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

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cse.unt.edu

Abstract

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 an integrated memory management solution for Prolog systems, with amortized cost dominated by normally occurring heap garbage collection costs.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Memory management; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Constraint and logic languages

General Terms Languages, Performance, Algorithms, Design

Keywords atom garbage collection, Prolog memory management, multi-engine Prolog, Prolog runtime system architecture, integrated memory management

1. Introduction

Most Prolog implementations use variants of the Warren Abstract Machine (WAM) architecture [1] that provides efficient compilation of unification, backtracking and indexing operations. In the WAM, Prolog terms, built of function symbols (called *functors*), 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 that can acquire new symbols at run-time, leading to the need for symbol garbage collection mechanisms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'11, June 4–5, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0263-0/11/06. . . $10.00

Symbol garbage collection is important in practical applications of programming 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. While serialization can be avoided in C-based systems using pointers to "blobs" on the Prolog heap and type castings, this is not an option in strongly typed Java, where such data would need to be put on the Prolog heap (an `int` array) in a serialized form, incurring significant processing time and memory costs. For instance, under 64 bit Java 1.6.x, the BigInteger representation of `0` is serialized to as much as 202 bytes and its BigDecimal `0.0` to 290 bytes. Not counting conversion time, the memory impact is itself prohibitive.

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. For instance, a Prolog clause can be contained in and can refer to an external HashMap while holding a handle to it. 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.

This paper describes the implementation of such a solution in our ongoing *Lean Prolog* system that supports sharing of arbitrary size external data, as divers as collections, graphs, GUI compo-

nents, arbitrary precision integers and decimals, between multiple logic engines.

Beyond Prolog implementation, our techniques are likely to be reusable for other scripting or domain specific languages implemented in languages like Java or `C#`.

We will discuss our design decisions and algorithms in the context of *Lean Prolog*'s lightweight BinWAM-based [13, 14, 22] runtime system, a minimalist Java kernel using logic engines as first class building blocks encapsulated as *interactors* [20, 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 this scenario, implemented as the compilation model used by jProlog, a BinWAM-based research prototype written by Bart Demoen in collaboration with the author back in 1996, [6], and some of its derivatives like Prolog Cafe [2] or `P#` [5], is that Java objects are too heavyweight for basic Prolog abstract machine functions. We have observed that a relatively plain, C-style integer based runtime system performs an order of magnitude faster than one using terms represented as Java objects, especially in the presence of "just-in-time" and "HotSpot" java compilers. The reader will find specific figures supporting our design decisions in section 5.

Unfortunately, giving up on Java objects for a low-level BinWAM engine [15, 24] 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` or Google's new `go` language [1]. 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 unreachable logic engines.

Fortunately, a number of simplifications of our runtime architecture, like separation of engines and threads (subsection 2.2) and a *tag-on-data* term representation (subsection 2.3) 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, arbitrary size numbers and logic engines (subsection 2.4).

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 2 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 3 first outlines in subsection 3.1, and then describes the major components of the symbol GC algorithm (3.2 opportunity detection, 3.3 work delegated to engines, and 3.4 work in the class implementing the atom table). Subsection 3.5 focuses on the "fine tuning" of our symbol GC policy. Section 4 discusses interaction with multi-threading. Section 5 provides evidence supporting some of our design decisions and discusses an empirical evaluation of the costs and benefits of the integration of symbol GC with other memory management tasks. Finally, sections 6 and 7 discuss related work and conclude the paper.

[1] 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.

2. Architectural aspects 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.

2.1 The Multi-Engine API

Our *Engines-as-Interactors API* has evolved progressively into a practical Prolog implementation framework starting with [17] and continued with [20] and [23]. 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 lightweight Prolog interpreter on a given clause database, together with a set of built-in operations. Engines are designed to interoperate with external resources in a modular way, 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 [17].

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 [21, 23].

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.

2.2 Decoupling engines and threads

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 [17, 20, 23], 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. Prolog 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* [21]. While it is possible to launch a logic engine as a separate thread, they are also heavily used in some built-ins like `findall`, that collects to a list all answers produced by a goal. Interestingly, our engine-based `findall` is about twice as fast as a direct findall implementation in which Prolog terms are saved as Java objects to an external ArrayList, as engine-to-engine communication uses significantly more compact heap representations.

