TLP **16** (5–6): 950–965, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000272

Lock-free atom garbage collection for multithreaded Prolog

JAN WIELEMAKER
VU University Amsterdam, The Netherlands
CWI Amsterdam, The Netherlands
(*e-mail:* `J.Wielemaker@vu.nl`)

KERI HARRIS
SecuritEase, New Zealand
(*e-mail:* `keri@gentoo.org`)

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

The runtime system of dynamic languages such as Prolog or Lisp and their derivatives contain a *symbol table*, in Prolog often called the *atom table*. A simple dynamically resizing hash-table used to be an adequate way to implement this table. As Prolog becomes fashionable for 24×7 server processes we need to deal with atom garbage collection and concurrent access to the atom table. Classical lock-based implementations to ensure consistency of the atom table scale poorly and a stop-the-world approach to implement atom garbage collection quickly becomes a bottle-neck, making Prolog unsuitable for soft real-time applications. In this article we describe a novel implementation for the atom table using lock-free techniques where the atom-table remains accessible even during atom garbage collection. Relying only on CAS (Compare And Swap) and not on external libraries, the implementation is straightforward and portable.

KEYWORDS: atom, symbol table, atom garbage collection, lock-free, hash table

1 Introduction

An important, but for a long time simple, component of the implementation of dynamic languages is the symbol or atom table. This table maps the string representation of a symbol (atom) to a *handle*. Using handles instead of the original strings avoids duplication, allows for fast equality testing (unification, clause indexing) and causes each symbol to require the same space, which simplifies the representation of data involving atoms. The atom table is traditionally implemented using a dynamically resizing *open hash table*. Such a table provides excellent performance and is easy to implement.

In languages that use symbols to represent only constants from the program, the above is completely adequate. Prolog programs, however, tend to use atoms also for representing constant as well as dynamic strings from data that is processed by the program. For example, NLP (natural language processing) programs tend to represent words from the text they process as atoms. As shown in, e.g., Creutz (2003), the number of unique words does not flatten out if more and more documents are being processed. This means we need *atom garbage collection* (AGC) to dispose of words from old documents if we want to realise programs that can process unbounded input.

As virtually all modern hardware has multiple cores and the number of cores is growing, several Prolog implementations have added support for multiple threads (see table 1). In most systems a thread runs a *goal* using its own stacks, while all threads share the same *program*. Typically, atoms are shared between all threads and thus access to the atom table needs to be synchronised between threads.

A naive way to solve this problem is described in Wielemaker (2003). It relies on a classical atom table for which the consistency is guaranteed using *mutexes* (also called *locks* or *critical sections*) that serialises atom lookup operations. Note that an atom lookup may either return an existing atom or create a new atom. In a typical application most atom lookup operations return an existing atom. Still, also lookup of an existing atom needs to be locked to deal with a table resize as well as AGC. The first implementation of concurrent AGC in SWI-Prolog used a *stop-the-world* approach: the thread that initiates AGC stops all threads, marks all atoms reachable from each thread, removes unmarked atoms from the hash table and finally resumes all threads. This became problematic for two reasons. First, we discovered that reliably 'stopping the world' is troublesome in MS Windows.[1] Second, as programs relying on dozens of threads were developed, the atom table lock became a bottleneck. Our requirements are:

- AGC that allows other threads to proceed, including performing atom lookups. This solves the above mentioned portability problem and makes the system better suitable for (soft) real-time applications.
- Scalable atom lookup, where the lookup time depends as little as possible on the number of threads performing concurrent lookups.

The first step to tackle this was taken in 2013 after we discovered the portability issue mentioned above. We replaced the stop-the-world collector with an asynchronous marking algorithm while using the global atom table lock to perform the collection phase safely. This implies that other threads can proceed during the marking phase, but will block when trying to lookup an atom. The design of this collector is the subject of section 3.2.

[1] This is claimed (`http://www.codeproject.com/Articles/7238/QueueUserAPCEx-Version-Truly-Asynchronous-User-M`) to be possible using a device driver. Using a device driver was not an option due to the required administrative privileges for installing a device driver as well as security considerations that cause many organizations not to accept software that require this. Another claim found is to use GetThreadContext() after SuspendThread(). This proved unreliable in our tests (around 2011) on Windows XP. Similar problems are reported at e.g., `http://stackoverflow.com/questions/3444190/windows-suspendthread-doesnt-getthreadcontext-fails`. Microsoft hints at this solution in a recent (2015) post at `https://blogs.msdn.microsoft.com/oldnewthing/20150205-00/?p=44743`.

