A Hitchhiker's Guide to Reinventing a Prolog Machine

PAUL TARAU

Department of Computer Science and Engineering
(*e-mail:* paul.tarau@unt.edu)

submitted March 16, 2017; revised ?; accepted ?

Abstract

We take a fresh, "clean-room" look at implementing Prolog by deriving its translation to an executable representation and its execution algorithm from a simple Horn Clause meta-interpreter. The resulting design has some interesting properties:

- the heap representation of terms and the abstract machine instruction encodings are the same
- no dedicated code area is used as the code is placed directly on the heap
- unification and indexing operations are orthogonal
- filtering of matching clauses happens without building new structures on the heap
- variables in function and predicate symbol positions are handled with no performance penalty
- a simple English-like syntax is used as an intermediate representation for clauses and goals
- solutions of (multiple) logic engines are exposed as answer streams that can be combined through typical functional programming patterns
- performance of a basic interpreter implementing our design is within a factor of 2 of a highly optimized compiled WAM-based system using the same host language

To help placing our design on the fairly rich map of Prolog systems, we discuss similarities to existing Prolog abstract machines, with emphasis on separating necessary commonalities from arbitrary implementation choices.

KEYWORDS: Prolog abstract machines, heap representation of terms and code, immutable goal stacks, natural language syntax for clauses, answer streams, multi-argument indexing algorithm.

1 Introduction

Forgetting how to implement a Prolog system is as hard as learning how to build one. While contaminated with episodic memories acquired through the not so few Prolog systems that we have built in the past (Tarau 2012d; Tarau 2012b; Tarau 2012a; Tarau 2012c), a fresh, "clean-room" attempt is described here to reinvent a Prolog machine by deriving it from the intuitions gleaned from the execution algorithm of a simplified two-clause meta-interpreter.

But why would one do this, when more than three decades of Prolog implementation seem to have fully saturated the search space of available implementation choices? Some of the details will unfold as our story progresses through the next sections, but an important reason is that we felt that existing systems, while possibly peeking out in terms of performance (Costa et al. 2012; Carlson and Mildner 2012; Swift and Warren 2012; Hermenegildo et al. 2012; Zhou 2012) or overall environment convenience and system usability (Wielemaker et al. 2012; Carlson and Mildner 2012; Hermenegildo et al. 2012), have left interesting implementation choices

unexplored. Another reason is that the natural chain of concepts leading the execution mechanism SLD-resolution to an efficient low-level implementation has stayed often in a "no-man's land" between theoretical work exploring the foundations of logic programming languages and implementation work focussed mostly on refining the gold-standard set by the Warren Abstract machine (Aït-Kaci 1991; Van Roy 1994).

While comparisons to WAM-based systems abound to help placing in context the alternatives we propose, the paper can also be seen as a self-contained shortcut allowing a "hitchhiker" to Prolog implementation to get the core of an efficient-enough Prolog system up and running in a few days. However, given the space constraints of the paper, our step-by-step derivation process will leave out some routine elements well-known to people modestly familiar with prolog implementation. The resulting design, implemented in an easily translatable to **C** subset of Java, around 1000 lines, is available at `http://www.cse.unt.edu/~tarau/research/2016/prologEngine.zip`. We refer to it for details that space constraints will force us to summarize or omit. It covers only Horn-Clause Prolog, as this is usually a good first step to evaluate the basic implementation choices and performance characteristics of a Prolog Machine. It does not cover orthogonal implementation issues like garbage collection, unification of cyclical terms, built-ins or host language interface, as our focus is on the inner workings of the Prolog machine seen as an unification+backtracking+indexing engine. We will start by sketching here some features that are not typically shared with most existing Prolog systems:

- a shared representation of executable code and terms on the heap
- the use of a natural language-style intermediate representation used as our "assembler language"
- the packaging of solutions as answer streams
- the decoupling of indexing and unification instructions
- interpreted, but fast-enough execution.

The paper is organized as follows. Section 2 derives (informally) the main lines of our design from a simple Horn Clause meta-interpreter and an equational representation of Prolog terms. Section 3 describes an execution-ready heap representation of Prolog clauses that also serves as instruction set for our abstract machine. Section 4 explains the execution algorithm seen as iterated unfolding of the first goal in the body of a clause, the generation of answer streams and the main run-time data structures. Section 5 overviews a generic indexing mechanism, orthogonal to the unification-based execution algorithm, designed as an add-on to the iterated unfolding interpreter loop. Section 6 shows some preliminary performance results. Section 7 overviews related work. Section 8 concludes the paper.

