Implementing User-level Value-weak Hashtables

Aaron W. Hsu

Indiana University

awhsu@indiana.edu

Abstract

Value weak hashtables retain only weak references to the values associated with either a strongly or weakly referenced key. Value-weak hashtables automatically remove entries with invalid weak references, which occur when the collector reclaims a weakly referenced value. Value-weak hashtables provide a convenient way to automatically manage a table of information whose entries do not need to exist once the values associated with the keys no longer need to exist. However, most Scheme implementations do not provide value-weak hashtables by default in their base library. Key-weak hashtables are more common. This paper presents an user-level technique for implementing value-weak hashtables that relies on features commonly found or that could be implemented in most implementations. This makes value-weak hashtables a practical reality for most Scheme users without requiring additional work on the implementation code itself. Programmers may, therefore, utilize value-weak hashtables in code that targets a wider range of implementations.

1. Introduction

Value-weak hashtables behave similarly to regular, strong hashtables except that they retain only weak references to the values associated with the keys of the table. They may or may not behave like key-weak tables by retaining a weak reference to the key. Weak hashtables simplify tables where the values in the table subordinate the keys. These sorts of tables often appear as "secondary" access points to objects. The keys in the table may represent a secondary, non-primary key. In such tables, once an entry's value is no longer referenced or used elsewhere, the entry does not serve any purpose and may be removed. A collector may invalidate a weak value reference during its collection cycle, which enables the hashtable to remove the invalidated entry from the table. This automatic management of the entries frees the programmer from explicitly managing these tedious operations.

Tables that weakly reference their value slots can be used to simplify the implementation of a number of "supporting" structures. A fairly simple example is when implementing Scheme symbols. One could implement `string->symbol` by associating the strings of symbols in the system with their symbol objects in a value-weak hashtable. Here, the strings are the keys, and the symbols the values. If the system can then prove at some later time that the symbol no longer needs to exist in the system and can be reclaimed, then the entry in this table will also be removed automatically by the storage manager.

The Guile Scheme manual [6] suggests that value-weak hashtables could be used to implement parts of a debugger, where line numbers are keys and source expressions are values. When the source expressions are no longer necessary, they can be removed automatically from the debugger without more book keeping. The author has also found such techniques useful when implementing web programs that carry around some state, such as in session tables, cookies, or the like. In this case, tables associating things like IP addresses and user names to a session object may hold that session object weakly, which automates the cleanup of these objects when the session objects are cleaned up or time out.

While very useful, the author is not aware of an easy to port library for value-weak hashtables, nor of any existing documents explaining how to implement this feature without directly programming the internal hashtable code of an implementation. A fairly common interface exists for using hashtables [11], but most Scheme implementations do not provide value-weak hashtables built-in. Without a portable library, programmers cannot readily target many Scheme implementations if their code depends on value-weak hashtables. Perhaps due to this limited availability, many Scheme programmers do not take advantage, nor are they commonly aware, of value-weak hashtables.

This paper presents an user-level implementation of value-weak hashtables that relies on more widely available features. Thus, a programmer can comfortably use value-weak hashtables on implementations that do not provide built-in support. The implementation is efficient enough to make the library practical for widespread use, and simple enough that programmers can port the interface to their Scheme implementation of choice. As an added advantage, the technique admits different strategies to manage the automatic collection of unneeded entries to suit specific application needs.

The following section discusses prerequisite information: strong and weak hashtables, weak references, garbage collection, and any other non-standard features necessary to understand the implementation in Section 3. It also describes how well various implementations support these features. Section 4 discusses variations on the primary implementation technique, and examines some of its limitations. Section 5 concludes.

2. Prerequisites/Background

This section describes the interfaces and background dependencies used to implement the value-weak hashtables described in Section 3. It also discusses how well a range of systems supports these features. Section 3 assumes that key-weak and strong hashtables, weak references, and some form of collection sensitive structures such as guardians or finalizers exist on the targeted Scheme system. Some of these features, such as key-weak hashtables, may be reasonably implemented using other structures, such as weak references, so the actual minimal requirements of the technique and structures described in Section 3 is less than what is listed here. However, to simplify the implementation, Section 3 uses a larger set of libraries and existing interfaces than strictly necessary. That set which extends beyond standard R6RS Scheme [11], as well as a subset of the R6RS hashtable interface is described below.

