Generation-Friendly Eq Hash Tables

Abdulaziz Ghuloum and R. Kent Dybvig
Department of Computer Science, Indiana University, Bloomington, IN 47408
{aghuloum,dyb}@cs.indiana.edu

Abstract

Eq hash tables, which support arbitrary objects as keys, distinguish keys via pointer comparison and often employ hash functions that utilize the address of the object. When compacting garbage collectors move garbage collected objects, the addresses of such objects may change, thus invalidating the hash function computation. A common solution is to rehash all of the entries in an eq hash table on the first access to the table after a collection. For a simple stop-and-copy garbage collector, which moves every element of a hash table, the rehashing overhead is proportional to the amount of work done by the collector and so the cost of rehashing adds only constant overhead. Generational copying collectors, however, may move a few or none of the entries, so rehashing a large table may cost considerably more than the garbage collection run that caused the rehash. In other words, such rehashing is not "generation friendly."

In this paper, we describe an efficient, generation-friendly mechanism for implementing eq hash tables. The amount of work required for rehashing is proportional to the work performed by the collector as only objects that actually move during a collection are rehashed. The collector supports eq hash tables and their variants via a simple new type of object, a *transport link cell*, the handling of which by the collector is nearly trivial.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—data structures; D.3.4 [*Programming Languages*]: Processors—memory management, runtime environments

General Terms Languages

Keywords Scheme, hash tables, generational garbage collection, generation-friendliness

1. Introduction

A hash table (Dumey 1956; Knuth 1998; Cormen et al. 2001) associates a set of keys with a corresponding set of values. A hash table employs a hashing function h that maps keys from the universe of possible keys to a limited range of integer values, which are used as indices into the table. Proper hashing depends on the property that h is a function in the mathematical sense: for every k, $h(k)$ must be the same every time $h(k)$ is computed. Efficient hashing (i.e., $O(1)$ cost for lookup and update operations) relies on the ability of the hashing function to produce uniformly distributed values.

Hash tables can be characterized by the predicates used to distinguish keys. Eq hash tables, which support arbitrary objects as keys, distinguish keys via pointer comparison, e.g,. Scheme's `eq?` predicate or C's `==` operator. Because keys are distinguished by their address in memory, the most effective eq hash functions are based on those addresses, since the address of an object is different from that of every other object and the addresses are already scattered in memory. Hash functions based on objects' addresses are also the most efficient, since no traversal of the objects' sub-parts is required. Unfortunately, compacting garbage collectors, which are used by many implementations, can change the address of an object at any time—in particular, between the time it is entered into an eq hash table and the time it is retrieved. An attempted retrieval based on the new address is likely not to succeed. A common solution to this problem is to rehash all of the entries in an eq hash tables on the first access of the table since the last garbage collection run. For simple copying collectors, which touch every element of a table in shifting it from old to new space, the rehashing overhead is proportional to the amount of work done by the collector and so adds only constant overhead. Generational copying collectors, however, may move none or only some of the entries. Rehashing a large table may cost considerably more than the cost of the garbage collection run that caused the rehash. In other words, such rehashing is not "generation friendly."

In this paper, we describe an efficient, generation-friendly mechanism for implementing eq hash tables. The amount of work required for rehashing objects that move during garbage collection is at most proportional to the work performed by the collector. Consequently, the cost of rehashing is $O(1)$ amortized over the lifetime of the hash table. Furthermore, the mechanism adds no overhead to programs that do not make use of hash tables. In addition, none of the common hash-table operations require synchronization with the collector, i.e., disabling the collector during the operation. The mechanism is also flexible in that it allows eq hash tables, and variants such as weak eq hash tables or hash tables based only partly on object address, to be created by the user. The collector supports eq hash tables and their variants via a simple new type of object, the *transport link cell*, which are inspired by transport guardians (Dybvig et al. 1993). Transport link cells are simple to implement efficiently within the collector and thus contribute almost nothing to the complexity of the implementation or the cost of a collection run.

The rest of this paper is organized as follows: Section 2 gives brief background information about hash tables and compacting generational garbage collection techniques. Section 3 describes three existing implementation techniques of hash table that are found in practice and how each fails to meet our performance criteria. Section 4 presents a prototype solution of implementing eq

Proceedings of the 2007 Workshop on Scheme and Functional Programming
Université Laval Technical Report DIUL-RT-0701

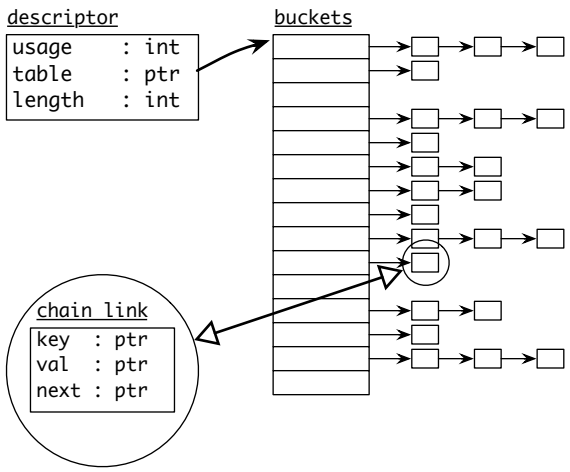

Figure 1. Chained Hash Tables: A hash table is represented as a descriptor holding (a) a counter indicating the number of key/value pairs stored in the table, (b) a pointer to a vector of buckets, and (c) the number of buckets in the vector. A (possibly empty) linked list of key/value pairs is stored in each bucket.

hash tables using transport guardians. Section 5 presents our final solution based on transport link cells and describes their implementation within the collector. Section 6 analyzes the performance of our implementation. To illustrate the flexibility of our approach, Section 7 presents some possible user-space extensions to the basic hash-table mechanism that employ the same transport link cell mechanism. Finally, section 8 presents our conclusions and discussions future research directions.

