Type Inference for Late Binding. The SmallEiffel Compiler.

Suzanne COLLIN, Dominique COLNET and Olivier ZENDRA
Campus Scientifique, Bâtiment LORIA,
Boîte Postale 239,
54506 Vandoeuvre-lès-Nancy Cedex
France
Tel. +33 03 83.59.20.93
Email: colnet@loria.fr

Centre de Recherche en Informatique de Nancy

Abstract. The SmallEiffel compiler uses a simple type inference mechanism to translate Eiffel source code to C code. The most important aspect in our technique is that many occurrences of late binding are replaced by static binding. Moreover, when dynamic dispatch cannot be removed, inlining is still possible. The advantage of this approach is that it speeds up execution time and decreases considerably the amount of generated code. SmallEiffel compiler source code itself is a large scale benchmark used to show the quality of our results. Obviously, this efficient technique can also be used for class-based languages without dynamic class creation : for example, it is possible for C++[10] or Java and not possible for Smalltalk.

1 Introduction and Related Works

Object-oriented languages, by their very nature, pose new problems for the *delivery of applications* and the construction of time-efficient, type-safe, and compact binaries. The most important aspect is the compilation of inheritance and late binding. The distinctive feature of late binding is that it allows some freedom about the exact type of a variable. This flexibility of late binding is due to the fact that a variable's type may dynamically change at run time. We introduce an approach to compilation which is able to automatically and selectively replace many occurrences (more than 80%) of late binding by static binding, after considering the context in which a call is made. The principle we will detail in this article consists in computing each routine's code in its calling context, according to the concrete type of the target. The computation of a specific version of an Eiffel [7] routine is not done for all the target's possible types, but only for those which *really exist* at run time. As a consequence, it is first necessary to know which points of an Eiffel program may be reached at run time, and then to remove those that are unreachable.

Our compilation technique, which requires the attribution of a type to each object, deals with the domain of type inference. Ole Agesen's recent PhD thesis

[2] contains a complete survey of related work. Reviewed systems range from purely theoretical ones [13] to systems in regular use by a large community [8], via partially implemented systems [11] [12] and systems implemented on small languages [5] [9].

Using Agesen classification [2], SmallEiffel compiler's algorithm can be qualified as *polyvariant* (which means each feature may be analyzed multiple times) and *flow insensitive* (no data flow analysis). Our algorithm deals both with *concrete types* and abstract types. The Cartesian Product Algorithm (CPA) [1] is a more powerful one. However, the source language of CPA is very different from Eiffel : it is not statically typed, handles prototypes (no classes) and allows dynamic inheritance ! The late binding compilation technique we describe for Eiffel may apply to any class-based language [6], with or without genericity, but without dynamic creation of classes. The generated target code is precisely described as well as the representation of objects at run time.

Section 2 introduces our technique's terminology and basic principle. Section 3 explains in detail and with examples the technique we used to suppress late binding points. Section 4 considers the problem of genericity. Section 5 show the impact of late binding removal. Our examples are written in Eiffel, and we use the vocabulary which is generally dedicated to Eiffel [7].

2 Overall Principles and Terminology

2.1 Dead Code and Living Code

In Eiffel, the starting execution point of a program is specified by a pair composed of an initial class and one of its creation procedures. This pair is called the application's *root*. The first existing object is thus an instance of the root class, and the very first operation consists in launching the root procedure with that object. First of all, our compilation process computes which parts of the Eiffel source code may or may not be reached from the system's root, without doing any data flow analysis. The result of this first computation is thus completely independent of the order of instructions in the compiled program. With regard to conditional instructions, we assume that all alternatives may a priori be executed. Starting from the system's root, our algorithm recursively computes the *living code*, that is code which may be executed. Code that can never be executed is called *dead code*. Living code computation is closely linked to the inventory of classes that may have instances a run time. For example, let's consider the following root :

```
class ROOT creation root feature
   a: A;
   root is
      do io.put_string("Enter a number :"); io.read_integer;
         !!a;                                -- (1)
         a.add(io.last_integer);             -- (2)
         a.sub(1);                           -- (3)
      end;
end -- ROOT
```

Of course, the root procedure of class ROOT belongs to the living code. Since this procedure contains an instantiation instruction for class A (at line (1)), instances of class A may be created. The code of routines `add` and `sub` of class A may be called (lines (2) and (3)). Consequently, the source code of these two routines is living code, from which our algorithm will recursively continue. In the end, if we neither consider input-output instructions nor the source code of routines `add` et `sub`, only two classes may have instances at run time: class A and class ROOT. By analogy with the terms dead and living code, a *living class* is a class for which there is at least one instantiation instruction in living code: A is a living class. Conversely, a *dead class* is a class for which no instance may be created in living code.

