Designing efficient and safe non-strong references in Eiffel with parametric types

Frederic Merizen* Olivier Zendra†

Dominique Colnet‡

Emails: {merizen,zendra,colnet}@loria.fr

LORIA / INRIA-Lorraine
615 Rue du Jardin Botanique
BP 101
54602 VILLERS-LES-NANCY Cedex
FRANCE

September 29, 2004

Abstract

In this partial report, we present the current state of our work in 2004 within the SmartEiffel project on the design and implementation of several kinds of object references. We introduce the known concept of weak references, reminding how this peculiar kind of references can be used to optimize and fine-tune the memory behavior of programs, thus potentially speeding up their execution. We show that genericity (parametric types in Eiffel) is the key to implementing weak references in a statically-checked hence safer and more efficient way. We compare our solution for weak references to similar notions in other languages and stress the advantages it offers. We present practical examples to support our claim that weak references can be both safe and efficient, and can help optimize programs memory-wise. We further extend our work to other kinds of references — soft references, tunable strength references and programmable references — that provide extra degrees of flexibility and correspond to actual practical needs.

Keywords: Eiffel, SmartEiffel, genericity, parametric types, safety, weak references, soft references, tunable strength references, memory management, optimization

*Henri-Poincaré University, Nancy 1 / LORIA.
†INRIA-Lorraine / LORIA.
‡University of Nancy 2 / LORIA.

Contents

1 Introduction

Weak references allow a program to maintain an object reference that does not prevent the garbage collector (GC) from considering the object as dead and from reclaiming the associated memory. There are many practical uses for weak references: caching data that is expensive to compute, recycling objects to lessen memory pressure on the garbage collector, keeping meta information about data without maintaining the data itself alive, and implementing observer/observable relationships.

Normally, when the application code references an object, the latter is considered accessible, belongs to the live set and may not be collected by the GC. This is a "normal", *strong reference* to the object. Weak references on the contrary allow the application to access the object, but do not prevent the GC from collecting it. How come this does not lead to dangling references ?

Actually, to access a weakly referenced object, the application must obtain a strong reference to the object. If the application obtains this strong reference before the garbage collector runs, then the GC may not collect the object because a strong reference to the object exists. But if the application asks for the strong reference after the object has been collected, a `Void` value[1] will be returned instead. By testing for this `Void` value, the application can know whether the object has been collected and act consequently.

In this paper, we aim at providing weak references for a high-level language, Eiffel, so that they can be easily compiled to code that is both efficient and safe. We show how *generic types* — Eiffel *parametric types* — are a great asset to reach these goals. Indeed, although similar concepts exist in various languages, our solution is the only one we know of that is based on generic types, which as we will show makes it safer and more efficient. The implementation issues we faced are explained within the context of SmartEiffel, The GNU Eiffel Compiler, which we develop at LORIA and that compiles Eiffel source code to portable C code or Java bytecode.

However, despite all the advantages of weak references, they may not offer all the possibilities and all the flexibility that may be required for fine-tuning the memory behavior of specific applications. Indeed, thay may be found too volatile (too weak) in some cases, while normal strong references are not enough (tto strong). We thus consider the problem from a more generalized point of view: providing various kinds of references with different strengths. In addition to the normal, strong references and to the weak references presented in this paper, we introduce three other kinds of references. First is the notion of soft references, which basically are weak references that *may* survive a collection, based on decisions from the GC itself. Then we design references for which the *application developer* can provide a strength indication, that is used for the GC to make it decision whether to collect or not the referenced object. Finally, (almost-) absolute control is given to the application developer with *programmable references*, for which s/he provides a heuristic method that decides the death or survival of the poined object.

Note that we do not consider in this paper special kinds of weak-reference-like objects such as ephemerons which are peculiar to solutions for the finalization problem.

[1] `Void` is the Eiffel equivalent of `null` in C, C++, Java...

This paper is organized as follows. First, section 2 presents a brief survey of existing concepts similar to weak references in various languages. Section 3 then introduces our solution for safe weak references in Eiffel, based on parametric types. Section 4 discusses and explains our design and implementation choices and how our solution generalizes. Section 5 presents our conceptual solutions for even more advanced kinds of references. Finally, section 6 concludes and opens perspectives for future work.

2 Weak-reference-like concepts in various languages

2.1 Proxy objects

In the weak reference concept we presented in the introduction, weakly referenced objects are accessed by obtaining a strong reference to them, and then working with this strong reference like with any other normal reference.

The Python language implements another, seemingly less cumbersome mechanism called *proxy objects*. Such proxies weakly reference an object, while exposing the very same interface as the object itself. All calls to the proxy's features are relayed to the proxied object if it still exists, otherwise an exception is thrown.

Python proxies are a syntactic sugar that hides one level of indirection, and is functionally equivalent to weak references. However, they are somewhat slower since the liveness of the referenced object has to be checked *each time* it is accessed. More importantly, they encourage a poor programming style by letting the developer use proxies (almost) anywhere normal objects are expected. Therefore, the introduction of a proxy in one single place can lead to unexpected exceptions being thrown in any part of the program, which makes correctness proving or even simply debugging a daunting task. Note that interestingly, Python also provides plain weak references.

2.2 Reference strengths

Weak references were made available in Java [GJS96] in the 1.2 specification through the `java.lang.ref.Reference` class and its descendants. Each Java `Reference` can be added to a queue that is used for asynchronous finalization. More interestingly, Java offers three flavors of weak references, corresponding to three different "strengths". They can be sorted in order of decreasing strength:

Soft references instruct the GC to *try and keep the referenced object alive* for some time after it ceases to be strongly reachable. Yet, objects that are only softly reachable may be destroyed, especially if they have been in that state for some time or memory becomes scarce. This kind of reference is well-suited to implement caches.

Weak references *do not keep objects alive*: if an object is found to be weakly reachable only, it is to be destroyed by the GC.

Phantom references are finalization tools, and cannot be dereferenced. They share the queue mechanism with the other two types of weak references.

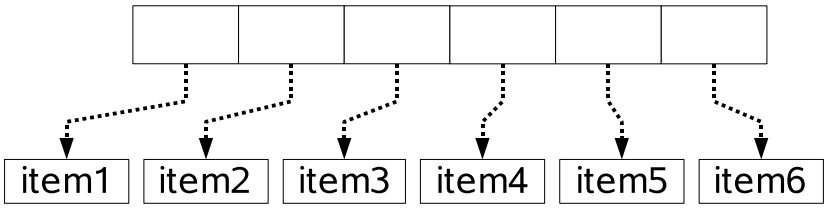


Figure 1: A weak vector

2.3 Weak collections

Weak references are typically used for groups of objects rather than isolated objects. Therefore, many languages, including Java and many Smalltalk [GR83] dialects offer one or more of the following weak collections.

Weak vectors are similar to vectors or arrays, except for the fact that their elements may be destroyed by the GC (see figure 1).

Weak key dictionaries hold key-value pairs. The key is weakly referenced, and when a key is collected by the GC, the whole pair is discarded from the dictionary. This works as if the value were strongly referenced by the key — see figure 2(a). However, it is not possible to actually add a reference inside the key, which is why the dictionary itself holds a strong reference to the value in addition to the weak reference to the key, and an extra finalization mechanism to discard the value when the key goes away — see figure 2(b).

Weak value dictionaries also hold key-value pairs, but they weakly reference the value. Of course, when a value is collected, the corresponding pair is discarded from the dictionary.

