

# A Hitchhiker's Guide to Reinventing a Prolog Machine

Paul Tarau

Department of Computer Science and Engineering  
University of North Texas

CICLOPS'17

# Outline

- 1 Deriving the execution algorithm
- 2 The heap representation as executable code
- 3 Execution as iterated clause unfolding
- 4 The execution algorithm
- 5 Exposing the answers of a logic engine to the implementation language
- 6 Multi-argument indexing: a modular add-on
- 7 Ready to run: some performance tests
- 8 Summary and conclusions

# 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)))) .
```

# Deriving the execution algorithm

# The equational form of terms and clauses

## An equation like

$T = \text{add}(s(X), Y, s(Z))$

can be rewritten as a conjunction of 3 equations as follows

$T = \text{add}(SX, Y, SZ), SX = s(X), SZ = s(Z)$

## When applying this to a clause like

$C = [\text{add}(s(X), Y, s(Z)), \text{add}(X, Y, Z)]$

it can be transformed to a conjunction derived from each member of the list

$C = [H, B], H = \text{add}(SX, Y, SZ), SX = s(X), SZ = s(Z), B = \text{add}(X, Y, Z)$

The list of variables ( $[H, B]$  in this case) can be seen as as a toplevel skeleton abstracting away the main components of a Horn clause: the variable referencing the head followed by 0 or more references to the elements of the conjunction forming the body of the clause.

# The “natural language equivalent” of the equational form

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

```
add SX Y SZ if SX holds s X and SZ holds s Z and add X Y Z.
```

- note and the correspondence between the keywords “if” and “and” to Prolog’s “:-” clause neck and “,” conjunction symbols
- Note also the correspondence between the keyword “holds” and the use of Prolog’s “=” to express 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”

# Example

## Prolog code:

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :-add(X,Y,Z) .  
  
goal(R) :-add(s(s(0)),s(s(0)),R) .
```

## Natural language-style assembler code:

```
add 0 X X .  
add _0 Y _1 and _0 holds s X and _1 holds s Z if add X Y Z .  
  
goal R if add _0 _1 R and  
  _0 holds s _2 and  
  _2 holds s 0 and  
  _1 holds s _3 and  
  _3 holds s 0 .
```

# The heap representation as executable code



# Representing terms in a Java-based runtime

- 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

```
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

compiles to

```
add _0 Y _1 and _0 holds s X and _1 holds s Z if add X Y Z .
```

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

```
[5]a:4 [6]c:add [7]r:10 [8]v:8 [9]r:13 [10]a:2 [11]c:s [12]v:12  
[13]a:2 [14]c:s [15]v:15 [16]a:4 [17]c:add [18]u:12 [19]u:8  
[20]u:15
```

- 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
- thus indexing, meant to speed-up the selection of matching clauses, is orthogonal to the core unification and goal reduction algorithm
- as we do not assume anymore that predicate symbols are non-variables, it makes sense to design indexing as a distinct algorithm
- we need a convenient way to plug it in as a refinement of our iterated unfolding mechanism (we will use Java 8 streams for that)

# Unification, trailing and pre-unification clause filtering

- unification descends recursively and binds variables to terms in corresponding positions  $\Rightarrow$  full unification can be expensive!
- our relatively rich tag system reduces significantly the need to call the full unification algorithm
- pre-unification:  $\Rightarrow$  we need to filter out non-unifiable clauses quickly
- “unification instructions” can be seen as closely corresponding to the tags of the cells on the heap, identified in our case with the “code” segment of the clauses
- we will look first at some low-level aspects of unification, that tend to be among the most frequently called operations of a Prolog machine

# The 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
- the `xS` register array: a copy of the outermost array of the goal element we are working with, holding dereferenced elements in it
- used to reject clauses that mismatch it in positions holding symbols, numbers or references to array-lengths
- 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, as our tags will tell us that first occurrences of variables do not need it at all, and that other variable-to-variable-references need it exactly once as a `getRef` step

# Unification and trailing

- as we unfolded to registers the outermost array of the current goal (corresponding to a predicate's arguments) we will start by unifying them, when needed, with the corresponding arguments of a matching clause
- a dynamically growing and shrinking `int` stack is used to eliminate recursion by the otherwise standard unification algorithm
- *trailing*: to emulate procedurally the execution of the two clause meta-interpreter, we need to keep an “undo list” (the trail) for variable bindings to be undone on backtracking
- to avoid unnecessary work, variables at higher addresses are bound to those at lower addresses on the heap and after binding, variables are trailed when lower than the heap level corresponding to the current goal



# Fast “linear” term relocation

- 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
- `pushHead` copies and relocates the head of clause at (precomputed) offset `b` from the prototype clause on the heap to the a higher area where it is ready for unification with the current goal
- similar code for `pushBody`
- we also relocate the skeleton `gs` starting with the address of the first goal so it is ready to be merged in the list of goals
- a new small class `Spine` will keep track of those runtime components

# 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 small methodless `Spine` class declares the following fields:
  - `int hd`: head of the clause
  - `int base`: base of the heap where the clause starts
  - `IntList gs`: immutable list of the locations of the goal elements accumulated by unfolding clauses so far
  - `int ttop`: top of the trail as it was when this clause got unified
  - `int k`: index of the last clause the top goal of the `Spine` has tried to match so far
  - `int[] regs`: dereferenced goal registers
  - `int[] xs`: index elements based on `regs`

# The execution algorithm

## Our interpreter: yielding an answer and ready to resume

- it starts from a `Spine` and works through 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