Hashtables. The R6RS Scheme standard [11] provides a library for hashtables, and this paper uses this interface. The implementation described in Section 3 focuses on EQ? hashtables, but readily adapts to other types of hashtables. R6RS hashtables have four primary operations: reference, assignment, deletion, and update. To create a hashtable, `make-eq-hashtable` can be used in one of two ways:

```
(make-eq-hashtable)
(make-eq-hashtable k)
```

When called with no arguments, `make-eq-hashtable` constructs an empty hashtable. When called with a positive exact integer argument, `make-eq-hashtable` creates an empty hashtable with an initial capacity of roughly k. [1] Reference is provided by `hashtable-ref`.

```
(hashtable-ref ht key default)
```

When passed a hashtable, key, and a default, if an association from the provided key to a value exists in the hashtable `ht`, then that value is returned. Otherwise, `default` is returned.

```
(hashtable-set! ht key val)
```

The assignment procedure will create an association from `key` to `val` in the `ht` hashtable. `hashtable-set!` replaces any existing association to `key`.

```
(hashtable-delete! ht key)
```

The deletion procedure will remove the association for key from the hashtable `ht` if it exists, and will do nothing if it does not exist.

```
(hashtable-update! ht key proc default)
```

The update procedure allows the programmer to update the value associated with a given key using a procedure. After applying `hashtable-update!`, the association of `key` will be the result of applying `proc` to the previous association value if one exists or to the default value if one does not exist. This allows more efficient updating of values in the hashtable when the new value may be based on the old value. [1]

Key-weak hashtables. In most implementations key-weak hashtables are used like regular hashtables, with the exception of a different constructor. This paper assumes that the constructor for key-weak hashtables is `make-weak-eq-hashtable` and that all normal hashtable procedures work on key-weak hashtables. Weak hashtables differ from strong hashtables only in that if the weakly referenced value or key becomes invalid, that entry becomes invalid and may be removed. [2]

Weak pairs [2]. Scheme implementations commonly support weak references through weak pairs or boxes. In a weak pair, one or both of the slots of the pair are weak references. A weak reference does not prevent the garbage collector from reclaiming an object. Specifically, if only weak references to an object exist, then a collector may reclaim that objects allocated space. After reclaiming an object weak references to that object, when evaluated, will return a special value indicating that the value originally references no longer exists. Note that a programmer cannot expect the collector to reclaim an object in any specific amount of time. The collector may never reclaim objects that it could reclaim, in which case the implementation may be wasting space, but it is not violating any of the basic behaviors of weak references.

Another way of thinking about weak references is in terms of reachability. An object is said to be weakly reachable whenever there exist only weak references to the object, and strongly reachable when one or more strong references to the object exist. Weakly reachable objects may be collected, while strongly reachable ones may not.

This paper assumes a `weak-cons` procedure that, given two values, returns a cons pair whose first (car) value is the first argument passed to weak-cons and whose second (cdr) value is the second. The returned cons pair behaves like a regular pair except that the car field of the pair is a weak reference. The cdr field of the returned pair is a strong reference. If the weakly held reference in the cons pair is reclaimed by the collector, and subsequently, `car` is applied to the pair, the call will return `#!bwp` instead of the original value. Weak pairs return true when passed to the `pair?` predicate.

Here is a short interaction which illustrates this behavior:

```
> (define v "value")
> (define p (weak-cons v '()))
> p
("value")
> (define v #f)
> (collect (collect-maximum-generation))
> p
(#!bwp)
```

Guardians and Finalizers. Section 3 requires some way to run code after certain objects have been proven unreachable by the collector, implementations usually provide this functionality through guardians or finalizers. Guardians are parameter-like structures that allow you to preserve an object for further action after the collector determines that it is safe to collect that object. One creates guardians using `make-guardian`, which returns a guardian procedure. Calling a guardian procedure on an object registers that object with the guardian. Calling the guardian procedure with no arguments will return one of the registered objects marked as safe for collection; otherwise, if no objects registered with the guardian are safe to collect, the guardian procedure returns false, instead. The guardian removes objects from its register whenever it returns them. Repeated calls to the guardian return different registered, reclaimable objects until no such objects remain. Thus, guardians allow a program to save objects from the collector for some further processing before finally releasing them for real. Often, one registers objects

that require additional clean up, such as memory allocated by a foreign program, to clean up automatically instead of forcing manual clean up calls to pollute the primary code base.

To give an example, suppose that we have the following interaction:

```
> (define g (make-guardian))
> (define v "value")
> (g v)
> (g)
#f
> (define v #f)
> (g)
#f
> (collect (collect-maximum-generation))
> (g)
"value"
```

Note how the guardian preserves a reference to the object in question, without interrupting the behavior of the collector. That is, while with weak references, after the collector has reclaimed storage, the weak reference becomes invalid, with a guardian, if the collector finds a reclaimable object (one that has no references, or that has only weak references to it) that is also registered with the guardian, it will not reclaim the storage for the object until the guardian returns and subsequently removes that object from its set of objects. The interaction between weak references, guardians and the storage manager can be understood a little easier by examining the following example:

```
> (define g (make-guardian))
> (define v "value")
> (define p (weak-cons v '()))
> (g v)
```