Doubly weak dictionaries hold key-value pairs that weakly reference both the key and the value. If either is collected, the pair is discarded from the dictionary.

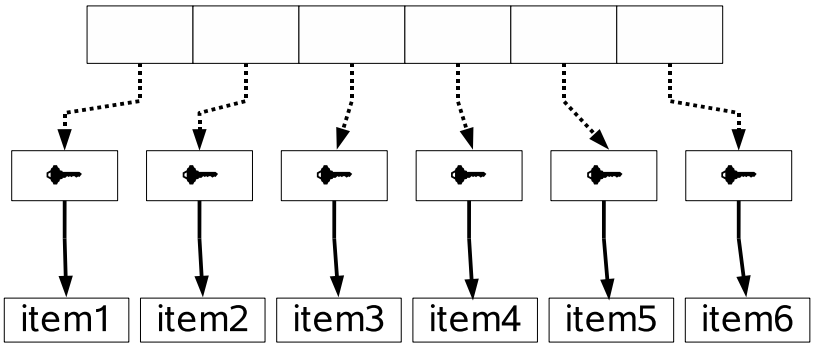
In some languages including Guile[2], weak collections are the only kind of weak reference available.

Weak vectors are ideal to implement object recycling, and weak dictionaries can be used to attach meta information to objects without keeping the objects live more than necessary.

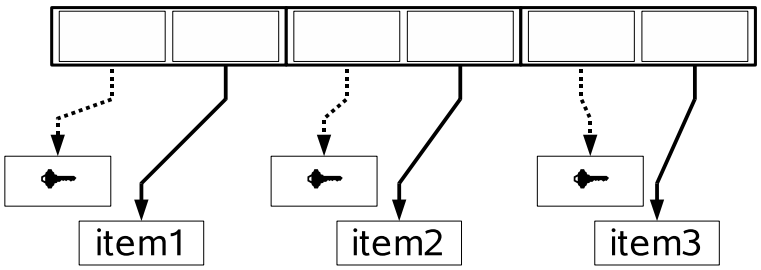
3 Designing safe weak references with parametric types

The previous section shows that the concept of weak reference is not new and has been integrated in several languages. However we think those weak references do not offer the best safety and performance possible.

[2]Guile [Fre] is the GNU implementation of the Scheme language.



(a) Concept



(b) Implementation

Figure 2: Weak key dictionaries. Plain lines represent strong references, dotted line weak references.

In the current section, we introduce our solution to better design and implement weak references. We first consider the conceptual model, then the application developer point of view, and finally the compiler and garbage collector implementation aspects, in an Eiffel compiler.

3.1 A conceptual model for weak references in Eiffel

In the model we designed for weak references in Eiffel, the garbage collector always frees the memory used by an object when it discovers this object is not strongly referenced anymore. The user shall then obtain a `Void` reference informing her/him the object has been collected and is no longer available through the weak reference. Our notion of weak reference is thus similar to Java's in this respect.

We think this is the most general and convenient kind of weak reference. Indeed, it does not alter the referenced object lifetime, which is highly desirable or even mandatory to semi-automatically manage memory and to instrument the GC. Furthermore, this kind of weak reference makes it possible to realize interesting data caches, if garbage collection is not too frequent. Finally, we consider this kind of reference as a sound basis upon which improvements such as soft references or ephemerons could be built if required.

We think there are nonetheless problems with most weak reference-like concepts, for example as implemented in Java. One of the main concerns is the total lack of type safety at compile time and the execution time cost. Indeed, in languages such as Java, a weak reference may hold any kind of object; the static type of its contents is thus `Object`. Thus, when a weakly referenced object is retrieved, it has to be cast to its actual, precise type, which incurs a cost at run-time. In our model, we address this issue by making the weak reference a *parametric type*, whose parameter is the type of the weakly referenced object. Therefore all type checks are performed at compile time instead of run-time, which is much safer and more efficient.

Note that Eiffel normally features two kinds of objects: "normal" ones, which are (strongly-) referenced objects, and so-called *expanded objects*, that is objects passed by value (see [Mey92] for details). While it obviously makes plenty of sense to *weakly* reference objects which are normally referenced, it seems to us a complete nonsense to try and weakly *reference* expanded objects that normally can only be used by value. Our model hence forbids weak references on expanded objects.

Finally, one important aspect, though not directly related to the semantic model of our weak references is that they should cost absolutely nothing when not used. Indeed, integrating a new possibility in a language and the corresponding compiler(s) should not negatively impact things when this concept is not relied on. This seems a tautology, but according to our experience is not always the case.

3.2 The user side: the `WEAK_REFERENCE` class

The interface we chose to offer weak references in Eiffel to the application developer is a very simple and convenient one: a `WEAK_REFERENCE[G]` class.

This class is used via two routines:

```
class WEAK_REFERENCE[G]
--
-- Weak reference to an object.
-- This kind of reference does not prevent the object from being
-- reclaimed by the garbage collector (in which case item returns Void).
-- Item makes it possible to get (a strong reference to) the object.
-- Inheriting from this class is prohibited.
--
creation
   set_item
feature
   item: G;
         -- Return a (strong) reference to the object
   set_item(i: like item) is
         -- Set the object to be weakly referenced
      do
         item := i
      ensure
         item = i
      end
end -- WEAK_REFERENCE
```

Figure 3: The `WEAK_REFERENCE` class.

- `set_item`, to set the weak reference so that it weakly references an object
- `item`, to query the weak reference, in order to obtain a strong reference on the weakly referenced object, or `Void` if the latter has been reclaimed by the GC

As a consequence, the typical way to access a weakly referenced object involves the following steps:

1. Get a normal — strong — reference on the object by querying the weak reference object.
2. Check this strong reference is not `Void`, otherwise nothing can be done.
3. Work with the non-`Void` reference as with any normal reference.
4. Take care not to maintain the object live with the strong reference once it is not in use anymore. If the reference is a local variable, it is automatically canceled at the end of the routine, otherwise it may be wise to set it to `Void` explicitly.

The `WEAK_REFERENCE[G]` class (shown in figure 3) is very simple, since it only contains one attribute, or instance variable, named `item`, which is seen by the developer as an argumentless function[3], and one procedure, `set_item`. Except for the uniform access to the `item` attribute, this is very similar of what can be done for other languages.

[3]This is called the uniform reference principle of Eiffel, see [Mey92].

However, this class features a fundamental peculiarity, allowed by the power of the Eiffel language: it is a *generic class*.

Generic classes are the Eiffel parlance for classes implementing *parametric types*. As can be seen from its name, the `WEAK_REFERENCE[G]` class accepts one type parameter, symbolized by the formal parameter name '`G`'. The application developer provides an actual type parameter when using the `WEAK_REFERENCE[G]` class: for example, s/he declares a variable of type `WEAK_REFERENCE[IMAGE]`, another of type `WEAK_REFERENCE[CUSTOMER]`, etc. This is much more than a syntactic shortcut: the `WEAK_REFERENCE` class may never be used alone, it does require the extra piece of information that is the type of the object it references. A `WEAK_REFERENCE[IMAGE]` is a completely different kind of object, type-wise, than a `WEAK_REFERENCE[CUSTOMER]`, as much as an `IMAGE` is different from a `CUSTOMER`. This means that even when using weak references, the program remains *fully and statically typed*.