2. Background

This section gives a brief overview of chained hash tables, generational garbage collection, and the problems associated with the interaction of the two subsystems. Readers who are familiar with these topics can skip this section without loss of continuity.

2.1 Chained hash tables

A chained hash table (see figure 1) is a data structure with the following components:

- *Hash table descriptor:* The descriptor contains information about the hash table such as the number of elements contained in the table, the number of buckets in the buckets array, a pointer to the buckets array, and the maximum capacity of the table.
- *Buckets Array:* A contiguous area of memory where every element of the array (a bucket) is a pointer to a chain of links.
- *Chain Links:* A link in the chain is composed of a key, a value associated with the key, and a pointer to the next link in the chain.

In addition to the data structures, a hash table uses a *hash function*, h, that maps a hash key, k, to an integer index $i = h(k)$. The hash function computes a hash value either based on the content of the key (e.g., symbol tables use the content of the input string to compute the hash value), or based on the address of the key (e.g., eq hash tables mapping arbitrary objects to values based on the identity of the key objects).

The set of required operations on hash tables varies among programming languages and their implementations. At a minimum, a hash table implementation must supports the following operations:

- `(hash-table-put! ht k v)`: associates the value *v* with the key *k* in the hash table *ht*.
- `(hash-table-get ht k d)`: retrieves the value associated with the key *k* in the hash table *ht*. If k does not exist in the table, the default value d is returned.
- `(hash-table-delete ht k)`: removes any association of the key *k* in the hash table *ht*.

To find the value associated with a key in a hash table, the index of the bucket containing the key is obtained by computing an index $idx = h(k) \bmod \text{length}(ht)$. The chain stored at that bucket idx is then searched sequentially by comparing the key of each chain link to the key requested.

Many programming languages (e.g., Common LISP (American National Standards Institute and Information Technology Industry Council 1996), Perl (Wall et al. 1996), Python (Rossum 1992), Scheme (Sperber et al. 2007), etc.) require additional operations on hash tables such as:

- `(hash-table-member? ht k)`: returns a boolean value indicating whether the hash table *ht* contains a binding for the key *k*.
- `(hash-table-keys ht)`: returns the list of keys stored in the hash table.
- `(hash-table-map ht p)`: applies the procedure *p* to every key/value pair in the hash table.

2.2 Compacting garbage collectors

Implementations of programming languages that support automatic memory management often use garbage collection to reclaim unused heap space (Wilson 1992). A large contiguous block of memory is pre-allocated to serve as an allocation heap from which individual objects are allocated. Exhaustion of the allocation area triggers the garbage collection routine which moves all live objects (e.g., objects that are reachable through the transitive closure of the program roots including the registers and the stack) to another area in memory, and thus freeing the old allocation area.

Generational garbage collectors separate heap objects into generations based on their age (Lieberman and Hewitt 1983; Sansom and Jones 1993; Appel 1989; Dybvig et al. 1994). Memory is divided into a (typically fixed) number of generations. Objects in younger generations are collected more frequently than those in older generations. New objects are allocated sequentially into the youngest generation (a.k.a. the *nursery*). Once the nursery is exhausted, the garbage collector is invoked to clean up the nursery by moving the surviving young objects to the first generation. Depending on some heuristics, the garbage collection run may clean up other generations in addition to the nursery area. Objects in generation i that survive a collection are moved to the next generation $i + 1$. Objects that survive the last generation are either tenured (i.e., moved to a permanent storage area and never collected again) or are moved back into the last generation.

2.3 The interaction between eq hash tables and compacting garbage collection

If the address of a key is used in part by the hashing function to obtain the index of the bucket in which the key is stored, and if the garbage collector changes the address of the object in memory, then lookup operations on the hash table will incorrectly fail to find some keys that *do* reside in the table. This is obviously a problem that implementations of eq hash tables must address.

3. Common Solutions

Implementations of programming languages that require supporting eq hash tables have adopted different strategies for coping with the interaction problem between hash tables and garbage collectors. This section summarizes some of the known solutions.

3.1 Extending all heap objects with a *hash value* field

One way of making hash tables work in garbage collected environments is by using a hash value that is different from the changing address of the object. With this technique, all heap objects are padded with an additional memory cell that holds a constant integer hash value (Agesen 1999).

Augmenting all memory objects with an additional field is easy to understand and to implement. Unfortunately, this solution makes the construction and collection of every object in the system more expensive. Languages whose paradigm rely on constructing many small temporary data structures (such as `cons` pairs in Lisp and Scheme, tuples in ML, closures, small vectors, etc.) suffer the most with this technique since most objects are not used as hash table keys.

