Stretching the storage manager: weak pointers and stable names in Haskell

Simon Peyton Jones[1], Simon Marlow[2], and Conal Elliott[3]

[1] Microsoft Research Ltd., Cambridge, simonpj@microsoft.com
[2] Microsoft Research Ltd., Cambridge, simonmar@microsoft.com
[3] Microsoft Research Ltd., Redmond, conal@microsoft.com

Abstract. Every now and then, a user of the Glasgow Haskell Compiler asks for a feature that requires specialised support from the storage manager. Memo functions, pointer equality, external pointers, finalizers, and weak pointers, are all examples.
We take memo functions as our exemplar because they turn out to be the trickiest to support. We present no fewer than four distinct mechanisms that are needed to support memo tables, and that (in various combinations) satisfy a variety of other needs.
The resulting set of primitives is undoubtedly powerful and useful. Whether they are *too* powerful is not yet clear. While the focus of our discussion is on Haskell, there is nothing Haskell-specific about most of the primitives, which could readily be used in other settings.

1 Introduction

"Given an arbitrary function `f`*, construct a memoised version of* `f`*; that is, construct a new function with the property that it returns exactly the same results as* `f`*, but if it is applied a second time to a particular argument it returns the result it computed the first time, rather than recomputing it."*

Surely this task should be simple in a functional language! After all, there are no side effects to muddy the waters. However, it is well known that this simple problem raises a whole raft of tricky questions. A memo table inherently involves a sort of "benign side effect", since the memo table is changed as a result of an application of the function; how should we accommodate this side effect in a purely-functional language? What does it mean for an argument to be "the same" as a previously encountered one? Does a memo function have to be strict? Efficient memo tables require at least ordering, and preferably hashing; how should this be implemented for arbitrary argument types? Does the memo function retain all past (argument,result) pairs, or can it be purged? Can the entire memo table ever be recovered by the garbage collector? And so on.

One "solution" is to build in memo functions as a primitive of the language implementation, with special magic in the garbage collector and elsewhere to deal with these questions. But this is unsatisfactory, because a "one size fits all" solution is unlikely to satisfy all customers. It would be better to provide a simpler set of primitives that together allowed a programmer to write a variety of memo-table implementations. The purpose of this paper is to propose just such a set of primitives. Our design proposes four related mechanisms:

1. The `unsafePerformIO` primitive allows the programmer to execute benign side effects (Section 3).
2. Typed *stable names* allow a stable (i.e. invariant under garbage collection) "key" to be derived from an arbitrary value (Section 4).
3. Typed *weak pointers* allow the programmer to avoid an otherwise-lethal space leak (Section 5).
4. Finalization allows the programmer to express a variety of policies for purging the memo table of unused values (Section 6).

Each of these four primitives also has independent uses of its own. The latter three have in common that they require integrated support from the garbage collector.

Compared to earlier work, our new contributions are these:

– We offer the first complete, integrated design that supports user-programmable memo tables in Haskell, a non-strict, purely-functional language.
– So far as we know, our stable-name proposal is new. The same underlying run-time system mechanism also supports both inter-heap references in GPH, our distributed implementation of Haskell [11], and Haskell references held by external agents such as GUI widgets or COM objects.
– Weak pointers, in contrast, have been in use since at least the early 80's. Our design has some neat wrinkles, and solves the little-known key-in-value problem. Though developed independently, our solution is very close to that of [4], but we believe that our characterisation of the (tricky) semantics of weak pointers is easier for a programmer to understand.

Everything we describe is implemented in the Glasgow Haskell Compiler (GHC). No single aspect of the design is startling, yet it has taken us a surprisingly long time to achieve, due to a number of interacting subtleties. One contribution of the paper is to summarise the folklore in this tricky area, though we believe that we have also developed it significantly.

2 Memo functions

We use memo functions as our running example because they highlight most of the awkward issues. The basic idea is very simple: if a function is applied a second time to a given argument, return the result computed the first time instead of recomputing it.

Memoisation is particularly attractive for a purely-functional language, because there are guaranteed to be no side effects that might change the result even if the argument is the same as before [7]. Hughes [5] studied the implications of memoisation in a lazy language. More recently, Cook and Launchbury [1] describe *disposable* memo functions, a variant of Hughes' lazy memo functions, and give an operational semantics that clarifies their behaviour. Hash-consing is a specialised kind of memo table application that remembers previously-built heap objects in case an identical object is required again. All these papers give applications that explain the usefulness of memo functions.

2.1 A design for memo functions

Following [1], the most elegant way to construct a memo function is by providing a higher-order function `memo`:

```
memo :: (a -> b) -> (a -> b)
```

That is, `memo` takes a function with arbitrary range and domain, and returns a memoised version of the function. The memoised function is a new value, in contrast to other approaches where memoisation is achieved by some kind of pragma or side effect.

The standard toy example is the Fibonacci function, whose complexity turns from exponential to linear if the function is memoised in this way:

```
fib :: Int -> Int          ufib :: Int -> Int
fib = memo ufib            ufib 0 = 1
                           ufib 1 = 1
                           ufib n = fib (n-1) + fib (n-2)
```

(Notice that the recursive call is made to `fib`, the memoised version of `ufib`).

In this example we defined a single memoised fibonacci function, but `memo` does not require that. Indeed, there may be many memoised versions of the same function in use at any time. Each such call to `memo` creates its own memo table, which should be garbage collected when the memoised function is discarded. For example, here is a version of `map` that might be used when the argument list is expected to have many occurrences of the same value:

```
memo_map f xs = map (memo f) xs
```

Here, a single memoised version of `f` is applied to each element of the list `xs`. A function of several arguments can easily be memoised on a particular argument. For example, here is how to memoise a three-argument function, `f`, on its second argument[1]:

```
memo_2_3 :: (a -> b -> c -> d) -> (a -> b -> c -> d)
memo_2_3 f = \ a b c -> mf b a c
           where
             mf = memo (\b a c -> f a b c)
```

Similarly, a function can easily be memoised on several arguments. The first use of `memo` maps the first argument to a function that is itself memoised:

```
memo2 :: (a -> b -> c) -> (a -> b -> c)
memo2 f = memo (\ a -> memo (f a))
```

[1] "\"is Haskell's notation for lambda

2.2 Variations on the theme

The first question that springs to mind is: how does `memo` decide whether a new argument is "the same" as one it has seen before? One could imagine at least three different variants of `memo`:

- Perform no evaluation on the argument; simply use pointer equality. Recall that Haskell is a lazy language and we would prefer it if `memo` did not change the strictness of the function it is memoising, and using pointer equality certainly has this property. On the other hand, pointer equality will not detect that the arguments `(1+2)` and `(4-1)` are the same, because they are thunks held at different addresses.
- Evaluate the argument to weak-head normal form, and then use pointer equality. This approach will produce more "hits", because two thunks that evaluate to the same value will match. It would also make the memoised version of the function strict. Even then we might worry that two thunks that both evaluate to `3`, say, might nevertheless evaluate to values held at distinct addresses.
- Perform a proper equality check on the argument. In this case, the type of `memo` must change, since it is no longer fully polymorphic[2]:

```
memoEq :: Eq a => (a -> b) -> a -> b
```

The main point is that there is more than one possible semantics for `memo`, a powerful argument for allowing the programmer to define it rather than building it in.

