Compiler Conrstuction Report
Prolog

Tareq Kirresh(TK),Alaa Shuqair(AS), Dua Abu Safiah(DAS), Ashjan Bakri(AB)

October 6, 2017

Contents

1 History

Prolog was designed in the 1970s by Alain Colmerauer and a team of researchers with the idea – new at that time – that it was possible to use logic to represent knowledge and to write programs. More precisely, Prolog uses a subset of predicate logic and draws its structure from theoretical works of earlier logicians such as Herbrand (1930) and Robinson (1965) on the automation of theorem proving. Prolog was originally intended for the writing of natural language processing applications. Because of its conciseness and simplicity, it became popular well beyond this domain and now has adepts in areas such as:

- Formal logic and associated forms of programming
- Reasoning modeling
- Database programming
- Planning, and so on.

Colmerauer started his work at the University of Montréal, and a first version of the language was implemented at the University of Marseilles in 1972. Colmerauer

and Roussel (1996) tell the story of the birth of Prolog, including their try-and-fail experimentation to select tractable algorithms from the mass of results provided by research in logic.

In 1995, the International Organization for Standardization (ISO) published a standard on the Prolog programming language. Standard Prolog (Deransart et al. 1996) is becoming prevalent in the Prolog community and most of the available implementations now adopt it, either partly or fully.

Prolog evolved out of research at the University of Aix-Marseille back in the late 60's and early 70's. Alain Colmerauer and Phillipe Roussel, both of University of Aix-Marseille, colaborated with Robert Kowalski of the University of Edinburgh to create the underlying design of Prolog as we know it today. Kowalski contributed the theoretical framework on which Prolog is founded while Colmerauer's research at that time provided means to formalize the Prolog language.

1972 is referred to by most sources as the birthdate of Prolog. Since its birth it has branched off in many different dialects. Two of the main dialects of Prolog stem from the two Universities of its origin: Edinburgh and Aix-Marseille. At this time the first Prolog interpreter was built by Roussel. The first Prolog compiler was credited to David Warren, an expert on Artificial Intelligence at the University of Edinburgh.

To this day Prolog has grown in use throughout North America and Europe. Prolog was used heavily in the European Esprit programme and in Japan where it was used in building the ICOT Fifth Generation Computer Systems Initiative. The Japanese Government developed this project in an attempt to create intelligent computers. Prolog was a main player in these historical computing endeavours.

Prolog became even more pervasive when Borland's Turbo Prolog was released in the 1980's. The language has continued to develop and be used by many scientists and industry experts. Now there is even and ISO Prolog standardisation (1995) where all of its individual parts have been defined to ensure that the core of the language remains fixed.

2 Language Syntax and Sematics

2.1 Data Types

Prolog's single data type is the term. Terms are either atoms, numbers, variables or compound terms.

- An atom is a general-purpose name with no inherent meaning. Examples of atoms include x, red, 'Taco', and 'some atom'.
- Numbers can be floats or integers. ISO standard compatible Prolog systems can check the Prolog flag "bounded". Most of the major Prolog systems support arbitrary precision integer numbers.
- Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A compound term is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. Examples of compound terms are truck_year

  ```
  ('Mazda', 1986)
  ```

  and 'Person_Friends'

  ```
  (zelda,[tom,jim]).
  ```

  Special cases of compound terms:

  - Lists: A List is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, []. For example, [1,2,3] or [red,green,blue].
  - Strings: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding, or Unicode if the system supports Unicode. For example, "to be, or not to be".

2.2 Rules and Facts

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: facts and rules. A rule is of the form

```
Head :- Body.
```

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's goals. The built-in predicate ,/2 (meaning a 2-arity operator with name ,) denotes conjunction of goals, and ;/2 denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

3 Examples

3.1 Simple Facts in a Knowledge Base

Lets say we have this knowledge base:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

by posing the query :

```
woman(mia).
```

we will get

```
yes
```

This is because we explicitly recorded in the knowldege base above. Similarly, if we pose this query :

```
playsAirGuitar(jody).
```

we will, again, get

```
yes
```

Because once more, this is a fact we stated in the knowledge base we are querying. Say we pose this query :

```
tatooed(jody).
```

We will get the answer

```
no
```

This is because we do not have this statement in our knowledge base, so Prolog will return no, since it doesn't know if it is true or not.