Some systems attempt to optimize the memory requirement by adding a hash value field *on-demand* (Agesen 1999). Unfortunately, such optimization is not always possible. For example, many optimizing compilers for Scheme and Common Lisp represent `cons` pairs as a tagged pointer to a two-word object holding the `car` and `cdr` fields of the pair. With such a tight representation, objects do not contain an extensible field to hold the hash value.

3.2 Rehashing as part of garbage collection

Another technique is to rehash the contents of each hash table during garbage collection (Hanson 2005). This technique does not penalize programs that do not use hash tables. Because the hash table data structures are owned by the garbage collector, however, operations that access hash tables must generally be performed with the garbage collector disabled, and the code to disable the collector may add considerable overhead to each operation. This also has the effect of complicating the task of the garbage collector in addition to prohibiting implementing hash tables as a user-level feature. Also, the cost of rehashing all hash tables on every garbage collection can dominate the cost of the garbage collection run. This is most noticeable on generational garbage collectors where minor collection runs are frequent and their cost is small.

3.3 Rehashing when lookup fails

An improvement over the second strategy is to delay rehashing until it is needed (Haible et al.; Dybvig 2007). If the lookup operation of an element fails, and a garbage collection occurred since the last rehash, the entire table is rehashed and lookup is retried on the newly rearranged table. The advantage of this approach is that not all hash tables are rehashed on every collection. Instead, only the hash tables that are used after collection are rehashed, and only if a lookup operation fails.

This technique performs well for applications in which operations rarely fail, e.g., a database from a finite domain where all lookup operations are performed on older-generation (and thus largely stationary) keys in the database. This approach also benefits hash tables that lay dormant for extended periods of time. There are, however, no guarantees on performance with this technique, since any failed lookup can trigger expensive rehashing.

This problem is recognized by implementors, as illustrated by the following excerpt from the implementation notes for GNU CLISP (Haible et al.):

> `EXT:FASTHASH-EQ`: This uses the fastest possible hash function. Its drawback is that its hash codes become invalid at every garbage-collection (except if all keys are immediate objects), thus requiring a reorganization of the hash table at the first access after each garbage-collection. Especially when generational garbage-collection is used, which leads to frequent small garbage-collections, large hash table with this test can lead to scalability problems.

4. Prototype Solution

There are two main problems with the solutions presented in Section 3. The performance of systems employing a *hash value* field suffers because every object occupies more memory, and thus the cost of allocation and garbage collection are increased. The performance of systems employing the rehashing strategy suffer because rehashing operations are performed on entire hash tables, even if only few or no keys are moved during garbage collection. To obtain an efficient, generation-friendly solution, the system must rehash *only* the keys that move during garbage collection. A key that does not move must *not* be rehashed.

Our prototype solution uses *transport guardians* (Dybvig et al. 1993), a mechanism by which the garbage collector can communicate with the hash table implementation the set of objects that have moved during collection. In this section, we first review transport guardians then show how they can be put to use to derive our implementation of hash tables.

4.1 Guardians

Guardians (Dybvig et al. 1993) allow programs to determine when the last reference to an object has been dropped and "resurrect" the object, often to perform some cleanup action (e.g., flushing and closing open file handles) before the storage associated with the object is finally released. From the programmer's perspective, a guardian is a procedure. If passed one argument, the guardian registers the object for resurrection; if passed no arguments, the guardian returns a resurrected object, if one has been resurrected, otherwise `#f`. Any number of guardians may be created, and each guardian may guard any number of objects. New guardians are created using `make-guardian`.

The simple example below illustrates the mechanism.

```
(let ([G (make-guardian)])
  (G (cons 'a '()))
  (let loop ()
    (cons 'b '())
    (or (G) (loop))))
```

The example creates a new guardian, which it calls `G`, creates a new one-element list `(a)`, and registers the list with the guardian. It then loops, allocating and discarding a new one-element list on each iteration. Assuming that the inner `cons` is not eliminated as useless code by the compiler, a collection is eventually required to reclaim the storage for the discarded lists. At this point, the original list `(a)` is recognized as no longer accessible except through the guardian and is thus resurrected. The next call to `G` returns the resurrected list and the loop terminates.

Guardians are handled in a cooperative manner by the collector and mutator. (The mutator is the code that runs outside of the collector, creating and releasing objects while the collector cleans up after it.) The two subsystems interact through a queue encapsulated within the guardian. When an otherwise inaccessible guarded object is found by the collector, it enqueues the object, and when the guardian is invoked without arguments by the mutator, the first object on the queue, if any, is dequeued.

The queue associated with each guardian is represented as a *tconc*, or *tail concatenation* object (Teitelman 1974). A tconc is a pair whose `car` points to the first pair of a nonempty list and

whose `cdr` points to the last pair of that list. Elements are enqueued by extending the list destructively through the last-pair pointer and dequeued through the first-pair pointer. The queue is empty when both pointers point to the same pair.

The procedure `make-guardian` allocates a new tconc and returns a procedure that encapsulates the tconc. When the guardian is passed an object to guard, it hands it off to the collector, which adds a pointer to it along with the guardian's tconc to an internal (per generation) list of guarded objects. After a given generation is collected, the collector scans that generation's list of guarded objects for any that are otherwise inaccessible, promotes them to the next generation if necessary, and enqueues each via its associated tconc. When the guardian is called without arguments, it compares the first-pair and last-pair pointers of the tconc and, if different, dequeues and returns one object. (See Figure 2.)