At this point the heap allocated string object "value" is referenced strongly by `v`, weakly by `p`, and is registered with the guardian `g`. In this state, the strong reference prevents the storage manager from reclaiming the storage for "value" since it is still needed. We can break that reference though, and then the collector would normally be free to collect the value. The weak reference in `p` will not hinder the garbage collector from reclaiming the storage. However, since "value" is registered with the guardian, when the collector tries to reclaim the storage, it will encounter the guardian's hold on the object, and instead of reclaiming the storage, the object now moves into the set of the guardian's reclaimable objects, to be returned at some point when the guardian is called with zero arguments.

```
> (define v #f)
> (g)
#f
> (collect (collect-maximum-generation))
> p
("value")
> (g)
"value"
```

Once the guardian returns the object, it is no longer protected from the collector unless a new strong reference is created or it is re-registered with the guardian. At this point, when the collector runs again, it may clear out the object, and the weak reference in `p` will be invalid.

```
> (collect (collect-maximum-generation))
> p
(#!bwp)
```

This paper makes use of guardians [3], but a suitably expressive alternative, such as Gambit Scheme's Wills [4], also suffices. Guardians make it easy to switch between different cleaning techniques discussed in Section 4. Whether using guardians, finalizers, wills, or other structures which accomplish the same purpose, the primary feature that makes this implementation possible is the ability to prevent the storage manager from collecting an object while guaranteeing that the object is not referenced elsewhere.

Given something like Wills or suitably expressive finalizers, an implementation of guardians is possible, and vice versa. Gambit's implementation of Wills defines a constructor `make-will` that takes a "testator" and an "action." The will then maintains a weak reference to the testator and a strong reference to the unary action procedure. When the testator becomes reclaimable (no strong references to the testator exist), the system will call the action procedure associated with the testator with the testator as its argument.

One could implement wills in terms of guardians using something similar to the following code. In this code, we assume that we have the ability to hook into the collector to run arbitrary code during collection cycles.

```
(define g (make-guardian))
(define action-list '())
(define (make-will testator action)
  (let ([p (weak-cons testator action)])
    (set! action-list (cons p action-list))
    (g testator)
    p))
(let ([handler (collect-request-handler)])
  (collect-request-handler
    (lambda ()
      (set! action-list
        (remp
          (lambda (x) (bwp-object? (car x)))
          action-list))
      (do ([res (g) (g)]) [(not res)]
        (for-each
          (lambda (x)
            (when (eq? res (car x))
              ((cdr x) res)))
          action-list))
      (handler))))
```

Similarly, using wills, one can implement guardians.

```
(define make-guardian
  (let ([claimable '()])
    (case-lambda
      [()
       (when (pair? claimable)
         (let ([res (car claimable)])
           (set! claimable (cdr claimable))
           res))]
      [(val)
       (make-will val
         (lambda (x)
           (set! claimable
             (cons x claimable))))])))
```

Of course, the above implementations are not complete, but illustrate the main concepts. Additional work must be done, for example, to ensure their correct behavior in threaded

applications. This is especially true in cases where the underlying implementation is either finalizers or wills, because there is less control as to when the action procedures will be executed. These sorts of tradeoffs are discussed in more detail in Section 4.

Implementation Support. The following technique for implementing value-weak hashtables requires the above features, whether they are provided as built-ins or as libraries. Some implementations have built-in value-weak hashtables. The author is aware of Gambit [9] and Guile [10] which have documented value-weak hashtable interfaces. Chez Scheme [7] and PLT Racket [5] both have support for the above features built-in. Chicken [8] appears to have some support for the above features, but the author was unable to verify the functioning of key-weak hashtables. Others Schemes have other, varying degrees of support, often with documentation that makes it difficult to determine whether or not a given feature is in fact supported.

3. Implementation

Essentially, a value-weak hashtable removes entries from the table whenever it can based on the reachability of the values of those entries. Put another way, the collection of a value associated with some key in the table should trigger the deletion of that entry. The following demonstrates a simple interaction with a value-weak hashtable.

```
> (define ht (make-value-weak-eq-hashtable))
> (define v1 "First value")
> (define v2 "Second value")
> (value-weak-hashtable-set! ht 'first v1)
> (value-weak-hashtable-set! ht 'second v2)
> (value-weak-hashtable-set! ht 'third v1)
```

At this point the hashtable `ht` now contains three entries, where the first and third entries both contain `v1` as their value.

```
> (value-weak-hashtable-ref ht 'first #f)
"First value"
> (value-weak-hashtable-ref ht 'second #f)
"Second value"
> (value-weak-hashtable-ref ht 'third #f)
"First value"
>
(eq? (value-weak-hashtable-ref ht 'first #f)
     (value-weak-hashtable-ref ht 'third #f))
#t
```

At this point, there is no apparent difference in behavior from a strong hashtable. However, suppose that we eliminate all of the strong references to the string referenced by `v1`, and then perform a collection.

```
> (define v1 #f)
> (collect (collect-maximum-generation))
```

