# A Simplified Virtual Machine for Multi-Engine Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

VMSS'2016

## Outline

## Prolog: Programming in Logic

- a YAVOL: yet another very old language: originating in the late 70's
- Robinson: unification algorithm - for better theorem proving
- motivations: Colmeraurer: NLP, Kowalski: algorithms = logic + control
- $\Rightarrow$ a computationally well-behaved subset of predicate logic
- the general case, after getting rid of quantifiers: `a ; b :- c,d,e.`
- Horn clauses: `a :- b,c,d.`
- all variables universally quantified $\Rightarrow$ we do not have put quantifiers
- multiple answers returned: (improperly) called "non-deterministic" execution
- newer derivatives: constraint programming, SAT-solvers, answer set programming: exploit fast execution of propositional logic
- like FP, and relational DB languages: a from of "declarative programming"

## Prolog: raising again?

Programming Language Ratings - from the Tiobe index

- 29 Lisp 0.630 %
- 30 Lua 0.593 %
- 31 Ada 0.552 %
- 32 Scala 0.550 %
- 33 OpenEdge ABL 0.467 %
- 34 Logo 0.432 %
- 35 Prolog 0.406 %
- 36 F# 0.391 %
- 37 RPG (OS/400) 0.375 %
- 38 LabVIEW 0.340 %
- 39 Haskell 0.287 %

a year or two ago: Prolog not on the list (for being above 50)

## Horn Clause Prolog in three slides

## Prolog: unification, backtracking, clause selection

```
?- X=a,Y=X. % variables uppercase, constants lower
X = Y, Y = a.

?- X=a,X=b.
false.

?- f(X,b)=f(a,Y). % compound terms unify recursively
X = a, Y = b.

% clauses
a(1). a(2). a(3).    % facts for a/1
b(2). b(3). b(4).    % facts for b/1

c(0).
c(X):-a(X),b(X).     % a/1 and b/1 must agree on X

?-c(R).              % the goal at the Prolog REPL
R=0; R=2; R=3.       % the stream of answers
```

## Prolog: Definite Clause Grammars

Prolog's DCG preprocessor transforms a clause defined with "-->" like

```
a0 --> a1,a2,...,an.
```

into a clause where predicates have two extra arguments expressing a chain of state changes as in

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

- work like "non-directional" attribute grammars/rewriting systems
- they can used to compose relations (functions in particular)
- with compound terms (e.g. lists) as arguments they form a Turing-complete embedded language

```
f -- > g,h.
f(In,Out):-g(In,Temp),h(Temp,Out).
```

Some extra notation: `{...}` calls to Prolog, `[...]` terminal symbols

## 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=add(s(X),Y,s(Z))
```

can be rewritten as a conjunction of 3 equations as follows

```
T=add(SX,Y,SZ),SX=s(X),SZ=s(Z)
```

When applying this to a clause like

```
C=[add(s(X),Y,s(Z)), add(X,Y,Z)]
```

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

```
C=[H,B],H=add(SX,Y,SZ),SX=s(X),SZ=s(Z), B=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”

## A small expressiveness lift: allowing variables in function and predicate symbol positions

our flat natural syntax allows the use of variables in function and predicate symbol position as in

```
She likes beer if She likes fries and She drinks alcohol.
```

- we can drop this Prolog restriction for a form of higher order syntax with a first order semantics
- as a convenient notational improvement, we can instruct our parser to expand

  ```
  Xs lists a b c
  ```

  to

  ```
  Xs holds list a _0 and _0 holds list b _1 and _1 holds list c nil
  ```

- two new keywords: "`list`" represent the list constructor and "`nil`" representing the empty list

## 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 tag system

- V=0 marking the first occurrence of a variable in a clause
- U=1 marking a second, third etc. occurrence of a variable
- R=2 marking a reference to an array slice representing a subterm
- C=3 marking the index in the symbol table of a constant (identifier or any other data object not fitting in a word)
- N=4 marking a small integer
- A=5 marking the “arity” (length) of the array slice holding a flattened term

## – continued –

To emulate the existence of distinct types for tagged words and their content we flip the sign when tagging and untagging:

```
final private static int tag(final int tag, final int word) {
  return -((word << 3) + tag);
}

final private static int detag(final int word) {
  return -word >> 3;
}

final private static int tagOf(final int word) {
  return -word & 7;
}
```

This is likely to trigger an index error at the smallest mis-step confusing address and value uses of `ints`.

