Towards safer aliasing with the Eiffel language

Olivier ZENDRA, Dominique COLNET

E-mail: {zendra, colnet}@loria.fr
LORIA
UMR 7503
(INRIA - CNRS - University Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex
FRANCE

Abstract

The SmallEiffel compiler source code makes intensive use of aliasing in order to achieve very good performance both in terms of memory and execution time. This technique is very appropriate for compilation, but can also be applied to a wide range of applications. Thanks to the design by contract capabilities of the Eiffel language, aliasing can be handled very safely. The singleton pattern appears to be crucial in implementing alias provider objects. We propose an efficient implementation of this pattern made easy by some Eiffel idioms.

1 Introduction

Aliasing has been known for a long time (see for example [Mey88]). It is generally regarded as a dangerous feature which should be avoided as much as possible in programs. Much work has thus been focused on alias detection, prevention and avoidance [Hog91, HLW$^+$92, Min96]. It seems however quite difficult, if not impossible, to avoid aliasing in object-oriented languages and thus techniques have also been sought to control aliasing.

We believe aliasing offers some significant advantages in terms of performance and thus may be worth relying on. For these reasons our Eiffel [Mey94] compiler — SmallEiffel, The GNU Eiffel Compiler [1] — makes intensive but carefully designed use of aliasing, trying to reconcile both performance and safety through use of techniques which are mostly already known.

This paper, which reports on our experience with the SmallEiffel compiler, is organized as follows. In section 2, we first explain why we consider

[1] `http://SmallEiffel.loria.fr`

aliasing especially useful in a compiler. Section 3 then introduces the concept of *alias provider* and details how it can be implemented in Eiffel by using the singleton pattern [GHJV94]. Then we describe in section 4 how design by contract and Eiffel assertions can be used to enforce a safe use of aliasing. Section 5 considers the problem of mutable object aliasing. The benefits of using aliasing in SmallEiffel are discussed in section 6. Finally section 7 concludes.

2 Aliasing: what for, in a compiler?

Aliasing is of course not specific to compiler implementations. However, compilers appear to be prone to using aliasing, for example when encountering an identifier.

One of the common tasks of a compiler is for example to check whether a variable is a local one, or an argument of the enclosing routine, or a global variable. To efficiently perform this task, one must be careful to avoid costly string content comparisons. Indeed, it is extremely frequent in a compiler to need to know whether two symbols are the same or not.

The basic idea of aliasing here is to use exactly the same string when several symbols have the same name. This way, it is possible to compare two symbol names with a mere pointer comparison, instead of having to look at the (potentially complete) contents of the name. This avoids having duplicates of the same string, which saves a lot of memory. There also seems to be a possible gain in terms of speed. However, the cost for managing such a symbol name aliasing system may offset part of its advantages and thus has to be more precisely evaluated.

Of course, strings are not the only "aliasable" objects in a compiler, but since they are very common and well understood objects, we will use them as a basic example all along this paper. Many other kinds of more complex objects are good candidates for aliasing, such as for example the various objects used to represent abstract syntax tree [AU77], or AST, nodes.

Furthermore, the use of aliasing is not limited to compilers, but can in fact be applied to most kinds of software, such as for example search engines, graphical interfaces, etc.

3 Using an alias provider

Shared symbol names are akin to the interned atoms supplied by Lisp or Smalltalk, or Java interned strings. Unlike these languages Eiffel does not provide such a mechanism which has thus to be explicitly emulated by the developer.

In SmallEiffel, we implemented this kind of aliasing with an *alias provider* (or *aliaser*). Indeed, although local aliased object management is quite easy and intuitive in a routine or small algorithm, this is not the case anymore when aliasing has to be managed on a system-wide basis.

In a compiler, for example, although the symbol names are created in the parser only, they are handled by a special object in charge of providing a unique, shared object for each name. This special object is called in SmallEiffel and in this paper the STRING_ALIASER. It is not dissimilar to a *bridge* giving access to *islands* of objects [Hog91]. However unlike the bridge object our alias provider is not intended to give only temporary access to the objects it holds (*dynamic* aliasing [HLW+92]) but also a controlled, long-term access (*static* aliasing).

To achieve its purpose, it is above all very important that the STRING_ALIASER object be unique in the whole system. A singleton pattern [GHJV94] perfectly suits this purpose.

