Architecture and Implementation Aspects of the Lean Prolog System

Paul Tarau

1 Introduction

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [1] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At a more theoretical level, it has been argued that *interactivity*, seen as fundamental computational paradigm, can actually expand computational expressiveness and provide new models of computation [2].

In a logic programming context, the Jinni agent programming language [3] and the BinProlog system [4] have been centered around logic engine constructs providing an API that supported reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [5], that have allowed the separation of the first-order language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins.

Building upon the *Fluents* API described in [5], this document will focus on bringing interaction-centered, agent oriented constructs from software design frameworks and design patterns to programming language level.

The resulting language constructs, that we shall call *Interactors*, will express control, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog builtins, including dynamic database operations.

Interruptible Iterators are a new Java extension described in [6]. The underlying construct is the `yield` statement providing multiple returns and resumption of iterative blocks, i.e. for instance, a `yield` statement in the body of a `for` loop will return a result for each value of the loop's index.

The `yield` statement has been integrated in newer Object Oriented languages like `Ruby` [7, 8] `C#` [9] and `Python` [10] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [11] and ICON [12].

Interactors can be seen as a natural generalization of Interruptible Iterators and Coroutine Iterators. They implement the the more radical idea of allowing clients to communicate to/from inside blocks of arbitrary recursive computations. The challenge is to achieve this without the fairly complex interrupt based communication protocol between the iterator and its client described in [6]. Towards this end, Interactors provide a structured two-way communication

between a client and the usually autonomous service the client requires from a given language construct, often encapsulating an independent component.

2 First Class Logic Engines

Our *Interactor API* is a natural extension of the *Logic Engine API* introduced in [5]. An *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution [13] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas. In our actual implementation, with all data areas dynamic, engines are lightweight and engine creation is extremely fast.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine. The query

```
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. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the `stop/1` operation that might or might not reclaim resources held by the engine. In our actual implementation we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected.

So far, these operations provide a minimal *Coroutine Iterator API*, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [5], as well as higher order operations like *fold* and *best_of*.

3 Logic Engines as Interactors

LeanProlog's Java-based predecessor, *Jinni* has been mainly used in various applications [14–17] as an *intelligent agent infrastructure*, by taking advantage of Prolog's knowledge processing capabilities in combination with a simple and easily extensible runtime kernel supporting a flexible reflexion mechanism [18]. Naturally, this has suggested to investigate weather some basic agent-oriented language design ideas can be used for a refactoring of pure Prolog's interaction with the external world.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [1, 19] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At a more theoretical level, it has been argued that *interactivity*, seen as a fundamental computational paradigm, can actually expand computational expressiveness and provide new models of computation [2].

It has been a long tradition of logic programming languages [20, **?**,21] to use multiple logic engines for supporting concurrent execution.

In this context, the Jinni Prolog agent programming framework [3] and the LeanProlog system have been centered around logic engine constructs providing an API that supports reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [5], that have allowed the separation of the first-order language interpreters from the multi-threading mechanism, while providing, at the same time, a very concise source-level reconstruction of Prolog's built-ins. Later we have extended the original *Fluents* with a few new operations [22] supporting bi-directional, mixed-initiative exchanges between engines.

The resulting language constructs, that we have called *Interactors*, express control, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog built-ins, including dynamic database operations.

In a wider programming language implementation context, a `yield` statement has been integrated in newer object oriented languages like `Ruby` [7, 8] `C#` [9] and `Python` [10] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [11] and ICON [12].

Interactors can be seen as a natural generalization of Coroutine Iterators. They implement the more radical idea of allowing clients to communicate to/from inside blocks of arbitrary recursive computations. The challenge is to achieve this

without the fairly complex interrupt based communication protocol between the iterator and its client described in [6]. Towards this end, Interactors provide a structured two-way communication between a client and the usually autonomous service the client requires from a given language construct, often encapsulating an independent component.

3.1 Logic Engines as Answer Generators

Our *Interactor API* has evolved progressively into a practical Prolog implementation framework starting with [5] and continued with [23] and [22]. We summarize it here while instantiating the more general framework to focus on interoperation of logic engines.

An *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution [13] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas.

In our newer implementations, with all data areas dynamic, engines are lightweight and engine creation is fast and memory efficient[1] to the point where using them as building blocks for a significant number of built-ins and various language constructs is not prohibitive in terms of performance.

3.2 Iterating over computed answers

Note that our logic engines are seen, in an object oriented-style, as implementing the *interface* `Interactor`. This supports a *uniform* interaction mechanism with a variety of objects ranging from logic engines to file/socket streams and iterators over external data structures.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine. The query

[1] The additional operation `load_engine(Interactor,AnswerPattern,Goal)` that clears data areas an initializes an engine with `AnswerPattern,Goal` has also been available as a further optimization, by providing a mechanism to reuse an existing engine.

```
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. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the

```
stop(Interactor)
```

operation, that might or might not reclaim resources held by the engine. In our latest implementation *Lean Prolog*, we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected. While this API clearly refers to operations going beyond Horn Clause logic, it can be shown that a fairly high-level pure prolog semantics can be given to them in a style somewhat similar to what one would do when writing a Prolog interpreter in Haskell, as shown in section 4 of [22].

So far, these operations provide a minimal API, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [5], as well as higher order operations like *fold* and *best_of* [22]. We give more details on emulations of some key constructs in section 4.

3.3 A yield/return operation

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
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.

Note that an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/get` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism simply by defining

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

When combined with a `catch(Goal,Exception,OnException)`, on the client side, the client can decide, upon reading the exception with `get/2`, if it wants to handle it or to throw it to the next level.

3.4 Coroutining Logic Engines

Coroutining has been in use in Prolog systems mostly to implement constraint programming extensions. The typical mechanism involves *attributed variables* holding suspended goals that may be triggered by changes in the instantiation state of the variables. We discuss here a different form of coroutining, induced by the ability to switch back and forth between engines.

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's client to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine,Data)
```

that is called by the client to send data to an Engine, and

```
from_engine(Data)
```

that is called by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   (b) 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*
   (c) 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

Using a metacall mechanism like `call/1` (which itself can be emulated in terms of engine operations [5] or directly through a source level transformation [24]), one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine,Query, Result):-
  to_engine(Engine,Query),
  get(Engine,Result).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

The predicate `ask_engine/3` sends a query (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an `engine_yield` operation. The query is typically a goal or a pattern of the form `AnswerPattern:-Goal` in which case the engine interprets it as a request to instantiate `AnswerPattern` by executing `Goal` before returning the answer instance.

As the following example shows, this allows the client to use, from outside, the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1⇒S2),sum_loop(S2).

inc_test(R1,R2):-
   new_engine(_,sum_loop(0),E),
   ask_engine(E,(S1⇒S2:-S2 is S1+2),R1),
   ask_engine(E,(S1⇒S2:-S2 is S1+5),R2).

?- inc_test(R1,R2).
R1=the(0 ⇒ 2),
R2=the(2 ⇒ 7)
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed. The goal can then access the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

4 Source level extensions through new definitions

To give a glimpse to the expressiveness of the resulting Horn Clause + Engines language, first described in [5] we specify a number of built-in predicates known as "impossible to emulate" in Horn Clause Prolog (except by significantly lowering the level of abstraction and implementing something close to a Turing machine).

4.1 Negation, `once/1`, `if_then_else/3`

These constructs are implemented simply by discarding all but the first solution produced by a Solver.

```
% returns the(X) or the atom no as first solution of G
first_solution(X,G,Answer):-
  new_engine(X,G,Solver),
  get(Solver,Answer),
  stop(Solver).

% succeeds by binding G to its first solution or fails
once(G):-first_solution(G,G,the(G)).
```

```
% succeeds without binding G, if G fails
not(G):-first_solution(_,G,no).
```

The same applies to an emulation of Prolog's if-then-else construct, shown here in a simpler form as the predicate `if/3`.

```
% if Cond succeeds, call Then otherwise call Else
if(Cond,Then,Else):-
  new_engine(yes,Cond,Solver),
  get(Solver,Answer),stop(Solver),
  select_then_else(Answer,Then,Else,Goal),
  Goal.

select_then_else(the(yes),Then,_Else,Then).
select_then_else(no,_Then,Else,Else).
```

Note that these operations require the use of CUT in typical Prolog library implementations. On the other hand, in the presence of engines one does not need to use CUT very often - and when one does, it is mostly for efficiency reasons.

4.2 Reflective Meta-Interpreters

The simplest Horn Clause+Interactors meta-interpreter **metacall/1** just *reflects* backtracking through **element_of/2** over deterministic engine operations.

```
metacall(Goal):-
  new_engine(Goal,Goal,E),
  element_of(E,Goal).

element_of(E,X):-get(I,the(A)),select_from(E,A,X).

select_from(_,A,A).
select_from(E,_,X):-element_of(E,X).
```

We can see `metacall/1` as an operation which fuses two orthogonal language features provided by an engine: *computing an answer of a Goal*, and *advancing to the next answer*, through the source level operations `element_of/2` and `select_from/3` which 'borrow' the ability to backtrack from the underlying interpreter. The existence of the simple meta-interpreter defined by `metacall/1` indicates that first-class engines lift the expressiveness of Horn Clause logic significantly.

4.3 All-solution predicates

All-solution predicates like `findall/3` can be obtained by collecting answers through recursion. The (simplified) code consists of `findall/3` that creates an engine and `collect_all_answers/3` that recurses while new answers are available.

```
findall(X,G,Xs):-
   new_engine(X,G,E),
   get(E,Answer),
   collect_all_answers(Answer,E,Xs).

collect_all_answers(no,_,[]).
collect_all_answers(the(X),E,[X|Xs]):-
   get(E,Answer),
   collect_all_answers(Answer,E,Xs).
```

Note that after the auxiliary engine created for `findall/3` is discarded, heap space is needed only to hold the computed answers, as it is also the case with the conventional implementation of `findall`.

4.4 Term copying and instantiation state detection

As standardizing variables in the returned answer is part of the semantics of `get/2`, term copying is just computing a first solution to `true/0`. Implementing `var/1` uses the fact that only free variables can have copies unifiable with two distinct constants.

```
copy_term(X,CX):-first_solution(X,true,the(CX)).
var(X):-copy_term(X,a),copy_term(X,b).
```

The previous definitions have shown that the resulting language subsumes (through user provided definitions) constructs like negation as failure, if-then-else, once, `copy_term`, `findall` - this suggests calling this layer *Kernel Prolog*. As Kernel Prolog contains negation as failure, following [25] we can, in principle, use it for an executable specification of full Prolog.

It is important to note here that the engine-based implementation serves in some cases just as a proof of expressiveness and that, in practice, operations like `var/1` for which even a small overhead is unacceptable in a system, are implemented directly as built-ins. Nevertheless, the engine-based source-level definitions provide in all cases a reference implementation usable as a specification for testing purposes.

4.5 Implementing Exceptions

While it is possible to implement an exception mechanism at source level as shown in [26], through a continuation passing program transformation (binarization), one can use engines for the same purpose. By returning a new answer pattern as indication of an exception, an efficient, constant time implementation of exceptions is obtained.

We have actually chosen this implementation scenario in the LeanProlog compiler which also provides a **return/1** operation to exit an engine's emulator loop with an arbitrary answer pattern, possibly before the end of a successful derivation. The (somewhat simplified) code is as follows:

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

catch(Goal,Exception,OnException):-
  new_engine(answer(Goal),Goal,Engine),
  element_of(Engine,Answer),
  do_catch(Answer,Goal,Exception,OnException,Engine).

do_catch(exception(E),_,Exception,OnException,Engine):-
  (E=Exception->
    OnException % call action if matching
  ; throw(E)    % throw again otherwise
  ),
  stop(Engine).
do_catch(the(Goal),Goal,_,_,_).
```

The `throw/1` operation returns a special exception pattern, while the `catch/3` operation stops the engine, calls a handler on matching exceptions or re-throws non-matching ones to the next layer. If engines are lightweight, the cost of using them for exception handling is acceptable performance-wise, most of the time. However, it is also possible to reuse an engine (using `load_engine/3`) - for instance in an inner loop, to define a handler for all exceptions that can occur, rather than wrapping up each call into a new engine with a catch.