The decoupling of engines and threads 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.

2.3 The *tag-on-data* term representation

When describing the data in a heap cell with a tag that indicates its type (variable, symbol, integer) 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* a term has to be processed. Like in our previous Prolog implementations [16, 18, 19] we choose the second possibility, which also supports a form of term compression [24].

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[2]. 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:

arity	symbol-number	2-bit tag

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.

2.4 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. For instance the term `f(X,Y)` would take `12` bytes in a WAM representation with 32-bit word size. The equivalent Java term would contain a functor object made of a string `"f"` and an array of length `2` containing arguments `X` and `Y` which in turn would be distinct variable objects containing an `int` field each. Assuming (conservatively, with a 32 bit JVM in mind) that the size of an object is `8` bytes + 4 bytes for each instance fields, we get `12 bytes` for `X` and `Y` each, `16 bytes` for `f/2` and `20` bytes for its argument array for a total of `12+12+16+20=60` bytes, not counting alignment constraints and possibly larger data sizes on a 64 bit JVM. The ratio between memory representations leads to comparable slowdowns of execution time. This justifies our design choice to use integer array representations for Prolog terms, and consequently a symbol table to interface between Java objects and their heap representations. Like in the case of Java's interning of a `String`, a single copy of the same Prolog symbol is shared and referenced as an `int` index in the symbol table, resulting in faster equality testing and memory savings.

Once the decision to have a symbol garbage collector is made, a number of consequences on the implementation follow, that break away from the design choices one would make in a typical `C`-based Prolog:

- 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 objects 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?

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.

3. 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 1. *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.*

PROPOSITION 2. *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:*

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

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

One might argue that a program like

```
loop :- loop.
```

does neither. Note however that such programs do not create new symbols, and therefore nothing is lost if they do not activate symbol GC.

PROPOSITION 3. *If a symbol does not occur on the heap, the active registers or in the registers saved in the choice points of any live engine, then the symbol can be safely reclaimed.*

PROPOSITION 4. *It is safe to use external (i.e. Java) Map, Set etc. 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.*

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 [23]. 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, i.e. added to the symbol table only when occurring in the dynamic clauses that have passed all "the indexing tests"[3] - 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.

3.1 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 i.e. ensuring that as engines advance in their internal virtual machine loops, 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

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.

[3] Such indexing tests succeed when toplevel functors of the arguments in the heads of the clauses in the database match corresponding arguments in a goal atom.

The heap garbage collector used in *Lean Prolog* is a simple mark and sweep algorithm along the lines of [29]. As most GC implementations, it competes with heap expansion, as 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 libraries like HashMap or possibly other Map implementations for our symbol tables. This has also simplified the decision logic and helped us to separate the detection of the need for symbol GC, from the activation of the collection process. This uncoupling had no negative impact on performance as the decision to call the symbol GC itself has been fine tuned as a self-contained process that is only activated when a significant amount of changes in the symbol table have occurred.

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 symbol GC 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 their symbol index 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.

3.2 Detecting the need for Symbol GC

The `symbol_gc` flag is raised when 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 3.5. We ensure that only one thread uses these methods at a given time either by sharing a symbol table only between coroutining engines or by synchronizing them in case of engines running on different threads. In this scenario it is ok if multiple threads raise `symbol_gc` 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.

3.3 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 [13, 22] does not use a local stack, an adaptation to conventional WAMs might require scanning for possible symbol cells there as well.

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` [24] representation in the BinWAM but it can be (with some care!) adapted also to Prologs using the conventional WAM's `tag-on-pointer` scheme.

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

One can infer from Prop. 2 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 Java `Map` object that *manages our symbols* and has access to the set of 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[4].

Next we iterate over all the engines and *perform the steps described in subsection 3.3*.

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, we skip it. A deliberate stopping happens, for instance, when the parent that has launched and engine is only interested in the first solution produced by the engine.

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 (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 dismantling) 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[5].

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

3.5 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 [25], 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 class citizens, they also have a place in the symbol table to allow references from other engines.

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[6].

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

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 the symbol GC 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[7] or if the growth since last collection is not large enough[8].

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[9].

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[10], 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[11].

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

3.6 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

[4] 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.

[5] 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.

[6] This does not preclude allowing the programmer to manage directly the life-span of objects like files or sockets using open and close statements.