With portable support for *atomic* memory operations, notably compare-and-swap (CAS) now being available for all major platforms as well as a wealth of described techniques for using these to build lock-free data structures (e.g., *Read-copy-update* (RCU), *Transactional Memory* (TM), *Hazard pointers* (Desnoyers et al. 2012; Harris *et al.* 2008; Michael 2004)) we decided to replace many of the shared data structures in SWI-Prolog with lock-free alternatives. A crucial and the most challenging data structure was the atom table. The design of this lock-free, resizing hash table and its synchronisation with AGC is the subject of section 4.

This article is organised as follows. In section 2 we discuss the state-of-the-art with regard to atom handling in Prolog and other related work. In section 3 and section 4 we describe our implementation of the atom garbage collector and the lock-free atom table. In section 5, we evaluate our implementation using a couple of real applications as well as an artificial benchmark.

2 Related work

Atom garbage collection is still not widespread in Prolog nor related languages such as Erlang or Ruby. Ruby supports symbol garbage collection as of version 2.2.[2] To our best knowledge, Erlang does not provide atom/symbol garbage collection. Table 1 summarises the support for threads and AGC in popular Prolog systems. AGC for Erlang has been proposed in Lindgren (2005) based on the same motivation as we have, atom are commonly used for the representation of data that is being processed and long running processes will thus collect too many atoms over time. The Erlang community deals with this by avoiding atoms, using lists of characters instead or by restarting nodes periodically. The Prolog community uses the same workarounds. Some systems, e.g., SWI-Prolog, ECLiPSe and LPA Prolog support packed strings to have a compact and natural representation for volatile text as well as avoid the need for AGC.

We find two types of related work in the literature. First, there is a quickly growing body of articles describing lock-free data structures. We particularly refer to Triplett *et al* . (2011), describing a lock-free hash tables based on kernel space RCU techniques. This paper has an extensive section on related techniques for lock-free hash tables. The hash table implementation described provides good lookup performance during resize, a property lacking in our implementation (see section 4). The downside is that it relies heavily on the RCU *wait-for-readers* action, the implementation of which is slow in user space and poorly portable. We considered using `liburcu`[3], but discarded it due to the lack of support for native MS-Windows as well the lack of portability in general that follows from the detailed list of supported CPUs and compilers.

Second, we looked at the work done in the area of multi-threaded symbolic languages. In Lindgren (2005), Thomas Lindgren describes the design of an atom

[2] `https://www.infoq.com/news/2014/12/ruby-2.2.0-released`
[3] `http://liburcu.org`

Table 1. *Thread and AGC support for some popular Prolog systems*

System	Threads	AGC	Notes
SWI-Prolog	Y	Y	
SICStus 4	N	Y	SICStus supports multiple independent runtime systems in one process
YAP 6.3	Y	Y	YAP supports AGC or threads, but not both
B-Prolog 8.1	N	N	
CxProlog 0.98.1	N	Y	
ECLiPSe 6.1	N	Y	ECLiPSe will soon support threads and AGC. The implementation is similar to Lindgren (2005) and Tarau (2011)
GNU Prolog 1.4.4	N	N	
JIProlog 4.1.4.1	N	N	
Lean Prolog 5.4.4	Y	Y	See section 2
Qu-Prolog 9.7	Y	N	
XSB 3.6.0	Y	N	

garbage collector for Erlang. The overall idea is to realise a *copying* collector where we have two symbol tables. If AGC is started, a new table is initialised with the permanent atoms. An Erlang process is moved to the current (new) table when it is resumed. The process scans its memory areas and moves each atom to the new table while updating the used atom-handle. If all processes have been moved, the old table can be discarded. Tarau (2011) describes the symbol garbage collector for *Lean Prolog*. This Java based minimalist Prolog implementation uses symbols as a generalisation for atoms that can also refer to Java objects such as large numbers.[4] The implementation follows the same copying approach as the Erlang proposal described above.

