

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

Paul Tarau

Department of Computer Science and Engineering  
University of North Texas

ISMM'11: Saturday, June 4, 2011, 2:30pm

# Motivation

- Logic Programming languages work on symbolic objects (term algebras) and rely on *unification*, *recursion* and *backtracking*  $\Rightarrow$
- complex memory allocation/deallocation happens automatically
- in “classic” Prolog we have to deal with multiple stacks, heaps, (dynamic) code areas
- in our multi-engine Prolog system **Lean Prolog**
  - *logic engines encapsulate independent computations*
  - logic engines exchange complex data - they can even dynamically inject code in other engines
  - symbols - dynamically created identifiers - are used for I/O, dynamic code, logic engines, handles to external objects
- $\Rightarrow$  **integrated memory management** - dynamic stacks, heap, symbol tables (including external objects), dynamic code area

# Outline

- “bird’s eye” views of the (Lean) Prolog VM
- explain some design/implementation choices
- safeness assertions
- outline of the symbol GC algorithm
  - work delegated to engines
  - work performed in the atom table
- unusual functions of our symbol GC
- the symbol GC activation policy
- symbol GC and multithreading
- empirical evaluation
- conclusion

# A “bird’s eye” view of Prolog Runtime Systems

- Warren Abstract Machine (WAM): provides efficient compilation of unification, backtracking and indexing operations.
- *terms*, built of function symbols having as arguments constant symbols and variables, are represented on a heap using references (pointers in C and integer indices in Java) to their sub-terms
- references to symbols are tagged integers pointing to entries in a symbol table
- **new symbols** can be added at run-time
- $\Rightarrow$  need for symbol garbage collection mechanisms

# A “bird’s eye” view of our **Lean Prolog** system

- “lean”: lightweight emulator + source-level built-ins, when possible
- Java-based Prolog - but using fast, integer-based WAM variant (BinWAM)
- multiple first-class logic engines
- engines are uncoupled from multithreading mechanism
- engines can multi-task either preemptively or cooperatively
- engines can create new terms and symbols
- engines exchange complex data - containing possibly deep references to other engines and handles to complex external objects - Maps, ArrayLists etc.

# First Class Logic Engines in Lean Prolog

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

# Why do we need symbol GC in Prolog?

- symbol garbage collection is important in practical applications of programming languages that rely on internalized symbols as their main building blocks
- in Prolog, natural language tools, XML processors, database interfaces and compilers rely on dynamic symbols 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
- most of these reasons apply to other dynamic languages as well!

# Why don't we use Java objects to represent Prolog terms?

- using Java objects would give us free memory management
- but representation sizes (and, implicitly, processing time!) would go up significantly

Term	Java Object size	Int array heap size
f(a)	40 bytes	8 bytes
f((g(X),h(Y)))	124 bytes	28 bytes
[1,2,3,4,5]	228 bytes	60 bytes

Figure: Java Object vs. int array WAM heap representation

Why don't we place serialized Java objects on the int array heap, instead of placing them in the **symbol table**?

Class	Constructor argument	Bytes
java.util.ArrayList	()	58
java.util.HashMap	()	82
java.util.LinkedHashMap	()	135
java.lang.Integer	0	81
java.lang.Double	0.0	84
java.math.BigInteger	"0"	202
java.math.BigDecimal	"0.0"	290

**Figure:** Serialized sizes of minimal instances of some Java objects used in our implementation

## Facts helping to decide *if/when* symbol GC should occur

- if a program creates new symbols in a multi-engine Prolog, they will eventually end up in the registers, choice points or the heap of at least one of the engines
- symbol GC needs only to happen when either:
  - heap GC occurs in at least one engine
  - at least one engine backtracks
- it is safe to use external references through a handle stored in the symbol table, provided that the interface ensures that new symbols contained in them are added to the symbol table on their first use
- given any chain of engine calls within the same thread, running the symbol GC right after an engine's 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*

# Outline of our integrated symbol GC algorithm

- enable symbol GC when too many symbols or engines have been created since the previous collection
- enable symbol GC when forced by a dramatic shortage of memory
- wait within safe bounds until the actual garbage collection can be performed i.e. let engines advance in their VM loops while it is safe to do so
- ensure soundness: make sure that at least one of the opportunities will occur without the engines being able to create new symbols and crash as a result of running out of memory
- note that heap gc might be triggered if symbol space runs low - hoping that this also reduces live symbols

# 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
- all the symbols found in these areas are added to a newly created symbol table
- “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
- facilitated by the `tag-on-data` representation in the BinWAM