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

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

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

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

[11] 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.

copying collector, for instance [8], 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.

4. 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 that are used to support *Lean Prolog*'s multithreading API, consequences of unsafe interactions between the two subsystems can be quite dramatic.

Fortunately, Lean Prolog uncouples the multi-engine and multi-threading APIs and provides a number of high-level "design patterns" encapsulated as higher-order predicates for both cooperative and preemptive multi-tasking [21].

This allows source-level implementation of various scenarios ensuring the "peaceful coexistence" of multi-engine symbol GC and multi-threading.

The first approach, similar to the one used in systems like SWI-Prolog [28] 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 agent-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.

A 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.

On the Prolog side `Hub` is introduced with a constructor `hub/1` and works with following generic API:

```
hub(Hub)
ask_interactor(Hub, Term)
tell_interactor(Hub, Term)
stop_interactor(Hub)
```

A group of related threads are created around a `Hub` that provides synchronization and data exchange services. The built-in

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 1. Java Object vs. int array heap representation

```
new_logic_thread(Hub, X, G, Clone, Source)
```

creates a new thread by either "cloning" the current Prolog code and symbol spaces or by loading new Prolog code in a separate name space from a `Source` (typically a precompiled file or a stream). The default constructor

```
new_logic_thread(Hub, X, G)
```

shares the code but it duplicates the symbol table to allow independent symbol creation and symbol garbage collection to occur safely in multiple threads without the need to synchronize or suspend thread execution.

We refer to [21] for ways to provide, using higher-order predicates, a convenient set of user-level built-ins that combine maximum flexibility in expressing concurrency while avoiding unnecessary implementation complexity or execution bottlenecks.

5. Empirical Evaluation

Comparing with C-based Prologs is quite tricky given the gap between the relative speeds of C and Java. Also the integrated management of conventional string symbols (atoms in Prolog parlance), arbitrary size numbers (these days present in most Prolog systems) and external Java objects is different from other Prologs' where symbol GC is restricted to managing string symbols. As a result, we will first justify with quantitative data the design decisions that entail some of our implementation choices and then focus on evaluating the performance benefits of our Symbol GC algorithm.

5.1 Evaluating the impact of our design decisions

Our first major design decision was to give up Java's free memory management in exchange for the speed-up provided by working with an `int` array-based BinWAM emulator. While we have not built a comparably refined implementation with Prolog terms represented as Java objects, one can estimate the performance gap based on the relative size of the Java objects representing Prolog terms and their direct representation on a heap implemented as a dynamic integer array, as shown in Figure 1. Note that Java object-size estimates are conservative, assuming atomic objects at 8-bytes, when in practice actual sizes can be much higher. For instance, a `String` object in Java 6 (HotSpot) starts with 38 bytes of overhead, to which one adds 2 times the number of 2 byte characters and one might lose a few more bytes due to 8-byte alignment requirements.

Figure 2 lists serialized byte-sizes of some of the Java objects that have been frequently used in our system when accessing Java libraries from Prolog, to give an idea on the space (and implicitly computation time) required for placing them on the heap in serialized form. As a further justification of our design decisions to use a shared symbol table for objects like BigIntegers and BigDecimals we have instrumented our runtime system to perform one serialization and one deserialization operation on the output value of arithmetic operations. As an estimate of how an alternative implementation using this technique would perform, Figure 3 shows the impact of these operations on two arithmetic-intensive benchmarks. As base line, we have provided also measurements for our actual emulator, with symbol GC turned off and on.

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 2. Serialized sizes of minimal instances of some Java objects used in our implementation

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 3. Impact of serialization on BigInteger performance

The *Factorial* benchmark is simply a tail-recursive computation of the factorial of 20000. Given the huge size of the integers computed, the performance gap is quite large in this case. Also the use of Symbol GC actually speeds up this benchmark as it reduces the overall memory footprint significantly. For comparison, we have also given the timings for the same, recursively coded factorial using BigIntegers, directly in Java. Interestingly, in the case when serialization/deserialization overhead is added, Java performs slightly slower than LeanProlog which switches between representations to plain `int` types when computing with small values and because recursive calls are faster in WAM-based LeanProlog than in the underlying Java VM.