3 Benign side effects

Although a purely-functional language has no visible side effects, the implementation overwrites heap objects all the time! When the value of a thunk (e.g. an unevaluated function argument) is demanded, the thunk is overwritten with the newly-computed value, so that any subsequent demands need not recompute it. Memo functions require a similar sort of "benign side effect", but if we are to program `memo` in Haskell then we must expose this ability to the programmer.

Side effects are expressed in Haskell using the `IO` monad [10]. In particular, the `IO` monad provides mutable cells with the following primitives:

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

[2] The notation `Eq a` means that the type `a` is a member of the `Eq` type class, i.e. it supports equality.

A value of type `IORef t` is a reference to a mutable cell holding a value of type `t`. The primitives to allocate, read, and write the cell are all in the `IO` monad.

The idea is to use an `IORef` to hold the memo table. But `memo` is polymorphic: it says nothing about `IO`. We need a way to express side effects, and yet claim that the overall effect is pure. So we provide one new primitive:

```
unsafePerformIO :: IO a -> a
```

This function takes an I/O performing computation that delivers a value of type `a`, and turns it into a value of type `a`. The I/O will be performed when (and if) the value is demanded. There is no guarantee when that will be, or how it will interleave with other I/O computations; that is why the function is unsafe. However "unsafe" is not the same as "wrong". It simply means that the programmer, not the compiler, must undertake the proof obligation that the program's semantics is unaffected by the moment at which all these side effects take place.

We are finally ready to give one possible implementation of `memoEq`; we choose this variant because it allows us to evade the issues of pointer equality for the present.

```
memoEq :: Eq a => (a -> b) -> a -> b
memoEq f = unsafePerformIO ( do { tref <- newIORef emptyTblEq
                                ; return (applyEq f tref)
                                })

applyEq :: Eq a => (a -> b) -> IORef (TblEq a b) -> a -> b
applyEq f tref arg
  = unsafePerformIO (
    do { tbl <- readIORef tref
       ; case lookupEq tbl arg of
           Just result -> return result
           Nothing     -> do { let res  = f arg
                             ; let tbl' = insertEq tbl arg res
                             ; writeIORef tref tbl'
                             ; return res
       }                     })

type TblEq a b = [(a,b)]
emptyEq  :: TblEq a b
lookupEq :: Eq a => TblEq a b -> a -> Maybe b
insertEq :: Eq a => TblEq a b -> a -> b -> TblEq a b
-- Implementations omitted
```

The first application of `unsafePerformIO` allocates a mutable cell that holds the memo table, of type `TblEq a b`. It then immediately returns the memoised function, a partial application of `applyEq`. When the latter is given an argument, it again uses `unsafePerformIO` to get hold of the memo table, query it, and perhaps write a new value into it. The memo table, here represented as a simple association list, contains argument-value pairs. In the context of memo tables we will often refer to the function argument as the *key*, and the result as the *value*.

Of course, an association list is hardly the most efficient structure for a memo table, a further reason for wanting memo tables to be programmable. We could instead use some kind of lookup tree, based on ordering (not just equality) of the argument. That would in turn require that the argument type was ordered, thus changing `memo`'s type again:

```
memoOrd :: Ord a => (a -> b) -> a -> b
```

`memoOrd` can be implemented exactly as above, except that the lookup and insert functions become more complicated. We can do hashing in a very similar way. Notation apart, all of this is exactly how a Lisp programmer might implement memo functions. All we have done is to make explicit exactly where the programmer is undertaking proof obligations — a modest but important step.

4 Stable names

Using equality, as we have done in `memoEq`, works OK for base types, such as `Int` and `Float`, but it becomes too expensive when the function's argument is (say) a list. In this case, we almost certainly want something like pointer equality; in exchange for the fast test we accept that two lists might be equal without being pointer-equal.

However, having only (pointer) equality would force us back to association lists. To do better we need ordering or a hash function. The well-known difficulty is that unless the garbage collector never moves objects (an excessively constraining choice), an object's address may change, and so it makes a poor hash key. Even the relative ordering of objects may change.

4.1 The `StableName` type

What we need is a cheap address-like value, or *name* that can be derived from an arbitrary value. This name should be *stable*, in the sense that it does not change over the lifetime of the object it names. With this in mind, we provide an abstract data type `StableName`, with the following operations:

```
data StableName a     -- Abstract

mkStableName   :: a -> IO (StableName a)
hashStableName :: StableName a -> Int

instance Eq  (StableName a)
instance Ord (StableName a)
```

The function `mkStableName` makes a stable name from any value. Stable names support equality (class `Eq`) and ordering (class `Ord`). In addition, the function `hashStableName` converts a stable name to a hash key.

Notice that `mkStableName` is in the `IO` monad. Why? Because two stable names might compare less-than in one run of the program, and greater-than in another run. Putting `mkStableName` in the `IO` monad is a standard trick that allows `mkStableName` to consult (in principle) some external oracle before deciding what stable name to return. In practice, we often wrap calls to `mkStableName` in an `unsafePerformIO`, thereby undertaking a proof obligation that the meaning of the program does not depend on the particular stable name that the system chooses.

Stable names have the following property: if two values have the same stable name, the two values are equal

$$(\dagger)\ \texttt{mkStableName}\, x = \texttt{mkStableName}\, y \quad \Rightarrow \quad x = y$$

This property means that stable names are unlike hash keys, where two keys might accidentally collide. If two stable names are equal, no further test for equality is necessary. An immediate consequence of (†) is this: if two values are not equal, their stable names will differ.

$$x \neq y \quad \Rightarrow \quad \texttt{mkStableName}\, x \neq \texttt{mkStableName}\, y$$

`mkStableName` is not strict; it does not evaluate its argument. This means that two equal values might not have the same stable name, because they are still distinct unevaluated thunks. For example, consider the definitions

```
p  = (x,x)
f1 = fst p
f2 = snd p
```

So long as `f1` and `f2` remain unevaluated, `mkStableName f1` will return a different stable name than `mkStableName f2`[3].