2 Distilling the "essence" of Prolog's execution algorithm

When seen through the eyes of a meta-interpreter for the Horn Clause subset of Prolog, the execution algorithm is astonishingly simple. We will next expand step-by-step the intuitions behind its implicit operations and derive an equational form that we will use to guide subsequent refinements into a self-contained interpreter.

2.1 Our starting point: a simplified Horn Clause meta-interpreter

We will start by inventing a simpler meta-interpreter a than the usual one, with a bit of help from a convenient clause representation. 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 its tail
   metaint(Bs).       % interpret the extended body
```

Clauses are represented as facts of the form cls/2 with the first argument representing the head of the clause followed by a (possibly empty) list of body goals and terminated with a variable returned also as the second argument of cls/2.

```
cls([
     add(0,X,X)                      %1
        |Tail],Tail).
cls([
     add(s(X),Y,s(Z)), add(X,Y,Z)    %2
        |Tail],Tail).
cls([
     goal(R), add(s(s(0)),s(s(0)),R) %3
        |Tail],Tail).
```

The actual content of the clauses, that we will use as our running example is marked with %1, %2 and %3. As one can verify with any Prolog system, it runs as expected when computing the successor arithmetic equivalent of $2+2=4$.

```
?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```

2.2 The equational form of terms and clauses

We will use the term *equation* to denote an unification step, typically between a variable and a compound term or constant. 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. One can see that a Prolog clause can be decomposed into a sequence of such equations, which, if executed as unification steps, build back the representation of the clause.

2.3 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. Note that we use here "natural language" with a grain of salt, as we are talking about a severely restricted form of controlled English.

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

Note that we keep the Prolog convention for the uppercase (or "`_`") as the fist character of variable names 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.

We can consider this syntax our "assembler language" to be read in directly by the loader of a runtime system, as well as the virtual machine *code* generated by a simple compiler translating Prolog clauses to it. A tokenizer splitting into words sentences delimited by "`.`" is all that is needed to complete a parser for this restricted English-style "assembler language".

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

Let us observe right away that our flat natural syntax allows the use of variables in function and predicate symbol position as in

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

corresponding to a Prolog syntax involving variables in predicate positions as in

```
Someone(likes, beer):-Someone(likes, fries),Someone(drinks, alcohol).
```

This suggests dropping this Prolog restriction for a form of higher order syntax with a first order semantics giving to our intermediate language a (touch of) the capabilities of HiLog (Chen et al. 1989). While this example can be seen as an indirect way to support the use if `likes` as an infix operator the use in

```
call(Operation,10,20,Result)
```

as

```
Operation 10 20 Result
```

hints about more interesting uses, with `Operation` working as a variable in predicate symbol position. 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
```

with the new keywords "`list`" representing the list constructor and "`nil`" representing the empty list.

3 The heap representation as the executable code

Derived from the equational representation of Prolog terms, our natural language form of the clauses is ready to leave Prolog for a conventional implementation language that does not provide, like our two clause meta-interpreter did, unification, recursion and backtracking for free. *It is basically an array representation with variables on the left side of our equations turned into array indices pointing to compound terms at higher addresses in the same array.*

For convenience to both the readers and the writer of this paper, we have picked a subset of Java, trivially translatable to **C** that does not make use of object oriented features, while benefitting from simplicity of automated memory management and safety coming from things like polymorphic types and index checking.

3.1 The tag system

We will instruct our tokenizer to recognize variables, symbols and (small) integers as primitive data types. As we develop a Java-based interpreter we represent our Prolog terms top-down (as first described in (Meier 1990)). Java's primitive `int` type is used for tagged words[1].

We will instruct our parser to extract as much information as possible by marking each word with a relevant tag (that will also be seen as a WAM-like instruction by the execution algorithm). We will use the following 3-bit tags:

- 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 of the array slice holding a flattened term (of size 1 + the number of arguments, to also make room for the "function symbol" - that could be an atom or a variable)

To ensure fast inlining in a language like Java we make most of our methods `final private static` - with similar annotations available in **C**. For clarity, we omit these annotations from our code snippets. To emulate the existence of distinct types for tagged words and their content we flip the sign when tagging and untagging:

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

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

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

The minus sign marking `word` is meant to trigger an index error at the smallest mis-step when a method would confuse an address use and a value use of an `int`. The same technique would help catching such errors in **c**.

Note that we will ensure that the Java compiler does as much inline expansion as possible by coding in a "**C**-friendly" style, avoiding inheritance and declaring most methods as `static`, `private` and `final`.

[1] In a **C** implementation one might want to choose `long long` instead of `int` to take advantage of the 64 bit address space.

3.2 The top-down representation of terms

Our top-down representation of Prolog terms closely follows our natural language-style assembler language syntax. After the clause