3.1 Implementing the singleton pattern in Eiffel

The role of the singleton pattern is to ensure a given class only has one canonical instance and to provide global access to it.

The proposed C++ implementation in [GHJV94] guarantees that the Singleton class is instantiated only once by using an explicit test instruction at runtime.

In Java it is possible to ensure at no runtime cost that only one instance of a class is ever created. This is done by making the constructor private, creating a single static instance in the static initialization block and redefining the `clone` method so that it returns a reference to the static instance.

In Eiffel, a singleton can be implemented by relying on the assertion and *once* mechanisms in the following way.

```
class A_SINGLETON

 -- Other features here...

feature {NONE}  -- Singleton part

   singleton_memory: A_SINGLETON is
      once
         Result := Current;
      end;

invariant

   is_actually_singleton: Current = singleton_memory

end  -- A_SINGLETON
```

Current is the Eiffel notation for the current object, like self in Smalltalk [GR83] or this in Java [GJS96] or C++ [Str86].

singleton_memory is an Eiffel once function [Mey94]. Such a function is executed only once, the first time it is called. Subsequent calls return exactly the same result which is automatically memorized for all instances of the same class. Since singleton_memory is exported to NONE, it is private and can only be called from objects of type A_SINGLETON or its heirs. This way, the class function singleton_memory memorizes the (reference to the) first object that calls it.

Invariants in Eiffel are assertions that are triggered each time a client calls a feature on an instance of the class enclosing the invariant. Invariants are inherited by descendants of the class they appear in. The is_actually_singleton invariant ensures that only one object of type A_SINGLETON (or subtype thereof) is ever created in the system. Indeed, it checks that the (reference to the) Current object is exactly the one stored by singleton_memory. If more than one object of type A_SINGLETON were to be created, this invariant would fail and trigger an exception on the second object, because Current would refer to this second object, whereas singleton_memory would refer to the first.

Is is important to note that like the Java solution mentioned above, the Eiffel implementation of a singleton incurs no overhead at all in optimized mode. Indeed, invariants, like other assertions, are designed to be used in the development phase of the code, whereas in the delivery code they are not active anymore.

Of course, because of the semantics of Eiffel once routines, each class which must be a singleton has to implement the singleton_memory function and is_actually_singleton invariant. Inheriting from A_SINGLETON is not a viable solution, because this way it would be impossible to have more than one singleton in the system.

3.2 Implementation of an alias provider

Given the above implementation for singletons, the specification of our alias provider is quite simple. In addition to the code for the singleton pattern, our STRING_ALIASER offers simply one function, item, in charge of providing a reference to the shared string corresponding to the specified model. Here is its interface:

```
class interface  STRING_ALIASER

   item (model: STRING): STRING

      require
         model /= Void
```

```
      ensure
         Result.is_equal(model)

invariant
   is_actually_singleton: Current = singleton_memory
```

The first time `item` is called with a certain name as an argument, it returns it and memorizes the `model` string in a private DICTIONARY — a kind of hash table. Subsequent calls with either the same model string or different model strings with the same contents will all return the (reference to) the very first, memorized, string object.

Here again, no overhead occurs at runtime, since preconditions (`require`) and postconditions (`ensure`) are not generated in optimized mode.

4 Enforcing the alias provider use

We have described in the previous sections how the uniqueness of the alias provider could be guaranteed. It is of course also very important that in the whole system all objects use the alias provider, and that none keeps its own duplicate versions of aliased strings. This way only one single, unique version of the string is ever created and is managed by the alias provider.

So all classes of the system have to respect the rules of the game, and use the alias provider. They also have to take part in the whole system security by checking that their providers themselves do use the alias provider. This can be easily done in Eiffel by using the preconditions and postconditions of routines.

The following example features LOCAL_NAME, a class used by SmallEiffel to represent an identifier corresponding to the name of a local variable. The creation routine `make` takes two arguments: one to record the position in the source code, the other for the string representing the identifier name.

```
class interface LOCAL_NAME

   --
   -- A local name in some declaration list.
   --

creation

   make (sp: POSITION; n: STRING)
      require
         sp /= Void;
         n = string_aliaser.item(n)  -- (#1)
```

```
ensure
   start_position = sp;
   to_string = n                -- (#2)
```

In the above code, assertion *(#1)* is a precondition which checks that the caller is passing an *aliased* string argument generated by the STRING_ALIASER. Indeed if the contents of `n` is the same as the one of a string memorized by the string aliaser, `string_aliaser.item` returns the memorized STRING object. Thus, if STRING `n` has been provided by the STRING_ALIASER, the memorized object corresponding to the contents of `n` is `n` itself.