It is easy to make `mkStableName` strict, by using Haskell's strict-application function "`$!`". For example, `mkStableName $! f1` and `mkStableName $! f2` would return the same stable name. Using strict application loses laziness, but increases sharing of stable names, a choice that only the programmer can make.

4.2 Using stable names for memo tables

Throughout the rest of this paper, we will make use of Stable Name Maps, an abstract data type that maps Stable Names to values (Figure 1). The implementation may be any kind of mutable finite map, for example an `IORef` to a standard finite map:

[3] A compiler optimisation might well have evaluated `f1` and `f2` at compile time, in which case the two calls would return the same stable name; another example of why `mkStableName` is in the `IO` monad.

```
data SNMap k v -- abstract

newSNMap    :: IO (SNMap k v)
lookupSNMap :: SNMap k v -> StableName k -> IO (Maybe v)
insertSNMap :: SNMap k v -> StableName k -> v -> IO ()
removeSNMap :: SNMap k v -> StableName k -> IO ()
snMapElems  :: SNMap k v -> IO [(k,v)]
```

Fig. 1. Stable Name Map Library

```
data SNMap k v = IORef (FiniteMap (StableName k) v)
```

or a real hash table (using a combination of `hashStableName` and equality on `StableName`):

```
data SNMap k v = IOArray Int [(StableName k, v)]
```

Using stable names it is easy to modify our memo-table implementation to use pointer equality (strict or lazy) instead of value equality. We give only the code for the `apply` part of the implementation

```
applyStable :: (a -> b) -> SNMap a b -> a -> b
applyStable f tbl arg
  = unsafePerformIO ( do { sn  <- mkStableName arg
                         ; lkp <- lookupSNMap tbl sn
                         ; case lkp of
                             Just result -> return result
                             Nothing     ->
                                do { let res = f arg
                                   ; insertSNMap tbl sn res
                                   ; return res
                         }         })
```

4.3 Implementing stable names

Our implementation is depicted in Figure 2. We maintain two tables. The first is a hash table that maps the address of an object to an offset into the second table, the *Stable Name Table*. If the address of a target changes during garbage collection, the hash table must be updated to reflect its new address. There are two possible approaches:

- Always throw away the old hash table and rebuild a new one after each garbage collection. This would slow down garbage collection considerably when there are a large number of stable names.

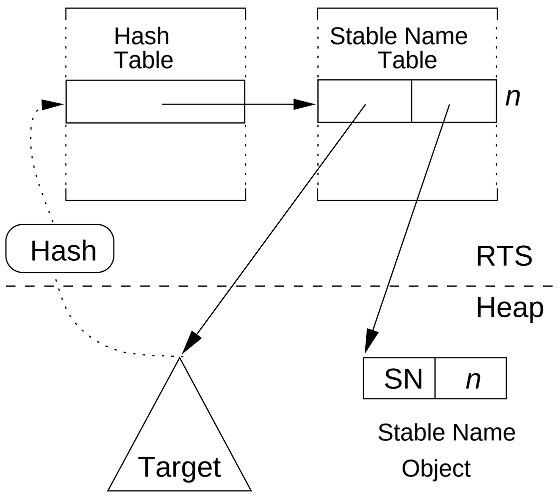

Fig. 2. Stable Name Implementation

– In a generational collector, we have the option of partially updating the hash table during a minor collection. Only the entries for targets which have moved during the current GC need to be updated. This is the method used by our implementation.

Each slot in the Stable Name Table (SNT) corresponds to a distinct stable name. The stable name can be described by its offset in the SNT, and it is this offset that is used for equality and comparison of stable names.

However, we cannot simply use this offset as the value returned by `mkStableName`! Why not? Because in order to maintain (†) we must ensure that we never re-use a stable name to which the program still has access, *even if the object from which the stable name was derived has long since died.*

Accordingly, we represent a value of type `StableName a` by a *stable name object*, a heap-allocated cell containing the SNT offset. It is this object that is returned as the result of `mkStableName`. The entry in the SNT points to the corresponding stable name object, and also the object for which the stable name was created (the *target*).

Now entries in the SNT can be garbage-collected as follows. The SNT is not treated as part of the root set. Instead, when garbage collection is complete, we scan the entries of the SNT that are currently in use. If an entry's stable name object is dead (not reachable), then it is safe to re-use the stable name entry, because the program cannot possibly "re-invent" it. For each stable name entry that is still live, we also need to update the pointers to the stable name object and the target, because a copying collector might have moved them.

Available entries in the SNT are chained on a free list through the stable-object-pointer field.

4.4 `hashStableName`

The `hashStableName` function satisfies the following property, for stable names a and b:

$$a = b \quad \Rightarrow \quad \texttt{hashStableName}\ a = \texttt{hashStableName}\ b$$

The converse is not true, however. Why? The call `hashStableName a` is implemented by simply returning the offset of the stable name `a` in the SNT. Because the `Int` value returned can't be tracked by the garbage collector in the same way as the stable name object, it is possible that calls to `hashStableName` on different stable names could return the same value. For example:

```
do { sn_a <- mkStableName a
   ; let hash_a = hashStableName sn_a
   ; sn_b <- mkStableName b
   ; let hash_b = hashStableName sn_b
   ; return (hash_a == hash_b)
   }
```

Assuming `a` and `b` are distinct objects, this piece of code could return `True` if the garbage collector runs just after the first call to `hashStableName`, because the slot in the SNT allocated to `sn_a` could be re-used by `sn_b` since `sn_a` is garbage at this point.

4.5 Other applications

An advantage of the implementation we have described is that we can use the very same pair of tables for two other purposes. When calling external libraries written in some other language, it is often necessary to pass a Haskell object. Since Haskell objects move around from time to time, we actually pass a *Stable Pointer* to the object. A stable pointer is a variant of a stable name, with slightly different properties:

1. It is possible to dereference a stable pointer to get to the target. This means that the existence of a stable pointer must guarantee the existence of the target.
2. Stable pointers are reference counted, and must be explicitly freed by the programmer. This is because a stable pointer can be passed to a foreign function, leaving no way for the Haskell garbage collector to track it.

We implement stable pointers using the same stable name technology. The stable name table already contains a pointer to the target of the stable name, hence (1) is easy. To support (2) we add a reference count to the SNT entry, and operations to increment and decrement it. The pointer to the target is treated as a root by the garbage collector if and only if the reference count is greater than zero.

We use exactly the same technology again for our parallel implementation of Haskell, Glasgow Parallel Haskell (GPH). GPH distributes a single logical Haskell heap over a number of disjoint address spaces [11]. Pointers between these sub-heaps go via stable names, thus allowing each sub-heap to be garbage collected independently. Weighted reference counting is used for global garbage collection [8].

The point here is simply that a single, primitive mechanism supports all three facilities: stable names, passing pointers to foreign libraries, and distributed heaps.

5 Weak pointers

If a memoised function is discarded, then its memo table will automatically be garbage collected. But suppose that a memoised function is long-lived, and is applied to many arguments, many of which are soon discarded. This situation gives rise to a well-known space leak:

- Since the memo table contains references to all the arguments to which the function has ever been applied, those arguments will be reachable (in the eyes of the garbage collector) even though the function will never be applied to that argument again.
- Not only that, but the result of applying the function to those arguments is also held in the memo table, and hence will be retained for ever.
- Finally, the memo table itself becomes clogged with useless entries that serve only to slow down lookup operations.

The first of these problems seems to go away when we use stable names, since it is the stable names that are retained in the memo table, not the argument itself; but the latter two problems remain, and the first reappears as an inability to recycle stable names.

5.1 Weak pointers

The standard solution to these woes is to use *weak pointers*. The garbage collector recovers all heap objects that are not *reachable*. A heap object is reachable if it is in the transitive closure of the *points-to* relation starting from the set of *root pointers*. A *weak pointer* is a pointer that is not treated as a pointer for the purposes of computing reachability. That is, even if object A is reachable, and A contains a weak pointer to another object B, the latter is not thereby considered reachable[4].

Object B may be reachable from the root set by some other path, of course, but if not, it is considered garbage. In this case, the weak pointer in object A no longer points to a valid object, and is replaced by a *tombstone*. The act of dereferencing a weak pointer will fail if the latter has been tombstoned.

Weak pointers help memo tables in the following way. Ignoring stable names for now, assume that the memo table refers to both the keys and values. If the pointer to the key is a weak pointer, then the memo table will not keep the key alive, thus solving the first problem. Periodically the memo table can be "purged", by searching for keys that have been tombstoned, and deleting their entry from the memo table, thus releasing the value as well.

5.2 A problem with weak pointers

A little-recognised problem with using weak pointers for memo tables is this: *if the value contains a pointer to the key, the entry will never be removed.* If the value refers to the key, then the memo table will keep the value alive, the value will keep the key alive, and the entry in the memo table can never be purged, which defeats the whole purpose of the weak pointer. We will refer to this as the key-in-value problem.

If this problem actually occurs in practice, it causes a potentially-lethal space leak, and one that is not easy to identify or cure. Unfortunately, the situation is by no means unusual. Consider a lookup table that maps a person's name to a record describing the person. It is quite likely that the record will include, among other things, the person's name.

5.3 A new design

In the light of these issues, we have developed a new design for weak pointers in Haskell, called *key/value weak pointers*. Here is (part of) the signature of the `Weak` module:

```
data Weak a            -- Abstract

mkSimpleWeak :: k -> v -> IO (Weak v)
deRefWeak    :: Weak v -> IO (Maybe v)
```

The function `mkSimpleWeak` takes a "key", a "value" of type `v`, and builds a *weak pointer object* of type `Weak v`. Weak pointers have the following effect on garbage collection:

– *The value of a weak pointer object is reachable if the key is reachable*[5].

[4] The alert reader may have noticed that an entry in the Stable Name Table of Section 4.3 effectively contains a weak pointer to its stable name object.

[5] Recall that the garbage collector recovers memory that is not reachable; and also note that the statement says "if", not "if and only if"

The specification says nothing about the reachability of the weak pointer object itself, so whether or not the weak pointer object is reachable does not affect the reachability of its key or value.

This simple, crisp, specification conceals quite a subtle implementation (Section 5.5), but it offers precisely the support we need for memo tables. It does not matter if the value refers to the key, because the value is not reachable unless the key is — or unless the value is reachable some other way, in which case the key is certainly reachable via the value.

`mkSimpleWeak` is in the `IO` monad because it has an important operational behaviour: before the call, the key and value are both reachable, but after the call the reachability of the value is dependent on the reachability of the key. This isn't a side-effect as such — it wouldn't change the meaning of the program if we delayed the operation — but to obtain the desired effect it's important that we can force the call to `mkSimpleWeak` to be performed at a certain time, hence we use the `IO` monad for sequencing.

The function `deRefWeak` dereferences a weak pointer, returning either `Nothing` (if the value has been garbage collected), or `Just v` (where `v` is the value originally given to `mkSimpleWeak`). The `deRefWeak` operation is in the `IO` monad for an obvious reason: its return value can depend on the time at which the garbage collector runs.

Though we developed our design independently, we subsequently discovered that Hayes's OOPSLA'97 paper [4] describes a much earlier implementation of the same core idea, there dubbed *ephemerons*, originally due to Bosworth. We contrast our designs in Section 9.

5.4 Memo table using key/value weak pointers

We can now give the code for a memo table that uses weak pointers, based on our earlier stable-name version.

```
applyWeak :: (a -> b) -> SNMap a (Weak b) -> a -> b
applyWeak f tbl arg
  = unsafePerformIO (do { sn  <- mkStableName arg
                        ; lkp <- lookupSNMap tbl sn
                        ; case lkp of
                            Nothing   -> not_found tbl sn
                            Just weak ->
                               do { val <- deRefWeak weak
                                  ; case val of
                                         Just result ->
                                           return result
                                         Nothing ->
                                           not_found tbl sn
                       }          })
  where
    not_found tbl sn = do { let res = f arg
```

```
                                    ; weak <- mkSimpleWeak arg res
                                    ; insertSNMap tbl sn weak
                                    ; return res
                                    }
```

The memo table maps a stable name for the argument to a weak pointer to the value. If the function has not been applied to `arg` before, the call to `lookupSNMap` will return `Nothing`, and the auxiliary function `not_found` will be called. The latter makes a weak pointer for the result, with a lifetime controlled by `arg`, and inserts this weak pointer into the memo table as before.

If the lookup is successful, `deRefWeak` is used to find the actual value. There is an awkward race condition here, because at the moment `deRefWeak` is called there might, conceivably, be no further references to `arg`. If that is so, and a garbage collection intervenes, the weak pointer might be tombstoned before `deRefWeak` gets to it. In this unusual case we simply call `not_found`. Strangely enough, doing so makes `arg` reachable in the continuation of `deRefWeak`, and thus ensures that `deRefWeak` will always succeed. This sort of weirdness is typical of the world of weak pointers.

5.5 Implementing weak pointers

The definition of reachability is simple, but it takes a little care to implement it correctly. Our implementation works as follows. We maintain a list of all current weak pointer objects, called the Weak Pointer List. When a new weak pointer object is created, it is immediately added to this list. Garbage collection proceeds as follows:

1. Mark all the heap reachable from the roots. (We will pretend that we are using a mark-sweep garbage collector, but everything works fine for copying collectors too.)
2. Examine each weak pointer object on the Weak Pointer List, *whether or not it is itself reachable*. If it has a key that is marked (i.e. is reachable), then mark all the heap reachable from its value field, and move the weak pointer object to a new list.
3. Repeat from step (2), until a complete scan of the Weak Pointer List finds no weak pointer object with a marked key.
4. For each remaining object on the Weak Pointer List, either tombstone it (if it is marked), or simply discard it (otherwise).
5. The list accumulated in step (2) becomes the new Weak Pointer List. Mark any unreachable weak pointer objects on this list as reachable, so that they will be retained by the garbage collector.

There are two subtleties in the implementation. The first is the iteration necessary in step (3). This is required, because making one value reachable may make the key of some other weak pointer object reachable; and so on. Notice that the reachability of the value of a weak pointer object is influenced only by the reachability of the corresponding key, and not at all by the reachability, or otherwise, of the weak pointer object itself.

The second subtlety is the relationship between *reachability* and *retainability*. The *reachability* criterion is used to determine which weak pointers to tombstone, but it is not the same as the set of objects retained by the garbage collector. The objects retained are precisely the reachable objects, plus any weak pointer objects which have reachable keys, but which are unreachable themselves at the end of the algorithm.

Although all live weak pointer objects are implicitly kept by the garbage collector regardless of whether they are reachable, it would be wrong to mark them all as reachable as a first step in the above algorithm. This is because doing so would preclude having a weak pointer object whose key is itself a weak pointer object, because the key would always be considered reachable. Weak pointers to weak pointers are a useful concept, as we shall see later (Section 9).

The above implementation can be extended straightforwardly to work with a generational garbage collector. The guiding principle is: any object which resides in a generation which we are not collecting is considered to be reachable for the purposes of this collection. So if the key of a weak pointer lives in the oldest generation, we will not be able to determine that the weak pointer is dead until we perform a major collection.

5.6 Other applications

Another situation where we found weak pointers to be "just the right thing" is when referencing objects outside the Haskell heap via proxy objects (a proxy object is an object in the local heap that just contains a pointer to the foreign object).

Consider a structured foreign object, to which we have a proxy object in the Haskell heap. The garbage collector will track the proxy object in order that the foreign object can be freed when it is no longer referenced from Haskell (probably using a finalizer, see the next section). If we are given a pointer to a subcomponent of the foreign object, then we need a suitable way to keep the proxy for the *root* of the foreign object alive until we drop the reference to the subcomponent.

A weak pointer solves this problem nicely: the *key* points to a proxy for the subcomponent, and the *value* points to the proxy for the root. The entire foreign object will thereby be retained until all references to the subcomponent are dropped.

6 Finalization

We did not present code for purging the memo table of useless key/value pairs. Indeed, the whole idea is less than satisfactory, because it amounts to *polling* the keys to see if they have died. It would be better to receive some sort of *notification* when the key died.

Indeed, it is quite common to want to perform some sort of clean-up action when an object dies; such actions are commonly called *finalization*. If it were possible to attach a finalizer to the key, then when the key dies, the finalizer could delete the entry from the memo table. A particular key might be in many memo tables, so it is very desirable to be able to attach multiple finalizers to a particular object.

Finalizers are often used for *proxy objects* that encapsulate some external resource, such as a file handle, graphics context, malloc'd block, network connection, or whatever. When the object becomes garbage, the finalizer runs, and can close the file, release the graphics context, free the malloc'd block, etc. In some sense, these proxy objects are the dual to stable pointers (Section 4.5): they encapsulate a pointer from Haskell to some external world, while a stable pointer encapsulates a pointer from the external world into Haskell.

Finalizers raise numerous subtle issues. For example, does it matter which order finalizers run in, if several objects die "simultaneously" (whatever that means)? The finalizer may need to refer to the object it is finalizing, which presumably means "resurrecting" it from the dead. If the finalizer refers to the object, might that keep it alive, thereby vitiating the whole effect? If not, how does the finalizer get access to the object? How promptly do finalizers run? And so on. [3] gives a useful overview of these issues, and a survey of implementations.

6.1 A design for finalizers

In our experience, applications that use weak pointers almost always require some sort of finalization as well, so we have chosen to couple the two. We add the following two new functions:

```
mkWeak   :: k -> v -> Maybe (IO ()) -> IO (Weak v)
finalize :: Weak v -> IO ()
```

`mkWeak` is like `mkSimpleWeak`, except that it takes an extra argument, an optional finalization action. The call `(mkWeak k v (Just a))` has the following semantics:

- If `k` becomes unreachable, the finalization action `a` is performed some time afterwards. There is no guarantee of how soon afterwards, nor about the order in which finalizers are run.
- Finalization of a weak object may be initiated at any time, by applying `finalize` to it. The weak pointer object is immediately replaced by a tombstone, and its finalizer (if it has one) is run. The `finalize` operation returns only on completion of the finalizer.
- The finalization action `a` is guaranteed to be performed *exactly once* during the run of the program, either when the programmer calls `finalize`, or some time after `k` becomes unreachable, or at the end of the program run.

The `mkSimpleWeak` operation is implemented in terms of `mkWeak`, by passing `Nothing` as the finalizer.

The finalization action `a` is simply an I/O action of type `IO ()`. Here, for example, is how one might arrange to automatically close a file that was no longer required:

```
fopen :: String -> IO Handle
fopen filename
 = do { hdl <- open filename
      ; mkWeak hdl () (Just (close hdl))
      ; return hdl
      }

open  :: String -> IO Handle
close :: Handle -> IO ()
```

Here, `fopen` uses `open` to open the file, and then calls `mkWeak` to attach a finalizer to the handle returned by `open`. (In this case the second parameter of `mkWeak` is irrelevant.) The finalizer `(close hdl)` is of type `IO ()`; when `hdl` becomes unreachable the finalizer is performed, which closes the file.

The following points are worth noticing about finalizers:

- In the `fopen` example, the finalizer refers to `hdl`. We are immediately faced with a variant of the key/value problem for memo tables (Section 5.2). It would be a disaster if the finalizer kept the key alive, which in turn would ensure the finalizer never ran! We solve this simply by modifying the reachability rule for weak pointers:
  - The value *and finalizer* of a weak pointer object are reachable if the key is reachable.
- *Any* value whatsoever (even a weak pointer object) can have a finalizer attached in this way – this is called *container-based finalization*. It contrasts with destructors in C++, which implement *object-based finalization* in which the finalizer is part of the object's definition.
- A value can have any number of finalizers attached, simply by making several calls to `mkWeak`. (This is essential if (say) a key is entered in several memo tables.) Each of the finalizers is run exactly once, with no guarantee of relative order.
- The program may discard the weak pointer object returned by `mkWeak` if it isn't required (as we did in the example above). The finalizer will still run when the key becomes unreachable, but we won't be able to call `finalize` to run the finalizer early.

6.2 Implementing finalization

Finalizers are relatively easy to implement. The weak pointer implementation of Section 5.5 needs modification as follows:

1. Mark all the heap reachable from the roots.

2. Scan the Weak Pointer List. If a weak pointer object has a key that is marked (i.e. is reachable), then mark all the heap reachable from its value *or its finalizer*, and move the weak pointer object to a new list.
3. Repeat from step (2), until a complete scan of the Weak Pointer List finds no weak pointer object with a marked key.
4. Scan the Weak Pointer List again. If the weak pointer object is reachable, then tombstone it. If the weak pointer object has a finalizer, then move it to the Finalization Pending List, and mark all the heap reachable from the finalizer. If the finalizer refers to the key (and/or value), this step will "resurrect" it.
5. The list accumulated in step (3) becomes the new Weak Pointer List. Mark any unreachable weak pointer objects on this list as reachable.

Subsequent to garbage collection, a dedicated *finalization thread* successively removes a item from the Finalization Pending List, and executes the finalizer. The finalization thread runs pseudo-concurrently with the program; if a finalizer shares state with the main program then suitable synchronisation must be used. We use the primitives of Concurrent Haskell for this purpose [9].

7 Memo tables with finalization

In this section we bring together stable names, weak pointers and finalizers in an implementation of a memo table that can purge itself of unneeded key/value pairs, and also release itself when the memoized function is no longer reachable. The implementation is given in Figure 3, and a diagram depicting the memo table structure is given in Figure 4.

The memo table representation is identical to the one given in Section 5.4, except that we now add a finalizer to each weak pointer in the table. When invoked, the finalizer will remove its own entry from the memo table, allowing the value (the memoized result of this computation) to be garbage collected.

This inadvertently creates a problem for garbage collecting the entire memo table: since each finalizer now needs to refer to the memo table, and by the reachability rule we gave for weak pointers with finalizers, this means that the memo table is reachable if the key of any weak pointer in the table is reachable. This is a disaster! Even if the memoized function dies, the memo table, including all the cached values, will live on until all the keys become unreachable.

The solution, not unsurprisingly, is to use another weak pointer. If all the finalizers refer to the memo table only through a weak pointer, we retain the desired reachability behaviour for the memo table itself. If a running finalizer finds that the memo table has already become unreachable, because `deRefWeak` on the weak pointer to the table returns `Nothing`, then there's no finalization to do.

We also add a finalizer to the memo table (`table_finalizer`), which runs through all the entries in the table calling `finalize` on each weak pointer. This is important because it allows all the values to be garbage collected at the same

```
type MemoTable a b = SNMap a (Weak b)

memo :: (a -> b) -> a -> b
memo f =
   let (tbl,weak) = unsafePerformIO (
          do { tbl <- newSNMap
             ; weak <- mkWeak tbl tbl (Just (table_finalizer tbl))
             ; return (tbl,weak)
             })
   in  memo' f tbl weak

table_finalizer :: SNMap a (Weak b) -> IO ()
table_finalizer tbl =
   do { pairs <- snMapElems tbl; sequence_ [ finalize w | (_,w) <- pairs ] }

memo' :: (a -> b) -> MemoTable a b -> Weak (MemoTable a b) -> a -> b
memo' f tbl weak_tbl arg = unsafePerformIO (
  do { sn <- mkStableName arg
     ; lkp <- lookupSNMap tbl sn
     ; case lkp of
        Nothing -> not_found
        Just w  -> do { maybe_val <- deRefWeak w
                      ; case maybe_val of
                           Nothing  -> not_found
                           Just val -> return val
                      }

      })
  where val = f arg
        not_found = do { weak <- mkWeak arg val (Just (finalizer sn weak_tbl))
                       ; insertSNMap tbl sn val
                       ; return val
                       }

finalizer :: StableName a -> Weak (MemoTable a b) -> IO ()
finalizer sn weak_tbl = do { r <- deRefWeak weak_tbl
                           ; case r of
                                Nothing -> return ()
                                Just mvar -> removeSNMap tbl sn
                           }
```

Fig. 3. Full Memo Table Implementation

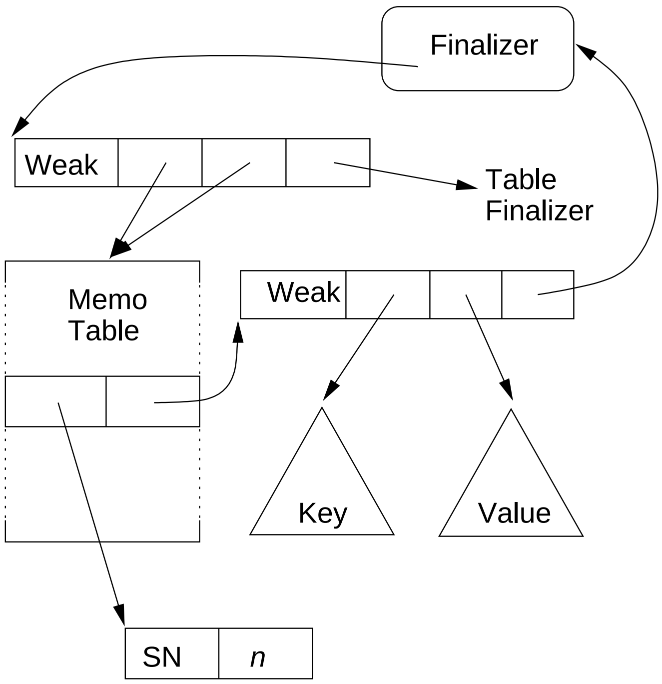

Fig. 4. Full Memo Table Implementation

time as the table; without this finalizer, the values would live on until their respective keys became unreachable.

8 Push-based Architectures

One compelling and intriguing application of key/value weak pointers turned up in a new implementation of the Fran reactive animation system, which is implemented in Haskell. In this section we sketch the problem and explain how weak pointers solve it.

8.1 Message Networks

A Fran program can usefully be thought of as a "message network". A message network is a directed acyclic graph (dag) that defines a conceptual flow of messages through a collection of hardware and software components. In such a network, each node may be a *source*, *sink* or *filter* of messages. For example,

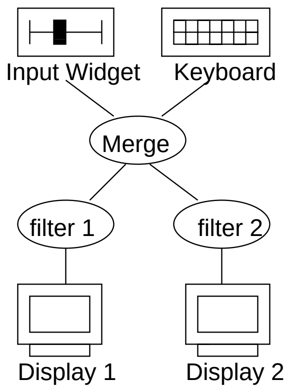

Fig. 5. Sample message network

digital input devices are sources, whether implemented in hardware (e.g., mice, head trackers, or video cameras), or software (e.g., slider and type-in widgets). Similarly, hardware and software digital output devices are sinks. Message "filters" come in two forms. Some filters pass all received messages through, but alter each one according to some algorithm. Other filters pass some messages through, while blocking others.

Pairs of nodes that are directly connected to each other in a message network are in an ordered "service/client" relationship. The provider of a message is the service and the recipient is the client. In general, source nodes can only be services, sinks can only be clients, and filters can act as either, and frequently act as both clients and services simultaneously.[6]

As a simple example, Figure 5 shows a message network with two sources (keyboard and button widget) whose messages are merged, then transformed by a calculation, then separately filtered and passed on to two separate display sinks.

So far, we have avoided saying how messages flow through a network. Two obvious candidates are pushing and pulling. In a push implementation, services take the initiative. Whenever a node receives a message, it pushes the message to each of its clients (possibly altering it first), unless it decides to block the message. The overall effect is that availability of source data drives work through the network toward sinks. In a pull implementation, on the other hand, clients take the initiative. Every time a node wants a message, it tries to pull one from

[6] Note that a sink is not just a node with no clients, but rather a node that has an external effect. A source or filter node could also lack clients merely because no one is currently interested in its messages. Similarly, a source is not just a node with no services.

each of its services. These demands eventually reach sources, where they will be satisfied if new data is available.

In order to decide whether to use a pull- or push-based implementation, one needs to examine the relative frequency of (a) data generation at sources, vs (b) data consumption at sinks. If sources generate values much more frequently than sinks need them (even considering rejection by filters), then pull may be the right choice. If, however, source data is infrequent compared to sink demands, then pushing is probably called for. The push model has an additional advantage, namely that it minimizes the latency between the external stimulus and its resulting response. Because we have been doing interactive graphics, we are interested both in high display rates and low latency reactions, so the push model is very appealing.

8.2 Garbage Collection

What does the question of push vs pull have to do with garbage collection? Considering again the example in Figure 5, which nodes should be kept alive during a garbage collection? The answer should be exactly the "useful" ones. Certainly any nodes that have references external to the message network should be kept, because they can be used by the reference holders. The sink (display) nodes are also useful, because they have external effect. The nodes that directly act as services with these sinks as clients are useful because they are useful to (i.e., they serve) the sinks. In general, for each service/client pair, if the client is useful, then the service is also.

All nodes in Figure 5 are useful, but what if `display 2` were turned off, i.e., became no longer useful? Then `filterB` no longer has any client, so it is no longer useful. The shared calculation node still has a useful client (`display 1`), so it is still useful, as are the rest of nodes in the graph.

This notion of "usefulness" has an intriguing similarity to the notion of reachability that underlies garbage collection. In a simple garbage collection setting (without weak pointers or finalizers), the set of accessible objects is the transitive closure of the set of root objects under the binary points-to relation. Similarly, the set of useful nodes in a message network is the transitive closure of the set of sinks under the binary *served-by* relation.

While the demands of our application area favor push, as described above, garbage collection favors pull. The reason is that for pull, served-by implies points-to. That is, for node A to be served by node B, A must point to B. By closure then, useful implies reachable, and thus the garbage collection will preserve all useful nodes. For push, however, the served-by and points-to relations have opposite orientation! For A to be served by B, B must point to A.

In our example, `display 2` was served by `filter 2`, so `filter 2` points to `display 2`. With conventional garbage collection, `filterB` would therefore sustain `display 2`, and the shared calculations node would sustain them both. The result is not only a space- but also an "effort-leak", since work is being done by the useless but sustained nodes.

To fix this space- and effort-leak, observe that the served-by relation is not a strong enough reason to keep a client alive, and thus should be represented by a *weak pointer* from service to client. This step goes too far, however, allowing every node in a message network to be reclaimed if it has no reference from *outside* of the network. In addition to making services not sustain clients, we must also make clients sustain services. This step is easy: merely construct a second weak pointer, this time with the client as the key and service as the value, and no finalizer. It is not even necessary to hold onto this second weak pointer object.

As an optimization, combine the two weak pointers into one, stored at the service. The key is the client, in order to control lifetime. The value contains the service, so it will be sustained, *and* the client, so that service can send it messages. What about the (optional) finalizer? Just as in the case of memo tables, a service node will in fact contain a set of these weak pointers. When the key (client) dies, the corresponding weak pointer is tombstoned. For efficiency, however, we would also like to have these weak pointers be removed from their sets. A finalizer is just the right tool for this job.

8.3 Observations

We have deliberately cast the discussion in general terms, because we believe that it illuminates a fundamental mis-match between traditional garbage collection and "push" applications. Solving the mis-match seems to require the full generality of key/value weak pointers. An open question is whether key/value weak pointers are "complete" (whatever that means), or whether some new application may require something yet more complicated.

9 Comparison

We are not aware of any other published work on stable names, although it seems likely that others have implemented similar mechanisms internally. Java's global and local references (part of the Java Native Interface, described in [6]) are similar to our stable pointers (Section 4.5) in that their primary function is to allow Java objects to be passed to foreign functions, by providing an indirection table and explicit freeing of references.

Weak pointers, on the other hand, are well known. Several language implementations include simple weak pointers, that is weak pointers that cannot express the key/value relationship and hence suffer from the problem we described in Section 5.2. These include Smalltalk, T, Caml, Moscow ML, SML/NJ, and several Scheme implementations. Java has no less than three kinds of weak pointer [6]: Soft References allow objects to be reclaimed when memory is short, Weak References are simple weak pointers, and Phantom References are a weaker form of Weak Reference.

Ephemerons, described by Hayes [4], are very similar to our weak pointers. They differ in subtle but important ways. First, the semantics of ephemerons is described by presenting a tricky garbage collection algorithm (similar to that in Section 5.5). We believe that our characterisation in terms of reachability is much more useful for programmers. This is a presentational difference, but there is a semantic difference too: the reachability rule for ephemerons is

- The value field of an ephemeron is reachable if *both* (a) the ephemeron (weak pointer object) is reachable, *and* (b) the key is reachable.

This semantics is actually a little more convenient than ours for the memo-table application, because it means there is no need to finalize the memo table itself (Section 7). We chose our semantics (i.e. delete clause (a)) for several reasons. First, it is simpler. Second, with the ephemeron semantics it is not clear when the finalizer should be run. When the key becomes unreachable? When the key *and* the ephemeron become unreachable? In fact, the choice made for ephemerons is neither of these: the finalizer of an ephemeron is run only if (a) the ephemeron is reachable and (b) the key is not. If the ephemeron itself is not reachable, the finalizer is never run. This contrasts with our guarantee that each finalizer is run precisely once.

Third, one can easily simulate the ephemeron reachability semantics with ours, but the reverse is not possible. The following function simulates the ephemeron semantics:

```
mkEphemeron :: k -> v -> Maybe (IO ())
       -> IO (Weak v)
mkEphemeron k v f
  = do { eph <- mkWeak k v f
       ; mkWeak eph () (Just (finalize eph))
       ; return eph
       }
```

The second call to `mkWeak` simply attaches a finalizer to the ephemeron, so that if the ephemeron ever becomes unreachable it is finalized, thus breaking the key-to-value link. This does not have the same finalization semantics as ephemerons do, but whether that is a bug or a feature is debatable.

Finalizers have been the subject of heated debate on the `gclist` mailing list. The conclusions of this debate, and of Hayes's excellent survey [3], are that

- A programmer should not rely on finalizers running promptly. Promptness is just too hard to guarantee. If promptness is required, then explicit finalization is indicated.
- No guarantees should be made about the order in which finalizers should run.

Dybvig proposed *guardians* for Scheme [2], a sort of batched version of finalizers. A (weak) pointer can be added to a guardian, and the guardian can be queried to find out which of the objects it maintains have become inaccessible. Dybvig also describes how to implement hash tables using guardians. The hash table he describes is capable of purging old key/value pairs, but only on activation of the lookup function (i.e. not asynchronously), and it also suffers from the key-in-value problem.

10 Conclusion

We have now described four mechanisms — `unsafePerformIO`, stable names, weak pointers, and finalization — that collectively allow us to implement memo tables in Haskell. If that were the sole application, we could be accused of overkill. But each of the mechanisms has independent uses of its own, as we have already indicated. What is surprising, perhaps, is that memo functions require such an elaborate armoury.

Many readers, ourselves included, will have a queasy feeling by this stage. What is left of the beauty of functional programming by the time all these primitives have been added? How can the unspecified "proof obligations" of `unsafePerformIO` be characterised and proved? Has the baby been thrown out with the bath water? These are justifiable criticisms. The baby is indeed in danger.

Our primary response is this: if we can simply provide a completely encapsulated implementation of `memo`, implemented as a primitive in (say) C, would that have been better? Far from it! The same functionality would have to be implemented, but with greater scope for error. Furthermore, it would take intervention by the language implementors to modify or extend the implementation. In any case, `memo` is but one of a whole raft of applications for the primitives we have introduced. So, we regard the primitives of this paper as *the raw material from which experienced system programmers can construct beautiful abstractions.* We wish that it were possible for the primitives to themselves be beautiful abstractions, but that aspiration seems to be beyond our reach.

So, our proposals have clear shortcomings. But the alternatives are worse. We could eschew weak pointers, finalizers, etc etc, and thereby exclude an important and useful class of applications. Or we could keep their existence secret, advertising only their acceptable face (such as `memo`). Instead, we have striven to develop as precise a characterisation of our primitives as we can, warts and all. We hope thereby to provoke a debate that may ultimately lead to new insights, and a better overall design.

11 Acknowledgements

We would like to thank the following people for helpful comments on earlier versions of this paper: Kevin Backhouse, Byron Cook, Barry Hayes, Fergus Henderson, Richard Jones, Andrew Kennedy, Sven Panne, and Julian Seward.

References

1. B. Cook and J. Launchbury. Disposable memo functions. In *Proceedings of the 1997 Haskell Workshop*, 1997.

2. R. Dybvig, C. Bruggeman, and D. Elby. Guardians in a generation-based garbage collector. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'93), Albuquerque*, pages 207–216, June 1993.
3. B. Hayes. Finalization in the collector interface. In Y. Bekkers and J. Cohen, editors, *Proceedings of the International Workshop on Memory Management (IWMM'92), St Malo*, pages 277–298. Springer Verlag LNCS 637, Sept 1992.
4. B. Hayes. Ephemerons: a new finalization mechanism. In *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, pages 176–183. ACM, Oct 1997.
5. R. Hughes. Lazy memo-functions. In *Proc Aspenas workshop on implementation of functional languages*, Feb 1985.
6. Java Software, Sun Microsystems, Inc., `http://java.sun.com/docs/`. *Java Development Kit 1.2 Documentation*.
7. R. Keller and M. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8:88–108, Jan. 1986.
8. D. Lester. An efficient distributed garbage-collection algorithm. In *Proc Parallel Architectures and Languages Europe (PARLE)*, pages 207–223. Springer Verlag LNCS 365, June 1989.
9. SL Peyton Jones, AJ Gordon, and SO Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, pages 295–308. ACM, Jan 1996.
10. SL Peyton Jones and PL Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, pages 71–84. ACM, Jan 1993.
11. P. Trinder, K. Hammond, J. Mattson, A. Partridge, and S. P. Jones. GUM: a portable parallel implementation of Haskell. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96), Philadelphia*. ApCM, May 1996.