This is a great asset for safety, since the correctness of the typing can be checked at compile time, whereas in other languages without parametric types, such as Java, a weak reference only contains an `Object`, which has to be back-cast at run-time to its actual dynamic type when the weak reference is queried. Parametric weak references also allow a better efficiency: in Java for example the cast is executed by the JVM at run-time, which incurs a cost, while our Eiffel weak references based on parametric types are compiled to the most efficient code. Indeed, each actual derivation of the `WEAK_REFERENCE[G]` class is compiled as a specific type, differently from the others. The actual type parameter thus determines the appropriate code to set and access the weakly referenced object.

As far as we know, no comparable solution for weak references based on parametric types exists, in any language.

3.3 The compiler side: impact on the garbage collector compilation

This section presents the impact of our weak reference implementation within the SmartEiffel compiler and with respect to the specialized mark-and-sweep GC[4] that is automatically generated for each compiled application. Most of the workings of the compiler and the GC fall beyond the scope of this paper; more details can be found respectively in [ZCC97, CZ99, ZC01] and [CCZ98].

3.3.1 Actually weakening the reference

The `WEAK_REFERENCE[G]` class as presented in figure 3 is not the only thing needed to have weak references in Eiffel. Indeed, when looking at its code for the `item` attribute, it seems it references quite normally — that is to say *strongly* — the referenced object and thus prevents it from being reclaimed by the GC.

To get the correct behavior and actually weaken the reference, it is of course necessary to slightly modify the GC code, which is generated by the compiler and automatically adapted to the compiled application. The modification lies in the generation of the marking functions. The function generated for the marking of `WEAK_REFERENCE`s has to be slightly different from other marking functions,

[4]Section 4.3 page 15 discusses implementation issues with other kinds of GCs.

since it must *not* propagate the marking process to its `item` attribute. Therefore, it just has to mark the weak reference object itself.

Note that this is an example of the power of code customization for optimization: the marking routine for `WEAK_REFERENCE[G]` objects is the simplest and the most efficient one, done by assigning a field a value. This very simple routine is thus an ideal candidate for inlining, which further reduces its cost.

3.3.2 `Void`-ing the reference when necessary

Beside making the weak reference actually weak, a second aspect is crucial to fully respect the semantics of the model we defined above: the reference has to be nullified, or `Void`-ed in Eiffel vocabulary, when the GC decides to reclaim the weakly referenced object. As explained in section 3.1, in this first simple version of weak references, this decision is made as soon as the collector knows the object is not longer strongly referenced.

Setting the `item` attribute of the `WEAK_REFERENCE[G]` to `Void` is performed quite logically during the sweep phase of garbage collection, after the mark phase has identified all the references and reachable objects in the system.

In the GC that SmartEiffel generates, all objects are segregated by *type* [CCZ98]. With memory chunks which are type-homogeneous, there is one sweeping routine for each type. This routine is specialized, in the sense that it knows where to find each object mark flag in the chunk, and statically knows the size separating two objects.

Setting the `item` to `Void` for `WEAK_REFERENCE[G]` implied modifying these sweeping functions to look at the mark flag of the object pointed by the `item` field. Indeed peeking at the mark flag of the object pointed by the `item` field makes it possible to know whether this object is still strongly referenced from somewhere else or not; in the latter case `item` is set to `Void` in the weak reference object. However, this modification is not as easy as the one described in section 3.3.1. Two issues complicated things: first, finding the weakly referenced object header, then understanding the status of this object, garbage-collection-wise.

Finding the weakly referenced object header In the memory layout of objects compiled by SmartEiffel, the object "header" that contains the mark flag lies *after the object* itself[5]. But `item` holds a pointer to the object that — like all object references in SmartEiffel — directly point to the *beginning of the object* payload, not to its internal bookkeeping header. Therefore, to access this header of the weakly referenced object, the sweeping function for the weak reference needs to know the size of the object to add it as a negative offset to the `item` pointer.

This is not an issue for objects which are instances of leaf types, that is types without heir. In this case indeed the object size, hence the location of the header, is known at compile time. The modification to the sweeping function for a kind of weak reference on leaf object thus relies on this size and generates a simple constant code to access the header.

However when a type is polymorphic, that is it has heirs, whose instances may have different sizes, the size of the object referenced by the `item` field may

[5] Except for resizable objects (arrays), but these are expanded objects and thus not subject to weak referencing, as explained in section 3.1.

not be known at compile time. The size thus has to be found at run-time, according to the dynamic type of the referenced object. This dynamic type is easy to find: such polymorphic types always contain a type identifier, used to resolve polymorphic calls as described in [ZCC97]. The sweeping function that is generated for a specific kind of weak reference on a non-leaf object thus has to be adapted to read this type identifier (which is always the first field of the object) and then access a table that associates to the type identifier the size of its instances. Note that this table is not only useful for weak references management, but also for the debugger.

Understanding the status of an object When the mark flag of an object that may be weakly referenced is found, its meaning is not immediate anymore, while it would be with normal objects, because now we are "peeking ahead" of the normal sweeping process.

The word that holds the mark flag in the weakly referenced object may also hold a pointer that is used for chaining free memory slots. Thus three states are possible for this word:

1. It may contain a pointer
2. It may contain `FSOH_MARKED`
3. It may contain `FSOH_UNMARKED`

The first case — a pointer — means the object has been found to be a dead object and has just been added in the free list of slots for this type[6]. In this case, the weak reference has to have its `item` field set to `Void`.

The second case — a `FSOH_MARKED` flag — means the object has been marked as live (strongly referenced) by the mark phase. The weak reference thus remains unchanged.

The third case — an `FSOH_UNMARKED` flag — is trickier. Indeed, since we are sweeping a weak reference and thus peeking at the mark flag of its `item` object, the latter may already or may not yet have been swept. If it has not been swept yet, the `FSOH_UNMARKED` flag means the object has not been marked live by the mark phase and is thus not strongly referenced anymore. As a consequence, the object is bound to be added into a free list later, when its memory chunk is swept. The weak reference thus has to have its `item` field set to `Void`. On the contrary, if the weakly referenced object has already been swept, the `FSOH_UNMARKED` flag means the object had been marked live by the mark phase, and then reset by the sweep phase to `FSOH_UNMARKED` in order to prepare it for the next mark-and-sweep cycle. In such a case, the weak reference must remain unchanged.

We thus see that knowing whether an object has already been swept is crucial when weak references are used, because of our "peek-ahead" for the `item` field. A small yet greatly effective change in the code generated by SmartEiffel made this possible, at no cost. Making sure objects chunks — hence objects — were swept monotonously, that is for example by increasing addresses, is enough. The sweeping routine generated for weak references just has to compare the address of the object pointed by the `item` field to that of the weak reference currently being swept.

[6]The free lists are rebuilt from scratch at each collection, as explained in [CCZ98].

3.4 Practical use: Caching widgets with Typed Weak References

In this section, we present a concrete example showing the actual usefulness of weak references and how our solution makes coding with typed weak reference an easy task.

This example shows how weak references can be used as caches to improve the speed of widget display. Indeed, hardware has improved a lot, yet some dialog boxes still display slowly. This can be explained by an increased complexity of modern dialog boxes, for example because of the addition of themeing and internationalization facilities to GUI toolkits, as well as features such as tabbed dialogs.