```
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 .
```

it is represented on the heap (starting in this case at address 5 and shown here marked with lower case tags and colons) as follows:

```
[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
```

Note the 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:`". As length information in kept in a separate word, 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

```
r:5 :- [r:16]
```

as the head of this clause starts at address `5` and its (one-element) body follows at address `16`.

3.3 Clauses as descriptors of heap cells

The parser places the cells composing a clause directly to the heap. At the same time, 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` (that one would mimic with a `struct` in **C**), 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.

4 Execution as iterated clause unfolding

As the meta-interpreter in section 2 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. Given also that we do not assume anymore that predicate symbols are non-variables, it makes sense to design indexing as a distinct algorithm, while ensuring that there's a convenient way to plug it in as a refinement of our iterated unfolding mechanism.

4.1 Unification, trailing and pre-unification clause filtering

In a way similar to the WAM's unification instructions, our relatively rich tag system reduces significantly the need to call the full unification algorithm. If one ignores the WAM's indexing instructions and avoids implementing an AND-stack by binarization (Tarau and Boyer 1990) or by placing terms directly on the heap, the remaining 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.

4.1.1 Dereferencing

The function `deref` walks, as usual in a Prolog implementation, 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. We put here their "low-level" code snippets (most likely well known to experienced Prolog implementors) as they are in the "innermost loop" of the system.

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

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

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

4.1.2 The pre-unification step: detecting matching clauses without copying to the heap

Independently of indexing, one can filter matching clauses by comparing the outermost array of the current goal with the outermost array of a clause head.

Interestingly, as a *prototype of each clause is already placed on the heap at loading and linking time*, one could tentatively unify it with the goal and then undo the bindings. On success, one would then redo the unification while progressively copying to the heap the subterms of the head that need to be newly created.

But even better, we can emulate WAM's registers as a copy of the outermost array of the goal element we are working with, holding dereferenced elements in it.

This "register"-array can be used to reject clauses that mismatch it in positions holding symbols, numbers or references to array-lengths. We can use for this the prototype of a clause head without starting to place new terms on the heap. At the same time, 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.

4.1.3 Unification

As we have 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 emulate recursion by the otherwise standard unification algorithm. Given that we we actually start by unifying the arguments of the outermost terms we split our algorithm in two methods, `unify_args` and `unify` that take turns pushing tasks on the stack and popping them off as they explore the structure of a the two Prolog terms.

4.1.4 Trailing

As usual, 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.

4.2 Fast linear term relocation

While the WAM's instructions (that need as well to be decoded by an interpreter loop in non-native implementations) spend effort on deciding which (new) terms are created on the heap ("write" mode) and which (old) terms are reused from below ("read" mode), we bet instead on a very fast relocation loop that speculatively places the clause head (including its subterms) on the heap. This "single instruction multiple data" operation would likely benefit from parallel execution simply by the presence of multiple arithmetic units in modern CPUs, or even more significantly, via a CUDA or OpenCL GPU implementation, especially if copies are speculatively created in parallel, based on predicted future uses.

As our variable and reference codes that need relocation (`V,U,R`) are `0,1` and `2`, we ensure that the `relocate` method is inlined by the Java compiler by defining it "`static private final`". Similar inlining would occur in today's **C** compilers.

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

Note that 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. To relocate a slice `<from,to>` from our *prototype clause*, placed on the heap ahead of time by the parser, we use another potentially inlineable method, `pushCells`:

```
void pushCells(int b, int from, int to,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.

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. The method `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.

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

As the code reuse coming from distinguishing between a "read" and a "write" mode during head-unification only increases heap usage by a small percentage (as most clauses are rather body heavy than head-heavy) we trade it for copying the complete head.