3.2 Simple Rules in a Knowledge Base

Say we have this Knowledge Base :

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2Music(mia).
playsAirGuitar(yolanda):- listens2Music(yolanda).
```

The First 2 statements are facts, the last 3 statements are rules. Suppose we pose this query :

```
playsAirGuitar(mia).
```

Prolog will respond yes. Why? Well, although it can't find playsAirGuitar(mia) as a fact explicitly recorded in KB2, it can find the rule

```
playsAirGuitar(mia):- listens2Music(mia).
```

Moreover, KB2 also contains the fact listens2Music(mia) . Hence Prolog can deduce that playsAirGuitar(mia) . lets pose this query :

```
playsAirGuitar(yolanda).
```

Prolog would respond yes. Why? Well, first of all, by using the fact happy(yolanda) and the rule

```
listens2Music(yolanda):- happy(yolanda).
```

Prolog can deduce the new fact listens2Music(yolanda) . This new fact is not explicitly recorded in the knowledge base — it is only implicitly present (it is inferred knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. In particular, from this inferred fact and the rule

```
playsAirGuitar(yolanda):- listens2Music(yolanda).
```

it can deduce playsAirGuitar(yolanda) , which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

3.3 Defining a Concept with a Knowledge Base

Say we have this knowledge base :

```
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).

jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

In effect, it is defining a concept of jealousy. It says that an individual X will be jealous of an individual Y if there is some individual Z that X loves, and Y loves that same individual Z too. The key thing to note is that this is a general statement: it is not stated in terms of mia , or pumpkin , or anyone in particular — it's a conditional statement about everybody in our little world. Lets pose this query :

```
jealous(marsellus,W).
```

This query asks: can you find an individual W such that Marsellus is jealous of W ? Vincent is such an individual. If you check the definition of jealousy, you'll see that Marsellus must be jealous of Vincent, because they both love the same woman, namely Mia. So Prolog will return the value :

```
W = vincent
```

3.4 Recursive Definitions in a Knowledge Base

Say we have this definition in a knowledge base :

```
gcd(u,v,u) :- v = 0.
gcd(u,v,x) :- v > 0,
y is u mod v,
gcd(v,y,x).
```

What this effectively does is express, using first order logic, the states we go through in order to find the greatest common divisor for a number recursively.

4 Modern Uses of Prolog

4.1 Artificial intelligence

Artificial intelligence is the branch of computer science concerned with making computers behave like humans. Prolog is a programming language centred around a small set of basic mechanisms, including pattern matching, tree-based data structuring and automatic backtracking. This small set constitutes a surprisingly powerful and flexible programming framework. Prolog is especially well suited for problems that involve objects - in particular, structured objects - and relations between them. Features like this make Prolog a powerful language for artificial intelligence (AI) and non-numerical programming in general.

Prolog being easier to use for teaching students Al. Lisp has been the primary language of artificial intelligence for many years, but it is a low-level language, too low for most students. Prolog has three positive features that give it key advantages over Lisp. First, Prolog syntax and semantics are much closer to formal logic, the most common way of representing facts and reasoning methods used in the artificial intelligence literature. Second, Prolog provides automatic backtracking, a feature making for considerably easier "search", the most central of all artificial intelligence techniques. Third, Prolog supports multidirectional (or multiuse) reasoning, in which arguments to a procedure can freely be designated inputs and outputs in different ways in different procedure calls, so that the same procedure definition can be used for many different kinds of reasoning. Besides this, new implementation techniques have given current versions of Prolog close in speed to Lisp implementations, so efficiency is no longer a reason to prefer Lisp.

