A Hitchhiker’s Guide to Reinventing a Prolog Machine

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

ICLP’17
Deriving the execution algorithm
The meta-interpreter `metaint/1` uses a (difference)-list view of prolog clauses.

Prolog: the two-clause meta-interpreter

The meta-interpreter `metaint/1` uses a (difference)-list view of prolog clauses.

- `metaint([]).` % no more goals left, succeed
- `metaint([G|Gs]):-` % unify the first goal with the head of a clause
  - `cls([G|Bs],Gs),` % build a new list of goals from the body of the
    - % clause extended with the remaining goals as tail
  - `metaint(Bs).` % interpret the extended body

- Clauses are represented as facts of the form `cls/2`
- The first argument representing the head of the clause + a list of body goals
- Clauses are terminated with a variable, also the second argument of `cls/2`.

```
cls([ add(0,X,X) |Tail],Tail).
cls([ add(s(X),Y,s(Z)), add(X,Y,Z) |Tail],Tail).
cls([ goal(R), add(s(s(0)),s(s(0)),R) |Tail],Tail).

?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```
The “natural language equivalent” of the equational form

As the recursive tree structure of a Prolog is flattened, it makes sense to express it as an equivalent “natural language-looking” sentence.

\[
\text{add } SX \ Y \ SZ \text{ if } SX \text{ holds } s \ X \text{ and } SZ \text{ holds } s \ Z \text{ and add } X \ Y \ Z.
\]

- note and the correspondence between the keywords “if” and “and” to Prolog’s “: −” clause neck and “, ” conjunction symbols
- the keyword “holds” (like the use of Prolog’s “=” ) expresses a unification operation between a variable and a flattened Prolog term
- the toplevel skeleton of the clause can be kept implicit as it is easily recoverable
- this is our “assembler language” to be read in directly by the loader of a runtime system
- a simple tokenizer splitting into words sentences delimited by “.” is all that is needed to complete a parser for this English-style “assembler language”
The heap representation as executable code
we instruct our tokenizer to recognize variables, symbols and (small) integers as primitive data types
we develop a Java-based interpreter in which we represent our Prolog terms top-down
Java’s primitive `int` type is used for tagged words
in a C implementation one might want to chose `long long` instead of `int` to take advantage of the 64 bit address space
we instruct our parser to extract as much information as possible by marking each word with a relevant tag
The top-down representation of terms

\[\text{add}(s(X), Y, s(Z)) :\neg\text{add}(X, Y, Z).\]

compiles to

\[\text{add }_0 Y _1\text{ and } _0\text{ holds } s X\text{ and } _1\text{ holds } s Z\text{ if } \text{add } X Y Z.\]

- on the heap (starting in this case at address 5):

  \[
  \]

- distinct tags of first occurrences (tagged “\(v:\)”) and subsequent occurrences of variables (tagged “\(u:\)”).

- references (tagged “\(r:\)”) always point to arrays starting with their length marked with tag “\(a:\)”.

- cells tagged as array length contain the arity of the corresponding function symbol incremented by 1

- the “skeleton” of the clause in the previous example is shown as:

  \[r:5 :- [r:16]\]
Clauses as descriptors of heap cells

- the parser places the cells composing a clause directly to the heap
- a descriptor (defined by the small class `Clause`) is created and collected to the array called “clauses” by the parser
- an object of type `Clause` contains the following fields:
  - `int base`: the base of the heap where the cells for the clause start
  - `int len`: the length of the code of the clause i.e., number of the heap cells the clause occupies
  - `int neck`: the length of the head and thus the offset where the first body element starts (or the end of the clause if none)
  - `int[] gs`: the toplevel skeleton of a clause containing references to the location of its head and then body elements
  - `int[] xs`: the index vector containing dereferenced constants, numbers or array sizes as extracted from the outermost term of the head of the clause, with 0 values marking variable positions.
Execution as iterated clause unfolding
The key intuition: we emulate (procedurally) the meta-interpreter

- as the meta-interpreter shows it, Prolog’s execution algorithm can be seen as iterated unfolding of a goal with heads of matching clauses
- if unification is successful, we extend the list of goals with the elements of the body of the clause, to be solved first
- as we do not assume anymore that predicate symbols are non-variables, it makes sense to design indexing as a distinct pre-unification step: detecting matching clauses without copying to the heap
- one can filter matching clauses by comparing the outermost array of the current goal with the outermost array of a clause head
- we use for this the prototype of a clause head without starting to place new terms on the heap
- dereferencing is avoided when working with material from the heap-represented clauses (none for first occurrences of variables, exactly once for others
Fast “linear” term relocation and the immutable goal stack

- we implement a fast relocation loop that speculatively places the clause head (including its subterms) on the heap
- this “single instruction multiple data” operation can benefit from parallel execution!
- new terms are built on the heap by the relocation loop in two stages: first the clause head (including its subterms) and then, if unification succeeds, also the body
- stretching out the Spine: the (immutable) goal stack
  - a Spine is a runtime abstraction of a Clause
  - it collects information needed for the execution of the goals originating from it
  - goal elements on this immutable list are shared among alternative branches
The execution algorithm
Our interpreter: yielding an answer and ready to resume

- it starts from a Spine and works though a stream of answers, returned to the caller one at a time, until the spines stack is empty
- it returns null when no more answers are available