The copying approach has two advantages. First, the size of the atom table is actually reduced and second, migrating the atoms is done by the target thread itself, which avoids the need for asynchronous scanning as described in section 3.2 and naturally distributes the workload over the running threads. In our view, there are also disadvantages. First, after starting a new symbol table, it needs to be populated with all permanent atoms, e.g., those appearing in the static part of the program. Second, all threads will concurrently populate the new table with a potentially large number of atoms. Third, SWI-Prolog threads are particularly designed to be embedded into C code and call arbitrary C code. This may cause long (even infinite) delays before all threads have migrated to the new table and we can discard the old one. This issue is raised by Lindgren but not resolved. Tarau does not mention this. Third, SWI-Prolog is actively used for processing *linked data* (RDF, Klyne and Carroll 2004). Applications like this use many atoms (we used up to 50 million atoms). Copying these atoms is expensive and uses a large amount of memory.

[4] Also, SWI-Prolog atoms are internally generalised to symbols that are also used as safe references to *foreign* objects such as streams, clause references, etc.

The collector described in this article allows threads to proceed their work during AGC, which includes running Prolog code, creating atoms as well as being blocked in system calls or expensive computations done in external languages. It reclaims the actual strings of unreachable atoms. The atom itself is not reclaimed, but reused when new atoms need to be created. It is not hard to imagine workloads where the copying approach is preferable to merely reusing atoms, neither the other way around. It is hard to make a fair judgement for real world usage.

3 A conservative atom garbage collector

This section describes AGC as it is currently implemented in SWI-Prolog. We briefly describe the simple single threaded and stop-the-world algorithms before introducing our current asynchronous marking algorithm. For AGC purposes we distinguish two types of references to atoms.

1. *Volatile* references come from highly dynamic memory areas. Currently these are the Prolog stacks (global and environment), the buffer area used by findall/3 and terms in *message queues* (streams of terms used to exchange messages between threads). Atoms referenced from these areas are identified by scanning these areas during the *mark* phase of AGC.
2. *Explicit* references come from mostly static data structures such as clauses, records, Prolog flags and code using the C interface. The number of such references is stored with the atom and maintained using `PL_register_atom()` and `PL_unregister_atom()`.

The atom lookup functions (`PL_new_atom()` and variations) increment the reference count of the returned atom to avoid it being collected immediately after creation. Functions such as `PL_put_atom_chars()`, which bind a Prolog term to an atom created from a string call `PL_new_atom()`, bind the term to the atom and decrements the reference count of the atom. Thus, calling `PL_put_atom_chars()` with a new unique string creates an atom that is referenced from a term and has its `references` field set to zero. The highest bit of the `reference` (`marked`) field is used for marking that there is a reference from a volatile memory area. Now, AGC performs the steps outlined in algorithm 1

Algorithm 1 Simple AGC

```
 1: function AGC
 2:     for all $volatile\_area$ do
 3:         MARK_ATOMS_IN($volatile\_area$)
 4:     end for
 5:
 6:     for all $a$ in $atoms$ do
 7:         if $a.references = 0$ then                ▷ no mark, no explicit rereferences
 8:             RECLAIM_ATOM($a$)
 9:         else
10:             CLEAR_MARK($a$)
11:         end if
12:     end for
13: end function
```

The above works for single-threaded execution. If we add threads to the picture, we need to take care of volatile references from other threads. We describe two approaches for this. First we provide a brief description of our old stop-the-world collector (Wielemaker 2003), followed by a description of our current conservative collector.

3.1 Stop the world AGC

Stop-the-world agc is similar to the single-threaded of algorithm 1. It merely has to mark the volatile areas of all threads. To do so, it stops each thread and marks all atoms reachable from the volatile areas of the stopped thread. Next, it collects the unreachable atoms and finally it resumes the stopped threads. Note that we cannot resume the threads immediately after marking because they may add new atoms to their volatile areas.

On Unix systems, each thread is signalled. The signal handler marks the reachable atoms and then suspends using `sigwait()`. On Windows, threads are stopped using `SuspendThread()`, after which the AGC thread marks the atoms of the suspended thread. Later we discovered that `SuspendThread()` returns immediately and only prevents the target thread from resuming after its current time slice finishes. After many workaround attempts we concluded there is no reliable way to suspend a thread and wait for it to be really suspended.

Although only one thread is accessing the thread's stacks during marking, the marking happens asynchronously as to avoid AGC (and thus all threads) having to wait until all threads reach a safe check point. This requires careful ordering of modifications to the stacks.

3.2 Conservative AGC

Blocking threads during the entire AGC process, the requirement to scan the stacks asynchronously and the portability issue around suspending threads were the major reasons to seek for another solution for marking the volatile areas. The inspiration came from the *Boehm-Demers-Weiser garbage collector* (BDWGC, Boehm 1993) which performs garbage collection on C data structures by scanning all memory for values that *can* be interpreted as a pointer to a location inside an allocated block of memory. Otherwise, BDWGC is a stop-the-world collector.