4.2 Automated Theorem Proving

Automated Theorem Proving, also known as ATP or automated deduction, is a subfield of automated reasoning and mathematical logic dealing with proving mathematical theorems by computer programs. Automated reasoning over mathematical proof was a major impetus for the development of computer science.

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus. It differs from Prolog in its use of unification with the occurs check for soundness, the model-elimination reduction rule that is added to Prolog inferences to make the inference system complete, and depth-first iterative-deepening search instead of unbounded depth-first search to make the search strategy complete. A Prolog technology theorem prover has been implemented by an extended Prolog-to-LISP compiler that supports these additional features. It is capable of proving theorems in the full first-order predicate calculus at a rate of thousands of inferences per second.

4.3 Expert Systems

In Artificial Intelligence, an expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented mainly as if–then rules rather than through conventional procedural code.

Features useful for expert systems:

- Inputs and updates During a query evaluation data can be input from a user, and assertions and rules can be added to the program, using special primitive relations. The evaluation of read(x) will cause x to be bound to the next term typed at the input terminal. The evaluation of assert(x) will cause the term which is the current binding of x to be added as a new rule. Thus, the rule :

  ```
  Ask-about(c) if print (c, "?") \& read (ans)
        \& ans = Yes \& assert (c),
  ```

  used to try to answer the query :

  ```
  Ask-about (Dirt is-fault-with Carburettor),
  ```

  will print the message :

  ```
  Dirt is-fault-with Carburettor?,
  ```

read the next term, and if it is the constant Yes it will add :

```
Dirt is-fault-with Carburettor
```

as a new assertion about the is-fault-with relation. Where this assertion is added is important. It can be added at the beginning of the list of clauses for the relation, or the end, or at some intermediate position. In this situation we would like it added at the beginning. Where it is added is an option that can be specified by the programmer. We shall not go into the details of this.

- Dynamic data base The rule :

```
u my is-fault-with x if y is-part-of x \& u is-fault-with y
```

must access assertions giving components and assertions about faults with components. We can use the Ask-about rule to allow the assertions about faults to be added dynamically as we try to solve the problem of finding a fault.

Instead of assertions about known faults with components we include in the initial data base only assertions about possible faults, knowledge that expert should have. We then include the rule :

```
u is-a-fault-with y if u is-a-poss-fault-with y \& Ask-about (u is-a-fault-with y)
```

Let us pause for a moment to consider the behaviour of our fault finder. When asked to find a fault with some device with a query

```
u is-fault-with Device
```

the use of the first rule for faults will cause the fault finder to walk over the structure of Device as described by the is-part-of assertions. When it reaches an atomic part it will query the user concerning possible faults with this component as listed in the is-poss- fault-with assertions. It will continue in this way, backtracking up and down the structure, until a fault is reported. As it currently stands, our expert system helps the user to look for faults.

- Generation of lemmas Sometimes it is useful to record the successful evaluation of an atomic query B by adding its derived substitution instance as a new assertion. Thus, suppose we have a rule :

```
R(tL.. ,tn) if Al\& . .\&Ak
```

and we want to generate a lemma each time it is successfully used to answer a query R(t' 1, , en). We add an extra assert condition at the end of the list of preconditions of the rule :

```
R (tl, . . ,tn) if Al\& . .\&Ak \& assert (R(tl, . . , tn))
```

If this solves the atomic query with answer substitution s then [R , en* will be added as new assertion. It will be added at the front of the sequence of clauses for R.

By adding asserts we can also simulate the use of rules with conjunctive consequences. Suppose that we know that both B and B' are implied by the conjunction Al& . . &An. Normally we would just include the two rules :

```
B if Al\& . .\&An
B' if Al\& . . \&An
```

in the program. The drawback is that we need to solve Al&..&An twice in derivations where both B and B' are needed. We can avoid the duplication by writing the two rules as :

```
B if Al\& ..\&An\&assert(B')
B' if Al\& . . \&An \&assert (B)
```

The successful use of either rule will add the other consequent as a lemma.

