Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler

Dominique COLNET, Olivier ZENDRA

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

Abstract

The design of the Eiffel language makes it possible to perform global optimizations on Eiffel programs. In this paper, we describe some of the techniques we used in SmallEiffel, The GNU Eiffel Compiler, to generate highly efficient executables for Eiffel programs. Most of these techniques — related to global analysis or not — may also be applied to other object-oriented languages.

1 Introduction

We presented in other papers the techniques for global optimization we used in SmallEiffel, The GNU Eiffel Compiler[1], to generate efficient code for late binding [CCZ97, ZCC97] as well as for the generation of a garbage collector (GC) customized to the system [CCZ98].

In this paper, after briefly recalling our main compilation technique, we describe more precisely some other kinds of optimizations implemented in SmallEiffel, whether they are specific to Eiffel [Mey94] or not.

This paper is organized as follows. Section 2 briefly recalls our compilation algorithm. Section 3 describes the implementation of expanded objects, while section 4 focuses on the optimization of dynamic dispatch for reference types. The handling of generic types is explained in section 5, and that of *once* objects in section 6. Finally, section 7 presents the kind of static evaluation allowed by global analysis and section 8 concludes.

2 Overall compilation method

The basic idea behind our algorithm [CCZ97] is to know the whole live code of a program. We build and analyze the call graph of the whole system, knowing all generic derivations used. This allows the computation of the set of all possible concrete types at runtime as

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

well as the set of live primitives. Thus, even when a polymorphic call (multiple possible target types) cannot be replaced by a monomorphic call (one target type), the number of possible target types is reduced.

Most programs are likely to contain polymorphic sites, even after the best type inference analysis (see [Age96] for a complete survey on type inference systems). Since these sites are called much more frequently [DMM96] than monomorphic sites, optimizing them is very important.

Our method to efficiently implement dynamic dispatch (see [DHV95] for a broad study of dispatch methods) without virtual function tables consists in duplicating and customizing each live feature (method or attribute name) according to the concrete type of the target. In [ZCC97], we detailed this method and showed the impact of optimization on the remaining polymorphic sites. We also showed the numerous inlinings this implementation of dynamic dispatch paired with global system analysis allows.

In SmallEiffel, dead code is never reached and thus never compiled, which avoids the cost of unnecessary compilation followed by dead code removal [Dha91, KRS94]. This kind of optimization, allowed by the global knowledge of the program, significantly speeds up the application execution and compilation times.

Knowing the whole live code also makes it possible to optimize the object structure, by generating only the attributes which may be used at runtime. Pruning unnecessary attributes can save a significant amount of space, especially when working with a deep inheritance hierarchy.

3 Expanded objects

SmallEiffel specifically handles expanded objects [Mey94] in order to guarantee not only their semantics but also that their actual implementation is an optimal one.

Because the type of an expanded object is always precisely and statically known, no type-id tag is needed in expanded objects. Hence the latter are as compact as possible since they contain only "useful" attributes. Given the Eiffel source code for PEACH in Appendix A, an object of type `expanded PEACH` would have the following C structure[2]:

```
struct STRUCT_PEACH { STRING* _name; };    /* Expanded #1 */
typedef struct STRUCT_PEACH PEACH;
```

The difference with a normal (non-expanded) PEACH object is quite obvious:

```
struct STRUCT_PEACH { int type_id; STRING* _name; };
```

Expanded objects are also "really" expanded by SmallEiffel. That is they are accessible without any indirection — not through another object referring to them — either in the stack for local expanded or in the heap for expanded attributes. For example, an Eiffel declaration like

```
local
   p1, p2: expanded PEACH;
```

is translated by the following C code (which also includes the default object initialization):

[2]In this paper, all our examples are based on the C code generated by the 23rd version of SmallEiffel, version -0.78, in -boost optimization mode.

```
{
   PEACH _p2 = { NULL };
   PEACH _p1 = { NULL };
```

An Eiffel assignment between two expanded objects is thus a direct C struct assignment, which can be optimized by the C compiler according to the underlying platform. For example, assuming that `p1` and `p2` have both been declared as shown in the previous paragraph, an Eiffel assignment `p1 := p2;` would be translated as:

```
_p1 = _p2;
```