When a dialog takes time to display, one easy way to try and improve display speed is consist in caching it for future use in case there is enough memory, through a weak reference. Figure 4 illustrates this idea, for an "open file" dialog.

```
open_file_dialog_cache : WEAK_REFERENCE [DIALOG_BOX]
display_open_file_callback is
   local
      my_open_dialog : DIALOG_BOX
   do
      -- get the cached open file dialog, if any:
      my_open_dialog := open_file_dialog_cache.item
      if my_open_dialog = Void then
         -- no open file dialog cached, create one:
         create my_open_dialog.make
         -- and cache it for future uses:
         open_file_dialog_cache.set_item (my_open_dialog)
      end
      my_open_dialog.display
   end
register_callbacks is
   do
      ...
      menu_bar.add_entry_callback("Open file",
                          agent display_open_file_callback)
      ...
   end
```

Figure 4: Caching a widget through a weak reference

Of course, the same thing may be applied to a number of, or all, dialogs in an application. In such a case, several `xxx_dialog_cache` attributes would be required.

Quite logically, improving on this idea would lead to put all these cached dialogs in a kind of collection, instead of several attributes. This idea is explained by figure 5.

Note that this fosters good programming practices, such as creating dialogs through a factory — possibly loading them from a resource file —- and using the same display callback for all dialogs.

```
cached_dialogs: ARRAY [WEAK_REFERENCE [DIALOG_BOX]]
factory: DIALOG_FACTORY
display_callback (dialog_id: INTEGER) is
   local
      my_dialog: DIALOG_BOX
   do
      -- get the possibly cached dialog:
      my_dialog := cached_dialogs.item (dialog_id).item
      if my_dialog = Void then
         -- dialog not cached, create one:
         my_dialog := factory.make_dialog (dialog_id)
         -- and cache it for future uses:
         cached_dialogs.item (dialog_id).set_item (my_dialog)
      end
      my_dialog.display
   end
register_callbacks is
   do
      ...
      menu_bar.add_entry_callback("Open file",
                     agent display_callback(open_file_dialog_id))
      menu_bar.add_entry_callback("Properties",
                     agent display_callback(properties_dialog_id))
      ...
   end
```

Figure 5: Using a weak vector to cache several widgets

4 Discussing weak references

In this section, we further discuss and explain some conceptual choices we made when designing our solution for safe and efficient weak references. We first address the keyword versus library component issue, then the by value versus by reference choice. Finally, we discuss how the solution we proposed in this paper generalizes, especially to other garbage collectors and languages.

4.1 Keyword versus library component

Weak references could be provided as a language extension instead of a standard library class. However, implementing a keyword inside the parser would freeze the syntax, making it harder to experiment with weak references. Furthermore, the design of Eiffel (see [Mey92]) is focused on keeping the core language relatively small to avoid unexpected interactions. Finally, if other types of versatile references (soft references, etc.) were to be added, a keyword for weak references would imply either other keywords, or a kind of sub-language to distinguish them all. A new class thus seemed less invasive, more flexible and more scalable.

4.2 Expanded or referenced weak reference

Since `WEAK_REFERENCE[G]` are Eiffel objects, they apparently could be either expanded or referenced — this boils down to a pass by value versus pass by reference choice. We decided our weak reference objects would be normal, referenced objects, not expanded ones. Let's see the reasons and implications of this choice.

4.2.1 Ease of use

Weak references are more often found in arrays than alone — just think of caches (see section 2.3 page 5). Therefore, it is crucial that arrays of weak references work seamlessly.

However, if `awr` is an array of *expanded* weak references, a call to `awr.item(i)` returns a *copy* of the weak reference found in the i^{th} position of `awr`. It follows that the natural Eiffel idiom for changing the object that is weakly referenced by a weak reference in an array, `awr.item(i).set_item(another_object)`, does not work for expanded weak references. Indeed, it makes a copy of the weak reference, and then applies `set_item` to this copy, instead of changing the reference that is stored in the array.

The correct way to change the object that is weakly referenced by a weak reference in an array is to use an auxiliary weak reference that is set to point to the wanted object and then is put into the array:

```
aux_weak_ref.set_item(another_object)
awr.put(aux_weak_ref, i)
```

This solution is quite cumbersome and counterintuitive, which makes expanded weak references very error-prone, all the more so as the solution that does not work is syntactically correct and does not trigger a compiler error or a warning.

Ease of use thus favors using normal, referenced `WEAK_REFERENCE[G]` objects, instead of expanded ones.

4.2.2 Efficiency

When compared to expanded objects, referenced objects incur a memory penalty of two machine words: one word for the pointer to the object, and another word for the object header. Weak references are tiny objects since they consist of a single pointer and only weight one machine word. So making them referenced objects implies a memory overhead of 200 %. Nonetheless, we expect weakly referenced objects to be fairly large, otherwise there would be no point in *weakly* referencing them. Therefore, the memory overhead should be negligible when taking into account the weakly referenced object.

4.2.3 Correctness

In SmartEiffel, expanded local variables are stored on the stack or in processor registers rather than in the heap. This is where the conservative part of SmartEiffel's semi-conservative GC [CCZ98] kicks in, and treats every word in the stack that seems to be a pointer to an object like an actual reference, by marking the referenced object as live. Since weak references contain an `item`

pointer, if one ends up in the stack, the object pointed by `item` will be marked... hence *strongly* referenced. While this misidentification may not cause any faulty program behavior, it would increase memory usage and negate the weak aspect of weak references that are put onto the stack. This would include all weak references declared as locals, or declared as attributes of expanded locals, and all weak references passed as arguments. All this seems a bit restrictive.

Of course, we could try to conceal the pointer in the weak reference from the GC. For example, a pointer might be effectively disguised by XORing it with a well-chosen bit pattern. But then, we would have to decide what to do when an object gets collected while it is referenced by a weak reference from the stack. The normal thing to do would be to `Void` the `item` field of the weak reference, but this is impossible.

Indeed, since SmartEiffel's GC currently has no knowledge about the types of the objects that lie in the stack — which is why it has to be semi-conservative — it is unable to decide whether a word that looks like a weak reference is actually one or not. Setting them all to `Void` would lead to critical errors, modifying words in the stack that were actually not weak references. The opposite policy also fails, because not setting to `Void` the weak references means that subsequent calls to `item` on them would return dangling pointers. Hiding weak pointers from the collector is thus not possible with a conservative algorithm.

To sum it up, expanded weak references can be implemented correctly, but at the cost of a "reduced weakness" caused by misidentifications for weak references found in the stack. On the contrary, referenced weak references are fine, since they live in the heap, where the GC is non-conservative (a.k.a type accurate).

4.3 Generalizing our solution

Although we rely on a semi-conservative mark-and-sweep collector in the SmartEiffel Eiffel compiler, our solution for weak references is also applicable to other kinds of collectors, and to other object-oriented languages — provided they feature parametric types.

In section 3.3.1 page 9, we detailed the modification to mark-and-sweep have it actually weaken the weak reference. Of course, this also applies almost verbatim to other kinds of collectors that rely on a marking phase followed by a kind of reclaiming phase, such as mark-and-compact collectors and copying ones. However, if a non-marking GC is relied on, for example a reference-counting algorithm, things change a bit. There, two semantics for weak references may be envisioned.