## 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

## Unification: a few examples

```
?- X=Y,Y=a.
X = a ,
Y = a

?- f(X,g(X,X))=f(h(Z,Z),U),Z=a.
X = h(a,a) ,
Z = a ,
U = g(h(a,a),h(a,a))

?- [X,Y,Z]=[f(Y,Y),g(Z,Z),h(a,b)].
X = f(g(h(a,b),h(a,b)),g(h(a,b),h(a,b))) ,
Y = g(h(a,b),h(a,b)) ,
Z = h(a,b)
```

## Dereferencing

- the function `deref` walks through variable references
- we ensure that the compiler can inline it, and inline as well the functions `isVAR` and `getRef` that it calls, with `final` declarations

```
final private int deref(int x) {
  while (isVAR(x)) {
    final int r = getRef(x);
    if (r == x) break;
    x = r;
  }
  return x;
}
```

two inlineable methods:

```
final private static boolean isVAR(final int x) {return tagOf(x) < 2;}

final int getRef(final int x) {return heap[detag(x)];}
```

## 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 `regs` register array: a copy of the outermost array of the goal element we are working with, holding dereferenced elements in it
  I 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 and "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 then the heap level corresponding to the current goal

## A stack-based unification algorithm

```
final private boolean unify(final int base) {
  while (!ustack.isEmpty()) {
    int x1 = deref(ustack.pop()); x2 = deref(ustack.pop());
    if (x1 != x2) {
     int t1 = tagOf(x1); int t2 = tagOf(x2);
     int w1 = detag(x1); int w2 = detag(x2);
      if (isVAR(x1)) { /* unb. var. v1 */
        if (isVAR(x2) && w2 > w1) { /* unb. var. v2 */
          heap[w2] = x1; if (w2 <= base) trail.push(x2);
        } else { // x2 nonvar or older
          heap[w1] = x2; if (w1 <= base) trail.push(x1);
        }
      } else if (isVAR(x2)) { /* x1 is NONVAR */
        heap[w2] = x1; if (w2 <= base) trail.push(x2);
      } else if (R == t1 && R == t2) { // both should be R
        if (!unify_args(w1, w2)) return false;
      } else return false;
    }
  } return true;
```

## The case of compound terms: unifying the arguments

```
final private boolean unify_args(final int w1, final int w2) {
   int v1 = heap[w1]; int v2 = heap[w2];
   final int n1 = detag(v1); // both should be A
   final int n2 = detag(v2);
   if (n1 != n2) return false;
   int b1 = 1 + w1; int b2 = 1 + w2;
   for (int i = n1 - 1; i >= 0; i--) {
     final int i1 = b1 + i;
     final int i2 = b2 + i;
     final int u1 = heap[i1];
     final int u2 = heap[i2];
     if (u1 == u2) continue;
     ustack.push(u2);
     ustack.push(u1);
   }
   return true;
 }
```

## 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 simply by the presence of multiple arithmetic units in modern CPUs
- via a CUDA or OpenCL GPU implementation, copies can be speculatively created in parallel, based on predicted future uses

```
final private static int relocate(final int b, final int cell) {
  return tagOf(cell) < 3 ? cell + b : cell;
}
```

- we compute the relocation offset ahead of time, once we know the difference between its source and its target, i.e., when the process for selecting matching clauses starts

## – continued –

- we relocate a slice `<from,to>` from our *prototype clause*, placed on the heap ahead of time by the parser

```
final private void pushCells(final int b, final int from, final int
                               final int base) {
  ensureSize(to - from);
  for (int i = from; i < to; i++) {
    push(relocate(b, heap[base + i]));
  }
}
```

- as our heap is a dynamic array, we check ahead of time if it would overflow with `ensureSize` to avoid testing if expansion is needed for each cell

## – continued –

- 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

```
final private int pushHead(final int b, final Clause C) {
    pushCells(b, 0, C.neck, C.base);
    final int head = C.gs[0];
    return relocate(b, head);
}
```

- 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
- he 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 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

```
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 `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

```
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

## A detour: first class logic engines

a richer API then what streams provided can be used

- a *logic engine* is a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *engine*
- very much the same thing as a programmer controlling Prolog's interactive toplevel loop:
  - launch a new goal
  - ask for a new answer
  - interpret it
  - react to it
- logic engines can create other logic engines as well as external objects
- logic engines can be controlled cooperatively or preemptively

## Interactors (a richer logic engine API, beyond streams): new_engine/3

new_engine(AnswerPattern, Goal, Interactor):

- creates a new instance of the Prolog interpreter, uniquely identified by `Interactor`
- shares code with the currently running program
- initialized with `Goal` as a starting point
- `AnswerPattern` : answers returned by the engine will be instances of the pattern

## Interactors: get/2, stop/1

get(Interactor, AnswerInstance):

- tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`
- if an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned
- is used to retrieve successive answers generated by an Interactor, on demand
- it is responsible for actually triggering computations in the engine
- one can see this as transforming Prolog's backtracking over all answers into a deterministic stream of lazily generated answers

stop(Interactor):

- stops the Interactor
- `no` is returned for new queries

## The `return` operation: a key co-routining primitive

return(Term)

- will save the state of the engine and transfer *control* and a *result* `Term` to its client
- the client will receive a copy of `Term` simply by using its `get/2` operation
- an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available

Application: exceptions

```
throw(E):-return(exception(E)).
```

## Exchanging Data with an Interactor

to_engine(Engine,Term):