Since the type of expanded objects is always precisely known, there is no dynamic dispatch on such objects. A call on an expanded object is always a direct, fast call. The following Eiffel code illustrates this point:

```
do_it_expanded is
   local
      peach: expanded PEACH;
   do
      peach.display;
   end;
```

The above `do_it_expanded` routine is translated into this C code (note that it is inlined):

```
{
   PEACH _peach = { NULL };
   (&_peach)->_name = _the_adequate_string;
   PEACHdisplay(&_peach);        /* direct call, no dispatch */
}
```

As a point of comparison, if one uses ordinary reference objects as shown in the following `do_it_reference` routine,

```
do_it_reference is
   local
      peach: PEACH;
   do
      !!peach.make;
      peach.display;
   end;
```

the generated C code looks like this (note that it is still inlined):

```
{
   PEACH* _peach = NULL;
   {
     PEACH * _n = newPEACH();
     _n->_name = _the_adequate_string;
     _peach = _n;
    }
    PEACHdisplay(_peach);
}
```

A loop calling the `do_it_expanded` routine 2 million times runs about twice as fast as its `do_it_reference` version (excluding the time spent for I/O operations). This is mainly because the GC is not triggered at all in the expanded version, because all objects are allocated on the stack, not in the heap.

4 Static binding for reference types

When a reference type is not involved in late binding, every use of it can be statically computed and there is thus no need for any type mark in the corresponding objects. The latter will thus be as compact as if they were expanded objects. Furthermore, direct calls will be performed on them, instead of dispatched ones. This has an important impact on performance (see [ZCC97]).

For example, if PEACH has no live heir, and is not the type of the source of an assignment to a variable whose type is an ancestor of PEACH, then PEACH need not be tagged. So these objects will have a structure identical to the one shown in the `do_it_expanded` example.

Global system analysis allows extra optimizations to take place when dealing with the `?=` Eiffel assignment attempt operator. In a nutshell, an assignment attempt consists first in checking whether the dynamic type of the source is compatible with the dynamic type of the target. Then, if this is the case, the assignment is performed, otherwise `Void` is assigned to the target.

When several concrete types are possible on the right hand side – the source – of the operator, a binary branching code is generated which tests the dynamic type of the source object. This is very similar to what is done for a late binding call site.

With global system analysis, the set of possible types is greatly reduced. It is not uncommon to have only one remaining possible concrete type, in which case a simple, direct `:=` assignment can be done.

Let us consider classes FRUIT, APPLE, PEACH (source code is in Appendix A). This example shows the impact of code customization for the `?=` assignment attempt operator.

```
class ASSIGNMENT_ATTEMPT_EXAMPLE
creation make
feature
   make is
      local
         object1, object2: ANY;
         fruit: FRUIT;
         boolean: BOOLEAN;
      do
         !APPLE!object1.make;
         !PEACH!object2.make;
         fruit ?= object1;    -- Attempt #1
         boolean := fruit.is_an_apple;
      end;
end
```

This example is of course a “toy program”, but such cases of assignment attempts arise when dealing with untyped collections of objects, retrieved for example from a persistent storage.

The generated C code for the instruction labeled `Attempt #1` above looks like this:

```
_fruit = (void*) _object1;
if (NULL != _fruit) {
   switch (_fruit->type_id) {
      case APPLEid:
      case PEACHid:
           break;
      default:
           _fruit = NULL;
   }
}
```

A particular example of this customization is an assignment attempt of the type:

```
x ?= Current;
```

Indeed, the dynamic type of Current is always known at compilation time. Furthermore, Current is known never to be `Void`, which removes an extra test.

As an example, let us now consider the instruction labeled `Attempt #2` in function `is_an_apple` of class FRUIT (see Appendix A). The Eiffel instruction is :

```
apple ?= Current;
```

As explained above, each routine is customized to each live type to which it belongs. The corresponding C code in the context of APPLE thus looks like this:

```
_apple = _Current;
```

Indeed, the type of Current is APPLE, in the context of the `is_an_apple` routine in class APPLE and is thus compatible with the type of the local variable `apple`. As we are sure that Current cannot be NULL, no extra test is needed.