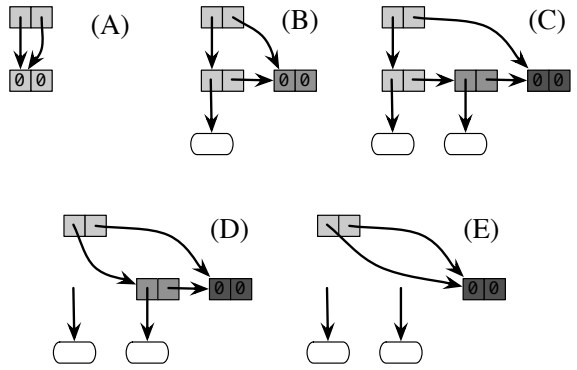

Figure 2. The lifetime of a guardian: (A) A guardian is a simple tconc pair with its head and tail pointing to the same object. (B, C) The garbage collector adds objects that have no strong pointers to the tail of the tconc. (D, E) The mutator pops the rescued elements from the head of the tconc. No synchronization between the mutator and the collector is needed since the mutator modifies the head of the tconc while the collector modifies its tail.

4.2 Conservative Transport Guardians

Transport guardians are a variant of guardians through which the garbage collector communicates to the mutator that an object's address has changed.

A conservative variant of transport guardians can be implemented on top of the general guardians using a simple trick (Dybvig et al. 1993). By constructing a new indirection object that is guaranteed to be younger than the guarded object, and by dropping the reference to the new object immediately after registering it with the guardian, we are guaranteed that whenever the collector is triggered, our new object will be found dead and is therefore resurrected and added to the tconc. This indicates that our initial object was potentially moved during the garbage run. Whenever the indirection object is popped from the tconc, it is registered again to monitor future collection runs. As the indirection object ages in the system, it eventually resides in the same generation as the original monitored object. Ideally, the the object should hold onto the guarded object via a weak pointer so that it does not cause otherwise inaccessible objects to be retained unnecessarily.

4.3 Combining Hash Tables and Transport Guardians

Transport guardians provide the means to detect which objects have moved, and therefore their addresses have changed, due to garbage collection. In order to utilize the guardians, we modify the representation of chained hash tables in two ways:

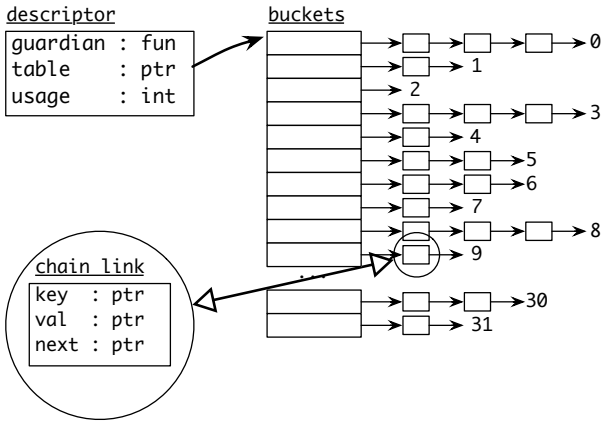

Figure 3. Modified Hash Tables: Each hash table contains a pointer to a transport guardian that keeps track of which keys have potentially moved during collection. Additionally, the buckets are terminated by an integer value denoting the index of the bucket in the table.

1. A transport guardian is constructed with every hash table to keep track of which hashed objects move during garbage collection.
2. The index of every chain is initially stored at the bucket. Therefore, the tail of every chain is the index of the chain in the table.

Figure 3 shows our modified hash table structure.

4.4 Adding elements to a hash table

Adding a key/value pair to a hash table involves the following steps: First, a new link is constructed to hold the key and value. Next, the link is registered with the hash table's transport guardian. Finally, the address of the key is obtained and is used to compute the index of the chain in which the object is stored.

```
(define (hash-table-add ht key val)
  (let ([link (make-link key val)])
    ((ht-transport-guardian ht) link)
    (insert-link ht link)))

(define (insert-link ht link)
  (let ([key (key-of link)])
    (let ([idx (mod (int-hash (address-of key))
                    (length-of ht))])
      (set-next-of link (hash-table-ref ht idx))
      (set-hash-table ht idx link))))
```

Two points are worth mentioning here. First, we are registering the link, rather than the key, with the transport guardian. This is because we need to obtain the link, and not the key, when the address of the key changes. If a general-purpose transport guardian is used, it will notify us when the link is moved. As long as the link is constructed after the key, this usage yields a correct, but sometimes conservative behavior. The guardian may signal some false positives when the link alone is moved until the link gradually ages and becomes as old as the key it holds. Section 5 shows how to implement more efficient transport guardians that do not signal such false positives.

Second, the address of the key *must* be obtained *after* the guardian is installed. Otherwise, the garbage collector may be triggered between the time of obtaining the address of the key and the

time a guardian is installed, resulting in incorrect hashing using an outdated address.

4.5 Link Lookup

A link lookup operation on a hash table is a procedure that takes a hash table and returns the link in which the key is stored, or false if the key is not in the table. Link lookup is useful for implementing both lookup and update operations.

When a key is added to the hash table, the link is registered with the transport guardian and the address of the key is used to compute the index. At the time of lookup, either the garbage collector has not moved the key, in which case computing the index during lookup will yield the same index that was used during insertion; or the garbage collector has moved the key, in which case the transport guardian would have captured the event and will return the links to all moved objects.