In the first semantics, the appropriate modification to actually weaken the reference consists in changing the handling of pointer assignments — including argument passing — so that the reference counter of an object is *not* changed when this object is assigned to the `item` attribute of a weak reference, and of course not changed either when `item` is reset to `Void`. This leads to very weak references, that may be used for aliasing (or sharing) and caching. For example, in the case of a log file 'shared' by several writers, it is important to open the file only once at a time, which is done by a a single common factory object. The latter holds a weak reference to the file and provides strong references to the writers. Thus, as soon as the file is not used by any writer anymore, hence not strongly referenced anymore, its reference counter becomes zero — since

the weak reference of the factory is not counted — and the file is immediately finalized (i.e. closed) and collected by the GC. However, these references are so weak that writing the following code

```
create my_weak_reference.set_item(create SOMETHING.make)
my_something := my_weak_reference.item
```

is meaningless since `my_weak_reference.item` always returns `Void`, because the reference counter of the newly created `SOMETHING` object remains zero.

A second semantics may thus appear more useful. It takes advantage of the fact that most reference counting collectors have a backup collector — often a mark-and-sweep — in order to reclaim cycles[7]. The situation thus becomes very similar to our solution based on mark-and-sweep. In this case, the weak reference behaves exactly like a strong reference — increasing and decreasing the object reference counter — except when a 'cleansing' (or 'backup') collection is triggered. There, all the reference counters are reset to zero and only the strong references are counted at mark-time. Then, all the weak references are swept (which is easier if the GC segregates them by type, as mentioned in section 3.3.2) and the item they reference is peeked at. If its reference counter is zero, the weak reference is collected since no strong reference maintains the object live; on the contrary, if the object is strongly referenced, each weak reference pointing to it increases its counter by one. Because backup collections occur infrequently, both examples shown with the first semantics work in a useful way with this second one.

In section 3.3.2 page 10, we addressed the issue of setting the `item` attribute of the weak reference to `Void`. Once again, for collectors that comprise a kind of reclaiming phase going throughout all the objects, this phase is used like the sweep phase of our mark-and-sweep and the modifications to add weak references are similar to the ones we did. However, with a collector that does not work with a reclaiming phase, such a reference-counting collector, setting the `item` field of the weak reference to `Void` can be significantly more complex.

Normally, in such a collector, an object can be reclaimed as soon as it is becomes dead, that is when its reference counter reaches zero. If this is the case, all weak references that point to this object should have their `item` attribute set to `Void`. But this would imply knowing them, which can be a problem. One way to do this could be to have a back pointer in the weakly referenced object, pointing to some list of weak reference objects pointing to it. Maintaining this list could be quite costly and burdensome, in addition to the memory overhead the list would imply, as well as the back pointer in the weakly referenced object. Using a hash table to handle these lists could avoid the use of a back pointer in each weakly referenceable object, but would still be quite expensive.

Another possibility would be to delay the actual reclaiming of the weakly referenced object when its reference counter reaches zero. In this case, the weak reference should update itself, when its `item` is queried. The algorithm for its `item` query then becomes: when `item` is already `Void`, return it; otherwise if the reference counter of the weakly referenced object is zero, set `item` to `Void` and return it; otherwise, return `item` that actually references a live object. In order to work, this also implies relying on a periodic mark-and-sweep cycle — which is anyway present in the reference counting collector because it is needed to

[7] See [JL96] for a detailed explanation of reference counting and backup algorithms.

reclaim cycles, that can not be detected by pure reference counting (see [JL96] for details). All the objects whose reference counter is zero would be actually reclaimed during this mark-and-sweep phase, and all the weak reference object updated like in our simple mark-and-sweep collector. This process is thus rather similar to what we do in our solution, except that it would imply a greater cost when accessing weak references and a delay for the reclaiming of memory.

Our solution for weak references is thus quite generalizable to other GCs, although reference counting collectors make it a bit tricky and less efficient.

We explained in section 4.2.3 page 14 how the semi-conservative nature of the mark-and-sweep GC in the SmartEiffel compiler influenced our decision to avoid expanded weak references and go for normal referenced objects. Note that this makes our solution easily generalizable to all the object-oriented languages that offer referenced objects, which means probably all of them.

If the GC were a non-conservative one, even for local variables, our choice for referenced weak references would still be valid. Indeed, removing the conservative scanning of the stack would simply make it possible for us to implement weak reference objects as expanded ones, but would add no extra constraint. Making the decision to go for expanded weak references would thus have to be balanced again against the arguments of sections 4.2.1 and 4.2.2 for ease of use and efficiency. Consequently, our solution for weak references does neither require a language with expanded types nor a semi-conservative GC.

5 The need for more advanced kinds of references

Quite obviously, the normal strong references are rather fundamental in object-oriented languages. In addition, we showed in section 3 that weak references were also quite useful, solving concrete issues, and that they could be designed and implemented in a safer and more efficient way thanks to parametric types.

However, we claim these references still do not fulfill all the needs of application developers who want a fine control over their application memory behavior. Indeed, going back to the example in figure 5 and scrutinizing it reveals it is far from being perfect. Indeed, since all weak references are treated equally by the garbage collector, when none is strongly referenced, they either all survive or are all collected. It thus seems interesting to handle them with a finer granularity, collecting some of them and not the others, which would improve the reuse rate.

We consider that this specific need, and other ones as well, may be better — and probably completely, in practise — addressed by providing other, more flexible, references, besides strong and weak ones. In what follows, we therefore present `SOFT_REFERENCE[G]` , a variation of `WEAK_REFERENCE[G]` (section 5.1 page 17), then on section 5.2 page 18 the more application developer oriented `TUNABLE_STRENGTH_REFERENCE[G]` and `PROGRAMMABLE_REFERENCE[G]` (section 5.3 page 25).

5.1 Soft references

One first improvement for the code of the example of figure 5 consists in using a different kind of references, `SOFT_REFERENCE[G]` . From a syntactic point of view, these would not change anything to the code in figure 5 besides the replacing of `WEAK_REFERENCE[G]` by `SOFT_REFERENCE[G]` . The novelty would

lie in the semantic of such references: SOFT_REFERENCE[G] behave exactly like WEAK_REFERENCE[G] , except that when the GC triggers, it may decide to reclaim only *some* SOFT_REFERENCE[G] . This decision would have to be made by the GC itself, according to some heuristics and the memory status of the system.

To the application developer, these SOFT_REFERENCE[G] bring only improvements over WEAK_REFERENCE[G] : they are as easy to use — the developer has to care about very little besides checking the Void value — and provide a higher chance of reuse for cached objects.

The burden is in fact put onto the GC designers — us — who has to implement the appropriate heuristics. The problem is that no heuristics is good for all situations. A natural solution could lie in several heuristics coded by the GC designers in the GC code, and let the application developer choose amongst them. However, this solution is far from being perfect. First, the burden put on the GC designers would be increased, since we would have to code not only one, but in fact several heuristics, or even many if we want to provide a good coverage of all possible needs. For example, the ideal heuristics for a given application could take into account one or several of various factors, in different ways: overall memory footprint, memory footprint for one type of objects, frequency of use of some type of objects or some objects in a type, object age, object (re-)creation cost, etc. Having all useful criteria interact in the correct way could be very tricky for the GC designers. Furthermore, another issue consists in presenting the application developer with an understandable and easy to use interface to specify the heuristics s/he wants.