The *Factoradics* benchmark computes 1234^{4578} then it converts this large integer to a *factoradic* representation and back. The impact is less significant in this case as the intermediate values are lists of mostly small integers, that LeanProlog detects and converts to tagged `int` objects on the heap. In this case, note that the same program written in Java is faster with or without serialization overhead, which, like in the case of LeanProlog, dominates costs. To help putting things in perspective, the last row in Figure 3 shows that a C-based Prolog, relying on the native code GMP (GNU Multiple Precision Arithmetic Library), outperforms both Java and LeanProlog, by a significant margin, on both benchmarks.

5.2 Performance benefits of our Symbol GC algorithm

We will divide our performance 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.

The table in Figure 4 summarizes our experimental evaluation on some artificial benchmarks. The table in Figure 5 summarizes our experimental evaluation on two fairly large applications.

As for the *Factorial* and *Factoradics* benchmarks, execution times have been measured for our 3 memory management operations on a lightly loaded, 8-CPU MacPro (with 2 2.26GHz Quad-Core Intel Xeon processors and 16GB of memory).

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 reducing the total number of symbols and engines, we give as a 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.

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 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. As this benchmark contains a fairly representative blend of Prolog predicates, we have also added in Figure 6 measurements of total Java memory usage and total symbol count with symbol gc off and on. We have manually called the collector after the end of the benchmark to show the additional amount of symbols that can be collected.

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 Prolog version of the Wordnet 3.0 database (available from *http://wordnet.princeton.edu/wordnet/download*). A significant improvement in execution time is observed in this case, due to the overall reduction of memory bandwidth.

6. Related work

The current version of Lean Prolog and a few related papers are available at *http://logic.cse.unt.edu/tarau/research/LeanProlog* .

We have designed and implemented our symbol garbage collector by 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 some similarities with the Erlang atom garbage collector proposal described in [11].

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 4. Time (in ms.)/space efforts and benefits of our integrated symbol GC algorithm on three benchmarks

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 5. Time (in ms.)/space efforts and benefits of our integrated symbol GC algorithm on two applications

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 6. Statistics for the Pereira benchmark

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 [11]).

While our paper is based on a finished working collector, [11] describes a proposal for an implementation. While our description provides enough detail to be replicable in another system, [11] 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 on one side, multiple Prolog engines on the other - with the important difference that Erlang processes run in parallel and communicate using message passing while engines are uncoupled from threads in LeanProlog and use a different communication protocol
- 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 class citizens are themselves represented as symbols in our case
- a discussion of an incremental version of the collector is given in [11]
- 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 symbol GC opportunity 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 [27] and its interaction with multi-threading is discussed in [28]. Instead of copying however to a new, compact symbol table, SWI-Prolog leaves a symbol-table with holes. While managing them with a linked list can avoid a 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 very little extra runtime cost to do so. Also, as Lean Prolog is finding some practical uses in applications working on terabytes of data, leaving holes in the symbol table would be an unpleasant limiting factor for the total size of the symbol space.

While not described in detail in a publication that we are aware of, the SICStus Prolog [3] description, in the user manual, of the built-in `garbage_collect_atoms/0`, 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 [29]. 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, a copying heap GC algorithm [7, 8, 26] 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 [9]. Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel [10] and OR-parallel [12] execution models are worth mentioning. While in these systems logic engines are not made available to the programmer through an API, independent virtual machine states are used internally, for instance, when implementing various scheduling policies.

In combination with multithreading our own engine-based API bears similarities with various other Prolog systems, notably [4, 28]. 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].

7. Conclusion

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.

The main novelty of our integrated symbol GC is *support for allocation and exchange between multiple Prolog engines of arbitrary Java objects (including big integers and decimals) and seamless interoperation with various Java libraries*. Such benefits are likely to be replicated by using a similar architecture when implementing scripting or domain specific dynamic languages on top of Java or `C#`.

As a practical consequence, Prolog programmers can benefit from services provided by various Java collection and map classes whose complex object graphs are internalized as needed to garbage collectable Prolog symbols.

Our empirical evaluation indicates that the costs of symbol GC are amortized by consistent reduction of the memory footprint of Prolog's data areas resulting in reduced GC effort on the Java side as well. As a result, we have observed that on some large practical programs, activating our symbol GC can bring not only space savings but also execution time benefits.