By now, there is no reason for the table to retain the entries which contain the string "First value" as their values. Therefore, the table may clean up these entries and remove them.

```
> (value-weak-hashtable-ref ht 'first #f)
#f
> (value-weak-hashtable-ref ht 'third #f)
#f
```

The reader may already be considering how to implement such a structure. Very likely, the idea of wrapping values in weak pairs has already presented itself. Indeed, the most obvious approach to implementing value-weak hashtables is to take a normal hashtable and use weak-pairs for all the values. We might then define a `hashtable-set!` procedure for value-weak tables like so:

```
(define (value-weak-hashtable-set! ht key val)
  (hashtable-set! ht key (weak-cons val '())))
```

The task of referencing this table would then be simply implemented using something like the following:

```
(define (value-weak-hashtable-ref ht key def)
  (let ([res (hashtable-ref ht key #f)])
    (if (not res)
        def
        (let ([val (car res)])
          (if (bwp-object? val)
              def
              val)))))
```

Indeed, this does work, up to a point. This code does enforce the most obvious feature of value-weak hashtable, that their value slots weakly reference their values. However, when the storage manager does collect the value, nothing cleans up the table. An entry with an invalid reference in it should not stick around as doing so causes a space leak in the program. Without some additional work, the only hope to an user of such a system is to walk through the entries of the hashtable and manually delete all of the invalid entries every so often. This solution undermines the running time of hashtables if we are forced to do this operation with any regularity and makes value-weak hashtables rather useless. Instead, we need a way to selectively delete entries automatically without having to scan the entire table. For this, we consider the use of cleaner thunks.

Using closures, cleaner thunks that delete a specific entry in the table if triggered at the right time accomplish the task of removing the entries. The implementation must call such a thunk when the collector reclaims the value associated with it. Using a key-weak hashtable, we can associate the value (now used as a key) with its cleaner, and if each cleaner has only a single reference to it through this table, then the collection of the value will trigger the collection of the appropriate cleaner. We can intercept this collection by registering the cleaners with a guardian. This guardian exposes the cleaners that are safe to run because the values associated with them have been collected.

Any newly introduced strong references to the entry value that persist after the entry is added will undermine the weakness of the value cell in the table, so the code must ensure that only weak references to the value persist after adding the entry. To achieve this, the value is placed in a weak pair, with the key in the strong cell. This pair is inserted into the primary hashtable instead of the value directly. Only the weak key reference in the cleaner table remains in addition to the pair reference, so the invariant is maintained.

Figure 1 illustrates an actual example of this idea. Suppose that we have already run the interaction earlier in the section to fill the hashtable, but we have not yet invalidated any of the references. Our value-weak table structure has two internal tables, a `db` table, and a `trail` table. It also has a guardian associated with it. In the `db` table, there are three entries, one for each of the entries in the value-weak