Conversely, in the context of class PEACH, the generated C code is similar to:

```
_apple = NULL;
```

Here, the type of Current is PEACH, which is known by the compiler to be incompatible with the type of the local variable `apple`.

5 Generic types

Generic types in Eiffel are parameterized types. For example, ARRAY is such a type, since it is declared as ARRAY[E], where E is a formal parameter type, and is used as ARRAY[INTEGER], ARRAY[FRUIT], etc.

Our algorithm considers that several types derived from a same generic class are all *distinct* live types. A given generic class is live if there is at least one live generic derived type for that class.

Given the classes FRUIT, APPLE, PEACH and the generic class BASKET (see Appendix A), let us consider the following example:

```
class GENERIC_TYPE_EXAMPLE
creation make
feature
    make is
      local
         peach: PEACH;  apple: APPLE;
         fruit_basket: BASKET[FRUIT];
         apple_basket: BASKET[APPLE];
      do
         !!apple.make; !!peach.make;
         !!fruit_basket.make(peach);
         fruit_basket.item.display;    -- Generic #1
         !!apple_basket.make(apple);
         apple_basket.item.display;    -- Generic #2
      end;
end
```

This illustrates the impact of code customization when using generic types mixed with polymorphism. In the above example, there are two kinds of BASKET in the live code: BASKET[FRUIT] and BASKET[APPLE].

Here is the generated C code for the instruction labeled `Generic #1`:

```
SWITCH_FRUITdisplay(_fruit_basket->_item);
```

The first part of the `Generic #1` instruction, the `fruit_basket.item` attribute access, has been translated by a simple C structure field access `_fruit_basket->_item`. One must note that even though there are two kinds of baskets in the live code (BASKET[FRUIT] and BASKET[APPLE]), there is no need to perform any dynamic dispatch for the `item` attribute because the corresponding C structures for types BASKET[FRUIT] and BASKET[APPLE] both have the same `item` field with the same offset. This is one occurrence of an Attribute Read Removal (ARR), as explained in our OOPSLA'97 paper [ZCC97].

Since the local variable `fruit_basket` is declared of type BASKET[FRUIT], the expression `fruit_basket.item` is of type FRUIT. Thus, the last part of the `Generic #1` instruction — the call to the `display` routine — is done through a dispatching routine, `SWITCH_FRUITdisplay`. The latter implements late binding on the `display` method for all kinds of FRUIT.

Let us now consider the above labeled instruction `Generic #2`. The corresponding C code is:

```
APPLEdisplay(_apple_basket->_item);
```

Since in GENERIC_TYPE_EXAMPLE the local variable `apple_basket` is declared of type BASKET[APPLE], `apple_basket.item` is of type APPLE. Thus, there is no need for dynamic dispatch and a direct call to the APPLEdisplay routine is generated.

6 Once routines

Once routines [Mey94] are a specificity of Eiffel. The body of such a routine is executed only once in the program lifetime, the first time the routine is called. Subsequent calls return without executing the routine body.

When the *once* routine is a function, its result is computed the first time the routine is called, and returned at each subsequent call.

This allows a number of optimizations in SmallEiffel, most of them related to garbage collection [JL96]. Details of our implementation of a customized mark-and-sweep garbage collector can be found in [CCZ98].

Objects returned by *once* functions, or *once objects*, live from the time they were allocated till the end of the program. Thus, exactly like the root object of a system, a *once* object must always be considered as marked by the collector, and may not be collected.

Once object management can thus be optimized by relying on specific marking and by completely avoiding sweeping.

For some simple *once* functions, it is possible to unambiguously know the type of the result at compile time. Such results can thus be pre-computed, that is created at the very beginning of the program [ZCC97]. Further optimizations, such as not checking whether the object is NULL, can be performed in the GC when dealing with these objects.

A manifest string can be considered as a special type of *once* function whose value is pre-computable at compile time. Consequently, all the manifest strings are allocated in specific memory areas and are not subject to sweeping. They also have a marking process customized and optimized as for pre-computable once functions.

7 Static evaluation of expressions

SmallEiffel also performs a static evaluation of expression whenever possible. This kind of optimization is a classic one in compilers, whatever the language.