By developing a suitable front end to PROLOG we can shield the programmer from the details of the lemma generation. We could allow him to write rules with conjunctive consequents and to specify which rules were lemma generation rules. The front end program would expand rules with conjunctive consequents into several rules and add the appropriate asserts to the end of each of these rules. It would also add an assert to the end of eich of the lemma rules.

- All solutions

  Sometimes we want to find all the answers to a query, not just one answer. This is an option in some of the PROLOGs. Where it is not we can make use of a meta-rule such as

  ```
  All (query, term) if query \& print (term) \& fail.
  ```

  This will print out [term]s for each answer substitution s to query. The "fail" is a condition for which we assume there are no clauses. With a slightly modified rule, we can define the relation :

  ```
    is-all term such-that query
  ```

  that holds when I is the list of all the instantiations of "term" for answer substitutions to "query". In DEC-10 PROLOG such a relation is now a primitive of the language. Using it we can write rules such as

  ```
  I is-a-list-of-faults-with x if 1 is-all u such-that
  u is-fault-with x
  ```

  The use of this rule will result in 1 being bound to a list of all the faults with x. We can now consider a very simple extension to our fault finder. Instead of asking for a single fault we can ask for a list of all the reported faults with corrective actions. We do this with a query of the form :

  ```
   is-all [u a] such-that
  u is-fault-with D \&a is-action-for u.
  ```

  To handle this query we must also include in our database a set of assertions giving actions for faults, information supplied by the expert. An evaluation of this new query will guide the user through the structure of device D, asking about faults in components. Each reported fault will be paired with its corrective action. Finally the list of pairs [reported-fault corrective-action] will be printed.

- Term rewriting

  A term rewriting system (TRS) is a rewriting system where the objects are terms, or expressions with nested sub-expressions. Term rewriting is a surprisingly simple computational paradigm that is based on the repeated application of simplification rules. It is particularly suited for tasks like symbolic computation, program analysis and program transformation. Understanding term rewriting will help you to solve such tasks in a very effective manner.

- Type inference: Type inference refers to the automatic deduction of the data type of an expression in a programming language.

4.4 Automated Planning and scheduling

Automated planning and scheduling, sometimes denoted as simply AI Planning,[1] is a branch of artificial intelligence that concerns the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots and unmanned vehicles. Unlike classical control and classification problems, the solutions are complex and must be discovered and optimized in multidimensional space. Planning is also related to decision theory.

4.5 Natural language processing

Natural language processing (NLP) is a field of computer science, artificial intelligence and computational linguistics concerned with the interactions between computers and human (natural) languages, and, in particular, concerned with programming computers to fruitfully process large natural language corpora.

5 References

http://fileadmin.cs.lth.se/cs/Personal/Pierre_Nugues/ilppp/chapters/appA.pdf.
http://www.mta.ca/~rrosebru/oldcourse/371199/prolog/history.html.
http://www.doc.ic.ac.uk/~rak/papers/the%20early%20years.pdf.
https://pdfs.semanticscholar.org/1dba/24320c41fa5eae94c51fd971085bc8176dd7.pdf
https://link.springer.com/article/10.1007%2FBF00297245?LI=true
https://en.wikipedia.org/wiki/Automated_theorem_proving#First_implementations
https://calhoun.nps.edu/bitstream/handle/10945/36984/Rowe_book_AI_thr_Prolog_preface.pdf https://softwa
https://en.wikipedia.org/wiki/Expert_system
https://en.wikipedia.org/wiki/Natural_language_processing
https://en.wikipedia.org/wiki/Automated_planning_and_scheduling
https://en.wikipedia.org/wiki/Type_inference https://pdfs.semanticscholar.org/1dba/24320c41fa5eae94c51f
lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse1
https://github.com/TareqK/Compiler-Construction-Course/blob/master/Notes/introduction.markdown