Similarly, postcondition *(#2)* also guarantees the correct behavior of the enclosing routine, that is that the aliased string has been memorized by the current LOCAL_NAME object, which is thus a new *alias* to the shared object.

The same applies to other kinds of names in the compiler, such as CLASS_NAME or ARGUMENT_NAME for example.

5 Aliasing mutable objects

Of course, enforcing the use of the aliaser is not enough to safely cope with aliasing. One also has to strictly control who modifies the aliased object and where.

Eiffel strings are mutable objects: their contents may be changed, extended, shortened, etc. In the context of symbols in the SmallEiffel compiler, however, an aliased string should never be modified. Doing otherwise is bound to wreak havoc in the compilation process.

Since a library class such as STRING contains many public modifier routines, the best way to control their use would be to encapsulate the STRING object inside, say, a IMMUTABLE_STRING which would feature accessors to the STRING object but no modifier at all, or at least restrictively exported ones. Subclasses of STRING — which currently do not exist in the SmallEiffel compiler context — could also be encapsulated the same way, either by IMMUTABLE_STRING or a subclass of it.

This solution, though very satisfying from a design point of view, has not been applied in SmallEiffel because it would have implied an important penalty both in terms of memory (extra SYMBOL objects) and execution time (extra level of indirection).

It is nonetheless possible to take measures to check that the aliased strings are indeed not modified. Assertions to this end can be added within the STRING_ALIASER. The STRING_ALIASER needs a second dictionary, in which copies (not aliases) of the aliased strings are kept. Then, each time `item` is called on STRING_ALIASER, an assertion has to be triggered which verifies that the contents of both dictionaries are equal. Since the copies in the second DICTIONARY are not exposed to the outside, an assertion failure at this point would show that one of the aliased objects has been modified, thus signaling incorrect behavior from one of the aliases. This is not perfect, however, since these verifications are performed asynchronously, that is only when `item` is called.

More generally, aliasing mutable (and actually *mutated*) objects causes less problems when dealing with objects which have been *designed to be aliased*.

An example of such objects are instances of type BASE_CLASS which represent classes after they have been parsed. These objects are much more complex than strings since they include all the information about the class they represent: name, ancestors, attributes, routines, etc. If a class FOO is present in the system, being defined in a `foo.e` file, only one instance of the corresponding BASE_CLASS is ever created, whatever the number of occurrences of FOO in client code. Obviously, aliasing this kind of objects is mandatory. Doing otherwise would only result in a tremendous amount of instances of BASE_CLASS and would be utterly inefficient.

The mechanism used is very similar to the one previously detailed for strings. The alias provider for BASE_CLASS objects is the EIFFEL_PARSER singleton. Its role is to ensure that each class source file is only (lazily) parsed once, for obvious performance reasons, and to provide aliases for BASE_CLASS objects. Indeed, the EIFFEL_PARSER is the only object allowed to create instances of BASE_CLASS. Thanks to the selective export mechanism of Eiffel, this property can be *statically guaranteed*. The following code excerpt shows the export clause for the only creation method of BASE_CLASS:

```
class BASE_CLASS
...
creation {EIFFEL_PARSER} make
...
```

This way, each time a BASE_CLASS object appears in client code, one statically knows it is an aliased one coming — directly or not — from the EIFFEL_PARSER. Thus the appropriate checks are performed at compilation time and no extra cost is incurred at runtime, whether in development or in optimized mode. This contrasts with the STRING objects we previously detailed, which may also be created outside the STRING_ALIASER and thus require extra code in the client to ensure their origin at runtime.

This selected export mechanism is also tremendously useful to get a fine-grained control over modifications to BASE_CLASS objects. Unlike for STRING objects, no extra wrapper object is required and no overhead incurs at runtime, whether in terms of memory or speed. Modifier routines (i.e. mutators) can be exported only to the type of objects that need access to them, like this:

```
class BASE_CLASS
```

```
...
feature {TYPE}
   smallest_ancestor(type, other: TYPE): TYPE is
...
feature {EIFFEL_PARSER}
   add_index_clause(index_clause: INDEX_CLAUSE) is
...
feature {RUN_CLASS, PARENT_LIST}
   collect_invariant(rc: RUN_CLASS) is
...
```

It is worth noting that this Eiffel language mechanism — selective export — neatly meets design goals without sacrificing efficiency. We are not aware of any direct equivalent of this in other common OO languages such as C++ or Java.