4.6 Interactors and Higher Order Constructs

As a first glimpse at the expressiveness of the Interactor API, we implement, in the tradition of higher order functional programming, a *fold* operation [27] connecting results produced by independent branches of a backtracking Prolog engine:

```
efoldl(Engine,F,R1,R2):-
  get(Engine,X),
  efoldl_cont(X,Engine,F,R1,R2).

efoldl_cont(no,_Engine,_F,R,R).
efoldl_cont(the(X),Engine,F,R1,R2):-
  call(F,R1,X,R),
  efoldl(Engine,F,R,R2).
```

Classic functional programming idioms like *reverse as fold* are then implemented simply as:

```
reverse(Xs,Ys):-
  new_engine(X,member(X,Xs),E),
  efoldl(E,reverse_cons,[],Ys).

reverse_cons(Y,X,[X|Y]).
```

Note also the automatic *deforestation* effect [28] of this programming style - no intermediate list structures need to be built, if one wants to aggregate the

values retrieved from an arbitrary generator engine with an operation like sum or product.

5 Extending the Prolog kernel using Interactors

We review here a few typical extensions of the Prolog kernel showing that using first class logic engines results in a compact and portable architecture that is built almost entirely *at source level*.

5.1 Emulating Dynamic Databases with Interactors

The gain in expressiveness coming directly from the view of logic engines as iterative answer generators is significant. The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

The key idea for implementing dynamic database operations with interactors is to use a logic engine's state in an infinite recursive loop.

First, a simple difference-list based infinite server loop is built:

```
queue_server:-queue_server(Xs,Xs).

queue_server(Hs1,Ts1):-
  from_engine(Q),server_task(Q,Hs1,Ts1,Hs2,Ts2,A),return(A),
  queue_server(Hs2,Ts2).
```

Next we provide the queue operations, needed to maintain the state of the database. To keep the code simpler, we only focus here on operations resulting additions at the end of the database.

```
server_task(add_element(X),Xs,[X|Ys],Xs,Ys,yes).
server_task(queue,Xs,Ys,Xs,Ys,Xs-Ys).
server_task(delete_element(X),Xs,Ys,NewXs,Ys,YesNo):-
  server_task_delete(X,Xs,NewXs,YesNo).
```

Then we implement the auxiliary predicates supporting various queue operations.

```
server_task_delete(X,Xs,NewXs,YesNo):-
  select_nonvar(X,Xs,NewXs),!,YesNo=yes(X).
server_task_delete(_,Xs,Xs,no).

select_nonvar(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
select_nonvar(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],
  select_nonvar(X,Xs,Ys).
```

Next, we put it all together, as a dynamic database API.

We can create a new engine server providing Prolog database operations:

```
new_edb(Engine):-new_engine(done,queue_server,Engine).
```

We can add new clauses to the database

```
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).
```

and we can return fresh instances of asserted clauses

```
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).
```

or remove them from the the database

```
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
  ask_engine(Engine,
    delete_element(Clause),the(yes(Clause))).
```

Finally, the database can be discarded by stopping the engine that hosts it:

```
edb_delete(Engine):-stop(Engine).
```

Externally implemented dynamic databases can also be made visible as Interactors and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases can be provided. This simplifies adding module, object or agent layers at source level. By combining database and communication Interactors, support for mobile code and autonomous agents can be built as shown in [29]. Encapsulating external stateful objects like file systems, external database or Web service interfaces as Interactors can provide a uniform interfacing mechanism and reduce programmer learning curves in practical applications of Prolog.

A note on practicality is needed here. While indexing can be added at source level by using hashing on various arguments, the relative performance compared to compiled code, of this emulated database is 2-3 orders of magnitude slower. Therefore, in our various Prolog systems we have used this more as an executable specification rather than the default implementation of the database.

5.2 Refining control: a backtracking if-then-else

Various Prolog implementations (SWI, SICStus, LeanProlog, Jinni Prolog etc.) also provide a variant of `if-then-else` that either backtracks over multiple answers of its `then` branch or switches to the `else` branch if no answers in the `then` branch are found. With the same API, we can implement it at source level as follows:

```
if_any(Cond,Then,Else):-
  new_engine(Cond,Cond,Engine),
  get(Engine,Answer),
  select_then_or_else(Answer,Engine,Cond,Then,Else).

select_then_or_else(no,_,_,_,Else):-Else.
select_then_or_else(the(BoundCond),Engine,Cond,Then,_):-
  backtrack_over_then(BoundCond,Engine,Cond,Then).
```

```
backtrack_over_then(Cond,_,Cond,Then):-Then.
backtrack_over_then(_,Engine,Cond,Then):-
  get(Engine,the(NewBoundCond)),
  backtrack_over_then(NewBoundCond,Engine,Cond,Then).
```

Note that, in contrast with the CUT used in implementing Prolog's conventional `If-Then-Else`, which forces `Then` to return at most one solution, the engine API allows for more flexible control.

5.3 Simplifying Algorithms: Interactors and Combinatorial Generation

Various combinatorial generation algorithms have elegant backtracking implementations. However, it is notoriously difficult (or inelegant, through the use of ad-hoc side effects) to compare answers generated by different OR-branches of Prolog's search tree.

Comparing Alternative Answers Optimization problems, selecting the "best" among answers produced on alternative branches can easily be expressed as follows:

- running the generator in a separate logic engine
- collecting and comparing the answers in a client controlling the engine

The second step can actually be automated, provided that the comparison criterion is given as a predicate

```
compare_answers(First,Second,Best)
```

to be applied to the engine with an `efold` operation:

```
best_of(Answer,Comparator,Generator):-
  new_engine(Answer,Generator,E),
  efoldl(E,compare_answers(Comparator),no,Best),
  Answer=Best.

compare_answers(Comparator,A1,A2,Best):-
  ( A1\==no,call(Comparator,A1,A2)->Best=A1
  ; Best=A2
  ).

?-best_of(X,>,member(X,[2,1,4,3])).
X=4
```

Encapsulating Infinite Computation Streams An infinite stream of natural numbers is implemented as:

```
loop(N):-return(N),N1 is N+1,loop(N1).
```

The following example shows a simple space efficient generator for the infinite stream of prime numbers:

```
prime(P):-prime_engine(E),element_of(E,P).

prime_engine(E):-new_engine(_,new_prime(1),E).

new_prime(N):-
  N1 is N+1,
  if(test_prime(N1),
    true,
    return(N1)
  ),
  new_prime(N1).

test_prime(N):-
  M is integer(sqrt(N)),between(2,M,D),N mod D =:=0
```

Note that the program has been wrapped, using the `element_of` predicate to provide one answer at a time through backtracking. Alternatively, a forward recursing client can use the `get(Engine)` operation to extract primes one at a time from the stream.

6 Some Interactor-based Practical Language Extensions

We sketch briefly some of the language extensions of LeanProlog and its derivatives facilitated by the use of Interactors as building blocks.

6.1 Interactors and Multi-Threading

As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog [30] and SWI-Prolog [31], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

While one can build a self-contained lightweight multi-threading API solely by switching control among a number of cooperating engines, with the advent of multi-core CPUs as the norm rather than the exception, the need for *native* multi-threading constructs is justified on both performance and expressiveness grounds. Assuming a dynamic implementation of a logic engine's stacks, interactors provide lightweight independent computation states that can be easily mapped to the underlying native threading API.

A minimal native interactor-based multi-threading API has been implemented on top of a simple thread launching built-in:

```
run_bg(Engine,ThreadHandle).
```

This runs a new thread starting from the engine's `run()` predicate and returns a handle to the Thread object. To ensure that access to the Engine's state is

safe and synchronized, we hide the engine handle and provide a simple producer/consumer data exchanger object, called a `Hub`. Some key components of the multi-threading API are:

- `bg(Goal)`: launches a new Prolog thread on its own engine starting with `Goal`.
- `hub(Hub)`: constructs a new `Hub` - a synchronization device on which `N` consumer threads can wait with `collect(Hub,Data)` (similar to a synchronized `from_engine` operation) for data produced by `M` producers providing data with `put(Hub,Data)` (similar to a synchronized `from_engine` operation).

6.2 Engine Pools

`Thread Pools` have been in use either at kernel level or user level in various operating system and language implementations to avoid costly allocation and deallocation of resources required by Threads. Likewise, for first-class logic engine implementations that cannot avoid high creation/initialization costs, it makes sense to build *Interactor Pools*. For this, an additional operation is needed allowing the *reuse* of an existing logic engine (`load_engine/3` in LeanProlog). An interactor pool can be maintained by a dedicated logic engine that keeps track of the state of various interactors and provides recently freed handles, when available, to `new_engine` requests.

7 Interactors and Logic Programming

We have shown that Logic Engines encapsulated as Interactors can be used to build on top of pure Prolog a practical Prolog system, including dynamic database operations, entirely at source level. We have also provided a sketch of an executable semantics for Logic Engine operations in pure Prolog. This shows that, in principle, their exact specification can be expressed declaratively.

In a broader sense, Interactors can be seen as a starting point for rethinking fundamental programming language constructs like Iterators and Coroutining in terms of language constructs inspired by *performatives* in agent oriented programming.

Beyond applications to logic-based language design, we hope that our language constructs will be reusable in the design and implementation of new functional and object oriented languages.

Among real world applications of these ideas, we have been pursuing a new model of natural language understanding [32] where multiple concurrently processing agents, using lightweight interpretation engines implemented as interactors transform text into semantic model structures for reasoning in the Oil and Gas exploration and production domain.

8 The LeanProlog Abstract Machine

We overview the *LeanProlog* system's compilation technology, runtime system and its extensions supporting first-class logic engines followed by a short history of its development and some of its newer re-implementations.

While LeanProlog's abstract machine diverges from the conventional WAM in a number of ways, the document discusses its orthogonal architectural features independently to facilitate selective reuse in future implementations.

We describe in detail *LeanProlog*'s compilation technique which replaces the WAM with a simplified *continuation passing* runtime system (the "BinWAM"), based on a mapping of full Prolog to *binary logic programs*, and a from of *term compression* using a "tag-on-data" representation mechanism.

9 Design Philosophy

LeanProlog's design philosophy has been minimalistic from the very beginning. In the spirit of Occam's razor, while developing an implementation as an iterative process, this meant not just trying to optimize for speed and size, but also to actively look for opportunities to refactor and simplify.

The guiding principle, at each stage, was seeking answers to questions like:

- what can be removed from the WAM without risking significant, program independent, performance losses?
- what can be done to match, within small margins, performance gains resulting from new WAM optimizations (like read/write stream separation, instruction unfolding, etc.) while minimizing implementation complexity and code size?
- can one get away with uniform data representations (e.g. no special tags for lists) instead of extensive specialization, without major impact on performance?
- what can be done to improve low level representations with impact on caching, register allocation, code and memory size?
- when designing new built-ins and extensions, can we use source-level transformations rather than changes to the emulator?