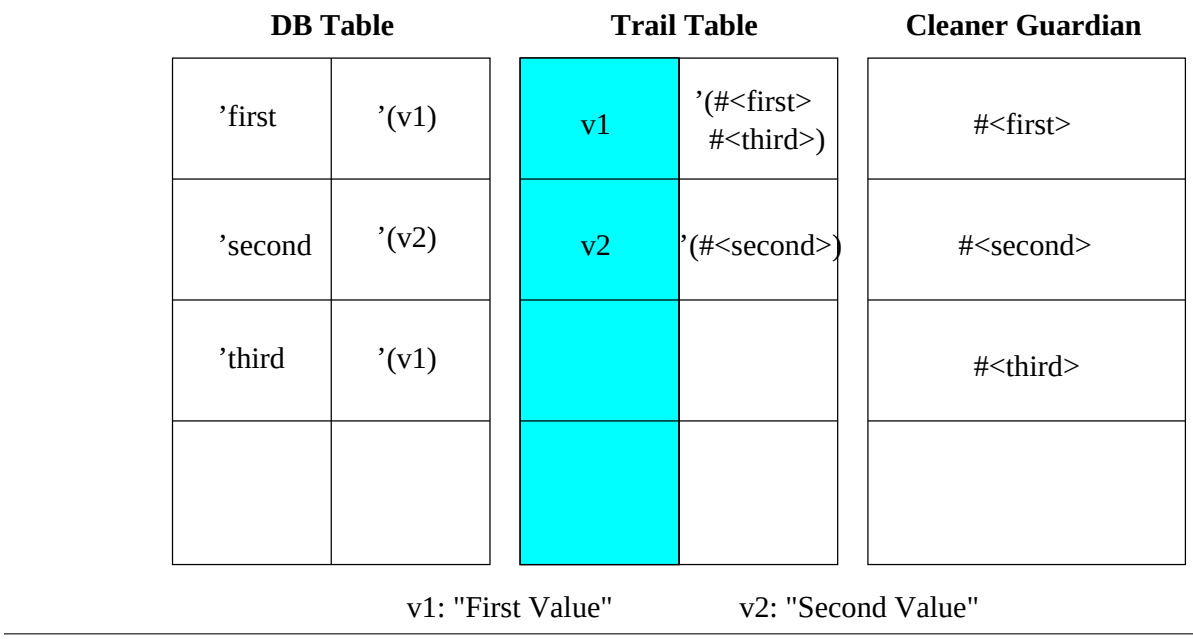


Figure 1. Basic Value-weak hashtable structure

table itself. The values of each of these entries is wrapped in a weak pair. In fact, this table represents the same naive technique that we started with above. This forms the core table for doing our operations. The `trail` table maintains an association and entry for each of the unique values that are indexed by the `db` table. In this case, there are two entries: our two strings. Each of these is associated with a list of cleaners. In the case of the second value string, only one key has been associated with that value in the main table, so only one element exists in the `trail` table. However, two entries in our table are associated with our first value string, so we have two cleaners in our list for that value in the `trail` table. In total, at this point, three cleaners were created. Each of these cleaners is registered with the guardian, and only one reference to the cleaners exists, which is the reference in the `trail` table.

At this point, if we invalidate the first value string, as we did in the above interaction, then at some point, the collector will see that there exists only weak references to that value, and it will reclaim that structure. This will in turn trigger the trail database to clean up its entry for that value, which has not been reclaimed. When this occurs, the references for the two cleaners associated with that value lose their references as well. When the collector sees this, the guardian will prevent them from being reclaimed and put them in the set of values that are safe to collect. We can then call this guardian at any point and, seeing the cleaners, know that they are safe to run. When we execute them they will delete their entries in the main hashtable if they were not already replaced, and finish the original task assigned to them.

Thus, the guardian of each table gives us a sort of stream like interface to the set of entries that we can remove. It tells us exactly which entries are safe to remove by returning only those cleaners that can be safely invoked. This works by carefully setting up our strong and weak references.

When implementing the user-level library all of these structures should be encapsulated inside a record. An R6RS version of this could look like this:

```
(define-record-type value-weak-eq-hashtable
  (fields db trail guardian)
  (protocol
    (lambda (n)
      (case-lambda
        [() (n (make-eq-hashtable)
               (make-weak-eq-hashtable)
               (make-guardian))]
        [(k) (n (make-eq-hashtable k)
                (make-weak-eq-hashtable k)
                (make-guardian))]))))
```

This defines a `make-value-weak-hashtable` procedure that creates a value-weak EQ? based hashtable.

While this structure enables collector based cleaning, an explicit procedure must still call each cleaner thunk. This procedure need only loop through the non-false values of the guardian:

```
(define (clean-value-weak-hashtable! guardian)
  (let loop ([cleaner (guardian)])
    (when cleaner
      (cleaner)
      (loop (guardian)))))
```

Each table operation calls clean-value-weak-hashtable! at the start of of the procedure, ensuring that the hashtable has removed any entries it can. Doing the cleaning here, on entry to every operation, and not at random and unpredictable collection times makes the implementation of table operations simpler because the code does not have to lock the structures before using them. Cleaning on entry, in essence, linearizes or serializes the operations on the tables and mostly eliminates the need to worry about thread interactions.

In order to make accessing the fields of the record easier, the following syntax serves as a binding form for the record fields.

```
(define-syntax let-fields
  (syntax-rules ()
    [(_ table (db trail guard) e1 e2 ...)
     (let
       ([db
          (value-weak-eq-hashtable-db table)]
        [trail
          (value-weak-eq-hashtable-trail
            table)]
        [guard
          (value-weak-eq-hashtable-guardian
            table)])
       e1 e2 ...)]))
```

Revisiting the previous naive hashtable code, how does it change? The reference operation is the easiest to implement. A simple check of the value of the pair associated with the key assures the intended behavior.

```
(define (value-weak-hashtable-ref
          table key default)
  (let-fields table (db trail guard)
    (clean-value-weak-hashtable! guard)
    (let ([res (hashtable-ref db key #f)])
      (if res
          (let ([val (car res)])
            (if (bwp-object? val)
                default
                val))
          default))))
```

So, in fact, the reference operation changes very little. Assignment needs more reworking.

```
(define (value-weak-hashtable-set!
          table key value)
  (let-fields table (db trail guard)
    (let ([data (weak-cons value '())])
      (clean-value-weak-hashtable! guard)
      (let ([cleaner
              (make-cleaner db data key)])
        (hashtable-set! db key data)
        (add-cleaner! trail value cleaner)
        (guard cleaner)))))
```

Here, we need to make sure that we create the cleaner and add it to the trail as well as do the insertion. We also register the cleaner with the guardian. The above implementation is simplified by our use of clean on entry, so that we don't have to disable the collector while editing our structures. This particular element is discussed in later sections.