We hope that our effort, that lifts symbol GC to a multi-engine context by integrating symbol and engine garbage collection, and generalizes it to manage handles to arbitrary external objects, will be useful to future implementors of logic languages as well as various scripting and domain specific languages built on top of Java or `C#`.

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 independently of their virtual machine architecture and implementation language.

Acknowledgement

We are grateful to Kostis Sagonas and the anonymous reviewers of ISMM 2011 for their salient comments and constructive criticism. We thank NSF (research grant 1018172) for support.

References

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

[2] M. Banbara, N. Tamura, and K. Inoue. Prolog Cafe: a Prolog to Java translator system. *Lecture Notes in Computer Science*, 4369:1, 2006.

[3] M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjoland. SICStus Prolog user's manual.

[4] Manuel Carro and Manuel V. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *ICLP*, pages 320–334, 1999.

[5] J.J. Cook. P#: A concurrent Prolog for the .NET Framework. *Software: Practice and Experience*, 34(9):815–845, 2004.

[6] B. Demoen and P. Tarau. jProlog home page (1996) http://www. cs. kuleuven. ac. be/˜ bmd.

[7] B. Demoen, P.L. Nguyen, and R. Vandeginste. Copying garbage collection for the WAM: To mark or not to mark? *Lecture notes in computer science*, pages 194–208, 2002.

[8] Bart Demoen, Gert Engels, and Paul Tarau. Segment Preserving Copying Garbage Collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, Philadelphia, February 1996. ACM Press.

[9] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/504083.504085.

[10] Manuel V Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In *Proceedings on Third international conference on logic programming*, pages 25–39, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-16492-8.

[11] T. Lindgren. Atom garbage collection. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*. ACM, 2005.

[12] Ewing Lusk, Shyam Mudambi, Ecrc Gmbh, and Ross Overbeek. 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.

[13] Paul Tarau. A Simplified Abstract Machine for the Execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.

[14] Paul Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In Andrei Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.

[15] Paul Tarau. Low level Issues in Implementing a High-Performance Continuation Passing Binary Prolog Engine. In M.-M. Corsini, editor, *Proceedings of JFPL'94*, June 1994.

[16] Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[17] Paul Tarau. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In John Lloyd, editor, *Computational Logic–CL 2000: First International Conference*, London, UK, July 2000. LNCS 1861, Springer-Verlag.

[18] Paul Tarau. Orthogonal Language Constructs for Agent Oriented Logic Programming. In Manuel Carro and Jose F. Morales, editors, *Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems*, Saint-Malo, France, September 2004. URL `http://clip.dia.fi.upm.es/Conferences/CICLOPS-2004/`.

[19] Paul Tarau. BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 2006. URL `http://www.binnetcorp.com/BinProlog`.

[20] Paul Tarau. Logic Engines as Interactors. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24-th International Conference, ICLP*, pages 703–707, Udine, Italy, December 2008. Springer, LNCS.

[21] Paul Tarau. Concurrent programming constructs in multi-engine prolog. In *Proceedings of DAMP'11: ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, New York, NY, USA, 2011. ACM.

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

[23] Paul Tarau and Arun Majumdar. Interoperating Logic Engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*, pages 137–151, Savannah, Georgia, January 2009. Springer, LNCS 5418.

[24] Paul Tarau and Ulrich Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In M. Hermenegildo and J. Penjam, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 73–87. Springer, September 1994.

[25] Satyam Tyagi and Paul Tarau. A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces. In I.V.

Ramakrishan and Gopal Gupta, editors, *Proceedings of PADL'2001*, Las Vegas, March 2001. Springer-Verlag.

[26] R. Vandeginste, K. Sagonas, and B. Demoen. Segment order preserving and generational garbage collection for Prolog. *Lecture notes in computer science*, pages 299–317, 2002.

[27] J. Wielemaker, M. Hildebrand, and J. van Ossenbruggen. Using Prolog as the fundament for applications on the semantic web. *Proceedings of ALPSWS2007*, pages 84–98, 2007.

[28] Jan Wielemaker. Native Preemptive Threads in SWI-Prolog. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2003. ISBN 3-540-20642-6.

[29] Qinan Zhou and Paul Tarau. Garbage Collection Algorithms for Java-Based Prolog Engines. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003*, pages 304–320, New Orleans, USA, January 2003. Springer, LNCS 2562.