2.2 Goals and Basic Principles

Object-oriented programming (OOP) is largely based on the fact that an operation always applies to a given object corresponding to a precise dynamic type. In Eiffel, as in Smalltalk, no operation may be triggered without first supplying this object, called *target* or *receiver*. Named `Current` in Eiffel, the target object determines which operation is called.

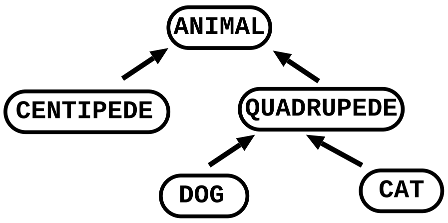

Fig. 1. Simple inheritance graph used in all the article.

Let's consider the simple inheritance example of figure 1, which will be used all along this article. Let's also assume that each sub-class of ANIMAL has its own routine `cry`, which displays its own specific cry. If the target is of type CAT, the routine `cry` of class CAT is used. If the target is of type DOG, it is the routine of class DOG. And so on... The main goal of our compilation technique consists in *statically* computing either — at best — the only routine which corresponds to a target object, or the reduced set of routines that are potentially concerned.

2.3 Representation of Objects at Run Time

Only living types are represented at run time. Basic objects corresponding to Eiffel's types INTEGER, CHARACTER, REAL, DOUBLE and BOOLEAN are directly represented with simple types of the target language. For example, an Eiffel object of type INTEGER is directly implemented with a C `int`. All other objects are represented by a structure (C `struct`) whose first field is the number that identifies the corresponding living type. Each of the object's living attributes is implemented inside this structure as a field bearing the attribute's name. If the attribute is directly an object (`expanded`), the field's type corresponds to the living type considered. If the attribute is a reference to another object, the field's type is "pointer to anything"(`(void *)` in C). For example, for the previously described class ROOT, the C structure is:

structure ROOT {int id; void *a;};

If attribute `a` of class ROOT is declared as `expanded` A, that is without intermediate pointer, the C structure would be:

structure ROOT {int id; A a;};
structure A {int id; ...; ...;};

3 Adapting Primitives

Considering that we know the set of living types, our compilation technique consists, for each of these types, in computing a specific version of primitives that can be applied to the type. The adaptation of a primitive is mainly based on the fact that the target's type is known: it is the living type in which the adaptation is realized. For all the following examples, we refer to the source text of classes ANIMAL, QUADRUPEDE, DOG, CAT and CENTIPEDE given in appendix.

To avoid needlessly complicating our explanations, the target language used is a pseudo-code close to Pascal or C. The actual C code produced by SmallEiffel can be obtained by compiling those examples, which are available with the compiler itself by `ftp` at address `ftp.loria.fr` in directory `/pub/loria/genielog/SmallEiffel`.

3.1 General Example

```
class ROOT1 creation root feature
 dog: DOG;
 root is
    do   !!dog;                  - (1)
         dog.set_name("Snowy");    - (2)
         dog.introduction;        - (3)
    end;
end - ROOT1
```

Fig. 2. The only living types are ROOT1 and DOG. Other types are dead.

The first root given in figure 2 is deliberately very simple. If we omit input-output instructions and class STRING, only types DOG and ROOT1 are alive. Type

```
structure ROOT1 {int id; void *dog;};
structure DOG {int id; void *name;};
procedure DOGset_name(DOG *target,void *my_name) {
 target->name = my_name;};
procedure DOGintroduction(DOG *target) {
 write("My name is :");
 write(target->name);};
procedure ROOT1root(ROOT1 *target) {
 target->dog = new(DOGid);                        - (1)
 DOGset_name(non_void(target->dog),"Snowy");      - (2)
 DOGintroduction(non_void(target->dog));}         - (3)
procedure main {
 ROOT1root(new(ROOT1id));}
```

Fig. 3. Generated code for the root of figure 2.