Creating cleaners does require a little more care than just blindly deleting the entry. Each cleaner cannot simply delete the entry with the right key from the table because an operation may have altered the value associated with the key. The cleaner must check this before it deletes the entry.

```
(define (make-cleaner db data key)
  (case-lambda
    [()
     (let ([res (hashtable-ref db key #f)])
       (when (and res (eq? res data))
         (hashtable-delete! db key)))]
    [(maybe-key) (eq? key maybe-key)]))
```

The above procedure provides two functionalities. The first, nullary version deletes the key from the table if it has not been changed since the cleaner was created. The second, unary version reveals the key to which the cleaner belongs. This second functionality makes it easier to write `drop-cleaner!` below.

A table may associate multiple keys to the same value, so the trail table must associate values to a set of cleaners. The following simple procedure encapsulates the process of adding a cleaner to the trail.

```
(define (add-cleaner! trail val cleaner)
  (hashtable-update! trail val
    (lambda (rest) (cons cleaner rest))
    '()))
```

It requires more work to delete a cleaner from the trail. Since we could have more than one cleaner for every value, we must walk through the cleaners, asking each in turn whether it is associated with the key we need to delete. In other words, we uniquely identify each cleaner by both the key and the value to which it is associated.

```
(define (drop-cleaner! db trail key)
  (let ([db-res (hashtable-ref db key #f)])
    (when db-res
      (let ([val (car db-res)])
        (unless (bwp-object? val)
          (let ([trail-res
                  (hashtable-ref
                    trail val '())])
            (if (or (null? trail-res)
                    (null? (cdr trail-res)))
                (hashtable-delete! trail val)
                (hashtable-update! trail val
                  (lambda (orig)
                    (remp (lambda (cleaner)
                            (cleaner key))
                          orig))
                  '()))))))))
```

The `drop-cleaner!` procedure handles most of the work for deletion, but technically, an implementation could omit this entirely, further simplifying the delete operation. One could avoid explicitly dropping the cleaners, assuming, instead, that at some point the collector will reclaim the values and their cleaners. Deletion costs less when ignoring the cleaners, but is subject to other limitations. Section 4 discusses these limitations and various tradeoffs in more detail. As drop-cleaner! handles almost all the work, the code can just call things in the appropriate order to implement deletion.

```
(define (value-weak-hashtable-delete!
          table key)
  (let-fields table (db trail guard)
    (clean-value-weak-hashtable! guard)
    (drop-cleaner! db trail key)
    (hashtable-delete! db key)))
```

Care should be taken when implementing the update operation. While straightforward, the update operation requires working with the weak pairs in the table. Thus far, we have avoided threading interactions, but the weak pairs do not benefit from the clean on entry strategy that allowed us to do this. The collector could suddenly invalidate the weakly referenced values. Since update requires us to check for this and then call the provided updater procedure on the value in

the pair, the code should extract the value in the pair only once, to prevent a possible timing inconsistency if the collector should happen to invalidate a reference between the check for validity and the actual point where the value is used.

```
(define (value-weak-hashtable-update!
          table key proc default)
  (let-fields table (db trail guard)
    (define (updater val)
      (let* ([res
              (if val
                  (let ([wp (car val)])
                    (if (bwp-object? wp)
                        default
                        wp))
                  default)]
             [new (proc res)]
             [data (weak-cons new '())]
             [cleaner
               (make-cleaner db data key)])
        (guard cleaner)
        (add-cleaner! trail new cleaner)
        data))
    (clean-value-weak-hashtable! guard)
    (hashtable-update! db key updater #f)))
```

Notice also that we choose to create an explicit, strong reference to the new result obtained from calling `proc` on the old value. This avoids the situation where the new result is not actually strongly referenced anywhere else, possibly causing the weak pair to have an invalid reference by the time we add a cleaner to it later, if the collector somehow runs between the point where we define the weak pointer and the time when we extract the value. While it does not appear particularly useful to return an immediately reclaimable object, the code should handle that possibility.

4. Discussion

The above section illustrates the core technique for implementing value-weak hashtables, but in various places, an implementor could choose a different strategy, based on the various tradeoffs.

An implementor should examine and weigh the various methods for collecting the unneeded entries from the table. In Chez Scheme, for instance, the code could use the collect-request-handler parameter to run the cleaners during collection cycles rather than at the start of every table operation [2]. In doing so, collections may occur at any time, which requires some synchronization of the table state to prevent cleaners from running while a critical section of a hashtable operation runs.

Our implementation does have some limitations. Obviously, normal hashtable procedures will not operate on value-weak hashtables because our record declaration creates a disjoint type for value-weak tables. Less obvious, we do not guarantee that a hashtable will remove any entries in a timely manner. We only remove entries when entering a value-weak operation on that table, if a table is not used, potentially unneeded entries may persist. This will not prevent the collection of the values stored in the table, though cleaners tied to those values will remain in the associated guardian.