It thus seems much more reasonnable to provide the application developer — who knows the application and its specifics — with ways to better express how the heuristics should work. Section 5.2 shows a first, simple, but incomplete mechanism to do this, while 5.3 presents a heavier mechanism where the application developer is given full control of the heuristics, in case the previous mechanism is not sufficient.

5.2 Tunable strength references

The first mecanism to give the application developer more control over the way the GC frees softly referenced objects consists in having a third kind of references, namely TUNABLE_STRENGTH_REFERENCE[G] .

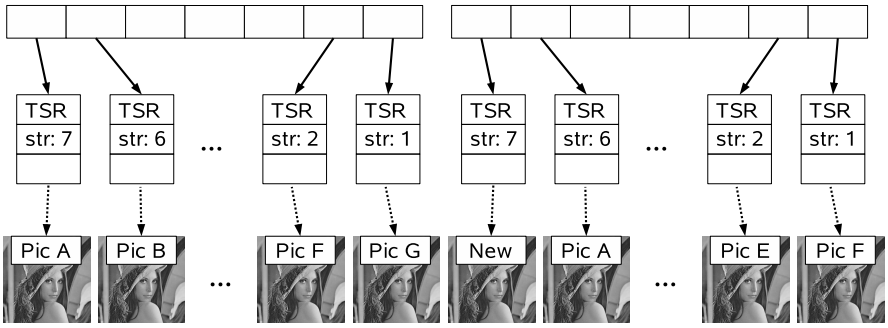
As its name implies, and as figure 6 shows, TUNABLE_STRENGTH_REFERENCE[G] are simply SOFT_REFERENCE[G] whose "strength" may be changed by the application developer. This makes it possible to provide *an ordering* between soft references, thus indicating which ones should be collected first and which ones are of greater value and should be kept as much as possible.

Note that this class makes it possible both to set a reference strength at creation time and to change it during execution. This is intended to give the application developer more flexibility, since the usefulness of a softly referenced object — hence the strenth of the referenced — could vary across several phases of the program. This also makes it easier to reuse the TUNABLE_STRENGTH_REFERENCE[G] objects themselves, even to reference new objects that have different strengths than the previously referenced ones.

Let us exhibit an example showing how these TUNABLE_STRENGTH_REFERENCE[G] come in handy, in a simplified image viewer program. Here, when image loading

```
class TUNABLE_STRENGTH_REFERENCE[G]

-- Versatile reference to an object, whose reference strength can be
-- queried and set.
--
-- This kind of reference does not always prevent the object from
-- being reclaimed by the garbaged collector (in which case item returns
-- Void). Objects referenced with the lowest strengths are more likely
-- to be collected than objects referenced with a higher strength.
-- Collected objects are collected in order of increasing strength.
-- 'item' makes it possible to get (a strong reference to) the object.
-- Inheriting from this class is prohibited.
--
-- When referenced by several (kinds of) references, the 'item' object
-- may be reclaimed by the garbage collector if and only if *all* the
-- references allow its collection.

insert REFERENCE_STRENGTHS
   -- to inherit strength constants and related convenience routines

creation  set_item_with_strength

feature {ANY}
   set_item (i: like item) is
      ensure
         item = i
      end
   set_item_with_strength(i: like item; s: like strength) is
      require
         is_valid_strength(s)
      ensure
         item = i
         strength = s
      end
   item: G
   strength: INTEGER_8
         -- Between 0 and 15
         -- with symbolic equivalents Min_xxx and Max_xxx
   set_strength(s: like strength) is
      require
         is_valid_strength(s)
      ensure
         strength = s
         (item = old item) or else (item = Void)
      end
end -- class TUNABLE_STRENGTH_REFERENCE[G]
```

Figure 6: Schetch of the `TUNABLE_STRENGTH_REFERENCE[G]` class.



(a) Initial state

(b) After loading a new image

Figure 7: Image cache using `TUNABLE_STRENGTH_REFERENCE[G]` .

and decoding is expensive, it appears quite natural to keep a cache of several previously displayed images, to save time for example when the user want to go back to previously seen pictures. When objects have to be discarded from the cache to save memory, it it quite obvious that an LRU algorithm is the correct way to proceed, freeing first the oldest images and trying to keep the most recent ones, which are more likely to be reused. This is illustrated by figure 7, whose upper part 7(a) represents the memory state of the cache before an image is loaded and whose lower part 7(b) shows its state after loading a new image.

An array of weak references as in figure 5 page 13, or even of soft references, would not do the desired job. On the contrary, an array that contains `TUNABLE_STRENGTH_REFERENCE[IMAGE]` is very adequate. Figure 8 details the code that initializes the cache. There, `image_cache` contains references of different strengths, able to hold `IMAGE` objects. The reference with the highest strength is used to hold the most recent image (at the lower index of the array), while the reference with the lowest strength contains the oldest image (upper index of the array). This appears clearly when looking at the `prepare_image_cache_array` routine whose role is to create the (empty) `TUNABLE_STRENGTH_REFERENCE[IMAGE]` objects and fills the `image_cache` array with them, ready to be used to cache images.

Figure 9 features a code excerpt showing how the cache is used. When a new image is loaded, in routine `load_new_image`, it becomes the most recent image in the cache. The latter is shifted right to make room for the new image, and discards the oldest one. Note that only the `item` fields of the `TUNABLE_STRENGTH_REFERENCE[IMAGE]` objects are changed in this example, the `strength` fields remain unchanged throughout the program execution.

When the cache is used to access in `get_previous_image` the previously displayed image, two cases may occur. First, if the previous image has not been collected by the GC, it is directly available. The cache just has to be shifted left to make the previous image the current one and so on[8]. On the contrary, if the previous image has been collected by the GC, it has to be reloaded from a file.

[8]This may not be optimal for a real image cache, but illustrates our point simply.

```
image_cache : ARRAY [TUNABLE_STRENGTH_REFERENCE [IMAGE]]
  -- most recent image at index 'lower'; older at index 'upper'
  -- strength of references decreases from 'lower' to 'upper'

prepare_image_cache_array is
   require
      enough_strengths: Max_number_of_cached_images <=
          (Maximum_reference_strength - Minimum_reference_strength + 1)
   local
      index, strength: INTEGER ; tsr: TUNABLE_STRENGTH_REFERENCE
   do
      create image_cache.make(1, Max_number_of_cached_images)
      from
         index := image_cache.lower
         strength := Maximum_reference_strength
      until index > image_cache.upper
      loop
         create tsr.set_item_with_strength(Void, strength)
            -- Void since no image is referenced yet
         image_cache.put(tsr, index)
         index := index + 1
         strength := strength - 1
      end
   end
```

Figure 8: Preparing an image cache based on an LRU algorithm and `TUNABLE_STRENGTH_REFERENCE[G]` .

```
load_new_image: IMAGE is
   local
      new_image: IMAGE ; index: INTEGER
      tsr, weaker_tsr: TUNABLE_STRENGTH_REFERENCE
   do
      from index := image_cache.upper - 1
      until index < image_cache.lower
      loop
         tsr := image_cache.item(index)
         weaker_tsr := image_cache.item(index + 1)
         weaker_tsr.set_item(tsr.item)
         index := index - 1
      end
      new_image := load_image_from_file(.....)
      tsr := image_cache.item(image_cache.lower)
      tsr.set_item(new_image)
      Result := new_image
   end

get_previous_image: IMAGE is
   local
      image: IMAGE ; index: INTEGER
      tsr, weaker_tsr: TUNABLE_STRENGTH_REFERENCE
   do
      tsr := image_cache.item(image_cache.lower + 1)
      image := tsr.item
      if (image /= Void) then -- cached image there, make it current
         from  index := image_cache.lower
         until index >= image_cache.upper
         loop
            tsr := image_cache.item(index)
            weaker_tsr := image_cache.item(index + 1)
            tsr.set_item(weaker_tsr.item)
            index := index + 1
         end
         weaker_tsr.set_item(Void)
      else -- previous not there, all cached images have been collected
         image := load_image_from_file(.....)
         tsr := image_cache.item(image_cache.lower)
         tsr.set_item(image)
      end
      Result := image
   end
```

Figure 9: Using an image cache based on an LRU algorithm and `TUNABLE_STRENGTH_REFERENCE[G]` .