We can therefore optimistically assume that the key was not moved and perform a direct lookup on the key, ignoring interaction from garbage collection. Only if that lookup fails do we check with the guardian.

```
(define (link-lookup ht key)
  (or (direct-lookup ht key)
      (rehash-lookup ht key)))
```

The `direct-lookup` operation involves computing the hash value of the key and traversing the chain at the computed index. The `rehash-lookup` operation invokes the guardian of the hash table, and if it returns a moved link, it rehashes that link. If the popped link contains the required key, it is returned and rehashing is terminated. Otherwise, the rehashing operation is retried until the key is found or all the moved objects are rehashed.

```
(define (rehash-lookup ht key)
  (let ([link ((transport-guardian-of ht))])
    (and link
         (begin
           (delete-link ht link)
           (insert-link ht link)
           (if (eq? (key-of link) key)
               link
               (rehash-lookup ht key))))))
```

Unlinking a chain link is performed by first obtaining the index of the bucket in which the link resides. The index of the bucket cannot be computed from the key since the address of the key no longer matches the address used initially to store the link. What we do instead is traverse the chain to the end to obtain the index we stored originally in the table.

```
(define (index-of x)
  (if (fixnum? x)
      x
      (index-of (next-of x))))
```

Once we obtain the index of the chain, we traverse the chain from the front and modify the pointer to the link to point to the next link in the chain.

```
(define (delete-link ht link)
  (let ([idx (index-of link)])
    (let ([chain (hash-table-ref ht idx)])
      (if (eq? chain link)
          (set-hash-table! ht idx (next-of link))
          (delete-loop chain link)))))
```

```
(define (delete-loop chain link)
  (let ([x (next-of chain)])
    (if (eq? x link)
        (set-next-of chain (next-of link))
        (delete-loop x link))))
```

4.6 Key Lookup and Update

Once we have a way of looking up the link in which a particular key resides in the hash table, implementing `hash-table-get` and `hash-table-put!` are straightforward as shown in the following code snippets.

```
(define (hash-table-get ht key default)
  (let ([x (lookup-link ht key)])
    (if x
        (value-of x)
        default)))

(define (hash-table-put! ht key val)
  (let ([x (lookup-link ht key)])
    (if x
        (set-value-of x val)
        (hash-table-add ht key val))))
```

5. Transport Link Cells

The preceding section demonstrated that transport guardians, which can be implemented on top of general guardians, allow the construction of generation-friendly eq hash tables. The mechanism does, however, have two drawbacks. First, it is conservative in the sense that some objects may be reported as having moved when in fact they have not, as described in the preceding section. Second, the use of guardians requires that the guarded objects be registered with the garbage collector, i.e., entered into a list of guarded objects. This in turn forces the collector to explicitly manage the objects even if the hash table employing the guardian has been dropped by the program, essentially adding undesirable deallocation overhead.

In this section, we describe our final solution based on *transport link cells*, a specialized variant of transport guardians. Transport link cells eliminate both drawbacks with the use of transport guardians. By providing accurate information about when objects are moved by the collector, no rehashing occurs for old objects that are not moved during a collection. Second, transport link cells need not be registered with the garbage collector so that when a cell is dropped, it incurs no further collection overhead. If links are reclaimed before collection, as is often the case for short-lived hash tables, they are never touched by the collector and are reclaimed for free.

Transport link cells, or TLCs, are used by the mutator as the *chain links* of the hash table buckets as illustrated in figure 3. TLCs are also used by the collector as a way of linking the guarded key and a guardian tconc. A TLC is an object with four fields:

- key: The key field serves two purposes: (1) It is used by the lookup and update operation of hash table for comparison, and (2) it is also used by the garbage collector for tracking address changes.
- value: The value associated with the key.
- tconc: This fields contains a pointer to a tconc into which the TLC is added once the key is moved. A null value means that the TLC was already added to the tconc and that no further action is required.
- next: A pointer to the next TLC in the bucket, or an integer that marks the index of the bucket in the hash table.

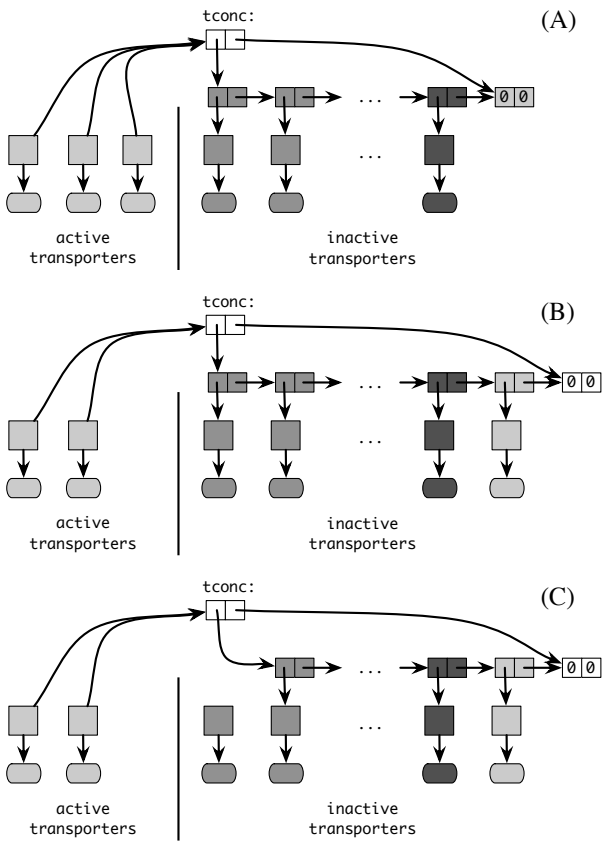

Figure 4. The lifetime of a transport guardian: (A) A transport guardians is constructed to point to a tconc and an object. (B) When the object is moved during garbage collection, its guardians are deactivated and appended to the tconcs they point to. (C) The program can then pop the inactive guardians from the tconc in order to perform operations on the moved objects.