The first result of this design, LeanProlog's BinWAM abstract machine has been originally implemented as a C emulator based on a program transformation introduced in [24]. It replaces the WAM with a simplified continuation passing logic engine [33] based on a mapping of full Prolog to binary logic programs (binarization). Its key assumption is that *as conventional WAM's environments are discarded in favor of a heap-only run-time system, heap garbage collection and efficient term representation become instrumental as means to ensure ability to run large classes of Prolog programs and to improve performance by reducing memory bandwidth.* The second architectural novelty, present to some extent in the original LeanProlog and a key element of its predecessor Jinni Prolog [16, 34] is the use of Interactors (and first-class logic engines, in particular) as a uniform

mechanism for the source-level specification (and often actual implementation) of key built-ins and language extensions [5, 23, 22].

We first explore, in the following sections, various aspects of the compilation process and the runtime system. Next we discuss source-level specifications of key built-ins and extensions using first-class logic engines.

The next sections of this document are organized as follows.

Sections 10 and 11 overview LeanProlog's key source-to-source transformation (*binarization*) and its use in compilation.

Section 12 introduces LeanProlog's unusual "tag-on-data" term representation (16.3) and studies its impact on term compression (12.2).

Section 13 discusses optimizations of the runtime system like instruction compression (13.1) and the implicit handling of read-write modes (13.2).

Section 3 introduces logic engines as Interactors and describes their basic operations.

Section 4 applies Interactors to implement, at source level, some key Prolog built-ins, exceptions (4.5) and higher order constructs (4.6).

Section 5 applies the logic engine API to specify Prolog extensions ranging from dynamic database operations (5.1) and backtracking if-then-else (5.2) to predicates comparing alternative answers 5.3 and mechanisms to encapsulate infinite streams 5.3.

Section 6 discusses the use of engines to support multi-threading (6.1) and engine pools (6.2).

Section **??** gives a short historical account of LeanProlog and its derivatives.

Finally, section 20 discusses related work and section 21 concludes the document.

10 The binarization transformation

We start by reviewing the program transformation that allows compilation of logic programs towards a simplified WAM specialized for the execution of binary logic programs (called BinWAM from now on). We refer to [24] for the original definition of this transformation.

Binary clauses have only one atom in the body (except for some inline 'built-in' operations like arithmetics) and therefore they need no 'return' after a call. A transformation introduced in [24] allows to emulate logic programs with operationally equivalent binary programs.

To keep things simple we describe our transformations in the case of definite programs. First, we need to modify the well-known description of SLD-resolution [35] to be closer to Prolog's operational semantics. We follow here the notations of [36].

Let us define the *composition* operator $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument.

Let $\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n$ and $\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m$ be two clauses (suppose $n > 0, m \geq 0$). We define

$$(\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n) \oplus (\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m) = (\texttt{A}_0\texttt{:-B}_1\texttt{,...,B}_m\texttt{,A}_2\texttt{,...,A}_n)\theta$$

with $\theta = \text{mgu}(\texttt{A}_1,\texttt{B}_0)$. If the atoms $\texttt{A}_1$ and $\texttt{B}_0$ do not unify, the result of the composition is denoted as $\bot$. Furthermore, as usual, we consider $\texttt{A}_0\texttt{:-true,A}_2\texttt{,...,A}_n$ to be equivalent to $\texttt{A}_0\texttt{:-A}_2\texttt{,...,A}_n$, and for any clause C, $\bot \oplus \texttt{C} = \texttt{C} \oplus \bot = \bot$. We assume also that "$\oplus$" renames at least one operand to a variant with fresh variables before attempting unification.

This Prolog-like inference rule is called LD-resolution and it has the advantage of giving a more accurate description of Prolog's operational semantics than SLD-resolution.

Before defining the binarization transformation, we describe two auxiliary transformations.

The first transformation converts facts into rules by giving them the atom `true` as body. E.g., the fact `p` is transformed into the rule `p :- true`.

The second transformation, eliminates the metavariables[2] by wrapping them in a `call/1` predicate, E.g., a clause like `and(X,Y):-X,Y` is transformed into `and(X,Y) :- call(X),call(Y).`

The transformation of [24] (*binarization*) adds continuations as extra arguments of atoms in a way that preserves also first argument indexing.

Definition 1 *Let P be a definite program and Cont a new variable. Let T and $E = p(T_1, ..., T_n)$ be two expressions.*[3] *We denote by $\psi(E, T)$ the expression $p(T_1, ..., T_n, T)$. Starting with the clause*

(C) $\quad A : -B_1, B_2, ..., B_n.$

we construct the clause

(C') $\quad \psi(A, Cont : -\psi(B_1, \psi(B_2, ..., \psi(B_n, Cont))).$

The set P' of all clauses C' obtained from the clauses of P is called the binarization of P.

The following example shows the result of this transformation on the well-known 'naive reverse' program:

```
app([],Ys,Ys,Cont):-true(Cont).
app([A|Xs],Ys,[A|Zs],Cont):-app(Xs,Ys,Zs,Cont).

nrev([],[],Cont):-true(Cont).
nrev([X|Xs],Zs,Cont):-nrev(Xs,Ys,app(Ys,[X],Zs,Cont)).
```

Note that `true(Cont)` can be seen as a (specialized version) of Prolog's `call/1`.

These transformations preserve a strong operational equivalence with the original program with respect to the LD resolution rule which is *reified* in the syntactical structure of the resulting program, where the order of the goals in the body becomes hardwired in the representation. This means that each resolution step of an LD derivation on a definite program P can be mapped to an SLD-resolution step of the binarized program P'. The followin holds:

[2] Variables representing Prolog goals only known at run-time.
[3] Atom or term.

Proposition 1 *Let G be an atomic goal and $G' = \psi(G, true)$. Then, the computed answers obtained querying P with G, are the same as those obtained by querying P' with G'.*

Note that the equivalence between the binary version and the original program can also be explained in terms of fold/unfold transformations as suggested by [37].

11 Binarization based compilation

Clearly, continuations become explicit in the binary version of the program. We refer to [26] for a technique to access and manipulate them in an intuitive way, by modifying LeanProlog's binarization preprocessor. We focus here only on their uses in LeanProlog's compiler and runtime system.

11.1 Metacalls as built-ins

Note that the first step of the transformation simply wraps metavariables inside a new predicate `call/1`, as most Prolog compilers do. The second step adds continuations as last arguments of each predicate and a new predicate `true/1` to deal with unit clauses. During this step, the arity of all predicates increases by 1 so that, for instance, `call/1` becomes `call/2`. Although we can add, for each functor `f` occurring in the program, clauses like

```
true(f(...,Cont)):-call(Cont).
call(f(...),Cont):-f(...,Cont).
```

as an implementation of `true/1` and `call/2`, in practice it is simpler and more efficient to treat them as built-ins [33].

LeanProlog actually performs the execution of `call/2` and `true/1` inline, but it has to look up in a hash-table to map the term to which a meta-variable points to its corresponding predicate-entry. As this happens only when we reach a 'fact' in the original program, it has relatively little impact on performance, for typical recursion intensive programs.

Note also that by placing pairs of symbol/arity corresponding to functors and predicates occurring in the program in a dictionary it is possible to avoid this lookup. However, to keep LeanProlog's design simple and save space we have decided to only add just the symbols and not the symbol/arity combinations to a dictionary. Instead, we kept the arity information in the same word as the symbol index and the tag (see subsection 16.3) as a mechanism for speeding up unification and indexing. In this context, the hashing algorithm simply mapped 2 integers to a value and its efficiency was acceptable when using a fast customized hashing algorithm.

11.2 Inline compilation of built-ins

Demoen and Mariën pointed out in [38] that a more implementation oriented view of binary programs can be very useful: a binary program is simply one that does not need an environment in the WAM. This view leads to inline code generation (rather than binarization) for built-ins occurring *immediately after the head*. For instance something like

```
a(X):-X>1,b(X),c(X).
```

is handled as:

```
a(X,Cont) :- inline_code_for(X>1),b(X,c(X,Cont)).
```

rather than

```
a(X,Cont) :- '>'(X,1,b(X,c(X,Cont))).
```

Inline expansion of built-ins contributes significantly to LeanProlog's speed and supports the equivalent of WAM's last call optimization for frequently occurring linear recursive predicates containing such built-ins, as unnecessary construction of continuation terms on the heap is avoided for them.

11.3 Early term construction vs. late term construction

In procedural and call-by-value functional languages featuring only deterministic calls it was a typical implementation choice to avoid repeated structure creation by using environment stacks containing only the variable bindings. The WAM [39] follows this trend based on the argument that most logic programs are deterministic and therefore calls and structure creation in logic programming languages should follow this model. A more careful analysis suggests that the choice between

- late and repeated construction (standard WAMs with AND-stack)
- eager early construction (once) and reuse on demand as in BinWAM

favor different programming styles. Let us note at this point that the WAM's (common sense) assumptions are subject to the following paradox[4]:

- If a program is mostly deterministic then it will tend to fail only in the guards (shallow backtracking). In this case, when a predicate succeeds, all structures specified in the body of a selected clause will eventually get created. By postponing this, the WAM will be only as good as doing it eagerly upon entering the clause past guards (as in BinWAM).

[4] We assume here that the Prolog program is compiled simply as a set of Horn Clauses with guards that are executed inline. Disjunctions and if-then-else in the body and their possible customized compilation in modern WAMs introduce interesting additional nuances that we are not discussing here.

– If the program is mostly nondeterministic then late and repeated construction (WAM) is not better than early creation (BinWAM) which is done only once, because it implies more work on backtracking. While the BinWAM will only undo bindings to trailed variables in the binarized body represented on the heap, a conventional WAM will repeatedly push/pop to its environment stack[5].

This explains in part why a standard AND-stack based WAM is not necessarily faster than a carefully implemented BinWAM. We revisit this in detail when discussing data representations and runtime system optimizations.

11.4 A simplified run-time system

A simplified OR-stack having the layout shown in Fig. 1 is used only for (1-level) choice point creation in nondeterministic predicates.

P ⇒	next clause address
H ⇒	saved top of the heap
TR ⇒	saved top of the trail
A_{N+1} ⇒	continuation argument register
A_N ⇒	saved argument register N
...	...
A_1 ⇒	saved argument register 1

Fig. 1: A frame on LeanProlog's OR-stack.

Given that variables kept on the local stack in conventional WAM are now located on the heap, the heap consumption of the program increases. It has been shown that, in some special cases, partial evaluation at source level can deal with the problem [40, 41] but as a more practical solution, the impact of heap consumption has been alleviated in LeanProlog by the use of an efficient copying garbage collector [42].

11.5 A simplified clause selection and indexing mechanism

As the compiler works on a clause-by-clause basis, it is the responsibility of the loader (that is part of the runtime system) to index clauses and link the code. It

[5] There are ways to avoid some of the of push/pop operations to environment stacks and there are also additional trailing costs that are needed to take into account. Interestingly enough, when such extra trailing occurs in the BinWAM, it can be seen as (efficiently!) mimicking the stack operations of conventional WAM.

uses a global $< key, key > \rightarrow value$ hash table seen as an abstract *multipurpose dictionary*[6].

A one byte mark-field is used to distinguish between load-time use and run-time use and for fast clean-up. Sharing of the global dictionary, although somewhat slower than the small $key \rightarrow value$ hashing tables injected into the code-space of the standard WAM, had the advantage to keep implementation as simple as possible. Also, as data areas were of fixed size initially, one big table provided overall better use of the available memory by sharing the hashing table for different purposes.

The *2-key hashtable* is used by the run-time system for the following services:

- to get the addresses of meta-predicates
- to perform first argument indexing
- support a user-level storage area (called "blackboard") containing global terms

This high level of code reuse contributes significantly to the small size and, arguably, to the overall speed of LeanProlog due to locality of reference in the bytecode.

Predicates are classified as *single-clause*, *deterministic* and *nondeterministic*. Only predicates having *all first-argument functors distinct*, are detected as deterministic and indexed.

In contrast to the WAM's fairly elaborate indexing mechanism, indexing of deterministic predicates in the BinWAM is done by a unique SWITCH instruction.