```java
final Spine yield() {
    while (!spines.isEmpty()) {
        final Spine G = spines.peek();
        if (hasClauses(G)) {
            if (hasGoals(G)) {
                final Spine C = unfold(G);
                if (C != null) {
                    if (!hasGoals(C)) return C; // return answer
                    else spines.push(C);
                } else popSpine(); // no matches
            } else unwindTrail(G.ttop); // no more goals in G
        } else popSpine(); // no clauses left
    }
    return null;
}
```
Exposing the answers of a logic engine to the implementation language
Answer streams

- to encapsulate our answer streams in a Java 8 stream, a special iterator-like interface called Spliterator is used
- the work is done by the tryAdvance method which yields answers while they are not equal to null, and terminates the stream otherwise

```java
public boolean tryAdvance(Consumer<Object> action) {
    Object R = ask();
    boolean ok = null != R;
    if (ok) action.accept(R);
    return ok;
}
```
- three more methods are required by the interface, mostly to specify when to stop the stream and that the stream is ordered and sequential
Multi-argument indexing: a modular add-on
The indexing algorithm

- the indexing algorithm is designed as an independent add-on to be plugged into the main Prolog engine.
- For each argument position in the head of a clause, it associates to each indexable element (symbol, number, or arity) the set of clauses where the indexable element occurs in that argument position.
- The clauses having variables in an indexed argument position are also collected in a separate set for each argument position.
- 3 levels are used, closely following the data that we want to index.
- Sets of clause numbers associated to each (tagged) indexable element are backed by an IntMap implemented as a fast int-to-int hash table (using linear probing).
- An IntMap is associated to each indexable element by a HashMap.
- The HashMaps are placed into an array indexed by the argument position to which they apply.
when looking for the clauses matching an element of the list of goals to solve, for an indexing element $x$ occurring in position $i$, we fetch the the set $C_{x,i}$ of clauses associated to it

If $V_i$ denotes the set of clauses having variables in position $i$, then any of them can also unify with our goal element

thus we would need to compute the union of the sets $C_{x,i}$ and $V_i$ for each position $i$, and then intersect them to obtain the set of matching clauses

instead of actually compute the unions for each element of the set of clauses corresponding to the “predicate name” (position 0), we retain only those which are either in $C_{x,i}$ or in $V_i$ for each $i > 0$

we do the same for each element for the set $V_0$ of clauses having variables in predicate positions (if any)

finally, we sort the resulting set of clause numbers and hand it over to the main Prolog engine for unification and possible unfolding in case of success
Ready to run: some performance tests
Trying out the implementation

- we prototyped the design described so far as a small, slightly more than 1000 lines of “C-friendly” Java
- available at https://github.com/ptarau/iProlog
- for more details: recording https://www.youtube.com/watch?v=SRYAMt8iQSw&list=PLJq3XDLIJKib2h2fObomdFRZrQeJg4UIW of our VMSS’2016 invited tutorial
- while implemented as an interpreter, our preliminary tests indicate, very good performance

<table>
<thead>
<tr>
<th>System</th>
<th>11 queens</th>
<th>perms of 11 + nrev</th>
<th>sudoku 4x4</th>
<th>metaint perms</th>
</tr>
</thead>
<tbody>
<tr>
<td>our interpreter</td>
<td>5.710s</td>
<td>5.622s</td>
<td>3.500s</td>
<td>16.556s</td>
</tr>
<tr>
<td>Lean Prolog</td>
<td>3.991s</td>
<td>5.780s</td>
<td>3.270s</td>
<td>11.559s</td>
</tr>
<tr>
<td>Styla</td>
<td>13.164s</td>
<td>14.069s</td>
<td>22.196s</td>
<td>37.800s</td>
</tr>
<tr>
<td>SWI-Prolog</td>
<td>1.835s</td>
<td>2.620s</td>
<td>1.336s</td>
<td>4.872s</td>
</tr>
<tr>
<td>LIPS</td>
<td>7,278,988</td>
<td>7,128,483</td>
<td>9,261,376</td>
<td>6,651,000</td>
</tr>
</tbody>
</table>

Timings and number of logical inferences per second (LIPS) (as counted by SWI-Prolog) on 4 small Prolog programs
Summary and conclusions

- by starting from a two line meta-interpreter, we have captured the necessary step-by-step transformations that one needs to implement in a procedural language that mimics it
- by deriving “from scratch” a fairly efficient Prolog machine we have, hopefully, made its design more intuitive
- we have decoupled the indexing algorithm from the main execution mechanism of our Prolog machine
- we have also proposed a natural language style, human readable intermediate language that can be loaded directly by the runtime system using a minimalistic parser
- the code and the heap representation became one and the same
- performance of the interpreter based on our design was able to get close enough to optimized compiled code
- future ports of this design can help with the embedding of logic programming languages as lightweight software or hardware components