ROOT1 is alive, as well as the `root` operation which serves as main program. Only class DOG may be instantiated (at line `(1)` figure 2). Procedures `set_name` and `introduction` of type DOG are alive, since they are used in operation `root` at lines `(2)` and `(3)`. Operation `set_name` uses attribute `name` (cf. definition of class ANIMAL), which is thus also alive. Let's note that classes ANIMAL and QUADRUPEDE are abstract ones (`deferred`) and may never correspond to a living type. Eventually, the generated pseudo-code is composed of two structure definitions and four procedures, including the launching procedure `main` (figure 3). Structures ROOT1 and DOG both begin with an `id` field that contains their type identifier. Field `dog` of structure ROOT1 corresponds to attribute `dog`, which is an object pointer. Similarly, attribute `name` of structure DOG is also an object pointer. The adaptation of procedure `set_name` in type DOG is procedure DOG-`set_name`. The latter always applies to a pointer to a structure DOG (the living type corresponding to the target). The body of this routine consists in setting field `name` in the target object's structure. In this procedure as in all others, we consider that argument `target` may never be a "pointer to nothing" (C NULL or Eiffel `Void`). This choice is perfectly coherent with an elementary principle of OOP: without target, no operation may be executed. The target's existence test is done by the caller, as can be seen in the code produced for instructions `(2)` and `(3)` of procedure ROOT1`root`. Function `non_void` has to verify that the target exists. If it does not, this function raises a run time error. Otherwise, this function returns its argument unchanged. Considering that we do not perform any data flow analysis, the call to `non_void` is always done, even when the previous instruction (the one in line `(1)`) gives the target a value.

3.2 Routines that Do not Use the Target

Let's now consider the root given in figure 4. In this example, only classes

```
class ROOT2 creation root feature
 centipede: CENTIPEDE; quadrupede: QUADRUPEDE;
 root is
  do
   !!centipede;                    - (1)
   centipede.cry;                  - (2)
   !DOG!quadrupede;                - (3)
   quadrupede.cry;                 - (4)
  end;
end - ROOT2
```

Fig. 4. Living types : ROOT2, CENTIPEDE and DOG. Other types are dead.

```
structure ROOT2 {int id; void *centipede; void *quadrupede};
structure CENTIPEDE {int id;};
structure DOG {int id;};
procedure DOGcry() { write("BARK");};
procedure CENTIPEDEcry() { write("SCOLO");};
procedure ROOT2root(ROOT2 *target) {
  target->centipede = new(CENTIPEDEid);        - (1)
  non_void(target->centipede); CENTIPEDEcry;   - (2)
  target->quadrupede = new(DOGid);             - (3)
  non_void(target->quadrupede); DOGcry; }      - (4)
```

Fig. 5. Target code generated for the root of figure 4.

CENTIPEDE and DOG are instantiated in the living code (instructions `(1)` and `(3)`). The two corresponding types are thus living ones. No instance of class CAT may exist at run time, since the living code does not contain any instantiation instruction for that class: type CAT is dead, as well as types ANIMAL and QUADRUPEDE. Indeed, these two types correspond to abstract (`deferred`) classes whose instantiation is forbidden. Code is thus produced only for the 3 living types ROOT2, DOG and CENTIPEDE. The full resulting code, given in figure 5, contains 3 structures and 4 procedures. Each living type structure comprises only living attributes. Consequently, type DOG does not contain field `name` any more, since attribute `name` is not used. Procedures DOG`cry` and CENTIPEDE`cry` do not use the target object (`Current`) which corresponds to their living type. They always print the same message, whatever their living target. Such routines that do not use their target are thus coded without any `target` argument. However, it is necessary to keep the target's existence test in the caller routine, by calling function `non_void` just before the actual call (cf. lines `(2)` and `(4)` of figure 5). For the instruction of line `(4)`, DOG`cry` must be called, even if the static type of variable `quadrupede` is QUADRUPEDE and not DOG. Indeed, the only living type which conforms to (dead) type QUADRUPEDE is DOG.