If the first argument dereferences to a non-variable, SWITCH either fails or finds the 1-word address of the unique matching clause in the global hash-table, using the *predicate* and the *functor of the first argument* as a 2-word key. Note that the basic difference with the WAM is the absence of intensive tag analysis. This is related also to our different low-level data-representation that we discuss in section 12.

A specialized JUMP-IF instruction deals with the frequent case of 2 clause deterministic predicates. To reduce the interpretation overhead, SWITCH and JUMP_IF are combined with the preceding EXECUTE and the following GET_STRUCTURE or GET_CONSTANT instruction, giving EXEC_SWITCH and EXEC_JUMP_IF. This not only avoids dereferencing the first argument twice, but also reduces unnecessary branching that breaks the processor's pipeline.

Note also that simplification of the indexing mechanism helps making backtracking sometime faster in the BinWAM than in conventional WAMs. This comes from its smaller and unlinked choice points, but, as mentioned in subsection 11.3, also from the sharing of all structures occurring in the body of a clause in the OR-subtree it generates, instead of repeated creation as in conventional

[6] A typical use for the first key as the functor of the predicate and the second key as the functor of the first argument. Another one, when implementing multiple dynamic databases is to use the first key as the name of the database and the second as the functor of a predicate.

WAM. This is a known property first pointed out in [38] as the typical case when binarized variants are faster than the original programs.

Our original assumption when trimming down the WAM's indexing instructions was that, for predicates having a more general distribution of first-arguments, a source-to-source transformation, grouping similar arguments into new predicates, can be used.

Later in time, while noticing that often well written Prolog code tends to be either "database type" (requiring multiple argument indexing) or "recursion intensive" (with small predicates having a few clauses, fitting well this simplified first argument indexing mechanism) it became clear that it makes sense to handle these two problems separately. As a result, we have kept this simplified indexing scheme (for "recursion intensive" compiled code) unchanged through the evolution of LeanProlog and its derivatives. On the other hand, our newest implementation, *Lean Prolog* handles "database type" dynamic code efficiently using a very general multi-argument indexing mechanism.

11.6 Binarization: some infelicities

We have seen that binarization has helped building a simplified abstract machine that provides good performance with help from a few low level optimizations. However, there are some "infelicities" that one has to face, somewhat similar to what any program transformation mechanism induces at runtime - and DCG grammars come to one's mind in the Prolog world.

For instance, the execution order in the body is reified at compile time into a fixed structure. This means that things like dynamic reordering of the goal in a clause body or AND-parallel execution mechanisms become trickier. Also, inline compilation of things like if-then-else becomes more difficult - although one can argue that using a source-level technique, when available (like it is in this case, by creating small new predicates) is a acceptable implementation anyway.

12 Data representation

We review here an unconventional data representation choice that turned out to also provide a surprising term-compression mechanism, that can be seen as a generalization of "CDR-coding" used in LISP/Scheme systems.

12.1 Tag-on-pointer versus tag-on-data

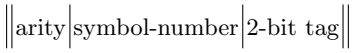
When describing the data in a cell with a tag we have basically 2 possibilities. We can put a tag in the same cell as the address of the data or near the data itself.

The first possibility, probably most popular among WAM implementors, allows one to check the tag before deciding *if* and *how* it has to be processed. We choose the second possibility as in the presence of indexing, unifications are more often intended to succeed propagating bindings, rather than being used as

a clause selection mechanism. This also justifies why we have not implemented traditional WAMs `SWITCH_ON_TAG` instruction.

We found it very convenient to precompute a functor in the code-space as a word of the form `<arity,symbol-number,tag>` [7] and then simply compare it with objects on the heap or in registers. In contrast, in a naively implemented WAM, one compares the tags, finding out that they are almost always the same, then compares the functor-names and finally compares the arities - an unnecessary but costly if-logic. Therefore we kept our unusual *tag-on-data* representation, while also ensuring to consuming as few tag bits as possible. Only 2 bits are used in LeanProlog for tagging Variables, Integers and Functors/Atoms[8].

With this representation a functor fits completely in one word:



As an interesting consequence, as we have found out later, when implementing a symbol garbage collector for a derivative of LeanProlog, the "tag-on-data" representation makes scanning the heap for symbols (and updating them in place) a trivial operation.

12.2 Term compression

If a term has a last argument containing a functor, with our tag-on-data representation we can avoid the extra pointer from the last argument to the functor cell and simply make them collapse. Obviously the unification algorithm must take care of this case, but the space savings are important, especially in the case of lists which become contiguous vectors with their N-th element directly addressable at offset `2*sizeof(term)*N+1` bytes from the beginning of the list, as shown in Fig. 2.

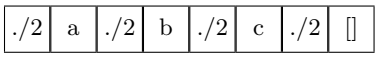


Fig. 2: Compressed list representation of [a,b,c]

The effect of this *last argument overlapping* on `t(1,t(2,t(3,n)))` is represented in Fig. 3.

This representation also reduces the space consumption for lists and other "chained functors" to values similar or better than in the case of conventional WAMs. We refer to [43] for the details of the term-compression related optimizations of LeanProlog.

[7] This technique is also used in various other Prologs e.g. SICStus, Ciao.

[8] This representation limits arity and available symbol numbers - a problem that, to some extent, went away with the newer 64-bit versions of LeanProlog.

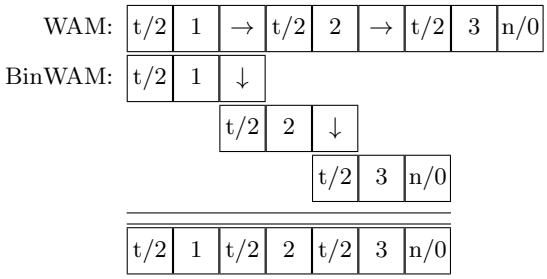


Fig. 3: Term compression.

13 Optimizing the run-time system: reducing the interpretation overhead

One can argue that the best way to reduce the interpretation overhead is by not doing it at all. However, the native code compilers we know of have a *compact-code* option that is actually emulated WAM. And being the most practical in term of compilation-time and code-size, this mode is used by the Prolog developer most of the time, except for the final product. On the other hand, some of the techniques that follow can be used to eliminate tests and jumps so they can be useful also in native code generation.

13.1 Instruction Compression

It happens very often that a sequence of consecutive instructions share some WAM state information. For example, two consecutive unify instructions have the *same mode* as they correspond to arguments of the same structure. Moreover, due to our very simple instruction set, some instructions have only a few possible other instructions that can follow them. For example, after an EXECUTE instruction, we can have a single, a deterministic or a nondeterministic clause. It makes sense to specialize the EXECUTE instruction with respect to what has to be done in each case. This gives, in the case of calls to deterministic predicates the instructions EXEC_SWITCH and EXEC_JUMP_IF as mentioned in the section on indexing. On the other hand, some instructions are simply so small that just dispatching them can cost more than actually performing the associated WAM-step.

This in itself is a reason to compress two or more instructions taking less than a word in one instruction. This optimization has been part of WAM-based Prolog systems like Quintus, SICStus, Ciao as well. Also having a small initial instruction set reduces the number of combined instructions needed to cover all cases. For example, by compressing our UNIFY instructions and their WRITE-mode specializations, we get the following 8 new instructions:

`UNIFY_VARIABLE_VARIABLE`

```
WRITE_VARIABLE_VARIABLE
UNIFY_VALUE_VALUE
WRITE_VALUE_VALUE
UNIFY_VARIABLE_VALUE
WRITE_VARIABLE_VALUE
UNIFY_VALUE_VARIABLE
WRITE_VALUE_VARIABLE
```

This gives, in the case of the binarized version of the recursive clause of append/3, the following code:

```
append([A|Xs],Ys,[A|Zs],Cont):-append(Xs,Ys,Zs,Cont).

TRUST_ME_ELSE */4,     % keeps also the arity = 4
GET_STRUCTURE X1, ./2
UNIFY_VARIABLE_VARIABLE X5, A1
GET_STRUCTURE X3, ./2
UNIFY_VALUE_VARIABLE X5, A3
EXEC_JUMP_IF  append/4 % actually the address of append/4
```

The choice of candidates for instruction compression was based on low level profiling (instruction frequencies) and possibility of sharing of common work by two successive instructions and frequencies of functors with various arities.

LeanProlog also integrates the preceding GET_STRUCTURE instruction into the double UNIFY instructions and the preceding PUT_STRUCTURE into the double WRITE instructions. This gives another 16 instructions but it covers a large majority of uses of GET_STRUCTURE and PUT_STRUCTURE.

```
GET_UNIFY_VARIABLE_VARIABLE
...
PUT_WRITE_VARIABLE_VALUE
....
```

Reducing interpretation overhead on those critical, high frequency instructions definitely contributes to the speed of our emulator. As a consequence, in the frequent case of structures of arity=2 (lists included), mode-related IF-logic is completely eliminated.

The following example shows the effect of this transformation:

```
a(X,Z):-b(X,Y),c(Y,Z). ⇒binary form⇒ a(X,Z,C):-b(X,Y,c(Y,Z,C)).

LeanProlog BinWAM code, without compression

a/3:
PUT_STRUCTURE        X4←c/3
WRITE_VARIABLE       X5
WRITE_VALUE          X2
WRITE_VALUE          X3
MOVE_REG             X2←X5
MOVE_REG             X3←X4
EXECUTE              b/3
```

```
LeanProlog BinWAM code, with instruction compression

PUT_WRITE_VARIABLE_VALUE  X4←c/3, X5,X2
WRITE_VALUE               X3
MOVE_REGx2                X2←X5, X3←X4
EXECUTE                   b/3
```

Not that, to some extent, the elaborate case analysis (safe vs. unsafe, x-variable vs. y-variable) makes the job of advanced instruction compression tedious in the case of standard WAM. Also, in the case of an emulated engine, just decoding the `init`, `allocate`, `deallocate` and `call` instructions might cost more than LeanProlog's simple `PUT_STRUCTURE` and `WRITE_VAR_VAL` and their straightforward IF-less work on copying 3 heap cells from the registers.

A very straightforward compilation to C [44] and the possibility of optimized 'burst-mode' structure creation in PUT instructions can be seen as facilitated by binarization. Arguably, such techniques would be harder to apply to AND-stack based traditional WAMs, which exhibit less uniform instruction patterns.

Simplifying the unification instructions of the BinWAM allows for very "general-purpose" instruction compression. Conventional WAMs often limit this kind of optimization to lists [9]. Overall, in a simplified engine, instruction compression can be made more "abstract" and therefore with fewer compressed instructions one can hit a statistically more relevant part of the code. In LeanProlog, for instance, arithmetic expressions or programs manipulating binary trees will benefit from our compression strategy while this may not be the case with conventional WAMs, unless they duplicate list-instruction optimizations for arbitrary structures.

An other point is that instruction compression is usually applied inside a procedure. As LeanProlog has a unique primitive EXECUTE instruction instead of standard WAM's CALL, ALLOCATE, DEALLOCATE, EXECUTE, PROCEED we can afford to do instruction compression across procedure boundaries with very little increase in code size due to relatively few different ways to combine control instructions. Inter-procedural instruction compression can be seen as a kind of 'hand-crafted' *partial evaluation* at implementation language level, intended to optimize the main loop of the WAM-emulator. This can be seen as a special case of the *call forwarding* technique used in the implementation of `jc` [45, 46]. It has the same effect as *partial evaluation* at source level which also eliminates procedure calls. At global level, knowledge about possible continuations can also remove the run-time effort of address look-up for meta-predicates and useless trailing and dereferencing.