On the other hand, we only copy the body once unification succeeds, by calling the method `pushBody` that also relies on the precomputed relocation offset `b`.

```
  int[] pushBody(int b, int head, Clause C) {
    pushCells(b, C.neck, C.len, C.base);
    int l = C.gs.length;
    int[] gs = new int[l];
    gs[0] = head;
    for (int k = 1; k < l; k++) {
      int cell = C.gs[k];
      gs[k] = relocate(b, cell);
    }
    return gs;
  }
```

Note also that we relocate the skeleton `gs` starting with the address of the first goal so it is ready to be merged in the immutable list of goals. At the end, an interesting assertion holds: *the heap is a stream of successive clauses connected among them by variable bindings*. And one can devise a mechanism where, based on profiling, these contiguous heap slices, corresponding to clauses, are created in advance, speculatively, on separate threads.

While we have not implemented it yet, we mention here an *alternative algorithm*, that like the WAM, only creates new terms originating form the head of the clause when needed, while also ensuring that term creation happens all at once, and only if full unification succeeds.

We start with the pre-unification matching, followed on success with full unification, but happening on the *prototype of the clause* located at a lower address on the heap. We take care to trail *every* variable binding. Then we scan the trail and handle the following two situations:

1. if a variable located in the prototype clause area points to a goal already on the heap we collect it for future unbinding, as we want to clear the prototype from all bindings, for reuse
2. if the binding comes from the goals already on the heap, above the prototype clauses area, we copy to the heap the subterm it points to, relocate the variable to point to it, while making sure to collect the variable for later placement on the trail as now it will point upwards.

An interesting aspect of this alternative is that one mimics Prolog's resolution step on the pre-built prototype and it trades some extra work on the trail, in exchange for less work and some space efficiency on the heap.

4.3 Stretching out the Spine: the immutable goal stack

A *Spine* can be seen as a runtime abstraction of a `Clause` instance, collecting the information needed for the execution of the goals originating from it. Implemented as the small methodless `Spine` class, it 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`

Like the meta-interpreter we have started with, a spine extends the goal stack with the contribution of a given clause's body. Note that (most of the) goal elements on this immutable list are shared among alternative branches. In a way, the illusion that the complete goal stack is local to each `Spine` instance is helpful as it matches the way they look in our simplified meta-interpreter from section 2, but has virtually no space overhead compared to a global procedurally managed goal stack as most of its (immutable) tail is safely shared among Spines.

4.4 The interpreter loop: yielding an answer and ready to resume

Our main interpreter loop 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, when it returns null, signaling that no more answers are available.

```
Spine yield() {
  while (!spines.isEmpty()) {
    Spine G = spines.peek();
    if (hasClauses(G)) {
      if (hasGoals(G)) {
        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;
}
```

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.

To achieve this, the class `Engine` defines in the method `ask()`. A 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. It 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;
    int res = answer(query.ttop).hd;
    Object R = exportTerm(res);
    unwindTrail(query.ttop);
    return R;
}
```

4.5 Playing with answer streams

We model our answer streams to match Java 8's stream API (Oracle Corp.), although as a more expressive alternative (*interactors*), that made it in our Prolog implementations starting with (Tarau 2000), can be considered instead.

A reason for choosing the Java 8 stream API is that it allows elegant embedding in cluster and cloud configurations using high-level functional programming constructs like `map, fold` and `filter` as well as automatic parallelization of complex data-flows as provided by frameworks like `Apache Flink`.

To encapsulate our answer streams in a Java 8 `stream`, a special iterator-like interface called `Spliterator` is used (Oracle Corp.). 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.

Once the `Spliterator` interface is implemented, the stream of answers encapsulating this engine is created with second argument `false`, specifying that it is not a parallel stream.