3.3 Late Binding

```
quadrupede: QUADRUPEDE;
root is
  local x: STRING;
  do
    x := get_user_answer("Dog or Cat? ");
    if equal("Cat",x) then !CAT!quadrupede else !DOG!quadrupede end;
    quadrupede.cry;  - (1)
  end;
```

Fig. 6. Instruction of line (1) need late binding.

```
procedure root(ROOT *target) {
 ...
 switchQUADRUPEDEcry(target->quadrupede); }      - (1)
procedure switchQUADRUPEDEcry(void *target) {
 switch (non_void(target)->id){           - (a)
    DOGid:   DOGcry; break;               - (b)
    CATid:   CATcry; break;               - (c)
    else error("non-conforming type");}}  - (d)
```

Fig. 7. Generated code for instruction (1) of figure 6.

In the example of figure 6, CAT and DOG are living types. The static type of the target of instruction `(1)` is QUADRUPEDE. Since the 2 living types conform to the target's type, a late binding operation is required to select either procedure `cry` of type CAT, or procedure `cry` of type DOG. When a late binding point exists in living code, an appropriate switching routine is defined (figure 7). For a given type, dead or alive, this routine performs the selection corresponding to a given operation name. Given the role of switching routines, their names remind the pair which corresponds to the selection ($type \times operation\_name$). In our example, the switching procedure must realize this selection among living types that conform to QUADRUPEDE, in order to call the suitable procedure `cry`. Hence the name `switch`QUADRUPEDE`cry`. In order to factorize it, the target's existence test is realized into the switching routine's body (line `(a)`). The operation called is the one which corresponds to the number that identifies the living type (lines `(b)` and (c)). Finally, to guard against all contingencies, an error is raised (line (d)) if the target does not correspond to any living type that conforms to QUADRUPEDE. Error **"non-conforming type"** of line (d) allows the detection of a potential problem of *system-level validity* .

Of course, this error may never occur for example of figure 6, because the target is always either of type CAT or of type DOG. Furthermore, the target is always an existing object, thanks to the call to **non_void**. More generally, the root of example 6 is said to be *system-valid* according to corresponding rules that are defined in the Eiffel reference manual [7] (*system-level validity*, page 357). A compiler which would be able to detect that a program is correct with regard to these rules could omit the run time error of line (d).

To simplify our presentation, we use a sequential switch in figure 7. Actually, one of our compiler's strong points is its ability to generate a dichotomic selection code, after sorting out the type identifiers. This speeds up the selection, which may have more than 2 alternatives, by decreasing the average number of tests performed. We may note that other compilers generally use a function pointer table whose goal is to realize the selection in a constant time. For reasons we will explain later (§3.5), we did not choose that solution.

3.4 Calls on Target Current

```
animal: ANIMAL;   quadrupede: QUADRUPEDE;
root is
  do
    x := get_user_answer("Dog or Cat? ");
    if equal("Cat",x) then !CAT!quadrupede else !DOG!quadrupede end;
    x := get_user_answer("Dog, Cat or Centipede ? ");
    if equal("Cat",x) then !CAT!animal
    elseif equal("Dog",x) then !DOG!animal
    else !CENTIPEDE!animal end;
    quadrupede.chameleon(animal);              - (1)
  end;
```

Fig. 8. Procedure chameleon is alive for types CAT and DOG (line (1)).