13.2 (Most of) the benefits of two-stream compilation for free

Let us point out here that in the case of GET_*_* instructions we have the benefits of separate READ and WRITE streams (for instance, avoidance of mode

[9] One can find out about the impact of this by changing the list-constructor of the NREV benchmark.

checking) on some high frequency instructions without actually incurring the compilation complexity and emulation overhead in generating them. As terms of depth 1 and functors of low arity dominate statistically Prolog programs, we can see that our instruction compression scheme actually behaves as if two separate instruction streams were present, most of the time!

13.3 Case Overlapping

Case-overlapping is a well-known technique that saves code-size in the emulator. Note that we can share within the main `switch` statement of the emulator a `case` when an instruction is a specialization of its predecessor, based on the well known property of the `case` statement in C, that in the absence of a `break;` or `continue;` control flows from a `case` label to the next. It is a 0-cost operation. The following example, from our emulator, shows how we can share the code between EXEC_SWITCH and EXEC.

```
case EXEC_SWITCH:
  .........
case SWITCH:
  .........
break;
```

Note that instructions like EXEC_SWITCH or EXEC_JUMP_IF have actually a global compilation flavor as they exploit knowledge about the procedure which is called. Due to the very simple initial instruction set of the LeanProlog engine these WAM-level code transformations are performed in C at load-time with no visible penalty on compilation time.

Finally, we can often apply both instruction compression and case-overlapping to further reduce the space requirements. As compressed WRITE-instructions are still just special cases of corresponding compressed UNIFY-instructions we have:

```
case UNIFY_VAR_VAL:
  .........
case WRITE_VAR_VAL:
  .........
break;
```

13.4 The benefits of simplification

In conclusion, the simplicity LeanProlog's basic instruction set due to its specialization to binary programs allowed us to apply low level optimizations not easily available on standard WAM. By applying simple optimizations like moving metapredicates as built-ins at WAM-level [47] to reduce the overhead introduced by binarization, the LeanProlog engine has proven itself as a viable simpler alternative to standard WAM, possibly explaining its relative popularity in new implementations (see section **??**).

14 Comparison with other Prolog Implementations

Most modern Prolog implementations are centered around the Warren Abstract Machine (WAM) [39, 48] which has stood amazingly well the test of time. In this sense LeanProlog's BinWAM is no exception, although its overall "rate of mutations" with respect to the original WAM is probably comparable to systems like Neng-Fa Zhou's TOAM or TOAM-Jr [49, 50] or Ian Wielemaker's SWI-Prolog [31] architectures and definitely higher, if various extensions are factored out, than the basic architecture of systems like GNU-Prolog [51], SICStus Prolog [52], Ciao [30], YAP [53] or XSB [54]. We refer to [55] and [56] for extensive comparisons of compilation techniques and abstract machines for various logic programming systems.

Techniques for adding built-ins to binary Prolog are first discussed in [38], where an implementation oriented view of binary programs that a binary program is simply one that does not need an environment in the WAM is advocated. Their paper also describe a technique for implementing Prolog's CUT in a binary Prolog compiler. Extensions to LeanProlog's AND-continuation passing transformation to also cover OR-continuations are described in [57].

Multiple Logic Engines have been present in a from or another in various parallel implementation of logic programming languages [58–60]. Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel [20] and OR-parallel [21] execution models are worth mentioning.

However, with the exception of [16, 61, 5, 29, 23, 22] we have not found an extensive use of first-class logic engines as a mechanism to enhance language expressiveness, independently of their use for parallel programming, with maybe the exception of [62] where such an API is discussed for parallel symbolic languages in general.

In combination with multithreading our own engine-based API bears similarities with various other Prolog systems, notably [30, 31].

The use of a garbage collected, infinitely looping recursive program to encapsulate state goes back to early work in logic programming and it is likely to be common in implementing various server programs. However, like a black hole, an infinitely recursive pure Horn Clause program will not communicate with the outside world on its own. The minimal API (`to_engine/2` and `from_engine/1`) described in this document provides interoperation with such programs, in a generic way.

15 Integrated Symbol, Engine Table and Heap Memory Management in Multi-Engine Prolog

We describe an integrated solution to symbol, heap and logic engine memory management in a context where exchanges of arbitrary Prolog terms occur between multiple dynamically created engines, implemented in a new Java-based experimental Prolog system.

As our symbols represent not just Prolog atoms, but also handles to Java objects (including arbitrary size integers and decimals), everything is centered around a symbol garbage collection algorithm ensuring that external objects are shared and exchanged between logic engines efficiently.

Taking advantage of a *tag-on-data* heap representation of Prolog terms, our algorithm performs in-place updates of live symbol references directly on heap cells.

With appropriate fine tuning of collection policies our algorithm provides a simple integrated memory management solution for Prolog systems, with amortized cost dominated by normally occurring heap garbage collection costs.

Symbol garbage collection is particularly important in practical applications of declarative languages that rely on internalized symbols as their main building blocks. In the case of Prolog, applications as diverse as natural language tools, XML processors, database interfaces and compilers rely on dynamic symbols (atoms in Prolog parlance) to represent everything from tokens and graph vertices to predicate and function names. A task as simple as scanning for a single Prolog clause in a large data file can break a Prolog system not enabled with symbol garbage collection.

The use in the implementation language of packages providing arbitrary length integers and decimals to support such data types in Prolog brings in similar memory management challenges. While it is common practice in Prolog implementations to represent fixed size numerical data directly on the heap (given the benefits of quick memory reclamation on backtracking), conversion from arbitrary size integers or decimals to serialized heap representations tends to be costly and can add complexity to the implementation[10].

Such problems can become particularly severe in multi-engine Prolog (defined here roughly as any Prolog system with multiple heap/stack/trail data areas) where design decisions on symbol memory management are unavoidably connected to decisions on symbol sharing mechanisms and engine life-cycle management. It is also important in this scenario to support sharing of data objects among engines and avoid copying between heaps as well as serialization/deserialization costs of potentially large objects.

Often, reference counting mechanisms have been used for symbol garbage collection. A major problem is that if the symbol table is used for non-atomic objects like handles to logic engines or complex Java objects that may also refer to other such handles, cycles formed by dead objects may go undetected. Another problem is that reference counting involves extensive changes to existing code - as every single use of a given variable in a built-in needs to be made aware of it.

These considerations suggest the need for an integrated solution to symbol and engine garbage collection as well as exchange of arbitrary Prolog terms between multiple engines.

---

[10] While serialization can be avoided in C-based systems using pointers to "blobs" on the heap and type castings, this is not an option in strongly typed Java.

We will now describe the implementation of such a solution in *Lean Prolog* that provides sharing among engines of arbitrary size external data, as divers as collections, graphs, GUI components, arbitrary precision integers and decimals.

We will discuss our design decisions and algorithms in the context of *Lean Prolog*'s lightweight BinWAM-based [24, 47] runtime system, a minimalist Java kernel using logic engines as first order building blocks encapsulated as *interactors* [22, 23].

One might still legitimately ask: why do we need heap and symbol garbage collection in a Java-based Prolog implementation, when, by using Java objects to represent Prolog terms, Java itself provides automatic memory management?

The original reason for dropping the compilation model used by jProlog[11] and some of the derivatives like Prolog Cafe [64] or `P#` [65, 66] was that Java objects where too heavyweight for basic Prolog abstract machine functions and we observed that a relatively plain, C-style integer based runtime system performs an order of magnitude faster, especially in the presence of "just-in-time" and later "HotSpot" java compilers.

Unfortunately, giving up on Java objects for a low-level BinWAM engine [67, 68] meant also having to manage memory directly. On one hand, as an advantage, the Java-based *Lean Prolog* model can be moved seamlessly to faster languages like `C` [12]. On the other hand, memory management becomes almost as complex as in `C`-based Prologs. In the case of our *Lean Prolog* implementation, this involves dynamic array management as well as heap and symbol garbage collection, the last task including also recovery of memory used by of unreachable logic engines.

Fortunately, a number of simplifications of our runtime architecture, like separation of engines and threads (subsection 16.2) and a *tag-on-data* term representation (subsection 16.3) allow for *naive* shortcuts to potentially tricky memory management decisions within good performance margins. Some of these decisions also lead to additional benefits like efficient engine-to-engine communication and a uniform handling, *as symbols*, of arbitrary external objects including maps, big numbers and logic engines (subsection 16.4).

As our approach to dynamic data areas and heap garbage collection (abbreviated from now on GC) is similar to typical Prolog systems, our focus will be on the symbol GC policy and algorithm and its integration with other memory management tasks.

The paper is organized as follows.

Section 16 overviews aspects of architecture of *Lean Prolog* that are relevant for the decisions involved in the design of our symbol GC algorithm and policy.

Section 17 first outlines in subsection 17.1, and then describes the major components of the symbol GC algorithm (17.2 opportunity detection, 17.3 work delegated to engines, and 17.4 work in the class implementing the atom table). Subsection 17.5 focuses on the "fine tuning" of our symbol GC policy.

---

[11] A BinWAM-based research prototype written by Bart Demoen in collaboration with the author back in 1996, [63].

[12] In fact, a `C` variant of *Lean Prolog* is now in the works, already covering pure Prolog + CUT + engines after just a few weeks of effort.

Section 19 discusses an empirical evaluation of the costs and benefits of the integration of symbol GC with other memory management tasks.

Finally, sections 20 and 21 discuss related work and conclude the paper.

16 Memory Management of *Lean Prolog*

We briefly overview the architecture of our Prolog implementation as it is relevant to the description of the memory management aspects that the paper will explore in detail.

Lean Prolog is based on a compositional, agent oriented architecture, centered around a minimalistic Java-based kernel and autonomous computational entities called *Interactors*. They encapsulate in a single API stateful objects as diverse as first class logic engines, Prolog's dynamic database, the interactive Prolog console, as well as various stream processors ranging from tokenizers and parsers to process-to-process and thread-to-thread communication layers. The Java-based kernel is extended with a parser, a compiler and a set of built-ins written in Prolog that together add as little as 40-50K of compressed Prolog byte-code. This design fits easily within the memory constraints of the hundred millions of resource limited embedded Java processors found in today's mobile appliances as well as those using Google's new Java-centered Android operating system.

16.1 The Multi-Engine API

Our *Engines-as-Interactors API* has evolved progressively into a practical Prolog implementation framework starting with [5] and continued with [23] and [22]. We will summarize it here while focusing on the interoperation of Logic Engines.

A *Logic Engine* is simply a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive top-level loop: launch a new goal, ask for a new answer, interpret it, react to it. Each *Logic Engine* runs a Horn Clause interpreter with LD-resolution [13] on a given clause database, together with a set of built-in operations. Engines are designed to be extensible in a modular way, through inheritance or delegation mechanisms, to provide additional functionality ranging from GUI components and IO to multithreading and remote computations.

The API provides commands for creating a new Prolog engine encapsulated as an `Interactor`, which shares code with the currently running program and is initialized with a given goal as a starting point. Upon request from their parent (a `get` operation), engines return *instances of an answer pattern* (usually a list of variables occurring in the goal), but *they may also return Prolog terms at arbitrary points in their execution.* In both cases, they suspend, waiting for new requests from their parent. After interpreting the terms received from an engine, the parent can, at will, resume or stop it. Such mechanisms are used, for instance, to implement exceptions at source level [5].

The operations described so far allow an engine to return answers from any point in its computation sequence, in particular when computed answers are found. An engine's parent can also *inject* new goals (executable data) to an arbitrary inner context of an engine with help of primitives used for sending a parent's data to an engine and for receiving a parent's data.