Furthermore, since in this example we assume that when an image is put into the cache no other reference to it exists anymore, the fact the previous image has been collected implies that all older images also have. Indeed, the objects referenced by `TUNABLE_STRENGTH_REFERENCE[G]` are always collected in order of increasing strength, and in our cache the previous image is referenced by the strongest `TUNABLE_STRENGTH_REFERENCE[IMAGE]`.

A second example shows how useful `TUNABLE_STRENGTH_REFERENCE[G]` are, but this time the application developer *dynamically changes the strengths of the references.*

Instead of having an image cache that loses the last image seen when going back (see the loop of the `get_previous_image` function of figure 9), let us design a smarter cache that keeps track of (a certain number of) loaded images and allows a backward *and forward* mavigation through them.

The creation code remains unchanged compared to figure 8 page 21. However, the code to use this advanced cache changes, as detailed in figure 10. First, a `current_image_index` is required, that keeps track of the index in the cache of the currently displayed image. Most of the time, this index is equal to `image_cache.lower` as in the previous example. But when previous images are navigated to, this index moves towards `image_cache.upper`. The invariant for this version of the cache is that the `TUNABLE_STRENGTH_REFERENCE[IMAGE]` object pointed by `current_image_index` has the `Maximum_reference_strength`, while others have a strength that lowers as the distance from `current_image_index` increases.

The code for the `get_previous_image` function explains how this index is used and how the cache works.

When `current_image_index` is already at the last index of the cache — no more cached image in the backward direction — the cached image it points to is simply replaced by reloading from disk the one that had been displayed before. Neither the index nor the strengths are modified, but all cached images are shifted left, to make room for the reloaded image at the upper index. The `load_image_for_upper_index` does this, and is thus almost identical to `load_new_image` of figure 9.

But in the general case there are indexes available corresponding to (potentially still cached) previously displayed images. `current_images_index` thus moves right one step, which implies updating all the strengths of the `TUNABLE_STRENGTH_REFERENCE[IMAGE]` objects in the cache to maintain the cache invariant ("the further from `current_images_index`, the lower the strength"). So when going right in `image_cache`, the strengths for the objects of type `TUNABLE_STRENGTH_REFERENCE[IMAGE]` have to be decreased for those ranked before `current_image_index` and increased for objects placed after it. This clearly appears in the `else` part of routine `get_previous_image`. Note that only the strengths are changed, the cached images are not moved.

The `get_previous_image` function is trivially symetrical to `get_next_image`.

`TUNABLE_STRENGTH_REFERENCE[G]` are thus a first mechanism that provides the application developer with more control over which soft references are collected first and which ones survive longer. It should be powerful enough for most needs, with its simplicity being an extra incentive to use it. However, in some cases the application developer may want even more control. The following 5.3 section provides another solution.

```
current_image_index: INTEGER

get_previous_image: IMAGE is
   local
      image: IMAGE ; index: INTEGER
      tsr: TUNABLE_STRENGTH_REFERENCE
   do
      if (current_image_index = image_cache.upper) then
         -- already at the last cached image
         load_image_for_upper_index
         tsr := image_cache.item(current_image_index)
         -- strengths are ok
         image := tsr.item
      else -- get the previous image
         current_image_index := current_image_index + 1
         tsr := image_cache.item(current_image_index)
         image := tsr.item
         if (image = Void) then
            -- image not cached, load and cache it
            image := load_image_from_file(.....)
            tsr := image_cache.item(current_image_index)
            tsr.set_item(image)
         end
         -- Update the strenghts, since current_image_index changed
         from  index := image_cache.lower
         until index >= current_image_index
         loop
            tsr := image_cache.item(index)
            tsr.set_strength(tsr.strength - 1)
            index := index + 1
         end
         from  index :=  current_image_index
         until index > image_cache.upper
         loop
            tsr := image_cache.item(index)
            tsr.set_strength(tsr.strength + 1)
            index := index + 1
         end
      end
      Result := image
   end
```

Figure 10: Using an image cache based on an LRU algorithm and `TUNABLE_STRENGTH_REFERENCE[G]` .

5.3 Programmable references

The best way to give the application developers full control over which soft referenced objects are collected and which are not is simply to let them *code themselves their own heuristics* and rules. This also has the advantage of lightening the burden on the GC designers.

One convenient way to do this consists in having a new type of soft references, namely `PROGRAMMABLE_REFERENCE[G]` (see figure 11). The latter is an abstract class containing an abstract boolean function, `item_reclamation_allowed`. This function is intended to be called by the GC in order for it to know what to do of the referenced object. When a specific heuristics is required to control how some soft references are collected, a new class has to be created that inherits from `PROGRAMMABLE_REFERENCE[G]` and provides the appropriate heuristics through an actual implementation for the `item_reclamation_allowed` function. When several heuristics are needed, several heirs of `PROGRAMMABLE_REFERENCE[G]` have to be written. The developer is also left the option to make these heirs parametric classes or not.

The following figures 12, 13 and 14 illustrate in the context of an image cache how `PROGRAMMABLE_REFERENCE[G]` can be used.

Figure 12 page 27 presents class `IMAGE_CACHE_REFERENCE`, which is an heir of `PROGRAMMABLE_REFERENCE[IMAGE]` and provides its own heuristics to manage soft references to images. Here, an image may be reclaimed when the overall memory footprint is too high (above the predefined `Max_total_footprint` constant). It may also be reclaimed when the image is too old, unless the total memory footprint of cached images is too low. Note that the age of each cached image, in number of garbage collections survived, is kept in `item_age`, an attribute (or instance variable) associated to each `IMAGE_CACHE_REFERENCE`, while the total memory footprint of all cached images is held by `total_image_footprint`, a `once` attribute (or class variable) shared between all instances. `item_age` is kept up-to-date by the application developer thanks to the `item_not_reclaimed` callback, used by the GC to notify that it has decided not to collect the referenced object, and thanks to the `item` routine that resets the age to zero when a strong referenced is obtained on the object (thus making it possible to know which cached images were most recently used). Similarly, `total_image_footprint` is updated when the GC informs the `IMAGE_CACHE_REFERENCE` that its item is about to be collected.

This `IMAGE_CACHE_REFERENCE` class makes it easy to implement the image cache we mentioned. Figure 13 page 28 shows how simply the application creates the image cache. Note that this creation is almost identical to the creation of an image cache based on `TUNABLE_STRENGTH_REFERENCE[IMAGE]` (see figure 8 page 21), except there is no need with `IMAGE_CACHE_REFERENCE` to take care of any reference strength.