```
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;
}
```

– continued –

- the active component of a `Spine` is the topmost goal in the immutable goal stack `gs` contained in the `Spine`
- when no goals are left to solve, a computed answer is `yield`, encapsulated in a `Spine` that can be used by the caller to resume execution
- when there are no more matching clauses for a given goal, the topmost `Spine` is popped off
- an empty `Spine` stack indicates the end of the execution signaled to the caller by returning `null`.
- a key element in the interpreter loop is to ensure that after an `Engine` yields an answer, it can, if asked to, resume execution and work on computing more answers

# Resuming the interpreter loop

- the class `Engine` defines in the method `ask()`
- the instance variable “`query`” of type `Spine`, contains the top of the trail as it was before evaluation of the last goal, up to where bindings of the variables will have to be undone, before resuming execution
- `ask()` also unpacks the actual answer term (by calling the method `exportTerm`) to a tree representation of a term, consisting of recursively embedded arrays hosting as leaves, an external representation of symbols, numbers and variables

```
Object ask() {
    query = yield();
    if (null == query) return null;
    final int res = answer(query.ttop).hd;
    final Object R = exportTerm(res);
    unwindTrail(query.ttop);
    return R;
}
```

# 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 `Splitterator` 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

```
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 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
- to be thrifter on memory, argument positions go up to a maximum that can be specified by the programmer
- for deep indexing, the argument position can be generalized to be the integer sequence defining the path leading to the indexable element in a compound term
- the clauses having variables in an indexed argument position are also collected in a separate set for each argument position

– continued –

- 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 `HashMap`s are placed into an array indexed by the argument position to which they apply

– continued –

- 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

# Indexing: two special cases

- for very small programs (or programs having predicates with fewer clauses than the bit size of a `long`)
  - the `IntMap` can be collapse to a `long` made to work as a *bit set*
  - alternatively, given our fast pre-unification filtering one can bypass indexing altogether, below a threshold
- for very large programs:
  - a more compact sparse bit set implementation or a Bloom filter-based set would replace our `IntMap` backed set, except for the first “predicate name” position, needed to enumerate the potential matches
  - in the case of a Bloom filter, if the estimated number of clauses is not known in advance, a *scalable Bloom filter* implementation can be used
  - the probability of false positives can be fine-tuned as needed, while keeping in mind that false positives will be anyway quickly eliminated by our pre-unification head-matching step
  - one might want to compute the set of matching clauses lazily, using the Java 8 streams API

# 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 generously commented Java program
- 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
- we have stayed away from Java's object oriented features
- $\Rightarrow$  a large `Engine` class hosts all the data areas
- a few small classes like `Clause` and `Spine` can be easily mapped to `C structs`
- while implemented as an interpreter, our preliminary tests indicate, very good performance
  - it is (within a factor of **2**) to our Java-based systems like Jinni and Lean Prolog that use a (fairly optimized) compiler and instruction set
  - is also within a factor of **2-4** from `C`-based SWI-Prolog

# Some basic performance tests

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

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

- the program `11 queens` computes (without printing them out) all the solutions of the 11-queens problem
- `perms of 11+nrev` computes the unique permutation that is equal to the reversed sequence of “ numbers computed by the naive reverse predicate
- `Sudoku 4x4` iterates over all solutions of a reduced Sudoku solver
- `metaint perms` is a variant of the second program, that runs via a two clause meta-interpreter



# 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 tokenizer and 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
- we believe that future ports of this design can help with the embedding of logic programming languages as lightweight software or hardware components