Instead of obtaining a root pointer to the current environment frame and choice point and examining all reachable frames and atoms from there, we simply scan the environment and global stack and mark anything that looks ‘atom-like’, but can of course be an accidental bit pattern appearing in, e.g., a floating point number or string. This is the *conservative* aspect: the marker might mark atoms that are in fact not referenced and thus the collector might not collect all atoms. The probability for false marks is reduced by using a tag on the lower bits of an atom handle that excludes clashes with aligned pointers.

As we do not want to stop threads, we need to deal with the fact that the target thread is running and changing the stacks as we mark them. Because we merely

```
Thread 1                           Thread 2

if(agc_is_running)
                                   agc_is_running=TRUE
                                   ...
  markAtom(a);
```

Fig. 1. Race when conditional marking is used.

examine the cells in the volatile areas, trying to interpret the bit pattern as an atom, our marker is safe as long as the location of the volatile areas of a thread remains unchanged while the area is scanned and atoms in the area are not somehow made invisible or moved. The location of volatile areas is changed during a stack shift, while atoms can be invisible and be moved during a thread-local stack-GC. Therefore, these two operations needs to be synchronized between the target thread and the marking thread using locks.

The running thread may create volatile references to new atoms as it proceeds, e.g., by pushing an atom to a stack. Without precaution, this atoms may not be marked. Therefore, we introduce a global variable to indicate that AGC is in progress. When AGC is in progress, operations that add an atom to one of the volatile areas also mark the atom. Similarly, `PL_unregister_atom()` must mark an atom if the last reference is lost while AGC is in progress. A simple conditional mark is insufficient due to the race condition illustrated in figure 1. This is solved by placing such atoms in a designated field in the thread structure that is marked by AGC as illustrated in algorithm 2. In this figure, `LD` represents the thread structure. Now, the atom is marked either by AGC thread scanning `LD->atoms.unregistering` or by the calling thread.

Algorithm 2 Safe conditional marking

```
function COND_MARK_ATOM($a$)
    $LD.atoms.unregistering \leftarrow a$
    if $agc\_is\_running = true$ then
        MARK_ATOM(a);
    end if
end function
```

With these changes the AGC implementation becomes as illustrated in algorithm 3. This implementation has the following properties:

- Threads continue during AGC marking. Currently, AGC is performed by the initiating threads. Future versions may pass this to a dedicated thread and may use multiple threads for the marking.
- Threads suspend on stack shifts or garbage collection while their volatile areas are being scanned. This is acceptable because the additional marking delay is proportional to the delay involved with GC or stacks shifts. More fine grained locking, e.g., by volatile area, is possible.
- Threads creation and destruction suspends during the marking phase of AGC. This also allows for more fine grained locking.
- Atom lookup suspends during the collect phase.

- No non-portable constructs such as suspending other threads are needed. The algorithm can be fully implemented using POSIX thread primitives, although our implementation uses atomic operations for, e.g., updating the atom reference count.

Algorithm 3 Conservative AGC control

```
function AGC
    LOCK(agc)
    if $agc\_is\_running = true$ then
        UNLOCK($agc$) return
    end if
    $agc\_is\_running \leftarrow true$
    for all $t$ in $threads$ do
        MARK_VOLATILE($t$)
    end for
    LOCK(atom_table)
    for all $a$ in $atoms$ do
        if $a.references = 0$ then
            RECLAIM_ATOM($a$)
        else
            UNMARK($a$)
        end if
    end for
    UNLOCK(atom_table)
    $agc\_is\_running \leftarrow false$
    UNLOCK($agc$)
end function
```

4 A lock free atom table

Although the in section 3.2 described atom garbage collector improves portability and reduces the time in which no thread can make progress, it does not avoid contention on the atom table lock and it still causes thread doing atom lookup to block during the collect phase. We describe our implementation that solves these problems in this section. We make the following assumptions and have the following requirements:

- A particular application requires up to a certain number of *live* atoms. This number is not known in advance and therefore the atom table needs to be resized until the required size is reached.
- Although it is necessary to remove no-longer-used atoms from the table, there is not much need to *reduce* the number of buckets in the atom table. This implies that after a startup period, atom table resize operations no longer take place and thus rather poor atom handling performance during the resize operations is acceptable.
- However, atom garbage collection may remain a frequent activity and thus atom lookup must perform well during AGC.
- The implementation must be portable to major operating systems and CPUs. In particular, we wish to limit the required synchronisation primitives to the POSIX mutexes (critical sections on Windows) and the atomic *Compare*