Figure 14 page 28 illustrates the use of `IMAGE_CACHE_REFERENCE`. No management of the cache semantics takes place in the `get_previous_image` routine, since all the heuristics are coded directly into `IMAGE_CACHE_REFERENCE`.

```
deferred class PROGRAMMABLE_REFERENCE[G]

-- Versatile reference to an object, whose life and death are determined
-- by a heuristics provided in 'item_reclamation_allowed'.
-- This class is intended to be insert-ed into actual programmed
-- references.
--
-- This kind of reference does not always prevent the object from
-- being reclaimed by the garbaged collector (in which case item returns
-- Void).
-- Item makes it possible to get (a strong reference to) the object.
--
-- When referenced by several (kinds of) references, the 'item' object
-- may be reclaimed by the garbage collector if and only if *all* the
-- references allow its collection.

creation set_item
feature {ANY}
   set_item (i: like item)
      ensure
         item = i
      end
   item: G is
      -- May be redefined (e.g. to keep track of accesses for heuristics)
     do
        Result := item_
     end
   deferred item_reclamation_allowed: BOOLEAN is
      -- Heuristics that is called by the garbage collector when it
      -- has to decide whether to reclaim 'item' or not.
      -- When True, the GC should reclaim 'item', unless other
      -- references change that decision (hence `item_reclaimed' and
      -- `item_not_reclaimed').
      require
 item /= Void
      end
   item_reclaimed is
 -- Routine called by the garbage collector as soon as it
 -- decides to reclaim 'item'
      require
 item = Void
   item_not_reclaimed is
 -- Routine called by the garbage collector as soon as it decides
 -- not to reclaim 'item'
      require
 item /= Void
feature {NONE}
   item_: like item -- actual referenced object
end -- class PROGRAMMABLE_REFERENCE[G]
```

Figure 11: Schetch of the `PROGRAMMABLE_REFERENCE[G]` class.

```
class IMAGE_CACHE_REFERENCE
inherit PROGRAMMABLE_REFERENCE[IMAGE]
    redefine item, item_reclaimed, item_not_reclaimed
creation set_item
feature {ANY}
   set_item(i: like item) is
      do
         item_storage := i ; item_age := 0 ;
         total_image_footprint.set_item(total_image_footprint.item +
                                        item_.memory_footprint)
      end
   item: IMAGE is
     do
        item_age := 0
        Result := item_storage
     end
   item_reclamation_allowed: BOOLEAN
      do
         Result := (total_image_footprint.item > Max_total_footprint)
                   or ( (item_age > Max_image_age) and
                      (total_image_footprint.item > Min_total_footprint) )
      end
   item_reclaimed is
      -- Routine called by the garbage collector before
      -- 'item' is reclaimed
      do
         total_image_footprint.set_item(total_image_footprint.item
                                         - item_.memory_footprint)
      end
   item_not_reclaimed is
      -- Routine called by the garbage collector when the GC decides
      -- not to reclaim 'item'
      do
         item_age := item_age + 1
      end
   Max_total_footprint: INTEGER is 15000000 -- max. 15 MB for all images
   Min_total_footprint: INTEGER is  1000000 -- do not reclaim below this
   Max_image_age: INTEGER is 35
   total_image_footprint: MEMO[INTEGER] is
      -- class variable, shared between all instances, that only
      -- contains an INTEGER item that can be set and queried.
      once
         create Result
      end
   item_age: INTEGER
      -- number of garbage collections 'item' survived
end -- class IMAGE_CACHE_REFERENCE
```

Figure 12: Example of descendant for the PROGRAMMABLE_REFERENCE[IMAGE] class.

```
image_cache : ARRAY [IMAGE_CACHE_REFERENCE]
  -- most recent image at index 'lower'; older at index 'upper'

prepare_image_cache_array is
   local
      index: INTEGER ; icr: IMAGE_CACHE_REFERENCE
   do
      create image_cache.make(1, Max_number_of_cached_images)
      from
         index := image_cache.lower
      until index > image_cache.upper
      loop
         create icr.set_item(Void)
            -- Void since no image is referenced yet
         image_cache.put(icr, index)
         index := index + 1
      end
   end
```

Figure 13: Preparing an image cache based on the `PROGRAMMABLE_REFERENCE[IMAGE]` class.

```
current_image_index: INTEGER

get_previous_image: IMAGE is
   local
      image: IMAGE ;   icr: IMAGE_CACHE_REFERENCE
   do
      if (current_image_index = image_cache.upper) then
         -- already at the last cached image
         load_image_for_upper_index
         icr := image_cache.item(current_image_index)
         image := icr.item
      else -- get the previous image
         current_image_index := current_image_index + 1
         icr := image_cache.item(current_image_index)
         image := icr.item
         if (image = Void) then
            -- image not cached, load and cache it
            image := load_image_from_file(.....)
            icr := image_cache.item(current_image_index)
            icr.set_item(image)
         end
      end
      Result := image
   end
```

Figure 14: Using an image cache based on the `PROGRAMMABLE_REFERENCE[IMAGE]` class.

6 Conclusion

In this partial report, we showed how a simple yet useful concept of weak references was added in a relatively easy way to the Eiffel language. We explained how this was achieved in a safe and efficient manner, thanks to the availability of parametric types in Eiffel (named generic types). We detailed how these weak references had to be implemented in the SmartEiffel compiler and the garbage collector it generates. We discussed various possible choices for weak references in Eiffel and exhibited the advantages of the solution we chose. We thus reached a mature point with a simple, elegant and efficient solution that gives the application developers much greater flexibility and control over the memory behavior of their program, when they need it, and with the minimal possible cost.

The novelty of the solution we detailed lies in the fact it provides safe and efficient weak references, thanks to parametric types. Obviously, both are long-known concepts. However, despite their tremendous advantages — types are checked at compile time whereas type casts occur at run-time — parametric types remain blatantly absent or, at best, under-used in mainstream and/or object-oriented languages. We thus feel they should definitely be available in all languages and largely used, hence the need — as we did in this paper — to stress in a clear and concrete way their advantages and how to rely on them for novel solutions to practical issues.

We decided to further improve our work by providing other kinds of references, besides the normal strong references and the weak references we first just described in this partial report. First, we wanted to make it possible to have weak references that are not *all* collected when the GC runs. We thus created and described typed soft references, which are very much like our typed weak references, but with a somewhat higher survival and reuse rate — if memory permits, of course. We also considered it would be a good thing to give the application developer more control over which weakly referenced objects are collected and which are not, at a given garbage collection cycle. To this end, designed typed "tunable soft references" a kind of reference where the application developer indicates, at creation time, the "strength" of the reference, hence providing a partial ordering on weakly referenced objects. The GC then has to choose how many it needs to collect. We also explored the possibility to give the developer full control over which weakly referenced objects are to be collected. This can be done by giving access to a lot of information on the system through introspection and reflexivity, and letting the developer use it to provide a decision about the life or dead of the weakly referenced object at run-time, when the GC has to run. Our typed "programmable references" make this possible.

The implementation for all these advanced kinds of references is ongoing work.

References

[CCZ98] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. In *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 154–165, October 1998.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99), Nancy, France*, pages 341–350. IEEE Computer Society, June 1999.

[Fre] Free Software Foundation. *The Guile Reference Manual*.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[Mey92] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall Inc., 1992.

[ZC01] Olivier Zendra and Dominique Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software - Practice and Experience*, 31(6):601–613, May 2001.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, Octobre 1997.