Note that bindings are not propagated to the original goal i.e. fresh instances are *copied* between heaps. Therefore, backtracking in the parent interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

16.2 Cooperative vs. preemptive uses of engines

A typical "cooperative" multitasking use case of the engine API is as follows:

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

As described in [5, 23, 22] the Interactor API encapsulates the essential building blocks that one needs beyond Horn Clause logic to build a practical Prolog system, mostly at source level and with surprisingly low performance hits. Built-ins like findall, setof, copy_term, catch/throw, assert/retract etc. can be all covered at source level, and even if performance considerations require faster native implementations, one can use the source level variants as specifications for testing and debugging.

An important feature of our *Lean Prolog* implementation is the decoupling of *engines* and *threads*. While it is possible to launch a logic engine as a separate thread, they are also heavily used in some built-ins like `findall` in a sequential setting, and, in the case of a new `lazy_findall` built-in, as cooperating coroutines.

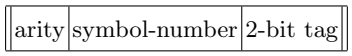
This decoupling removes the need of thread synchronization in cases where engines are used in sequential or cooperative multitasking operations and allows for organizing multi-threading as a separate layer where synchronization and symbol garbage collection are aware of each other.

16.3 Symbol GC in the presence of the *tag-on-data* term representation

When describing the data in a heap cell with a tag we have basically 2 possibilities. One can put a tag in the same cell as the address of the data (pointer) or near the data itself.

The first possibility, probably most popular among WAM implementors, allows one to check the tag before deciding *if* and *how* it has to be processed. Like in our previous Prolog implementations [4, 16, 3] we choose the second possibility, which also supports a form of term compression [67].

At the same time, it is convenient to precompute a functor in the code-space as a word of the form `<arity, symbol-number, tag>` and then simply compare it with objects on the heap or in registers[13]. Only 2 bits are used in *Lean Prolog* for tagging variables, small integers and functors/atoms. With this representation a functor or atom fits completely in one word:



As an interesting consequence, useful for symbol GC, the "tag-on-data" representation makes scanning the heap for symbols (and updating them in place) a trivial operation.

16.4 The case for internalizing all Java objects as symbols

Besides Prolog's atoms and functors, various Java objects can be *internalized* by mapping them to integers using hashing. Such integers are much "lighter" than Java's objects i.e. once one makes sure that a unique integer is assigned to each external object at creation time, using them in Prolog operations like unification or indexing becomes quite efficient.

Once the decision to have a symbol garbage collector is made, a number of consequences on the implementation follow, that break away from the usual wisdom that has been distilled over a few decades of Prolog implementation experience:

- serializing has minimal costs for an array of integers, but serializing an object graph containing complex objects, like HashMaps, TreeMaps or Java3D scene hierarchies is likely to be costly - that makes placing such object on the heap less appealing
- while in a single-engine implementation heap reclamation on backtracking efficiently discards heap represented objects, the lifespan of objects created in an engine might extend over the lifespan of the engine
- placing an object in the symbol table is essentially a lazy operation, in contrast to eager serialization - what if the control flow never reaches the object - and the effort to serialize it is spent in vain?

[13] This technique is also used in various other Prologs e.g. SICStus, Ciao.

One is then tempted by the following architectural choice: *if symbol garbage collection is available and sharing is possible and needed between multiple independent computations, then all external objects (not just string atoms) can be treated as Prolog symbols.*

Besides simplifying implementation of arbitrary size integers and decimals, internalizing everything provides cheap unification, as equality tests are reduced to integer comparisons and bindings to integer assignments. In particular, *internalizing logic engines* (Java objects at implementation level) allows treating them as any other symbols subject to garbage collection. This avoids likely memory leaks resulting from programmers forgetting to explicitly delete unused engines.

17 The multi-engine symbol garbage collection algorithm

We will first state a few facts that allow some flexibility with the policies on deciding *if* symbol GC should be performed and also on *when* that should happen.

Proposition 2 *If a program creates new symbols and no backtracking occurs, even in a multi-engine Prolog, they will eventually end up in the registers or the heap of at least one of the engines.*

Proposition 3 *Checking for the opportunity to call the symbol GC algorithm, given that a flag has been raised by the addition of a new symbol, needs only to happen when either:*

- *heap GC occurs in at least one engine*
- *at least one engine backtracks*

Proposition 4 *If a symbol does not occur on the heap, the registers or in the registers saved in the choice points of any live engine, then the symbol can be safely reclaimed.*

The multi-engine aspect of triggering the Symbol GC algorithm is covered by the following fact, easily enforced by an implementation:

Proposition 5 *Given any chain of engine calls, happening all cooperatively within the same thread, executing the Symbol GC algorithm when one of the engines calls the heap GC, or when one of the engines backtracks, results in no live symbols being lost, if the heaps of all the engines are scanned at that point.*

Together, these assertions ensure that symbol GC can safely wait until "favorable" conditions occur, resulting in increased efficiency and overhead reduction.

As a side note, we have in *Lean Prolog* two implementations of the dynamic database that a user can chose from: one is a lightweight, engine based, all source level dynamic database [22]. The other one is a higher performance, multi-argument indexed external database that relies on Java's garbage collector and manages its symbols internally. Interestingly, both benefit from the work of the symbol GC, although for different reasons. In the first case, symbols in the

database are handled by our collector as any other symbols on the heap of an engine. In the second case, symbols are lazily internalized, only for the dynamic clauses that have passed all the indexing tests - usually a small subset. In this case, from the symbol GC's perspective, database symbols are handled the same way as if read from a file or a socket.

17.1 Outline of the Symbol GC algorithm

As it is typical with GC algorithms, there are two phases:

1. heuristically recognizing that too many symbols or engines have been created since the previous collection (or being forced by a dramatic shortage of memory) - case in which a flag - let's call it `symgc_flag` is set to true
2. waiting within provably safe bounds until the actual garbage collection can be performed

As we want to make the symbol garbage collection available upon user request from a goal or the interactive prompt, we also need to ensure that the collector can be called safely right away in that case.

The heap garbage collector used in *Lean Prolog* is a simple mark and sweep algorithm along the lines of [69]. As most GC implementations, it competes with heap expansion in the sense that at a given time a decision is made if one or the other is retained as a solution to a heap overflow. On the other hand, we have avoided tight coupling of our GC algorithm with symbol table expansion, partly because we wanted to be free to use Java's Map libraries like HashMap or possibly other Map implementations for our symbol tables, in the future. This has also simplified the decision logic and helped us to separate the detection of the need for symbol GC, from the enactment of the collection process.

Note that while engines are internalized as symbols a separate *engine table* is kept providing the *roots* for the GC process.

The outline of our algorithm is as follows:

- detect the need for symbol GC (automatically or on user demand) and raise the `symbol_gc` flag
- collect to the new table all live symbols from the heaps of all the live engines and relocate in the process all heap references using symbols numbers in the new table
- remove all dead engines from engine tables
- replace the old symbol table with the new table

We will now expand this outline, filling in the details as needed.

17.2 Detecting the need for Symbol GC

The `symbol_gc` flag is raised a new object is added to the symbol table (by the `addObject` method) or a new engine is added by the `addEngine` and some heuristic conditions are met. We will postpone the details of the *policy* describing how

to fine tune such heuristics to subsection 17.5. These methods are *synchronized* to ensure only one thread uses them at a given time, as needed in case some of the engines run on different threads. Note however that it is ok if multiple threads raise this flag independently as we will only act on it when opportunity to do it safely is detected.

We start by describing the work done by individual engines as this is the simplest part of the algorithm.

17.3 Performing the symbol GC: work delegated to the engines

Individual engines are given the task to collect their live symbols. These symbols can be in one of the following *root* data areas.

- WAM registers
- temporary registers used for arguments of inlined built-ins
- in registers saved in choice points
- on the heap

Note that while the BinWAM [33] does not use a local stack, an adaptation to conventional WAMs might require scanning for possible symbol cells there as well. Anyway, experiments to move local and choice point data to the heap have been also initiated for conventional WAMs as shown in [70].

The scanning algorithm simply adds all the symbols found in these areas to the new symbol table. At the same time, some "heap surgery" is performed: the integer index of the symbol pointing to the old symbol table is replaced by the integer index that we just learned as being its location in the new symbol table. The same operation is applied to all roots. In our case, this is facilitated by the `tag-on-data` [67] representation in the BinWAM but it can be (with some care!) adapted also to Prologs using the conventional WAM's `tag-on-pointer` scheme.

17.4 Performing the GC: as seen from the class implementing the atom table

One can infer from Prop. 3 that we can avoid multiple flag testings in the inner loop of the emulators by only adding the test for the `symbol_gc` flag *after a heap garbage collection occurs* and when moving from a clause to the next, *on backtracking*.

We now focus on the tasks encapsulated in the class `AtomTable` a Map that *manages our symbols* and has access to the set logic engines contained in a separate Map.

First, a new `AtomTable` instance, called `keepers`, is created. This will contain the *reachable* symbols, that we plan to keep alive in the future.

The `keepers` table is preinitialized with all the compile time symbols, including built-in predicates, I/O interactors, database handles etc[14].

[14] A total of about 1250 symbols in the case of our *Lean Prolog* runtime system. This includes 2 engines, the parent driving the top-level and the worker used to catch exceptions on running goals entered by the user.

Next we iterate over all the engines and *perform the steps described in subsection 17.3*.

If an engine (with a handle also represented as a symbol) has been stopped by exhausting all its computed answers, or deliberately by another engine (typically the parent) we skip it.

If an engine is `protected` i.e. *it is the root of an independent thread group or managing the user's top-level*, the engine is added to `keepers`.

If the engine has made it so far, the task to filter the current symbol table will be delegated to it. We will defer the details of this operation to the next section.

A check for self-referential engines is made at this point: if the engine did not make it into `keepers` before scanning its own roots and it made it there after, it means that it is the only reference to itself (like through a pending current_engine(E) goal, in the continuation still on the heap). In this case, the engine is removed from `keepers`.

The next steps involve removal (and killing) of dead engines.

An engine qualifies as dead if it is not a protected engine and it is stopped, as well as if it is unreachable from the new symbol table. At this point an engine's `dismantle()` method is called that discards all resources held by the engine[15].

Finally, the new symbol table replaces the old one and threads possibly waiting on the symbol GC are given a chance to resume.

17.5 Fine tuning the activation of the symbol collector

A simple scenario for symbol GC is to be always user activated. The `symgc` built-in of *Lean Prolog* does just that. A priori, this is not necessarily bad, users of a bare-bone Prolog system can learn very quickly that most new symbols are brought in by using operations like `atom_codes/2` and reading/writing from/to various data sources.

However, once the symbol GC algorithm is there and working flawlessly, few implementors can resist the temptation to design various extensions under this assumption.

In the case of *Lean Prolog*, components ranging from arbitrary length integer arithmetic to the indexed external database rely on symbol GC. Components like the GUI use symbols as handles to buttons, text areas, panels etc. The same applies to file processing, sockets and thread control. Moreover, *Lean Prolog*'s reflection API, built along the lines of [18], makes available arbitrary Java objects in the form of Prolog symbols. And, on top of that, we have dynamic creation of new Prolog engines that can be stopped at will. As engines are first order citizens, they also have a place in the symbol table to allow references from other engines.

[15] The main difference is that a stopped engine can still be queried, in which case it will indicate that no more answers are available. In contrast, trying to query a dismantled engine would be an indication of an error in the runtime system, generating an exception.

Clearly, predicting the dynamic evolution of this ecosystem of symbols with a wide diversity of life-spans and functionalities cannot be left entirely to the programmer anymore.

In this context, the fine-tuning of the mechanism that automatically initiates symbol GC i.e. a sound collection *policy* becomes very important. The process is constrained by two opposite goals:

- ensure that memory never overflows because of a missed symbol GC opportunity
- ensure that the relatively costly symbol GC algorithm is not called unnecessarily
- the GC initiation algorithm should be simple enough to be able to prove that invariants like the above, hold

We will now outline our symbol GC policy, guided by the aforementioned criteria.