A TLC is allocated in the same manner as a pair or other small heap-allocated object, with the tconc and key fields initialized during allocation, i.e., not mutated after allocation. The address of the key is determined only after the TLC is constructed; otherwise, the address might change if the collector is triggered between the time the address is extracted and the time the TLC is constructed. The key field is never mutated. Thus, it never points to a younger object, allowing the generational collector to ignore the key field when tracking pointers from older to younger generations.

When the garbage collector forwards a TLC that contains (1) a non-null tconc field and (2) a key whose location has changed, it (a) adds the TLC to the tail of the tconc and (b) sets the tconc field to null in order to prevent further tracking. This minor chore is all the collector must do for TLCs beyond what it must do for other types of objects.

6. Analysis

A generation-friendly implementation of eq hash tables must possess two properties:

1. The cost of collecting a generation must be proportional to the size of live objects in that generations.
2. The cost of rehashing in the mutator must not exceed the amount of work performed by the collector to move the rehashed keys.

The first condition is true in our implementation of eq hash tables that utilizes TLCs because the TLC objects are allocated, traversed, and relocated in a manner similar to that of any other data structure in the system with the exception that it performs one additional check to see if the key of the TLC is moved during the collection or not. If the key is moved, a pair is constructed and appended to the tconc. The cost of the extra check and the conditional construction of these pairs is $O(1)$ per collected TLC and therefore adds only (small) constant overhead to the cost of collection.

The second condition is also true in our implementation. First, because we only rehash the TLCs that are popped from the hash table's tconc, we are guaranteed to rehash only the TLCs whose keys have actually moved during collection. Second, the cost of rehashing a single TLC is $O(1)$ because popping it from the tconc, deleting it from its original bucket and adding it to the new bucket are all constant time operations. This is based on the assumption that the buckets of any hash tables are of small constant size (the quality of the hashing function is in question should this not be the case).

Empirical Results

We have implemented generation-friendly eq hash tables based both on conservative transport guardians (Section 4) and on TLCs (Section 5), as well as eq hash tables that rehash on the first failed access after a collection (Section 3.3), in both Chez Scheme (Dybvig 2007) and Ikarus Scheme (Ghuloum 2007). Both Scheme systems employ generational garbage collectors.

We compared the three hash table mechanisms in each Scheme systems to determine how each behaves when hash tables of various sizes are constructed, collected, and both collected and accessed. The results for Chez Scheme are described below; the Ikarus results were similar.

The comparisons were performed on a dual-processor 2.6Ghz AMD Opteron system with 4GB of RAM. Each of the tests were run five times and the minimum times of each run was used for comparison.

Table Construction Times: We first measure the cost of constructing hash tables of various sizes. For each table size and hash-table mechanism, we construct a list of the desired size, then measures the time required to insert every pair of that list into a hash table.

The results of this benchmark are shown in Figure 5. The conservative transport-guardian mechanism (labeled CTG) is slightly slower than the transport-link-cell mechanism (TLC) for all sizes because of the additional storage overhead for the transport guardians. The rehash-on-failed-lookup mechanism (ROFL) is actually faster at smaller table sizes because the pairs used for the hash table buckets are smaller than the transport-link cells. It suffers at larger table sizes due to the cost of rehashing. As the table grows, garbage collections are triggered. On the first insertion of a new element after a collection, all elements of the table are rehashed. While we took care to make sure that the rehashing procedure performs no new allocation (by reusing existing bucket cells), this rehashing is significantly more costly for ROFL because it rehashes all objects rather than just those in younger generations. Although not obvious from the logarithmic table, ROFL takes approximately seven times longer than TLC to construct a table with 10 million elements table.

Garbage Collection: Next, we measure the effect a fully populated hash table has on garbage collection times. To do this, we construct a hash table as before, keep a pointer to the table in order

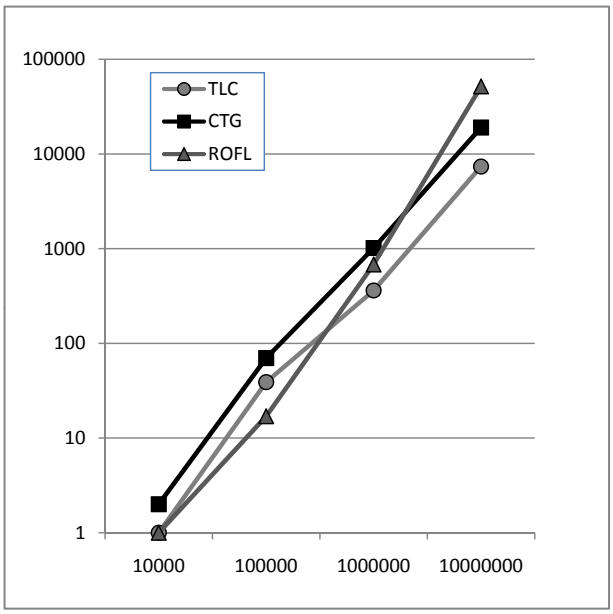

Figure 5. Hash Table Construction. Hash table sizes are shown on the horizontal axis, and table-construction times on the vertical axis. Both axes are in scaled logarithmically.