With BASE_CLASS aliasing it is also very important to make it easier to lazily perform some computations and avoid duplicating them. Looking up the inheritance tree to find the origin of an inherited feature is a very frequent occurrence of such a computation. Since it involves expensive computations which propagate to the ancestors of the BASE_CLASS, it is much better to trigger it only once and then modify the parent list by caching intermediate results for later retrieval. Thanks to aliasing, all occurrences of a given class name in the code share the same BASE_CLASS object, and thus the same aliased list of ancestors. Thus, once a semantic check is performed on one of these occurrences, the corresponding information becomes immediately and inexpensively available to all of them.

One should note that this way, aliases are not notified of mutations in the aliased object. Indeed, there was no need for notification in our application. It is nonetheless extremely easy to add a notification system if needed, thanks to the EIFFEL_PARSER which handles all requests for aliases.

SmallEiffel also features other aliasers based on the same principles, for example for AST nodes, type IDs, customized features [ZCC97], etc.

6 Experimental results

As we have shown above, aliasing is very present throughout the SmallEiffel compiler and concerns many kinds of objects. In order to precisely quantify the impact of aliasing, two versions of the compiler should be run: one with (full) aliasing and another without any. The latter would have meant major and lengthy rewriting. Furthermore, as mentioned above, it even seems impossible to write a compiler without some kind of aliasing, e.g. for BASE_CLASS. Consequently, we decided to restrict our benchmarking to string aliasing only.

In order to quantify the impact of string aliasing during the compilation process we thus built a second version of SmallEiffel without string aliasing. It is worth noting this was made easier by the fact that the whole aliasing process was in the STRING_ALIASER. All reference equality tests between two STRINGs were also replaced by the `is_equal` structure comparison.

We benchmarked both versions of the compiler by compiling SmallEiffel itself [2]. This is a significant benchmark because SmallEiffel is a real-world program, fully bootstrapped in Eiffel and represents about 70,000 lines of code for about 300 live classes.

Memory footprint seems to be able to benefit from string aliasing. The total number of strings (words) encountered during the bootstrap is precisely 79,838. All these words contain 637,014 bytes, which amounts to an average word length of about 8 bytes.

Knowing that one STRING object has a memory footprint of 3 words plus the actual contents of the string (the characters), it is easy to compute the total theoretical memory footprint of all unaliased STRING objects on a 32 bits machine:

$$\begin{array}{rcrcll} \text{nb\_of\_strings} & \times & (\text{string\_object\_size} & + & \text{average\_content\_length}) & = \\ 79{,}838 & \times & (12 & + & 8) & = \\ & & 1{,}596{,}760 \text{ bytes} & = & 1{,}559 \text{ Kbytes} & \end{array}$$

These 79,838 string occurrences correspond to only 4,265 different strings, whose memory footprint can be estimated at:

$$4,265 \times (12 + 8) = 85,300 \text{ bytes} = 83 \text{ Kbytes}$$

Thus when string aliasing is used, the theoretical memory footprint of STRING objects in the compiler is decreased by 94.6% compared to what it is without string aliasing.

These theoretical figures do not take into account the extra space needed for memory management, such as C `malloc` headers or garbage collector data [CCZ98]. Experimental results (figure 1) show that in fact the amount of memory saved by aliasing is 2,456 Kbytes.

Since the whole compiler footprint with aliasing is about 14,708 Kbytes, the benefit *from the sole string aliasing* represents a more than 14% decrease and is thus quite significant. This is even truer considering that SmallEiffel is not only a research project, but also a free product used worldwide by numerous individuals on small personal computers, for which a RAM of 32 or even 16 Mbytes is not uncommon at all.

This footprint reduction is also a good thing in terms of execution speed, since decreasing memory use also eases the burden on the garbage collector

[2] The results were obtained with a PC Pentium II 333 MHz with 128 Mbytes RAM, running Linux, kernel 2.0.36, and with SmallEiffel -0.78 and egcs-1.1.1.