First, we ensure that the symbol GC process should not be called from threads that try to add symbols when it is already in progress. This is achieved by atomically testing/setting a flag.

We will also avoid going further if the size of the (dynamically growing) symbol table is still relatively small[16] or if the growth since last collection is not large enough[17].

On the other hand, upon calling the `addEngine()` method, one has to be more aggressive in triggering the symbol collector that also collects dead engines, given that recovering engines not only brings back significant memory chunks, but it also has the potential to free additional symbols. A heuristic value (currently an increase of the number of new engines by 256), is used. As engine number increases are usually correlated with generation of new symbols, this does not often bring in unnecessary collections[18].

Next, by iterating over all live (i.e. not stopped) engines, we compute the sum of their heap sizes. If the size of the symbol table exceeds a significant fraction of the total heap size[19], it is likely that we have enough garbage symbols to possibly warrant a collection, given that we can infer that live symbols should be somewhere on the heaps. Next we estimate the relative cost of performing the symbol GC and we decline the opportunity if the GC has been run too recently[20].

Otherwise we schedule a collection by raising the `sym_gc` flag.

---

[16] This is decided by checking against a compile time constant `SYMGC_MIN`.

[17] This is decided by checking against a compile time constant `SYMGC_DELTA`.

[18] Nevertheless, future work is planned to dynamically fine tune this parameter.

[19] A compile time constant empirically set to 0.25, planned also to be dynamically fine tuned in the future.

[20] This is computed by the number of discarded attempts to initiate symbol GC since the time it has actually been performed, currently a heuristic constant set to 10.

17.6 An optimization: synergy with copying heap garbage collectors

An opportunity to run our symbol collector arises right after heap garbage collection. While we are using a mark-and-sweep collector in *Lean Prolog*, it is noteworthy to observe that in the case of a copying collector, for instance [42], running in time proportional with live data, one might want to trigger a heap gc in each engine just to avoid scanning the complete heap. Even better, one can instrument the marking phase of the heap garbage collector to also collect and relocate symbols. Or, one can just run the marking phase if heap GC is not yet due for a given engine and collect and reindex only the reachable symbols. We leave these optimizations as possible future work.

18 Symbol GC and Multi-Threading

Clearly, in the presence of multithreading special care is needed to coordinate symbol creation and even reference to symbols that might get relocated by the collector. Moreover, as our collector can also reclaim the engines themselves that are used to support *Lean Prolog*'s multithreading API, consequences of unsafe interactions between the two subsystems can be quite dramatic.

One solution, used in systems like SWI-Prolog [31] is to ensure that all threads wait while the collector is working.

Alternatively, one can simply use a separate symbol table per thread and group together a large number of engines cooperating sequentially within each thread. In this scenario, when communication between threads occur, symbols are internalized on each side as needed. If one wraps up the communication mechanism itself, to work as a transactional client/server executing one data exchange between two threads at a time, safety of the multi-engine ecosystem within each thread is never jeopardized.

The other requirement for this scenario is the ability to have multiple independent symbol tables, a design feature present in *Lean Prolog* also to support an atom-based module system and object oriented extensions.

We have set as default behavior in the case of *Lean Prolog* the second scenario, mostly because we have started with a design supporting up front multiple independent symbol tables and strong uncoupling between the Engine API and the multithreading API.

However, for "system programming" tasks like adding *Lean Prolog's* networking, remote predicate call, Linda blackboard layer as well for supporting encapsulated design patterns like *ForkJoin* or *MapReduce* that are used as building blocks for distributed multi-agent applications, we have provided a simple synchronization device between threads, allowing full programmer control on the interactions with aspects involving sequential assumptions like the symbol garbage collector.

The device, called a `Hub`, coordinates `N` producers and `M` consumers nondeterministically, i.e. consumers are blocked until a producer places a term on the `Hub` and producers are blocked until a consumer takes the term on the `Hub`. Threads

are always created with associated `Hubs` that are made visible to their parent and usable for coordinated interaction.

At this point in time, we are still exploring ways to provide a convenient set of user-level built-ins that combine maximum flexibility in expressing concurrency while avoiding unnecessary implementation complexity or execution bottlenecks.

19 Empirical Evaluation

We will divide our evaluation to cover two orthogonal aspects of the usefulness of our integrated multi-engine symbol garbage collector.

First, we evaluate, as usual, the relative costs of having the algorithm on or off on various benchmarks.

Second, we evaluate the benefits it brings to a system by comparing time and resource footprints with and without the collector enabled.

Observed/Benchmark	Devil's Own	Findall	Pereira
Syms NO SYMGC	765692	1295	759001
Engines NO SYMGC	4	12	953
Total time NO SYMGC	18629	5280	19738
Syms SYMGC	530892	1295	69579
Engines after SYMGC	4	2	3
Time for useful work	8173	3647	18035
Time for SYMGC	666	1	43
Time for Heap GC	4352	1008	270
Time for exp/shrink	7724	721	406
Total with SYMGC	20915	5377	18754

Fig. 4: Time/space efforts and benefits of our integrated symbol GC algorithm on three benchmarks

The table in Fig 4 summarizes our experimental evaluation on some artificial benchmarks. The table in Fig 5 summarizes our experimental evaluation on two fairly large applications. To make the experiments as realistic as possible, in each benchmark memory management operations are triggered automatically. This also tests the effectiveness of our collection policies. To measure the effectiveness of the symbol GC algorithm on both symbol and engine totals we give as a

Observed/Benchmark	SelfCompile	Wordnet
Syms NO SYMGC	2017	613183
Engines NO SYMGC	2	2
Total time NO SYMGC	10482	412345
Syms SYMGC 2017	28590	
Engines after SYMGC	2	2
Time for useful work	9358	367172
Time for SYMGC	0	14
Time for Heap GC	15	235
Time for exp/shrink	686	21428
Total with SYMGC	10429	388849

Fig. 5: Time/space efforts and benefits of our integrated symbol GC algorithm on two applications

baseline what happens when the symbol GC is switched off. These totals give indirectly an idea on the memory savings resulting from the use of symbol GC.

Execution times have been measured for our 3 memory management operations.

Given that *Lean Prolog*'s data areas are managed as dynamic arrays that expand/shrink as needed, expand/shrink operations, being often in the inner loops of the runtime interpreter actually dominate time spent on memory management.

As our symbol GC policy triggers symbol collection right after a heap GC in the engine that is most likely to have been created most of the symbols, the cost of symbol GC is dominated by heap GC costs. Proceeding right after garbage collecting this "dominant heap" ensures that only live objects are scanned on the heap so relatively few unnecessary symbol creation operations happen when the old symbol table is replaced by the new one. *This also explains why symbol GC times are significantly lower than time spent on other memory management tasks.*

We have created a dedicated *"Devil's Own"* symbol GC stress test that uses 4 threads creating concurrently long lists of new symbols that are always alive on the heaps. This is the only benchmark where overall execution time is significantly slower with the symbol GC enabled. On the contrary, the memory bandwidth reduction that can be seen as an indirect consequence of the symbol GC, has in 3 other benchmarks beneficial effects on execution time.

The *Findall* benchmark computes a list of all permutations of length 8. As *Lean Prolog*'s `findall` is implemented using engines this benchmark focuses exclusively on the effect of the symbol GC collector on engines.

The *Pereira* benchmark tests a wide variety of operations. In particular, assert/retract operations and findall/bagof/setof operations benefit significantly from the presence of symbol GC, to the point that overall execution time improves.

The *SelfCompile* application benchmark measures *Lean Prolog*'s time on recompiling its own compiler and libraries. As the symbols are all already there, no costs or benefits are incurred with or without symbol GC.

Finally, the `Wordnet` application benchmark reads in (and indexes) the complete Wordnet 3.0 database. A significant improvement in execution time is observed in this case, due to the overall reduction of memory bandwidth.

20 Related work

We have designed and implemented our symbol garbage collector starting from scratch through an iterative process that first worked with a single engine with serialized heap-represented external objects. Very soon, it has evolved to also manage arbitrary length arithmetic objects and Java handles. At the end, our overall architecture turned out to have a some similarities with the Erlang atom garbage collector proposal described in [71].

The most important commonality is copying of live symbols into a new table, based on scanning, followed by symbol relocation in all roots (see also section 2.2 in [71]).

While our paper is based on a finished working collector, [71] describes a proposal for an implementation. While our description provides enough detail to be replicable in another system, [71] is fairly general and often ambiguous about how things actually get worked out. This makes a detailed comparison difficult, but we were able to point out a number of similarities and differences, as follows.

Among the similarities:

- comparable contexts: multiple Erlang processes one one side, multiple Prolog engines on the other
- separation in "epochs" with special handling of compile time symbols
- live atoms are migrated from an old table to a new one i.e. both approaches are "copying collectors"

Among the differences:

- engines, as first order citizens are themselves represented as symbols in our case
- a discussion of an incremental version of the collector is given in [71]
- constraints related to the use of the symbol table as an interface to arbitrary external objects in our case
- a detailed discussion on the policy used to trigger the collection is given in our case

– in contrast to Erlang, detection of the presence of a large number of garbage symbols is complicated in our case by independent backtracking in multiple engines

In the world of Prolog systems symbol garbage collectors have been in use even in early pre-WAM implementations (Prolog1). Among them, SWI Prolog's symbol garbage collector, using a combination of reference counting and mark-and-collect has been shown valuable for processing large data streams and semantic web applications [72] and its interaction with multi-threading is discussed in [31]. Instead of copying however, SWI-Prolog leaves a symbol-table with holes. While managing them with a linked list can avoid linear scan in the case of SWI's implementation, we have chosen to edit symbol cells in-place, partly because our tag-on-data representation made this operation simple to implement and partly because it added no extra runtime cost to do so.

While not described in detail in a publication that we are aware of, the SICStus Prolog built-in `garbage_collect_atoms/0`'s description in the user manual [52] mentions about scanning all data areas for live atoms.

The heap GC algorithm (a simple "mark and sweep") used by *Lean Prolog* is the one described in [69]. The heap scanning for symbols is, in our case, proportional to the total size of the heaps of all engines, (possibly after running the heap GC as well in some). With this in mind, copying heap GC algorithm [42, 73, 74] is likely to provide also better symbol GC performance for programs with highly volatile heap data. The impact of such algorithms on integration with symbol GC remains to be studied.

Multiple Logic Engines have been present in a from or another in various parallel implementation of logic programming languages [58–60]. Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel [20] and OR-parallel execution models are worth mentioning. In combination with multithreading our own engine-based API bears similarities with various other Prolog systems, notably [30, 31]. However, a distinctive feature of *Lean Prolog*, that allowed us to separate concerns related to thread synchronization, is that our engine API is completely orthogonal with respect to multithreading constructs.

21 The Impact of Symbol Garbage Collection

While both reference counting and data area scanning symbol collection algorithms have been implemented in the past in various Prolog systems (and other related languages), we have not found in the literature a detailed, replicable description of all the aspects covering a complete implementation.

Our empirical evaluation indicates that the costs of symbol GC are amortized by improved memory bandwidth and that it usually brings not only space savings but also execution time benefits.

As virtually all Prolog and related logic programming systems in use today that support some form of concurrent execution can be seen as "multi-engine" Prologs, it is likely that they may benefit from an adaptation of our the integrated symbol and heap garbage collection algorithm.