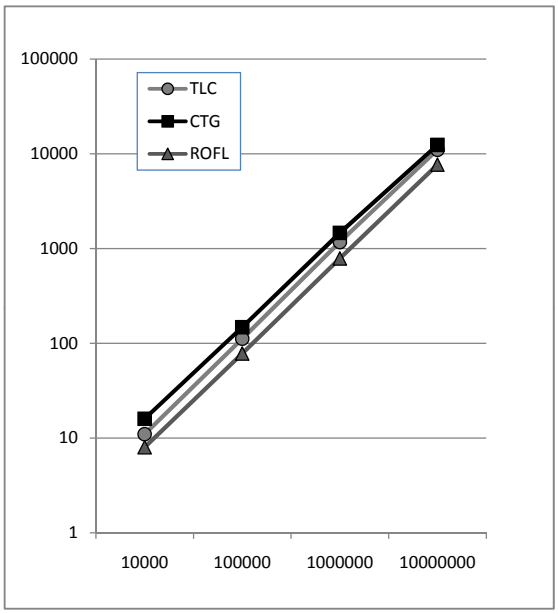

Figure 6. Garbage Collection Times. Hash table sizes are shown on the horizontal axis and the time required to perform one thousand garbage collection runs on the vertical axis. Both axes are scaled logarithmically.

to prevent it from being reclaimed, trigger the garbage collector 1000 times, and measure the amount of time required versus the amount of time required to collect the base system's heap 1000 times.

The results of this benchmark are shown in Figure 6. As we would expect, collection times for each mechanism increase linearly with table size. Collection is faster for the ROFL mechanism than for TLC and faster for TLC than for CTG due to the relative sizes of the internal data structures. None of the mechanisms have to rehash elements since there are no accesses to the hash table.

Rehashing Times Next, we measure the effects of rehashing after collections, independent of table allocation. As in the previous benchmark, we construct a fully populated hash table in advance and trigger the garbage collector 1000 times. The difference here is that we attempt to access a (non-existent) key from the table after every garbage collection run. This forces ROFL to rehash all elements, while TLC rehashes only those that have actually moved, and CTG rehashes those that it's mechanism cannot prove did not move.

The results of this benchmark are shown in Figure 7. TLC again has a small advantage over CTG, and both have a sizable advantage over ROFL. This advantage is even more clearly illustrated by the table below, which gives the percentage increase in run times for this third benchmark (collection followed by a failed access) over the second benchmark (collection only).

	TLC	CTG	ROFL
10,000	82%	81%	7,938%
100,000	59%	79%	8,319%
1,000,000	52%	73%	8,046%
10,000,000	43%	50%	8,803%

While the rehashing costs are close to collection costs for TLC and CTG, rehashing costs for ROFL are much larger.

Our results clearly establish the superiority of the transport-link cell and conservative transport guardian mechanisms over the rehash-on-failed-access mechanism for large hash tables. The advantage of transport-link cells over conservative transport guardians is less extreme, but still significant. The latter runs 1.8–2.8 times slower than the former on the allocation test, 1.1–1.2 times slower on the collection test, and 1.2–1.4 times slower on the collection and access test.

7. Extensions

To illustrate the generality of transport link cells (TLCs), we describe three extensions to eq hash tables that may be implemented using TLCs. Each can be implemented as a user level library, i.e., without modifying the underlying run-time architecture or garbage collector.

7.1 Weak Hash Tables

Weak hash tables are a variant of hash tables in which the keys are weakly held by the collector. Objects that are reachable only via weak pointers can be reclaimed and the weak pointers to these objects are replaced by a special *broken weak pointer* object. Weak hash tables enable reclaiming key/value pairs when the key is no longer reachable through any strong pointers.

Some programming languages or implementations support weak boxes (a box that weakly holds a single value) or weak pairs (a pair whose head is weakly held). In such systems, adding support for weak hash tables can be achieved simply by making the key field of the TLC point to a weak pointer to the actual key. As long as the weak box never resides in an older generation than the key it points to, the generation tracking by the TLC is sufficient to guarantee that no key is moved in memory without its weak box also moving in memory.