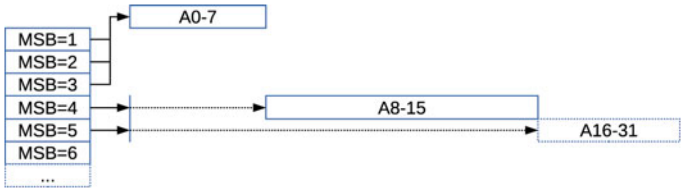

Fig. 2. Dynamic array data structure.

And Swap (CAS) operation for pointers and integers up to the size of a pointer. Modern C compilers generally provide CAS through documented primitives such as GCC's `__sync_bool_compare_and_swap`(*ptr*,*old*,*new*) rather than relying on embedded assembly code.

4.1 Data structures

The atom-table consists of a dynamic array of atom structures. The dynamic array is implemented using an array of base pointers to chunks that double in size as illustrated in figure 2. Typically, the array is statically initialized to a specified size (8 in the figure). Using this representation, the atom at index I ($I > 0$) can be requested using $atomArray[\mathrm{MSB}(I)][I]$.[5] The dynamic array can be extended by allocating a new block and adding it to the MSB index.

The atom structures in the dynamic array have the fields described below. In the actual implementation they have more fields, but these are not relevant for our description of the atom garbage collector.

next Pointer to the next atom in the open hash table.
name Pointer to the represented text.
references The atom reference count. The top three bits are named `reserved`, `valid` and `marked`. The `marked` bit is used for marking references from volatile areas, the `reserved` bit is used to indicate that the atom is not available for creating a new atom and the `valid` bit indicates the atom is fully alive.
next_invalid Pointer to next invalidated (but not yet reclaimed) atom. The use of this field is clarified in algorithm 7.

The `atom_table` is a classical open hash table. Following the RCU approach, the atom table is represented using a structure that is atomically replaced by a new (resized) version by making the global `atomTable` pointer point at the new version. Old versions remain reachable through the `prev` pointer until they can safely be reclaimed. Reclaiming old structures is described in section 4.2. See figure 3.

4.2 Algorithm

This section describes the algorithm to manage the atom hash table as well as reclaiming atoms from AGC. Note that the AGC mark phase described in section 3.2

[5] Many CPUs provide hardware support to compute the *Most Significant Bit*. For example, GCC provides access to this using `__builtin_clzl`().

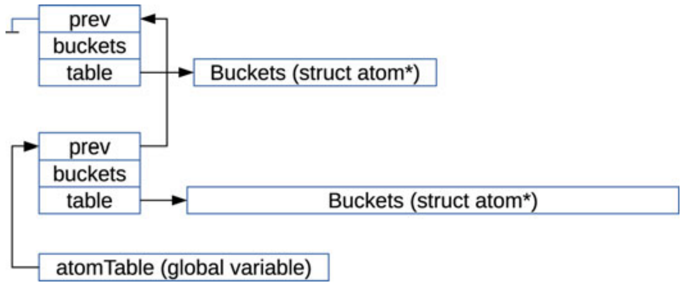

Fig. 3. The atom_table structure.

is not affected. The algorithm is provided as pseudo code. The description below summarises the algorithm while pointing at the relevant fragments of the pseudo code implementation.

1. Fetch the current global atom-table and do a classical open hash table lookup. If the atom is found and marked `valid`, increment `references` using CAS while validating that the atom remains marked as `valid` and return the atom. See algorithm 4, lines 7 to 21 and algorithm 5.
2. If the table is too small (algorithm 4, lines 24 to 26), resize the table (algorithm 6). While the resize is in progress, the `next` pointers linking atoms in the same bucket are generally incorrect. If we have not found the atom and the current atom table is too small we must either resize the table or some other thread is doing that and we wait for the resize to complete. That is why the resize is locked (algorithm 6, lines 4 and 14). If the atom table changed or the current bucked changed (algorithm 4, lines 27 to 29), our lookup may have failed because the table was being resized or a new atom was inserted. We restart the search using the latest table and bucket.
3. Now, if we did not find the atom, it is not in the table. We reserve a new atom by allocating it in the dynamic atom array (see algorithm 9). Next, if we can CAS the reserved atom into the *head* and the table has not changed (algorithm 6, lines 33 to 36) we added a unique atom to the table. We make it valid and return it. If something changed, we reset the references to zero, invalidating the atom and restart the search. This deals with three scenarios: (1) the table was resized while we added the atom, (2) someone else inserted the same atom or (3) someone else inserted a different atom in the same bucket. The last scenario make us redo the lookup and insert for no reason, but this only happens when two threads create two different atoms in the same bucket which should be rare.