However, in the case of Eiffel (and more generally of object-oriented languages), inheritance incurs some extra work to reach the best level of optimization. Indeed, *template methods* are very common in OOP [GHJV94]. It is often the case that part of a template method cannot be statically computed in the context of the class where it is defined, but can be in some of its heirs, where more contextual information is available.

Let us for example consider the simple `display_more` procedure which is defined in class FRUIT. This template method uses the deferred feature `stone_fruit` of FRUIT, which is then defined as the `true` constant in class PEACH and as the `false` constant in class APPLE.

As any other procedure, the template method `display_more` is customized to each live type to which it belongs. When the target is a PEACH, the `stone_fruit` expression inside the `display_more` function is always true. Thus, in this case, the `then` part of the `if-then-else` instruction is always executed while the `else` part is never reached. Taking this into account, the customized C code emitted by SmallEiffel looks like this:

```
void PEACHdisplay_more(PEACH* _Current) {
    PEACHdisplay(_Current);
    printf("This is a stone fruit.\n");
}
```

Inside the previous C function, the first instruction is a direct call to the customized `PEACHdisplay` procedure because the target `_Current` is known to be a PEACH. The second C instruction is the mapping of the always-reached `then` part.

Conversely, in class APPLE, the `else` part is always executed and the C code for class APPLE looks like this:

```
void APPLEdisplay_more(APPLE* _Current) {
    APPLEdisplay(_Current);
    printf("This is not a stone fruit.\n");
}
```

8 Conclusion

We showed in this paper that Eiffel is a high-level, object-oriented programming language which is well adapted to global code optimizations. We detailed some of the latter that we implemented in SmallEiffel, The GNU Eiffel Compiler. They make it possible to reach C-like performance, while keeping a much higher level of expressiveness, lightening the programmer's burden. This makes Eiffel a language suited to a wide range of applications, including very demanding embedded systems.

It seems possible to further increase the expressiveness of Eiffel by adding for example more dynamic aspects to the language. However, this could significantly reduce the potential for static optimizations.

Although the optimizations we described are all static ones, they do not prevent the integration of various dynamic optimizations in SmallEiffel. For example, the addition of profile-guided analysis [HCU91, CG94, AH96] would provide more information on the most commonly used types at execution time, and thus allow further optimization of dynamic dispatch sites.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions. We are also especially grateful to Jean-Michel Drouet who proofread early versions of this paper.

A Source code of the examples

```
deferred class FRUIT
feature
   name: STRING;
   display is
      do
         print("Fruit name is ");
         print(name);
         print("%N");
      end;
   is_an_apple: BOOLEAN is
      local
         apple: APPLE;
      do
         apple ?= Current; -- Attempt #2
         Result := apple /= Void;
      end;
   stone_fruit: BOOLEAN is
      deferred
      end;
   display_more is
      do
         display;
         if stone_fruit then
            print("It is a stone fruit.%N");
         else
            print("It is not a stone fruit.%N");
         end;
      end;
end
-------------------------------------------
class APPLE
inherit FRUIT;
creation make
feature
   make is
      do
         name := "APPLE";
      end;
   stone_fruit: BOOLEAN is false;
end
-------------------------------------------
class PEACH
inherit FRUIT;
creation make
feature
   make is
      do
         name := "PEACH";
      end;
   stone_fruit: BOOLEAN is true;
end
-------------------------------------------
class BASKET[E]
creation make
feature
   item: E;
      -- For the sake of simplicity,
      -- a BASKET contains only one element.
   make(i: like item) is
      do
         item := i;
```

```
        ensure
           item = i
        end;
end
```

References

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Department of Computer Science of Stanford University, Published by Sun Microsystem Laboratories (SMLI TR-96-52), 1996.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Sciences*, pages 142–166. Springer Verlag, 1996.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference, JMLC'97*, 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.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs. In *21st Annual ACM Symposium on the Principles of Programming Languages*, pages 397–408, January 1994.

[Dha91] D.M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 253–282. Springer Verlag, 1995.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *11th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 292–305, 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1994.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Sciences*, pages 21–38. Springer Verlag, 1991.

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

[KRS94] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–, 1994.

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

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