Another approach to implementing weak hash tables is by making the TLCs' key field a weak pointer by default, making every hash table in the system a weak hash table. Implementing strong

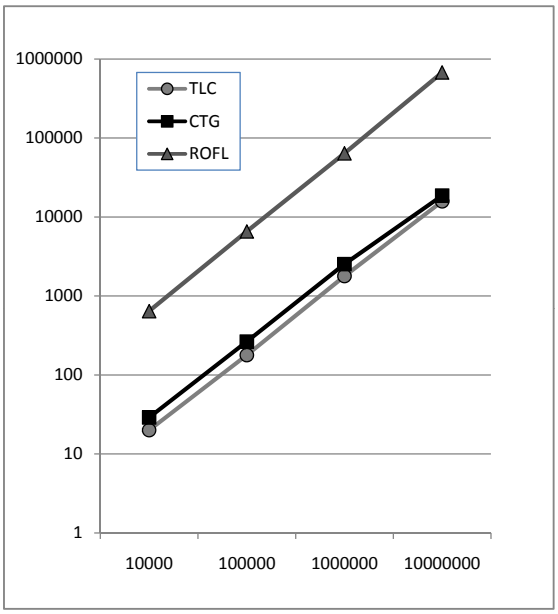

Figure 7. Garbage Collection and Access. Hash table sizes are shown on the horizontal axis and, on the vertical axis, the time required to perform one thousand garbage collection runs while accessing the hash table after every collection. Both axes are scaled logarithmically.

hash tables in terms of weak hash tables is a simple matter of adding a separate strong pointer to the key. This can be achieved by making the value field point to a tuple containing both the key and value pair.

7.2 Eqv Hash Tables

Some programming languages (e.g., Scheme (Sperber et al. 2007)) require an *Eqv Hash Table* in which some object types are hashed by identity while other objects (e.g., numbers, characters, etc.) are hashed by value. Additionally, some implementations (e.g., GNU CLISP (Haible et al.)) define a user-extensible class of objects that contain an internal hash value. Implementing hash tables in which the address of the key may not enter into the value of the hash function is achieved by using TLCs that are initialized by having a null value in their initial tconc field. This effectively disables the unneeded address tracking for TLCs that point to keys that are not hashed by identity.

7.3 Iteration Operations

The implementation of hash tables that we explored supported lookup and delete operations on hash tables. Iteration operations are a natural extension to hash tables and may be required my some programming languages (e.g., Scheme (Sperber et al. 2007), Common LISP (American National Standards Institute and Information Technology Industry Council 1996), etc.).

One may think that since the garbage collector cannot disturb the locations in which the keys reside, one can traverse the hash table directly, bucket by bucket, while performing iteration operations. This approach is incorrect however if any lookup or update operation is performed on the table during iteration. Performing lookup may rehash some elements in the table and consequently may cause some elements to be used zero, one, or more than one time.

In order to overcome this difficulty, we augment our hash tables with an auxiliary doubly-linked list that holds pointers to all the elements stored in the table. The value field of a TLC would point to a data structure that contains pointers to the next and previous TLC in the hash table in addition to the value associated with the key.

8. Conclusion and Future Work

In this paper, we presented a generation-friendly implementation strategy for eq hash tables. Our implementation is efficient since the cost of rehashing objects that moved by the garbage collector is $O(1)$ amortized over the lifetime of the hash table. Our implementation is also extensible since many variants of hash tables can be implemented without the need to modify the garbage collector. The transport-link cells that are used in our implementation provide accurate address tracking, efficiently, and without complicating the garbage collector.

For implementations that already support guardians, hash tables based on conservative transport guardians rather than transport-link cells are an attractive alternative. While the cost differential is significant in our tests, the difference will be less significant in applications for which hash table operations do not account for a large share of overall run time. Transport-link cells are simpler to implement than guardians, however, and should be chosen over guardians if the additional generality of the latter is not required.

Our present implementation of eq hash tables is not thread-safe. We could make it so easily and with reasonable efficiency by performing direct lookups optimistically and locking the table only if rehashing is required. Any locking is undesirable, however, so one goal for future research is to extend our mechanism to allow lock-free threaded execution.

Acknowledgments

Suggestions by the anonymous reviewers led to improvements in the final version of this paper.

References

Ole Agesen. Space and time-efficient hashing of garbage-collected objects. *Theory and Practice of Object Systems*, 5(2):119–124, 1999. URL `citeseer.ist.psu.edu/agesen98space.html`.

American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. 1996.

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989. URL `citeseer.ist.psu.edu/appel88simple.html`.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

Arnold I. Dumey. Indexing for rapid random access memory systems. 5(12):6–9, dec 1956. ISSN 0010-4795, 0887-4549.

R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2007.

R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *PLDI '93*, pages 207–216, June 1993.

R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BiBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University, March 1994.

Abdulaziz Ghuloum, September 2007. Ikarus (optimizing compiler for Scheme), Version 2007-09-05.

Bruno Haible, Michael Stoll, and Sam Steingold. Implementation notes for gnu clisp version 2.40.

Chris Hanson. *MIT/GNU Scheme 7.7.90+*. 2005.

Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1998.

Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. URL `citeseer.ist.psu.edu/lieberman83realtime.html`.

Guido Van Rossum. *Python Language Reference Manual*. 1992.

Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *Functional Programming Languages and Computer Architecture*, pages 106–116, 1993. URL `citeseer.ist.psu.edu/sansom93generational.html`.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised (5.97) report on the algorithmic language Scheme, June 2007. URL `http://www.r6rs.org/versions/r5.97rs.pdf`. With R. Kelsey, W. Clinger, J. Rees, R. B. Findler, and J. Matthews.

Warren Teitelman. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, October 1974.

L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl, 2nd edition*. O'Reilly Associates, 1996.

Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag. URL `citeseer.ist.psu.edu/wilson92uniprocessor.html`.