The above describes lookup of an atom, resizing the table and adding a new atom to the table while maintaining the unique atom-to-string mapping.

Two issues still need to be addressed. First, AGC may find the atom is ready to be collected. This is realised by algorithm 7, where we use CAS to clear the valid bit (lines 4 to 7). This, together with algorithm 5 which is used to return a found atom from the table ensures that while doing a lookup of an atom that is being invalidated by AGC either makes the lookup win, cancelling collection by AGC or AGC wins and a new atom with the same string is created by the lookup. Note

Algorithm 4 Atom lookup

```
 1: global var $atomTable$
 2: thread var $LD$
 3:
 4: function LOOKUP_ATOM(string)
 5:     var $table, buckets, head$
 6:     loop
 7:         $LD.atomTable = atomTable$
 8:         $table \leftarrow LD.atomTable.table$
 9:         $buckets \leftarrow LD.atomTable.buckets$
10:         $key \leftarrow$ HASH$(string)\&(buckets - 1)$
11:         $head \leftarrow table[key]$
12:         $LD.atomBucket = \&table[key]$
13:
14:         for $a \leftarrow head; a; a \leftarrow a.next$ do
15:             $references \leftarrow a.references$
16:             if IS_VALID$(references) \bigwedge string = a.name$ then
17:                 if BUMP_REF$(a, references)$ then
18:                     $LD.atomBucket = LD.atomTable = NULL$
19:                     return $a$
20:                 end if
21:             end if
22:         end for
23:
24:         if table too full then
25:             RESIZE_ATOM_TABLE
26:         end if
27:         if $table$ or $bucket$ not current then
28:             continue loop
29:         end if
30:
31:         $a \leftarrow$ RESERVE_ATOM$(string)$
32:         $a.next \leftarrow table[key]$
33:         if CAS$(\&table[key], head, a) \bigwedge table$ is current then
34:             $a.references \leftarrow 1|VALID|RESERVED$
35:             $LD.atomBucket = LD.atomTable = NULL$
36:             return $a$
37:         else
38:             $a.references \leftarrow 0$
39:         end if
40:     end loop
41: end function
```

that if AGC wins the atom changes identity. As the old identity is not in use, this is harmless. Second, we must reclaim old data structures: (1) tables that have been resized accessible through the `prev` from *atomTable* and (2) invalidated atoms that are linked into *invalidAtoms* in lines 8 and 9 of algorithm 7. For this, we keep a pointer at the table and bucket being processed in the thread's local data. These pointers are updated in algorithm 4, lines 7, 12, 18 and 35. At the end of AGC, algorithm 8 is called. This collects all bucket pointers in use by all threads and actually reclaims the atom if none of the buckets in which the atom must appear in any of the tables is referenced by any thread. Note that the collected bucket pointers is just a snapshot, but as none of the current buckets contain the atom, new bucket pointers will never encounter the atom. Likewise, old tables (`prev`) that are not in use by any thread are reclaimed. This step is trivial and not included in the pseudo code.

Algorithm 5 Claim an atom as valid

```
function BUMP_REF(a, references)
    loop
        if CAS($\&a.references, references, references + 1$) then
            return true
        else
            $references \leftarrow a.references$
            if $\neg$IS_VALID($references$) then
                return false
            end if
        end if
    end loop
end function
```

Algorithm 6 Resize atom table

```
global var $atomTable$

function RESIZE_ATOM_TABLE
    LOCK($agc$)
    if table too full then
        $newtable \leftarrow$ ALLOC_ATOM_TABLE($atomTable.buckets * 2$)
        $newtable.prev \leftarrow atomTable$
        for all atom $a$ in $atomTable$ do
            if IS_VALID($a.references$) then
                ADD_TO_TABLE($newtable$)
            end if
        end for
    end if
    $atomTable \leftarrow newtable$
    UNLOCK($agc$)
end function
```

Algorithm 7 Invalidate atom (during AGC collect phase)

```
global var $invalidAtoms$

function INVALIDATE_ATOM(a, references)
    $newrefs \leftarrow references\&\tilde{}valid$                    ▷ Clear valid bit
    if $\neg$CAS($\&a.references, references, newrefs$) then
        return false
    end if
    $a.next\_invalid \leftarrow invalidAtoms$
    $invalidAtoms \leftarrow a$
    return true
end function
```

5 Evaluation

We evaluated the atom table using an artificial test that stresses the atom table to the limit. Although many applications hardly stress the atom table, we also identified scenarios from existing applications where the new atom table significantly improves performance.