As a living operation is always adapted in the corresponding living type, calls on the receiver (target **Current**) are coded as immediate calls : there is no target's existence test, since **Current** may never be a pointer to a non-existent object, and there is also no need for any selection by switching routine. Let's consider the root given in figure 8. The set of living types is composed of 4 types : ROOT4, CAT, DOG and CENTIPEDE. Instruction (1) is a call to procedure **chameleon** with a target whose static type is QUADRUPEDE. This procedure is

```
procedure root(ROOT *target) {
  ...
  switchQUADRUPEDEchameleon(target->quadrupede,target->animal);}   - (1)
procedure switchQUADRUPEDEchameleon(void *target, void *other) {
  switch (non_void(target)->id)
     DOGid:DOGchameleon(target,other); break;                       - (a)
     CATid:CATchameleon(target,other); break;                       - (b)
     else error("non-conforming type");} }
procedure DOGchameleon(DOG *target, void *other) {
  DOGcry;                                                            - (c)
  write(" imitates ");
  switchANIMALcry(other);                                            - (d)
  write(" = ");
  switchANIMALcry(DOGimitation(target,other)); }                     - (e)
```

Fig. 9. Switch (QUADRUPEDE $\times$ `chameleon`) and the code of procedure `chameleon` for type DOG. Coding for the root of figure 8.

thus alive in types CAT and DOG, and dead in type CENTIPEDE because CENTIPEDE is not a type that conforms to QUADRUPEDE. The code for instruction `(1)` given in figure 9 includes the corresponding switch and the version of procedure `chameleon` adapted to type DOG. This procedure, whether it is adapted to type CAT or type DOG, uses its target. Consequently, internal calls to the switch (lines `(a)` and `(b)` in figure 9) pass down 2 arguments: the target, and argument `other`. Operation `chameleon` adapted to type DOG begins with a call to procedure `cry` on target `Current` (see class ANIMAL). As each living operation is duplicated in the corresponding living type, target `Current` is always a non-void pointer to an object of the corresponding living type. A call whose target is `Current` is thus coded with a direct — static — call, without even checking that the pointer exists (line `(c)`). The call to `cry` with target `other` of static type ANIMAL relies on the corresponding switch (line `(d)`). To follow Eiffel's order of evaluation (left to right for qualified calls), the leftmost elements in Eiffel become the rightmost and most nested arguments in the generated C code. So, instruction "`Current.imitation(other).cry;`" is coded (line `(e)`) by a direct call to `imitation` (adapted to type DOG), which serves as an argument of the switch (ANIMAL$\times$`cry`). Using this switch is mandatory, because in DOG's context, one has to consider the definition of `imitation` given in class QUADRUPEDE.

3.5 Further Optimizations

When a program is considered to be valid, either because it was intensively tested or because it has been proven system-valid, the produced code may be simplified in several ways:

– The test of target existence "`non_void`" may be suppressed.

– In a switch, the default test `"non-conforming type"` may be omitted.
– Simple operations, such as procedures that set or functions that read an attribute, may be inlined.
The various alternatives in a switch may thus result in different forms of coding : function calls, inlined function calls or mere variable accesses. This explains why we decided not to use function pointers arrays to realize our switching routines. Another advantage of our choice is that it is possible to factorize alternatives which correspond to the same C code.

4 Dealing with Genericity

```
root is
  local
    pussy:CAT; cat_ar:ARRAY[CAT]; doggy:DOG; quad_ar:ARRAY[QUADRUPEDE];
  do
    !!pussy; !!doggy;
    cat_ar := «pussy»;
    cat_ar.item(1).cry;              - (1)
    quad_ar := «pussy,doggy»;
    quad_ar.item(1).cry;             - (2)
    cat_ar.item(1).set_name("Felix"); - (3)
  end;
```

Fig. 10. Using genericity . Types ARRAY[CAT] and ARRAY[QUADRUPEDE] are alive.

```
procedure root {
 ...
 non_void(ARRAYofCATitem(cat_ar,1)); CATcry;              - (1)
 ...
 non_void(quad_ar);
 switchQUADRUPEDEcry(ARRAYofQUADRUPEDEitem(quad_ar,1));    - (2)
 ...
 non_void(cat_ar);
 CATset_name(ARRAYofCATitem(cat_ar,1),"Felix");}          - (3)
```

Fig. 11. Code generated for the root of figure 10.

To handle genericity, we also apply a similar technique, considering that several types derived from a same generic class are all *distinct* living types. A given generic class is alive if there is at least one living generic derived type for that class. For example, in figure 10, «`pussy`» represents an instantiation of type ARRAY[CAT] by creation of an array initialized with one single element. Type ARRAY[CAT] is thus alive. