The Weak-Reference Interface

Marcus Amorim Leal and Roberto Ierusalimschy

Departamento de Informática, PUC-Rio
Av Marquês de São Vicente 225, Rio de Janeiro-RJ, 22453-900, Brazil
{mleal, roberto}@inf.puc-rio.br

Puc-RioInf.MCC30/04 August, 2004

Abstract: Weak references are a powerful programming language abstraction. Among other uses, they can be employed as a finalization mechanism with the advantage of circumventing several difficulties associated with regular finalizers. Nevertheless, weak references are still modestly known by programmers in general, and have received erratic support by garbage-collected language implementations. In this paper we examine weak references' most common and relevant applications, present a survey on how this abstraction is supported by actual programming languages, and discuss weak-reference semantics and implementation alternatives.

Keywords: memory management, garbage collection, finalization

Resumo: Referências fracas constituem um poderoso recurso de programação. Dentre outras aplicações, elas podem ser empregadas como um mecanismo de finalização, mas evitando diversas dificuldades associadas a finalizadores tradicionais. Não obstante, referências fracas são pouco conhecidas por programadores em geral, e tem recebido um suporte inconsistente em linguagens com gerenciamento automático de memória. Neste artigo nós examinamos os usos mais comuns e relevantes de referências fracas, apresentamos um levantamento de como esta abstração é suportada por diversas linguagens de programação e discutimos vários aspectos importantes relacionados à sua semântica e à sua implementação.

Palavras-chave: gerenciamento de memória, coleta de lixo, finalização

1 Introduction

Garbage collectors free programmers from the burden and the potential errors of explicit memory management, without necessarily modifying a programming language basic semantics. However, in a few situations, applications demand more flexibility than that provided by a totally transparent garbage collector. To handle such requirements many languages are extended with some sort of facility that provide an interface to the garbage collector. Typically, this is represented by *finalizers* and *weak references*.

Weak references, also known as *weak pointers* or *virtual references*, are a special kind of reference that does not prevent an object (the *referent*) from being reclaimed by the garbage collector. Many garbage-collected language implementations, at least as early as the mid 1980s [17, 22], have supported forms of weak references.

One of the original motivations behind this early adoption was the inherent difficulty associated with the disposal of cyclic references in reference-count garbage collectors [6]. The use of weak references in complex data structures allowed programmers to avoid memory leaks.

More recently, language designers have shown a renewed interest in weak references as an elegant alternative to gain some control over the interaction between user-level applications and the garbage collector. Since the late 1990s many programming languages adopted some variant of this mechanism. In spite of that, there does not seem to be much agreement on a standard programming interface for weak references, and language implementations display significant semantic differences. This contrasts with finalizers, which are more uniformly supported by programming languages, especially object-oriented ones. Moreover, finalizers' semantics have been thoroughly discussed [4, 12, 8, 5], while weak-reference semantics have not.

In this paper we try to fill such gap by discussing several important issues related to weak references that, as far as we know, have not been explored by other authors. Section 2 examines weak references' most common and significant applications, which are illustrated with a few examples. Section 3 presents a survey on how weak references are supported by actual programming languages, with a focus on more recent implementations. Section 4 discusses weak-reference semantics, implementation issues, and their main implications. Finally, Section 5 presents our conclusions.

2 Applications

Weak references, if adequately supported, are a valuable language facility that can solve many common design problems. Among weak references' most relevant applications, we highlight the following:

- **Memory Leak Avoidance** - Pure reference-count garbage collectors are unable to reclaim cyclic data structures. As already mentioned, this difficulty can be easily overcome by partially replacing regular references with weak ones, in such a way that any cycle is composed of at least one weak reference.

  Likewise, weak references can enable garbage collection in a less common situation: applications that use proxy objects with reciprocal references to external data structures. As an example, consider a scripting language that uses a widget from a graphical user interface (GUI) library written in C. The script can represent the widget as a proxy that references the actual C object. If the widget object supports user events through a callback mechanism, it may also need to reference the proxy object to execute the registered callbacks. The problem is that this reference to the proxy object should not avoid its reclamation, which can be assured using weak references.

- **Property Tables** - Sometimes it is necessary to associate extra attributes to an object without using the object's internal states. Reasons for that are many. For example, in several applications we need strong flexibility to dynamically assign different, and perhaps unpredictable responsibilities to an individual object, but not to its entire class.

  Surely this information can be saved in some sort of external data structure, such as a hash table. But this solution has a critical flaw. If we use the object as a key in the hash table, the internal hash table reference will prevent the object from being reclaimed by the garbage collector. To correct this behavior it is just necessary to wrap the hash table key with a weak reference.

- **Caches** - Applications that make recurrent use of large data structures loaded from a device or built dynamically may improve their performance significantly by keeping these structures resident in memory. However, as a drawback, this can lead to fast memory exhaustion. Weak references provide a simple solution to implement caches that preserve data only as long as memory is not scarce.

  Consider for instance an application that uses several large database tables. After loading and processing a particular table the application can remove all references to it except for a weak one. Whenever it becomes necessary to use the same table again, the application first tries to access it through the weak reference, and only if not successful reloads the table from the database.

Another common use of caches are memoized functions [14, 15]. The computational effort necessary to process a complex function can be significantly reduced by saving its results. When the function is reinvoked with the same argument, it just returns the saved value. In a long running application the storage used by memoized functions can grow to prohibitive levels. The use of weak references in this case can transparently preserve the most recently (and probably the most frequently) accessed values without compromising memory availability.

It is important to notice that garbage collectors that use reference counting generally dispose an object immediately after its count reaches zero. As a consequence, in systems that adopt this type of collector, it is not possible to construct memory caches based solely on simple weak references.

- **Weak Sets** - Weak sets are useful whenever objects have to be processed as a group without interfering with each object's life-cycle. A common example is the *Observer* design pattern which defines "a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [9]. This notification is generally done by the subject object, which must keep references to all its observers. In a loosely coupled application, these references must not prevent the observers' collection. A straightforward solution is to store the references in a weak set.

- **Finalization** - Weak references may also be employed to convey information from the garbage collector to user-level applications, which can learn when an object becomes disposable or finalizable and execute routines associated with such events. This is called *container-based finalization* and contrasts with class instance finalizers, which implement *object-oriented finalization* [13, 15].

  As we will explore further, weak-reference finalization is more flexible than typical object-oriented finalization, and can be adopted as a language's sole finalization mechanism. In spite of that, very few garbage-collected programming languages make this choice (e.g., Glasgow Haskell).

3 Implementation Survey

In this section we review weak-reference support in several garbage-collected programming languages, with a focus on more recent implementations. We delay a deeper discussion of semantics until Section 4.

Our survey includes Java [19], C# [18], Python [20], Gnu Smalltalk [7], Perl [2], Lua [14] and Glasgow Haskell [3]. Other languages that provide weak-reference support are Lisp [10], Scheme [11] and Modula-3 [1]. Considering that these languages have a rather simple weak-reference interface, which are covered by the other implementations reviewed here, we opted not to mention them any further.

3.1 Java

Since version 1.2, the Java platform provides three different forms of weak references: *strong*, *weak* and *phantom* [16]. Each of those references is implemented by a subclass of the abstract `Reference` class, and represents progressively weaker degrees of object reachability:

- An object is *strongly reachable* if it can be reached by traversing any chain of regular references that start in a stack or global variable (the program *root-set*).
- An object is *softly reachable* if it is not strongly reachable and it can be reached from the root-set by traversing a chain of references that includes at least one soft reference.
- An object is *weakly reachable* if it is neither strongly nor softly reachable and can be reached from the root-set by traversing a chain of references that includes at least one weak reference.
- An object is *phantom reachable* if it is not strongly, softly nor weakly reachable, it can be reached from the root-set by traversing a chain of references that includes at least one phantom reference, and it has been finalized.

Soft and weak references are semantically identical. The only actual difference between them, which is entirely dependent of the implementation, is the moment when they are cleared by the garbage collector. A weak reference *will be* cleared as soon as the garbage collector determines that the referent is weakly reachable, while a soft reference *may be* cleared as soon as the garbage collector determines that the referent is softly reachable. In both cases, until the references are cleared, the referent can be retrieved.

Phantom references, on the other hand, have a less orthodox semantics. First an object can never be retrieved using a phantom reference, not even while it is still strongly reachable. Furthermore a phantom reference is not automatically cleared by the garbage collector. The referent will be reclaimed only after the program explicitly invokes the `clear` method on the reference object[1].

Needless to say, phantom references by themselves are useless. You must use them always coupled with objects from the `ReferenceQueue` helper class (which

[1] It is not very clear why the designers of Java included the requirement of explicitly releasing the phantom referent. We have no knowledge of any application that benefits from this feature.

can be used with any `Reference` subclass). The only way a program can learn that an object became phantom reachable is by registering the phantom reference with a reference queue and polling the queue. The garbage collector will enqueue the reference sometime after the referent becomes phantom reachable.

Java also provides a weak map represented by the `WeakHashMap` class. Its implementation encapsulates a `HashMap` (a regular map implementation) and a `ReferenceQueue`. When an entry is inserted in a `WeakHashMap` its key is wrapped with a weak reference and inserted in the associated `HashMap`. The `ReferenceQueue` is checked whenever one of the weak map's methods is invoked, allowing the automatic removal of any cleared weak references and the corresponding entries.

3.2 C#

The .Net Framework Class Library provides two forms of weak references, called *short* and *long* weak references, both represented by the class `WeakReference` [18]. Short references are cleared by the garbage collector as soon as it determines that the referent is weakly reachable, while long references are cleared only after the referent is finalized. For classes that do not define finalizers, short and long weak references behave identically.

Since .Net makes no restrictions on referent access, long weak references, unlike Java's phantom references, allow the resurrection of objects that have been finalized.

3.3 Python

In Python weak references are implemented by the `weakref` module, which has a fairly large API [20]. The simplest way to create a weak reference is by calling the `ref` function and passing the referent object as its parameter. Its return value is an object of type `WeakReference`. As expected, the referent can be retrieved by calling the reference object if the referent is still alive. The `ref` function has as an optional parameter, a callback that will be invoked when the referent is about to be finalized. The weak reference object will be passed as the sole parameter to the callback, and the referent will no longer be available. If more than one callback is registered for an object, all the callbacks will be called from the most recent to the oldest registered.

An alternative way to create a weak reference in Python is with the `proxy` function. This function receives the same parameters as `ref` but returns a proxy that can be used as the object itself in most situations without requiring explicit dereferencing. The only main restriction is that proxy objects are not hashable, and therefore cannot be used as hash keys. This prevents problems related to the inherent dynamic nature of weak references. For instance, if a proxy is cleared, how should this change affect its hash value?

The `weakref` module also includes two different implementations of weak maps. The `WeakKeyDictionary` class provides a map with weak keys and strong values, and the `WeakValueDictionary` class provides a map with strong keys and weak values. In both classes entries are automatically removed after the weak reference is cleared.

3.4 Smalltalk

The Gnu Smalltalk implementation includes several weak containers, like sets, maps and even arrays [7]. Weak sets in particular support all typical set operations, and set members are transparently removed soon after they are determined to be disposable.

In addition to that, any Smalltalk object can be transformed in a *weak object* with the `makeWeak` message. A weak object retains its original type, but all its attributes become weak references. The garbage collector will clear weak object's attributes as soon as it determines that the referents are disposable.

3.5 Perl

Weak references were incorporated in Perl 5.8 core distribution as a way to overcome the difficulties of disposing circular references (Perl uses a simple reference-count garbage collector) [2]. Their implementation is coupled with *hard references*, which are identical to regular references and can be created by using the backslash operator on any variable, subroutine or value. Any hard reference can be turned into a weak one by using the function `weaken()`.

The only additional member of Perl weak-reference interface is the `isweak` function, that informs if a reference has been weakened.

3.6 Lua

Lua 5.0 implements weak references as associative arrays (known as *Lua tables*) [14]. The programmer, using metadata, can create a *weak table* where only the key field is weak, only the value field is weak, or both fields are weak. If any weak field becomes weakly reachable, the corresponding entry will be removed from the table by the garbage collector.

3.7 Haskell

Glasgow Haskell (GHC) provides an interesting implementation of weak references [15]. In GHC a weak reference is represented as a key/value pair where only the key is weakly referenced[2]. Thus the value will not be collected as long as the key is strongly reachable, but may be collected as soon as the garbage collector determines that the key is disposable. While the value is alive, it will prevent the disposal of any other objects to which it may refer, with the sole exception of the key.

Another important aspect of the GHC implementation is that it allows the association of finalizer *actions* with weak references. As expected, the finalizer will be invoked after the key is cleared. If multiple weak pointers are created with the same key, the finalizers for each of these weak pointers will all be run in some arbitrary order, or perhaps concurrently, after the key becomes unreachable. Differently from Python, a finalizer can contain references to the key, which are treated in the same way as references from the value to the key: they do not prevent the key's disposal.

4 Semantics and Implementation

Despite the simplicity of the weak-reference concept, we can notice from the survey just presented that there are several semantic and implementation issues that should be carefully analyzed by language designers. In this section we explore those issues together with their respective implications.

4.1 Referent Disposal

Although a weak reference, by definition, is ignored by the garbage collector, in practice it need not be so. Memory caches built with weak references generally will perform better when the referents' disposal is deferred until memory becomes effectively scarce. But this deferral is not always appropriate, and may even hamper application performance if resources are freed too slowly. To adequately fulfill these distinct requirements, weak references can be implemented with different "grades"of weakness, each with a different disposal priority. As discussed in Section 3.1, this is the solution adopted by Java.

Generational garbage collectors can support these different types of weak references with a straightforward implementation: automatically promote "stronger"weak-reference referents to an older generation, i.e. transfer the referent to a subheap that is less frequently processed by the garbage collector, therefore delaying its disposal. Of course this task can also be left to the programmer, as long as the garbage collector supports a mechanism to explicitly promote objects.

The .Net platform, as we have seen, provides only one type of weak reference. However, it is possible to emulate Java´s soft references in this system due to

[2]In purely functional languages, side effects and global states can be expressed with the use of *monads*. The reader interested in this concept should refer to [21]. In GHC weak references are implemented by the `IO` monad.

a peculiarity of its memory management mechanism. Microsoft Common Run Time uses a generational garbage collector that promotes objects with finalizers whenever they become finalizable [18]. Therefore, to implement an extend cache it suffices to wrap objects with a proxy class that defines a dummy finalizer, and reference it with a long weak reference.

4.2 Notification Mechanism

A weak-reference API can be extended with a notification mechanism. There are two main implementation options: queues and callbacks. In the former, soon after the garbage collector determines that a referent is disposable (or that its reachability has changed), it enqueues the associated weak reference. To learn about these events the application has to explicitly poll the queue[3]. In the latter, the garbage collector instead of enqueueing the reference, arranges for the registered callback to be invoked, passing either the reference itself or the referent as a parameter.

Weak references augmented by a callback notification facility allow the definition of instance-specific finalizers, which are very similar to regular class instance finalizers. The execution of both finalizers and weak-reference callbacks face the same issues:

- In languages that employ tracing garbage collectors, finalizer and weak-reference callback invocation are generally asynchronous, and hence unpredictable. Furthermore, most implementations give no guarantees on the order in which these routines are invoked or even whether they will be invoked at all. For instance, garbage collectors that use a "conservative"tracing algorithm may retain objects after they become unreachable, and finalization routines may not be invoked for live objects before the application terminates.

- There are at least three different alternatives for finalizer and weak-reference callback execution: they may be executed by the garbage collector thread, by a dedicated thread, or even by the application thread itself. Whatever the choice, this will introduce concurrency in the system, something that is not always appropriate nor desirable. Moreover, since these routines will be executed asynchronously, this can lead to race conditions, and in extreme cases, even deadlocks [5].

Notification queues, on the other hand, can be used to implement much more complex finalization dynamics, albeit not automatically executed. The application can wait for specific conditions, including previous object finalization, and

[3]Notice that this information can be obtained without a notification service by polling every single reference of interest. Obviously this has a significantly higher processing cost.

explicitly invoke the routines responsible for the object's finalization. This differs significantly from callback invocation, and avoids the synchronization problems discussed above.

Regardless of the notification mechanism, the promptness with which objects are determined to be unreachable still depends on the garbage-collector implementation. If the notification mechanism is to be used in the release of timely critical resources, such as file handles, the time lag between an object's reachability change and the corresponding notification can become a significant constraint.

As a final remark, notification queues can be effortlessly implemented by callbacks, but emulating callbacks with the former is only possible if the language provides a sufficiently powerful concurrency abstraction.

4.3 Referent Removal and Resurrection

If a language supports finalizers, the garbage collector has to determine when an object becomes finalizable and when it has been finalized (usually the memory associated with an object can only be deallocated after the corresponding finalization routine has finished executing). Weak references can be automatically cleared before or after finalization. In the .Net platform for example, the programmer must specify when the reference should be cleared.

In some cases it can be interesting to lock the referent and do not allow its collection until the application explicitly clears the reference. *Guardians* [8] is an early proposal to control object disposal and finalization execution (including finalization ordering) similar to weak references. Registered objects are automatically moved to a special set soon after they become disposable. Finalizers are invoked by the system only after the corresponding object is explicitly released from that set[4]. Java's phantom reference is another example of a weak reference that demand the explicit release of the referent.

Referent removal is particularly relevant when weak references are used as a finalization mechanism. It is important to define, for example, if a notification callback is able to access the referent, and eventually resurrect it. This issue also emerges in object-oriented finalizer semantics, and most languages do not make any restrictions on resurrection[5].

When resurrection is performed with finalizers, objects pass through an unreachable state before being resurrected and becoming reachable again. While in this "object limbo", objects cannot interact with the user-level application, but still retain the potential to influence the future behavior of the latter. This situation is completely circumvented when resurrection is performed only with weak

[4]Notice that guardians are equivalent to a weak-reference notification queue that enqueues the referents instead of the weak references.

[5]It is curious to note that Java allows object resurrection with finalizers, but not through phantom references.

references. In this case, if an object can not be reached either by regular or weak references, it no longer matters to the user-level application. Hence, there is no "object limbo".

In order to support resurrection, garbage collectors must preserve all unreachable objects referenced by a resurrectable object, otherwise applications may enter an invalid state. This requirement will postpone object reclamation and consequently increase memory usage. On the other hand, in systems that provide lower-level routines to gauge both memory usage and availability, resurrection can be employed to partially transfer the memory management task to the user-level application. Based on memory and other state information, the application can decide if resurrectable objects should be effectively resurrected or just left to die.

Resurrection can be entirely avoided if the information needed by finalization routines is stored in a weak property table. If the objects' attributes are not required for finalizing the object itself, memory can be reclaimed as soon as the object becomes unreachable. As a negative aspect, this design breaks object encapsulation.

4.4 Weak-Reference Types

Weak references do not need to be implemented as specific types, and languages can provide mechanisms to mark any ordinary reference as weak (examples of this design are found in Perl and Smalltalk). Although this semantics may seem attractive, it can become the source of critical programming errors. Consider the following pseudo-code:

```
Object o;
Vector v = new Vector();
v.weaken(); // mark v as a weak reference
...
if (v!=null)
  v.add(o);
```

The conditional clause does not guarantee that `v` is non-null when the `add` method is invoked. For that it would be necessary to attribute the object referenced by `v` to a non-weak reference and only then perform the test on the latter. But in this case you loose transparency, and may end up with a complex code that is harder to understand. Proxy types that automatically deference a weak reference, such as supported by Python, suffer from the same problem.

4.5 Weak Containers

Any of the traditional containers typically supported by programming languages can be extended with a "weak"version, but sets and maps are by far the most common and, as we will see, cannot be effectively implemented using only basic weak references.

Weak sets hold objects only while the garbage collector has not determined that they are disposable. Usually at least two methods are provided to manipulate these sets: one to add objects to the set, and another to retrieve all the objects. Since weak sets can be easily implemented with weak maps, fewer languages explicitly implement the former.

Weak maps can be defined with any combination of weak keys/weak values, but most languages support only a weak hash table where the keys are wrapped with weak references and the values are held by regular references. In any case, after an entry's weak member is cleared, the whole entry is transparently removed from the map.

In an ideal semantics, weak maps with a strong field should avoid the collection of a referent only as long as that object can be reached by a reference path that originates outside the map. But this is not how most implementations behave. Whenever there is an internal reference cycle, for example, when two entries each have the other key as its value, all the entries in the cycle become locked. In this case the keys are reachable in a traditional sense, however since their references exist only inside the weak map, it would be convenient if the referents were considered garbage.

Even when there are no cycles, entry collection can take longer than generally expected. If a weak field becomes unreachable, but the associated field is strong and contains a reference to another weak field in the map, the garbage collector may determine that the referent entry is also collectable only in the cycle following that in which its reference was collected. A map with n chained entries will take at least n garbage-collector cycles to have all its entries referents disposed. This may be useful for creating an n-level cache, but can lead to memory exhaustion even though weakly reachable objects have not been disposed. Lua weak tables for example behave in this manner. In Java things get even more complicated, because not only the garbage collector must run n times, but between each run the weak map has to be explicitly accessed by the user-level application to trigger the internal entry removal routine.

A possible solution to collect unreachable weak map entries is to replace strong references by a new reference type called *ephemerons*[13]. Ephemerons can be viewed as semi-strong references. They behave like weak references when pointed to objects that are internally referenced by the map, but as strong references in any other case. With this solution any map entry that cannot be reached through an external reference will be deemed unreachable, and eventually will be removed. Glasgow Haskell is the only implementation that we know that employ this kind of design.

5 Conclusion

Many applications require some kind of object topology information similar to that gathered by garbage collectors. Therefore, it is natural to extend the automatic memory management mechanism with an interface that allows user-level notification, and perhaps some control over objects' disposal. In many languages this interface is composed by two distinct facilities: weak references and finalizers.

Weak references are useful in several applications. In particular, when coupled with a notification facility, they can serve as a finalization mechanism. Weak-reference callback notification semantics is equivalent to finalizers', but with two advantages. First, the finalizable object can be reached by the application at any time before the finalization routine is actually invoked. Second, the finalization routine can be decoupled from the finalizable object, enabling a swifter memory reclamation. Notification queues, in addition to these advantages, can avoid many of the synchronization issues associated with finalizers' execution.

Despite their usefulness and expressive power, weak references are still modestly known by programmers in general, and thus have been underexplored. Even among language designers there seems to be no common view on how to support this abstraction, and language implementations certainly reflect that.

The issues discussed in this paper can serve as a relevant reference for further investigations on weak references, and also help to guide actual implementations.

Referências

[1] Critical Mass Modula-3 (CM3) 5.1 Documentation. http://www.elegosoft.com/cm3/doc, 2001.

[2] Perl 5.8.0 Documentation. http://www.perldoc.com, 2004.

[3] The Glasgow Haskell Compiler User's Guide, version 6.2. http://www.haskell.org/ghc, 2004.

[4] Martin C. Atkins and Lee R. Nackman. The active deallocation of objects in object-oriented systems. *Software – Practice and Experience*, 18(11):1073–1089, November 1988.

[5] Hans-J. Boehm. Destructors, finalizers, and synchronization. In *Proceedings of the ACM symposium on Principles of Programming Languages*, pages 262–272. ACM Press, June 2003.

[6] David R. Brownbridge. Cyclic reference counting for combinator machines. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

[7] Steven B. Byrne, Paolo Bonzini, and Andy Valencia. GNU Smalltalk User's Guide. http://www.gnu.org/software/smalltalk/gst-manual/gst.html, August 2002.

[8] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 207–216. ACM Press, 1993.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison Wesley, 1995.

[10] Bruno Haible and Sam Steingold. Implementation notes for GNU CLISP. http://clisp.cons.org/impnotes/index.html, 2004.

[11] Chris Hanson. MIT Scheme reference. http://www.swiss.ai.mit.edu/projects/scheme/.

[12] Barry Hayes. Finalization in the collector interface. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *LNCS*, pages 277–298. Springer-Verlag, September 1992.

[13] Barry Hayes. Ephemerons: A new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 176–183. ACM Press, 1997.

[14] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2003.

[15] Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages, 11th International Workshop*, volume 1868 of *LNCS*, pages 37–58. Springer-Verlag, 2000.

[16] Monica Pawlam. Reference objects and garbage collection. Sun Developer Connection, August 1998.

[17] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T manual*. Yale University Computer Science Department, fourth edition, January 1984.

[18] Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[19] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.0: API Specification*, 2002.

[20] Guido van Rossum. *Python Library Reference*, 2.3.3 edition, December 2003.

[21] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

[22] Xerox Palo Alto Research Center (PARC), Palo Alto, CA. *InterLISP Reference Manual*, October 1985.