For the artificial test we enumerate all answers of the ISO predicate `sub_atom/5` where the first argument is instantiated to an atom consisting of the (Unicode) characters 0..1000. This tests looks up 502,503 atoms. The test is run on multiple threads concurrently. The hardware is a dual Intel Xeon E5-2650 CPU system ($2 \times 8 = 16$ cores, 32 threads) running Ubuntu 14.04. We ran the tests in four

Algorithm 8 Reclaim invalidated atoms (during AGC collect phase)

```
global var $invalidAtoms$
global var $atomTable$

function DESTROY_ATOMS
    $buckets$ = ATOM_BUCKETS_IN_USE                      ▷ Collects $LD.atomBucket$ of threads
    for all atom $a$ in $invalidAtoms$ do
        if DESTROY_ATOM($a, buckets$) then
            remove $a$ from $invalidAtoms$
        end if
    end for
    FREE(buckets)
end function

function DESTROY_ATOM(a, buckets)
    $t \leftarrow atomTable$
    $key \leftarrow$ HASH($a.string$)
    while $t != NULL$ do
        $v = key\&(t.buckets - 1)$
        if $\&t.table[v]$ in $buckets$ then
            return false                                 ▷ A thread scans this bucket
        end if
        $t \leftarrow t.prev$
    end while
    $a.name \leftarrow NULL$
    $a.references \leftarrow 0$
    return true
end function
```

Algorithm 9 Reserve a new atom

```
global var $atomArray$

function RESERVE_ATOM
    loop
        for all $a$ in $atomArray$ do
            $refs \leftarrow a.references$
            if IS_FREE($refs$) $\bigwedge$ CAS($\&a.references, refs, refs|$reserved) then
                return $a$
            end if
        end for
        Add new block to $atomArray$ (locked)
    end loop
end function
```

conditions, comparing version 6.5.1 (prior to conservative AGC) to 7.3.20 and both while collecting the volatile atoms and pre-allocating these atoms, testing only lookup. The results are shown in table 2. We make the following observations:

- Concurrent lookup (rows 13 . . . 24) shows that, if atom lookup is dominant, there is no speedup from using multiple cores when using a lock based atom table. Our lock-free version shows good scalability up to 16 threads (the number of physical cores).
- With AGC reclaiming the volatile atoms (rows 1 . . . 12) we see a similar reduction of the total process CPU usage, but a much smaller reduction in wall time usage. The AGC *time* column gives a hint. AGC time is small in the old version, where marking is done by the threads in parallel. It is high in the new version, where the AGC thread performs all the marking.

Table 2. *AGC performance for old (6.5.1) and new (7.3.20) versions of SWI-Prolog. Note that the last row of each section relies on hyper-threading.*

		Time (sec)		Atom GC		
Row	# Threads	Process	Wall	# Invocations	Reclaimed bytes	Time
6.5.1, AGC active						
1	1	0.657	0.657	49	660,818,627	0.049
2	2	3.587	2.140	85	1,167,071,837	0.159
3	4	10.725	3.965	169	2,299,293,054	0.339
4	8	33.082	5.098	183	2,304,928,223	0.380
5	16	117.574	8.295	54	719,046,656	0.176
6	32	429.078	30.666	849	9,388,585,427	3.121
7.3.20, AGC active						
7	1	0.632	0.633	49	660,941,810	0.049
8	2	1.506	0.788	98	668,102,982	0.110
9	4	3.018	0.803	49	682,083,694	0.215
10	8	8.351	1.648	238	2,783,054,946	4.987
11	16	20.365	4.791	491	9,103,288,495	19.816
12	32	45.590	12.369	811	18,112,260,091	44.880
6.5.1, atoms pre-allocated						
13	1	0.746	0.746	0	0	0.000
14	2	3.554	2.067	0	0	0.000
15	4	9.273	3.439	0	0	0.000
16	8	27.471	4.009	0	0	0.000
17	16	117.049	7.918	0	0	0.000
18	32	296.947	21.847	0	0	0.000
7.3.20, atoms pre-allocated						
19	1	0.595	0.595	0	0	0.000
20	2	1.708	0.876	0	0	0.000
21	4	2.454	0.715	0	0	0.000
22	8	4.811	0.718	0	0	0.000
23	16	10.851	0.687	0	0	0.000
24	32	28.506	1.188	0	0	0.000