```
public Stream<Object> stream() {
  return StreamSupport.stream(this, false);
}
```

5 Multi-argument indexing: a modular add-on

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 (up to a maximum that can

be specified by the programmer) it associates to each indexable element (symbol, number or arity) the set of clauses where the indexable element occurs in that argument position. 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.

Sets of clause numbers associated to each (tagged) indexable element are supported 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. We will not actually compute the unions, however. Instead, 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.

Two interesting special cases can benefit from custom variants of the algorithm.

For very small programs or programs having predicates with fewer clauses than the bit size of a long (64), 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 challenging overall memory capacity, a more compact sparse bit set implementation like (Wooldridge 2016) or a Bloom filter-based set (Bloom 1970) would replace our `IntMap`-based 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 (Almeida et al. 2007) implementation can be used. In both cases, 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. Finally, especially for very large programs, one might want to compute the set of matching clauses lazily, using the Java 8 streams API (Oracle Corp.).

6 Some basic performance tests

We prototyped our design as a small, slightly more than 1000 lines of generously commented Java program. However, as a more natural target for a system developed around it would use **C**, we have stayed away from Java's object oriented features by using a large `Engine` class hosting all the data areas and a few small classes like `Clause` and `Spine` that can be easily mapped to **C** `structs`. While implemented as an interpreter, our preliminary tests (Fig. 1) indicate, somewhat surprisingly, that performance is close, (within a factor of **2**) to our Java-based systems like Jinni and Lean Prolog that use a (fairly optimized) WAM-based instruction set and a factor of **2**-4 from **C**-based SWI-Prolog. While this an order of magnitude slower than today's **C** based Prologs the "apples-to-apples" comparison with our fast WAM-based Prolog implemented in the same language - Java - is a more accurate indication that this lightweight interpreter design is actually quite fast and likely to be within a factor of 2 to 3 of today's optimized WAM-based Prologs if implemented in **C**. The program `11 queens` computes (without printing them out) all

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

Fig. 1. Timings and number of logical inferences (as counted by SWI-Prolog) on 4 small Prolog programs

the solutions of the 11-queens problem. `Sudoku 4x4` does the same for a reduced Sudoku solver and `perms of 11+nrev` computes the unique permutation that is equal to the reversed sequence of " numbers computed by the naive reverse predicate. The fourth program, `metaint perms` is a variant of the second program, run this time via the two clause meta-interpreter that we have started from. in section 2 to derive our execution algorithm.

For a more conclusive performance comparison, future work is planned on first deriving a WAM-like compiled instruction set from our interpreter. Also, we expect that a **C**-based system, even if kept as an interpreter, is likely to boost performance slightly above slower compiled systems like SWI-Prolog, from which our Java-based interpreter is within a factor of 2-4 on the Horn Clause subset, for small programs.

7 Related work

The closest Prolog implementation is our own Styla system (Tarau 2012c), a Scala-based interpreter, itself a derivative of our Java-based Kernel Prolog (Tarau 2012b) system. They both use a clause unfolding interpreter along the lines of (Tarau 1999), but contrary to our current design, they rely heavily on high-level features of the implementation language, including an object-oriented term hierarchy and a unification algorithm distributed over various term subtypes. First-class, "resumable" Prolog engines have been present in our systems since (Tarau 2000) and there's some renewed interest in them (as reflected by a few dozen recent messages in comp.lang.prolog in September 2015) as well as in a similar, thread-based model used in SWI-Prolog's Pengines (Lager and Wielemaker 2014).

Locating function symbols and arity in separate words is different to the typical "symbol+arity" used in most other Prolog systems we are aware of. The translation from HiLog (Chen et al. 1989) to a first order form, using `apply/N` predicates, is similar to our natural language assembler, where (unnamed) arrays of various length are essentially the same thing as HiLog's `apply/N` predicate wrappers. Consequently, work on a first-order semantics for HiLog (Chen et al. 1989) also covers our natural assembler programs.

There are some clear commonalities with the WAM (Aït-Kaci 1991) and more closely with the BinWAM variant of it (Tarau 2012d) where transformation to binary clauses implicitly emulates a form of goal stacking. In a way, we also end up emulating something close to the WAM's registers that are saved in our `Spine` stack, itself playing a role similar to the WAM's OR-stack. Placing them on a stack, rather then mapping them to a register vector, is also similar to B-Prolog's stack frame-based representation (Zhou 2012).

The major commonality with the BinWAM (Tarau 2012d), not present in the WAM, is that the implicit goal-stacking of the BinWAM is replaced here with an explicit and immutable goal stack seen as present in each `Spine`. As most (except the few topmost) goal elements are shared among