References

1. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In Wooldridge, M., Müller, J.P., Tambe, M., eds.: ATAL. Volume 1037 of Lecture Notes in Computer Science., Springer (1995) 347–360
2. Wegner, P., Eberbach, E.: New Models of Computation. Comput. J. **47**(1) (2004) 4–9
3. Tarau, P.: Orthogonal Language Constructs for Agent Oriented Logic Programming. In Carro, M., Morales, J.F., eds.: Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems, Saint-Malo, France (September 2004)
4. Tarau, P.: BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp. (2006)
5. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.
6. Liu, J., Kimball, A., Myers, A.C.: Interruptible iterators. In Morrisett, J.G., Jones, S.L.P., eds.: POPL, ACM (2006) 283–294
7. Matsumoto, Y.: The Ruby Programming Language. (June 2000)
8. Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter. In Johnson, R., Gabriel, R.P., eds.: OOPSLA Companion, ACM (2005) 158–159
9. Microsoft Corp.: Visual `C#`. Project URL http://msdn.microsoft.com/vcsharp.
10. van Rossum, G.: A Tour of the Python Language. In: TOOLS (23), IEEE Computer Society (1997) 370
11. Liskov, B., Atkinson, R.R., Bloom, T., Moss, J.E.B., Schaffert, C., Scheifler, R., Snyder, A.: CLU Reference Manual. Volume 114 of Lecture Notes in Computer Science. Springer (1981)
12. Griswold, R.E., Hanson, D.R., Korb, J.T.: Generators in Icon. ACM Trans. Program. Lang. Syst. **3**(2) (1981) 144–161
13. Tarau, P., Boyer, M.: Nonstandard Answers of Elementary Logic Programs. In Jacquet, J., ed.: Constructing Logic Programs. J.Wiley (1993) 279–300
14. Tarau, P.: Towards Inference and Computation Mobility: The Jinni Experiment. In Dix, J., Furbach, U., eds.: Proceedings of JELIA'98, LNAI 1489, Dagstuhl, Germany, Springer (October 1998) 385–390 invited talk.
15. Tarau, P.: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In: Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents, London, U.K. (1999) 109–123
16. Tarau, P.: Inference and Computation Mobility with Jinni. In Apt, K., Marek, V., Truszczynski, M., eds.: The Logic Programming Paradigm: a 25 Year Perspective, Springer (1999) 33–48 ISBN 3-540-65463-1.
17. Tarau, P.: Agent Oriented Logic Programming Constructs in Jinni 2004. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20-th International Conference, ICLP 2004, Saint-Malo, France, Springer, LNCS 3132 (September 2004) 477–478
18. Tyagi, S., Tarau, P.: A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces. In Ramakrishan, I., Gupta, G., eds.: Proceedings of PADL'2001, Las Vegas (March 2001) Springer-Verlag.
19. FIPA: FIPA 97 specification part 2: Agent communication language (October 1997) Version 2.0.
20. Hermenegildo, M.V.: An abstract machine for restricted and-parallel execution of logic programs. In: Proceedings on Third international conference on logic programming, New York, NY, USA, Springer-Verlag New York, Inc. (1986) 25–39

21. Lusk, E., Mudambi, S., Gmbh, E., Overbeek, R.: Applications of the aurora parallel prolog system to computational molecular biology. In: In Proc. of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems, Washington DC, MIT Press (1993)
22. Tarau, P., Majumdar, A.: Interoperating Logic Engines. In: Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, Georgia, Springer, LNCS 5418 (January 2009) 137–151
23. Tarau, P.: Logic Engines as Interactors. In Garcia de la Banda, M., Pontelli, E., eds.: Logic Programming, 24-th International Conference, ICLP, Udine, Italy, Springer, LNCS (December 2008) 703–707
24. Tarau, P., Boyer, M.: Elementary Logic Programs. In Deransart, P., Maluszyński, J., eds.: Proceedings of Programming Language Implementation and Logic Programming. Number 456 in Lecture Notes in Computer Science, Springer (August 1990) 159–173
25. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog: The Standard. Springer-Verlag, Berlin (1996) ISBN: 3-540-59304-7.
26. Tarau, P., Dahl, V.: Logic Programming and Logic Grammars with First-order Continuations. In: Proceedings of LOPSTR'94, LNCS, Springer, Pisa (June 1994)
27. Bird, R.S., de Moor, O.: Solving optimisation problems with catamorphism. In Bird, R.S., Morgan, C., Woodcock, J., eds.: MPC. Volume 669 of Lecture Notes in Computer Science., Springer (1992) 45–66
28. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2) (1990) 231–248
29. Tarau, P., Dahl, V.: High-Level Networking with Mobile Code and First Order AND-Continuations. Theory and Practice of Logic Programming **1**(3) (May 2001) 359–380 Cambridge University Press.
30. Carro, M., Hermenegildo, M.V.: Concurrency in prolog using threads and a shared database. In: ICLP. (1999) 320–334
31. Wielemaker, J.: Native preemptive threads in swi-prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345
32. Majumdar, A., Sowa, J.F., Stewart, J.: Pursuing the goal of language understanding. In: Conceptual Structures: Knowledge Visualization and Reasoning. Volume 5113 of LNCS., Springer (2008) 21–42
33. Tarau, P.: A Simplified Abstract Machine for the Execution of Binary Metaprograms. In: Proceedings of the Logic Programming Conference'91, ICOT, Tokyo (7 1991) 119–128
34. Tarau, P.: The Jinni Prolog Compiler: a fast and flexible Prolog-in-Java (2008) http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html.
35. Lloyd, J.: Foundations of Logic Programming. Symbolic computation — Artificial Intelligence. Springer-Verlag, Berlin (1987) Second edition.
36. Tarau, P., De Bosschere, K.: Memoing with Abstract Answers and Delphi Lemmas. In Deville, Y., ed.: Logic Program Synthesis and Transformation. Springer-Verlag, Louvain-la-Neuve (July 1993) 196–209
37. Proietti, M.: On the definition of binarization in terms of fold/unfold. (June 1994) Personal Communication.
38. Demoen, B., Mariën, A.: Implementation of Prolog as binary definite Programs. In Voronkov, A., ed.: Logic Programming, RCLP Proceedings. Number 592 in Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, Springer-Verlag (1992) 165–176
39. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (October 1983)

40. Demoen, B.: On the transformation of a prolog program to a more efficient binary program. In: LOPSTR. (1992) 242–252
41. Neumerkel, U.: Specialization of Prolog Programs with Partially Static Goals and Binarization. Phd thesis (1992) Technische Universität Wien.
42. Demoen, B., Engels, G., Tarau, P.: Segment Preserving Copying Garbage Collection for WAM based Prolog. In: Proceedings of the 1996 ACM Symposium on Applied Computing, Philadelphia, ACM Press (February 1996) 380–386
43. Tarau, P., Neumerkel, U.: Compact Representation of Terms and Instructions in the BinWAM. Technical Report 93-3, Dept. d'Informatique, Université de Moncton (November 1993) available by ftp from clement.info.umoncton.ca.
44. Tarau, P., Demoen, B., De Bosschere, K.: The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog. In George, K., Carrol, J., Deaton, E., Oppenheim, D., Hightower, J., eds.: Proceedings of the 1995 ACM Symposium on Applied Computing, Nashville, ACM Press (February 1995) 152–176
45. Gudeman, D., De Bosschere, K., Debray, S.: `jc`: An Efficient and Portable Sequential Implementation of Janus. In Apt, K., ed.: Joint International Conference and Symposium on Logic Programming, Washington, MIT press (November 1992) 399–413
46. De Bosschere, K., Debray, S., Gudeman, D., Kannan, S.: Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland/USA, ACM (January 1994) 409–420
47. Tarau, P.: Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In Voronkov, A., ed.: Logic Programming, RCLP Proceedings. Number 592 in Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, Springer-Verlag (1992) 462–473
48. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
49. Zhou, N.F., Takagi, T., Ushijima, K.: A matching tree oriented abstract machine for prolog. (1990) 159–173
50. Zhou, N.F.: A register-free abstract prolog machine with jumbo instructions. In Dahl, V., Niemelä, I., eds.: ICLP. Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 455–457
51. Diaz, D., Codognet, P.: Design and implementation of the gnu prolog system. Journal of Functional and Logic Programming **2001**(6) (2001)
52. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjoland, T.: SICStus Prolog user's manual
53. da Silva, A.F., Costa, V.S.: The design and implementation of the yap compiler: An optimizing compiler for logic programming languages. In Etalle, S., Truszczynski, M., eds.: ICLP. Volume 4079 of Lecture Notes in Computer Science., Springer (2006) 461–462
54. Swift, T., Warren, D.S.: An abstract machine for SLG resolution: definite programs. In Bruynooghe, M., ed.: Logic Programming - Proceedings of the 1994 International Symposium, Massachusetts Institute of Technology, The MIT Press (1994) 633–652
55. Van Roy, P.: 1983-1993: The wonder years of sequential prolog implementation. Journal of Logic Programming (1994)
56. Demoen, B., Nguyen, P.L.: So many wam variations, so little time. In: CL '00: Proceedings of the First International Conference on Computational Logic, London, UK, Springer-Verlag (2000) 1240–1254

57. Lindgren, T.: A continuation-passing style for prolog. In: ILPS '94: Proceedings of the 1994 International Symposium on Logic programming, Cambridge, MA, USA, MIT Press (1994) 603–617
58. Shapiro, E.: The family of concurrent logic programming languages. ACM Comput. Surv. **21**(3) (1989) 413–510
59. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. Program. Lang. Syst. **23**(4) (2001) 472–602
60. Ueda, K.: Guarded horn clauses. In Wada, E., ed.: Logic Programming '85, Proceedings of the 4th Conference, Tokyo, Japan, July 1-3, 1985. Volume 221 of Lecture Notes in Computer Science., Springer (1985) 168–179
61. Tarau, P.: Multi-Engine Horn Clause Prolog. In Gupta, G., Pontelli, E., eds.: Proceedings of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Las Cruces, NM (November 1999)
62. Casas, A., Carro, M., Hermenegildo, M.: Towards a high-level implementation of flexible parallelism primitives for symbolic languages. In: PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation, New York, NY, USA, ACM (2007) 93–94
63. Demoen, B., Tarau, P.: jProlog home page (1996) http://www. cs. kuleuven. ac. be/~ bmd
64. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe: a Prolog to Java translator system. Lecture Notes in Computer Science **4369** (2006) 1
65. Cook, J.: P#: A concurrent Prolog for the .NET Framework. Software: Practice and Experience **34**(9) (2004) 815–845
66. Cook, J.: Optimizing P#: Translating Prolog to more Idiomatic C#. Proceedings of CICLOPS 2004 (2004) 59–70
67. Tarau, P., Neumerkel, U.: A Novel Term Compression Scheme and Data Representation in the BinWAM. In Hermenegildo, M., Penjam, J., eds.: Proceedings of Programming Language Implementation and Logic Programming. Number 844 in Lecture Notes in Computer Science, Springer (September 1994) 73–87
68. Tarau, P.: Low level Issues in Implementing a High-Performance Continuation Passing Binary Prolog Engine. In Corsini, M.M., ed.: Proceedings of JFPL'94. (June 1994)
69. Zhou, Q., Tarau, P.: Garbage Collection Algorithms for Java-Based Prolog Engines. In Dahl, V., Wadler, P., eds.: Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, USA, Springer, LNCS 2562 (January 2003) 304–320
70. Demoen, B., Nguyen, P.: Allocating wam environments/choice points on the heap. Report CW **455** (July 2006)
71. Lindgren, T.: Atom garbage collection. In: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang, ACM (2005)
72. Wielemaker, J., Hildebrand, M., van Ossenbruggen, J.: Using Prolog as the fundament for applications on the semantic web. Proceedings of ALPSWS2007 (2007) 84–98
73. Vandeginste, R., Sagonas, K., Demoen, B.: Segment order preserving and generational garbage collection for Prolog. Lecture notes in computer science (2002) 299–317
74. Demoen, B., Nguyen, P., Vandeginste, R.: Copying garbage collection for the WAM: To mark or not to mark? Lecture notes in computer science (2002) 194–208