The first real-world evaluation was performed using ClioPatria[6]. ClioPatria is a linked data platform running on SWI-Prolog. Node identifiers (IRIs) are represented as atoms. Likewise, RDF *literals* are represented as atoms and a *token* → *literal* index is created to allow for full text search. The Linked Politics project converted the European parliament speeches to RDF, creating 26 million triples that require 9 million atoms to represent as described above. We timed the loading time. ClioPatria loads the different sources (graphs) in parallel. The test ran on the same hardware as above, using 32 threads for loading the data. The results are shown in table 3.

[6] `http://cliopatria.swi-prolog.org`

Table 3. *ClioPatria load time for 26M triples. Times are in seconds.*

	Graphs	Triples	Wall time	CPU %	CPU time
7.2.3 (conservative AGC; Linux)	47	26,192,652	1667.73	1641	27365
7.3.20 (lock-free AGC; Linux)	47	26,192,652	861.85	1409	12140

Table 4. *Time to answer FIX messages*

	Mean Time (ms)	Stddev	Max Time
7.2.3 (conservative AGC; Linux)	24.82	6.22	54.91
7.3.20 (lock free AGC; Linux)	16.66	0.42	17.99
7.2.3 (conservative AGC; Windows)	41.04	22.10	259.71
7.3.20 (lock free AGC; Windows)	24.32	0.50	26.59

The second real-world evaluation was performed using the SecuritEase stock-broking system running on SWI-Prolog. The application was placed under a representative load, decoding Financial Information eXchange (FIX) messages. For the test, 500 client requests were processed. The time to service each request was logged. Each test cycle used the same FIX message workload and client request workload. The hardware used for this test is an Intel i7 2720-QM CPU system (4 cores, 8 threads). Results were obtained for Gentoo Linux 4.1.15 and Windows Server 2008 R2. The results are presented in table 4. Using the lock-free atom table, the mean time to service requests was noticeably reduced. Of particular note is the low variance in timings using the lock-free atom table.

6 Conclusions

We have presented a practical and portable approach to implement lock-free access to the symbol table for concurrent dynamic languages such as Prolog, Erlang or Ruby. Lookup of existing atoms scales nearly perfect up to 16 threads on 16 physical cores. Atom garbage collection only cause thread heap expansion and garbage collection to suspend. Performance can be further enhanced by using multiple threads for the marking phase and more fine-grained locks that protect the marking phase.

Acknowledgments

We thank Paulo Moura, Joachim Schimpf and Paul Tarau for their input for table 1. The development of SWI-Prolog is supported by the Dutch national program COMMIT/.

References

Boehm, H.-J. 1993. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93. ACM, New York, NY, USA, 197–206.

Creutz, M. 2003. Unsupervised segmentation of words using prior distributions of morph length and frequency. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03. Association for Computational Linguistics, Stroudsburg, PA, USA, 280–287.

Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R. and Walpole, J. 2012. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23,* 2, 375–382.

Harris, T., Marlow, S., Jones, S. L. P. and Herlihy, M. 2008. Composable memory transactions. *Commun. ACM 51,* 8, 91–100.

Klyne, G. and Carroll, J. J. 2004. Resource description framework (RDF): Concepts and abstract syntax. World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210.

Lindgren, T. 2005. Atom garbage collection. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05. ACM, New York, NY, USA, 40–45.

Michael, M. M. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15,* 6 (June), 491–504.

Tarau, P. 2011. Integrated symbol table, engine and heap memory management in multi-engine Prolog. In *Proceedings of the International Symposium on Memory Management*. ISMM '11. ACM, New York, NY, USA, 129–138.

Triplett, J., McKenney, P. E. and Walpole, J. 2011. Resizable, scalable, concurrent hash tables via relativistic programming. In *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*, J. Nieh and C. A. Waldspurger, Eds. USENIX Association.

Wielemaker, J. 2003. Native preemptive threads in SWI-Prolog. In *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9–13, 2003, Proceedings*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, 331–345.