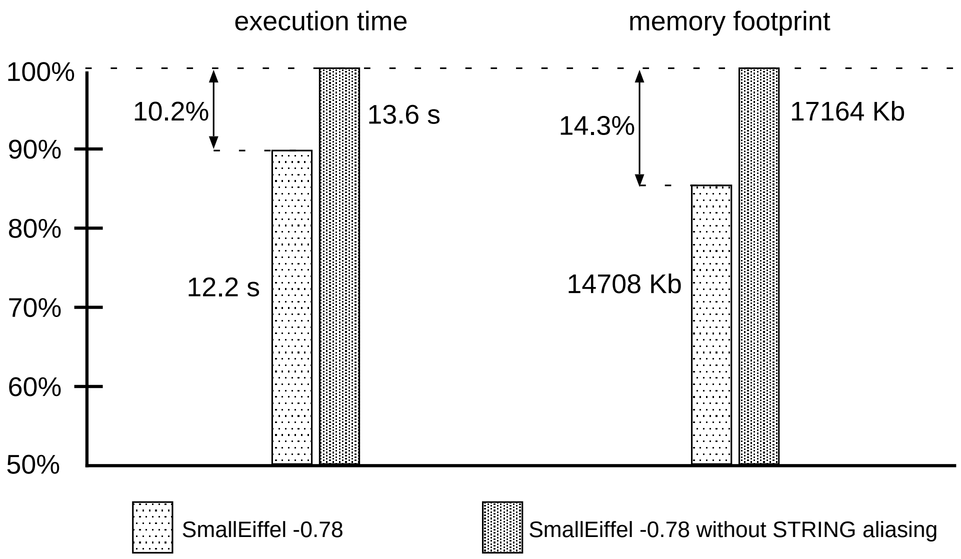

Figure 1: Performance comparison: SmallEiffel with string aliasing vs. SmallEiffel without aliasing

[CCZ98] and lowers swapping.

In fact, the effect of string aliasing on execution time is more than an indirect consequence. Indeed as we wrote in section 2, one of the main reasons for using string aliasing in the SmallEiffel compiler is to be able to do fast string comparisons in client code by merely comparing two references with the `=` operator instead of comparing (the contents of) two objects with the `is_equal` function.

However, the downside of string aliasing is that the management of the STRING_ALIASER has a cost. Indeed, each time a new string is found, it is necessary to check whether it is already in the aliaser and in case it is not the new object has to be created and then inserted in the aliaser. It was thus reasonable to wonder whether the cost of interning strings would outweigh the benefits of mere pointer comparisons.

The experiments we made show (figure 1) that *string aliasing alone* actually leads to an improvement of more than 10% on the whole compilation time.

Once again, it is important to remember that these significant performance improvements come from the sole string aliasing. It is not unreasonable to think that the other kinds of aliasing (on BASE_CLASSes, LOCAL_NAMEs, etc.) also cause footprint and speed improvements which are quite significant. Aliasing under various forms thus seems to play a major part in the very good performance [CCZ97] of the SmallEiffel compiler.

7 Conclusion

Although aliasing has significant drawbacks and is often considered with (at least) suspicion, it also offers a certain number of advantages that may be of use to developers.

We outlined these advantages in the compilation domain and described the way we implemented aliasing in SmallEiffel, The GNU Eiffel Compiler. We showed how reasonable design decisions allowed us to do this in a safer way and how a few idioms of the Eiffel language made safer aliasing easy to implement. In this respect design by contract, implemented with Eiffel assertions, as well as selective export rules, were powerful and most helpful tools. The singleton pattern, for which we proposed a novel Eiffel implementation, has also been crucial.

Results in terms of memory use and execution time have been precisely characterized and proved to be very positive. By using aliasing on the sole STRING objects, we were able to reduce the memory footprint of these objects twenty-fold. On the whole program footprint, this amounts to a decrease of more than 14%. Similarly, execution speed was improved by more than 10%

The very good performance achieved by SmallEiffel in terms of speed [ZCC97] thus seems to owe a lot to aliasing. Further work nonetheless needs to be done to more precisely assess the impact of aliasing on various kinds of programs, related to compilation or not.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions, and Jean-Michel Drouet who proofread early versions of this paper.

References

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81. Springer-Verlag, 1997.

[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.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

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

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

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), April 1992.

[Hog91] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, volume 26, pages 271–285, November 1991.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey94] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[Min96] Naftaly Minsky. Towards Alias-Free Pointers. In *European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Sciences*, 1996.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Series in Computer Science, 1986.

[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, October 1997.