## Symbol GC: as seen from the class `AtomTable`

- first, a new `AtomTable` instance, called `keepers`, is created, that will contain the *reachable* symbols
- the `keepers` table is preinitialized with all the compile time symbols, built-in predicates etc.
- we iterate over all the engines and perform the steps described in the previous slide

# Dealing with the engines

- 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, we skip it
- if an engine is `protected` i.e. it is the root of an independent thread running a set of related Prolog engines sharing the same symbol table and code the engine is added to `keepers`
- if the engine has made it this far, it will be scanned for live symbols
- a check for self-referential engines is made: if the engine did not make it into `keepers` before scanning its own roots and it made it there after, it is removed from `keepers`

## Some unusual functions of Lean Prolog's symbol GC

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

# Fine tuning the activation of the symbol collector

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 the following 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

## Outline our symbol GC policy

- postpone symbol GC if the size of the (dynamically growing) symbol table is still relatively small
- postpone symbol GC if the growth since last collection is not large enough
- trigger the symbol collector when a significant number of engines have been created, as collecting dead engines not only brings back significant memory chunks, but it also has the potential to free additional symbols
- if the size of the symbol table exceeds a significant fraction of the total engine heap size, 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

# A future optimization: synergy with copying heap garbage collectors

- a good opportunity to run our symbol collector arises right after heap garbage collection
- we are using a mark-and-sweep collector in *Lean Prolog*
- by using a copying collector, 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
- one can even instrument the marking phase of the heap garbage collector to also collect and relocate symbols
- $\Rightarrow$  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

# Symbol GC and Multi-Threading

- 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
- as our collector can also reclaim the engines that are used to support the multithreading API, consequences of unsafe interactions between the two subsystems can be quite dramatic
- ⇒ when possible, use cooperative multi-tasking
- ⇒ when possible, use safe “concurrency patterns” encapsulated as higher-order predicates
- also, one could suspend all threads sharing a symbol table - but that requires waiting until they all reach safe points

# Our Preferred Solution

- Lean Prolog supports multiple independent symbol and code spaces
- $\Rightarrow$  use a separate symbol table per thread and group together a large number of engines cooperating sequentially within each thread

# Thread coordination

- a simple synchronization 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
- we encapsulate thread coordination as higher-order predicates, that combine maximum flexibility in expressing concurrency while avoiding unnecessary implementation complexity or execution bottlenecks

# Empirical evaluation: what if we used serialized heap representations?

BigIntegers/BigDecimals with the runtime system instrumented to perform one serialization and one deserialization on each output:

Feature	Factorial	Factoradics
With serialization	10.30s	6.45s
With symbol GC off	1.82s	3.85s
With symbol GC on	1.53s	3.85s
Coded with BigIntegers in Java	0.41s	0.70s
With serialization overhead, in Java	11.66s	3.44s
SWI-Prolog with GMP integers	0.15s	0.26s

Figure: Impact of serialization on BigInteger performance

## Benefits of symbol GC on some artificial benchmarks

Impact on 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 expand/shrink	7724	721	406
Total with SYMGC	20915	5377	18754

Figure: Time (in ms.) and space benefits of symbol GC



## Benefits of symbol GC on two large programs

Impact on 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 expand/shrink	686	21428
Total with SYMGC	10429	388849

Figure: Time (in ms.) and space benefits of symbol GC



# Effectiveness of symbol GC on a typical blend of Prolog predicates - the Pereira benchmark

Memory Usage	Symbol GC off	Symbol GC on
Java Memory	158 MBytes	98 Mbytes
Symbols, after run	759793	69579
Symbols, after final GC	2670	2670

Figure: Statistics for the Pereira benchmark

# Conclusion

- our integrated symbol GC supports exchange between multiple Prolog engines of arbitrary Java objects, including big integers and decimals
- other “multi-engine” Prologs may benefit from an adaptation of our the integrated symbol and heap garbage collection algorithm
- the architecture is replicable when implementing scripting or domain specific dynamic languages on top of Java, Scala or C#
- efficient access to Java collection and map classes whose complex object graphs are internalized as lightweight garbage collectable Prolog symbols
- costs of symbol GC are amortized by consistent reduction of the memory footprint resulting in reduced GC effort on the Java side
- our symbol GC can bring not only space savings but also execution time benefits

# Questions?

Lean Prolog and a few related papers are at:

- <http://logic.cse.unt.edu/research/LeanProlog>