has been thoroughly studied. GNU Prolog comes with an amazing list of features.

The standout feature of GNU Prolog is that it can produce stand-alone executables. Stand-alone executable are programs that can be invoked on it's on the command prompt and does not depend on any other program. This might not be possible in other Prolog Systems as they require the presence of the emulator at run-time. The size of the executables produced in GNU Prolog can be small as it can avoid linking the code with unused built-in predicates. It is a compiler with constraint solving over finite domains.

We have thus taken **GNU Prolog** as the basis of our study to show the architecture of a compiler that can produce stand-alone executables.

5. Acknowledgement

We would like to thank our Professor Sujit Kumar Chakrabarti for giving us a giving us the opportunity to study and work on such a fascinating and intriguing project topic. We would also like to take this opportunity to thank our dear Professor for giving us a significant head start in the field of Programming Languages

A heartful thanks to our TAs for helping us out by answering to our queries and assisting us when requested.

6. References

- https://dl.acm.org/doi/pdf/10.1145/214956.214960?casa_token=6Rfx6-VP_yMAAAAA:1uHXHL9OlM6hgnyA0ky5xjHaITpCf0244zZjavjUy7p-4Lmd9HsS4xB8qsSmgdOu8AoBL-CL_SS2
- https://en.wikipedia.org/wiki/Prolog
- https://en.wikipedia.org/wiki/Warren_Abstract_Machine

- http://www.gprolog.org/