`Spines`, the overall time and space costs are comparable with a mutable goal stack managed by saving in our Spines pointers to its top. Another commonality with a BinWAM-optimization (as implemented in BinProlog) is that arguments placed in registers are dereferenced once and then matched against several clause candidates. On the other hand, the WAMs instruction set ensures subterms are only built on the heap as needed, while our interpreter tries instead to eliminate up front the non-matching clauses by borrowing pre-unification information from a prototype clause at a lower heap location before unification is called. On unification success, the complete body of the clause is relocated, and, as this heap-to-heap copy is quite fast, we do not get a significant performance hit as a result. Thus some of the unexplored implementation choices that materialized through our "from scratch" design steps result in comparable performance, while staying intuitively closer to the view offered by the two clause meta-interpreter, that we have started with in section 2.

Another feature not present in typical WAM-based Prolog systems, is the decoupling of indexing and unification instructions, although one might argue that the same philosophy is motivating the just-in-time indexing schemes of YAP (Costa et al. 2012) and SWI-Prolog (Wielemaker et al. 2012). In fact, when generalized to arbitrary paths, reaching constants occurring deep in a term, our indexing algorithm has more in common with the ones used in theorem proving systems like (Riazanov and Voronkov 2003), than with the WAM's tightly interleaved indexing instructions (Aït-Kaci 1991). We believe that besides separating naturally independent concerns, decoupling indexing favors deployment scenarios where the Prolog code is distributed on a cluster or cloud, and fetched as needed. In this case, indexing might need to happen at a different site than the one where the call is made.

8 Conclusions

We hope that by trying to forget as much as we could about the long polished art of Prolog implementation, we have obtained a genuinely more intuitive view of Prolog's execution algorithm. By deriving our Prolog machine as naively as possible, 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.

In the process, we have lifted some restrictions on Prolog syntax, like the need for the function or predicate symbol to be a constant, and 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. And the interpreter based on our design was able to get close enough (within a factor of two but often less) to optimized compiled code as shown by our preliminary performance tests. With only slightly more than 1000 lines of Java code, we believe that future ports of this design can help with the embedding of logic programming languages as lightweight software or hardware components.

References

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

ALMEIDA, P. S., BAQUERO, C., PREGUIÇA, N., AND HUTCHISON, D. 2007. Scalable bloom filters. *Inf. Process. Lett. 101,* 6 (Mar.), 255–261.

BLOOM, B. H. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 422–426.

CARLSON, M. AND MILDNER, P. 2012. SICStus Prolog – The first 25 years. *Theory and Practice of Logic Programming 12*, 35–66.

CHEN, W., KIFER, M., AND WARREN, D. 1989. HiLog: A first-order semantics for higher-order logic programming constructs. In *1st North American Conf. Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cleveland, OH, 1090–1114.

COSTA, V. S., ROCHA, R., AND DAMAS, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12*, 5–34.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LOPEZ-GARCIA, P., MERA, E., MORALES, J. F., AND PUEBLA, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12*, 219–252.

LAGER, T. AND WIELEMAKER, J. 2014. Pengines: Web logic programming made easy. *TPLP 14,* 4-5, 539–552.

MEIER, M. 1990. Compilation of compound terms in Prolog. In *Proceedings of the 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Massachusetts London, England, 63–79.

ORACLE CORP. Java 8 Streams package. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

RIAZANOV, A. AND VORONKOV, A. 2003. *Efficient Instance Retrieval with Standard and Relational Path Indexing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 380–396.

SWIFT, T. AND WARREN, DAVID, S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming 12*, 157–187.

TARAU, P. 1999. Inference and Computation Mobility with Jinni. In *The Logic Programming Paradigm: a 25 Year Perspective*, K. Apt, V. Marek, and M. Truszczynski, Eds. Springer, Berlin Heidelberg, 33–48. ISBN 3-540-65463-1.

TARAU, P. 2000. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In *Computational Logic–CL 2000: First International Conference*, J. Lloyd, Ed. London, UK. LNCS 1861, Springer-Verlag.

TARAU, P. 2012a. Jinni Prolog: a Java-based Prolog compiler and runtime system . `https://code.google.com/archive/p/jinniprolog/`.

TARAU, P. 2012b. Kernel Prolog: a Java-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy. `https://code.google.com/archive/p/kernel-prolog/`.

TARAU, P. 2012c. Styla: a Lightweight Scala-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy. `https://code.google.com/archive/p/styla//`.

TARAU, P. 2012d. 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.

TARAU, P. AND BOYER, M. 1990. Elementary Logic Programs. In *Proceedings of Programming Language Implementation and Logic Programming*, P. Deransart and J. Maluszyński, Eds. Number 456 in Lecture Notes in Computer Science. Springer, Berlin Heidelberg, 159–173.

VAN ROY, P. 1994. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming 19,* 20, 385–441.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12*, 67–96.

WOOLDRIDGE, B. 2016. Sparse Bit Set. https://github.com/brettwooldridge/SparseBitSet.

ZHOU, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12*, 189–218.