Of course, non-deterministically collecting the table entries has its own cost. While entries can be collected more routinely in situations where the clean-on-entry approach may not run any cleaners at all, it forces synchronization costs every time the code operates on the table. Fortunately, both these techniques may co-exist in a single implementation, so the user can enable or disable these various approaches individually for each table.

The above code only demonstrates the implementation of an `eq?` based (pointer based) hashtable. A value-weak hashtable taking an arbitrary hash procedure and equivalence predicate can also be constructed, but the various `eq?` comparisons on the keys should be replaced throughout the code.

One can tweak the code to allow for key and value-weak references in a single hashtable, but this requires more work. To do so, the strong references to the above key would have to be removed, and perhaps replaced with weak references. This creates more work in the code to handle the potentially invalid key references, but otherwise the technique remains the same.

The user of value-weak tables should be aware of their behavior with immediate values. In the above implementation, immediate values can be associated with keys. Such entries will never be automatically reclaimed, because the collector does not reclaim immediate values. These entries can only be removed manually.

Someone implementing this library for their system and domain should consider whether or not to delete cleaners from the trail when the entry is removed from the table. Not doing so improves the overhead of the deletion operation, but it also means that the cleaners will not disappear until the values with which they are associated are first collected. In cases such as immediate values, where these values will never be collected, the cleaners will remain indefinitely. This creates a leak in the general case, but may be worth it to those implementing the library for some special application.

The discussion of guardians and finalizer type mechanisms deserves some attention. The above implementation would work in a native implementation of finalizers or wills, but synchronization issues must be handled. Guardians have a benefit in this particular task because they are deterministic, which allows the implementor more control over when code attached to the guardian or reliant on its behavior runs. Finalizers are non-deterministic in that they may occur at any point the system determines they should run, and there is often little the programmer can do to control this. The discussion of synchronization above applies to finalizers and finalizer-like solutions in particular.

An implementation based on guardians can choose and switch between deterministic and non-deterministic cleaners easily. This can also be done with finalizers, but requires that the programmer implement guardian-like functionality to do so. Care should be taken to ensure the safety of this abstraction.

A completely different implementation strategy exists for implementations that provide a generational garbage collector. The implementation can avoid using a trail at all. This will result in all the cleaners becoming reclaimable in the guardian at some point. If the cleaners are re-registered with the guardian if their entries still exist, then as the program runs, those entries which stick around longer will have their cleaners in progressively older generations. This will result in the cleaners being run less and less often until they reach the oldest generation. While this doesn't provide the same type of guarantees as the trail based implementation does, it has the benefit of greatly reducing the overhead for

certain operations. On the other hand, it also creates a level of overhead itself since the cleaners are essentially being cycled through by the collector, even if they are doing so with different generations. The author has not thoroughly studied the effects and differences of this method against the trail based method, but in the benchmarks below, switching to this method almost halves the overhead of insertion and update, while increasing the timing for reference, which has very little overhead in the trail based implementation. Using this generation trick also forces this guardian/cleaner overhead on other procedures and the rest of the system, even if the value-weak hashtables are rarely, if ever, used.

5. Performance

The above implementation of value-weak hashtables has at least two important design choices which directly affect its performance. Firstly, because we rely on two hashtables to implement another, we incur more than double the cost of storage. We also perform more book keeping than normally occurs in native hashtable operations. However, we do not expect to encounter any worse asymptotic performance.

The appendix shows the timings for a fairly rough and informal test of four hashtable operations: insertion, deletion, update, and reference. As expected, there is a fairly high constant factor associated with operations which must handle book keeping between the two internal tables. Reference suffers the least because it has the lowest extra burden for tracking the entries. Both update and set operations have the highest, which seems consistent with their more complex definitions.

The benchmarks for each consisted of constructing a table and performing N number of operations of that given type, and taking the overall time. The smallest tested N was 1,000, and the largest was 3,000,000, since after this, the test machine ran out of RAM. The tests compare the performance of the native key-weak and strong EQV? based tables in Chez Scheme against the above implementation of value-weak tables.

The author has not attempted to tune the performance of these operations.

6. Conclusion

This paper illustrates how to build a value-weak hashtable abstraction without access to the internals of the Scheme system. This makes it easier to port to other Scheme implementations, provided that a few more basic and more commonly implemented structures are also available. The above implementation strategy is flexible enough to support a number of different design preferences, such as differing collection strategies and different cleaning guarantees. While the technique does not easily allow for the direct integration of the abstraction into existing hashtable operations, it is practical. It provides an avenue for programmers to implement value-weak hashtables easily and to encourage their use.