- used to send a client's data to an Engine

from_engine(Term):

- used by the engine to receive a client's Data

## Typical use of the Interactor API

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*:
    - the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
    - the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
    - the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

## What can we do with first-class engines?

- define the complete set of ISO-Prolog operations at source level
- in fact, one can define the engine operations in Horn clause Prolog - with a bit of black magic (e.g. splitting a term into two variant terms)
- implement (at source level) Erlang-style messaging - with millions of engines
- implement Linda blackboards
- implement Prolog's dynamic database at source level
- build an algebra for composing engines and their answer streams
- implement "tabling" a from of dynamic programming that avoids recomputation

## 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 thriftier 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 then 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 `http://www.cse.unt.edu/~tarau/research/2016/prologEngine.zip`
- as a more natural target for a system developed around it would use **C** (possibly accelerated with a GPU API like CUDA), 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 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

## A short history of Prolog machines

## The Warren Abstract Machine (WAM)

- designed by D.H.D. Warren in the early 80'
- uses registers, two stacks (for goals and clause choices), heap and trail
- cited this days as [1], a very good tutorial introduction
- improvements over the years, but basic architecture unchanged
  - simplified WAM, using a transformation to binary clauses [13] [12]
  - just-in-time indexing schemes of YAP [4] and SWI-Prolog [15]
  - alternative designs: stack frames based [16]
  - tabling, first-order semantics for HiLog [8], [3]
- implemented both natively and as a software virtual machine
- an early overview of WAM derivatives: [14]
- a TCLP issue dedicated to some of today's Prolog machines: [4, 2, 8, 5, 16, 12]

## This design, in context

- the closest Prolog implementation is our own Styla system [11], a Scala-based interpreter, itself a derivative of our Java-based Kernel Prolog [10] system
- they both use a clause unfolding interpreter along the lines of [9]
- contrary to this design, they rely heavily on high-level features of the implementation language
- an object-oriented term hierarchy and a unification algorithm distributed over various term sub-types
- first-class, “resumable” Prolog engines have been present in our systems since the mid-90s
- there’s some renewed interest in them (as reflected by a few dozen recent messages in comp.lang.prolog in September 2015)
- a thread-based model is used in SWI-Prolog’s Pengines [6]
- locating function symbols and arity in separate words is present in the ECLiPSe Prolog system [7]

H. Aït-Kaci.
*Warren's Abstract Machine: A Tutorial Reconstruction.*
MIT Press, 1991.

Mats Carlson and Per Mildner.
SICStus Prolog – The first 25 years.
*Theory and Practice of Logic Programming*, 12:35–66, 1 2012.

W. Chen, M. Kifer, and D.S. Warren.
HiLog: A first-order semantics for higher-order logic programming constructs.
In E.L. Lusk and R.A. Overbeek, editors, *1st North American Conf. Logic Programming*, pages 1090–1114, Cleveland, OH, 1989. MIT Press.

Vitor Santos Costa, Ricardo Rocha, and Luis Damas.
The YAP Prolog system.
*Theory and Practice of Logic Programming*, 12:5–34, 1 2012.

M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. F. Morales, and G. Puebla.
An overview of Ciao and its design philosophy.
*Theory and Practice of Logic Programming*, 12:219–252, 1 2012.

Torbjörn Lager and Jan Wielemaker.
Pengines: Web logic programming made easy.
*TPLP*, 14(4-5):539–552, 2014.

Joachim Schimpf and Kish Shen.
ECLiPSe âĂŞ From LP to CLP.
*Theory and Practice of Logic Programming*, 12:127–156, 1 2012.

Terrance Swift and S. Warren, David.
XSB: Extending Prolog with Tabled Logic Programming.
*Theory and Practice of Logic Programming*, 12:157–187, 1 2012.

Paul Tarau.
Inference and Computation Mobility with Jinni.

In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48, Berlin Heidelberg, 1999. Springer.
ISBN 3-540-65463-1.

Paul Tarau.
Kernel Prolog: a Java-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy, May 2012.
https://code.google.com/archive/p/kernel-prolog/.

Paul Tarau.
Styla: a Lightweight Scala-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy, May 2012.
https://code.google.com/archive/p/styla//.

Paul Tarau.
The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines.
*Theory and Practice of Logic Programming*, 12(1-2):97–126, 2012.

Paul Tarau and Michel Boyer.
Elementary Logic Programs.
In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173, Berlin Heidelberg, August 1990. Springer.

Peter Van Roy.
1983-1993: The Wonder Years of Sequential Prolog Implementation.
*Journal of Logic Programming*, 19(20):385–441, 1994.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjorn Lager.
SWI-Prolog.
*Theory and Practice of Logic Programming*, 12:67–96, 1 2012.

Neng-Fa Zhou.
The language features and architecture of B-Prolog.
*Theory and Practice of Logic Programming*, 12:189–218, 1 2012.

## 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