Acknowledgements

The author would like to especially thank the anonymous reviewers for their comments, and the PL Wonks group at Indiana University for their useful and detailed feedback. The author would also like to acknowledge Taylor Campbell for providing initial motivation.

References

[1] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[2] R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

[3] R. Kent Dybvig, Carl Bruggerman, and David Eby. Guardians in a generation based garbage collector. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 207–216, June 1993.

[4] Marc Feeley. Gambit-c manual chapter 11.2.1. http://www.iro.umontreal.ca/ gambit/doc/gambit-c.html#Wills, July 2010.

[5] Matthew Flatt and PLT. Reference: Racket. http://docs.racket-lang.org/reference/index.html, June 2010.

[6] Free Software Foundation. *Guile Reference*, July 2010. http://www.gnu.org/software/guile/docs/master/ guile.html/index.html.

[7] Chez Scheme. http://www.scheme.com, July 2010.

[8] Chicken Scheme. http://www.call-with-current-continuation.org, July 2010.

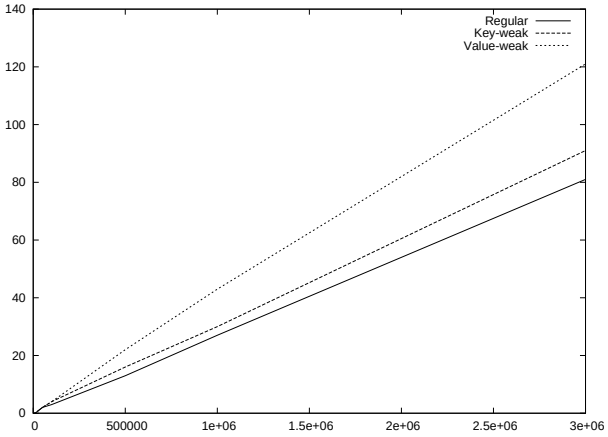
[9] Gambit Scheme. http://dynamo.iro.umontreal.ca/ gambit/wiki/index.php/Main_Page, July 2010.

[10] Guile Scheme. http://www.gnu.org/software/guile/, July 2010.

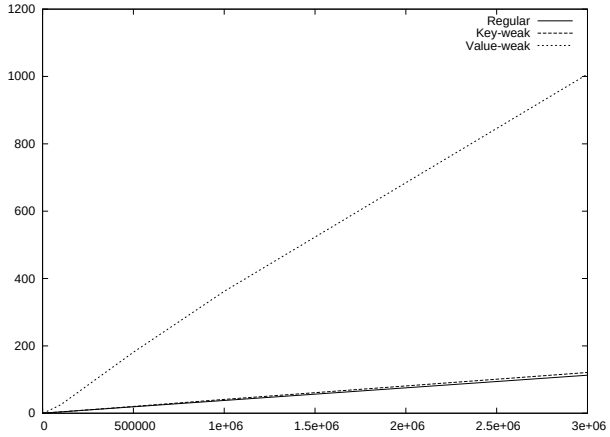
[11] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, pages 1–301, August 2009.

Appendix

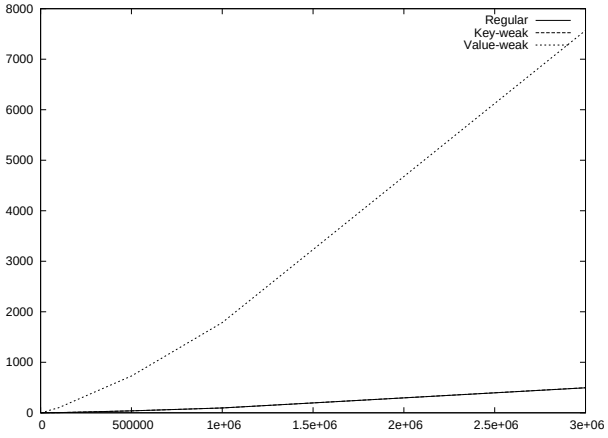
All timings are in miliseconds.



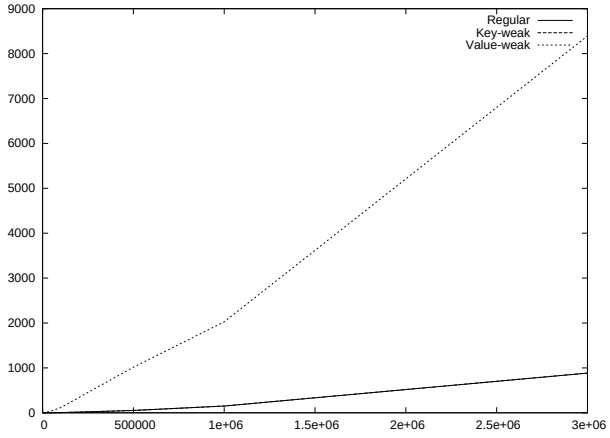
Reference



Deletion



Insertion



Update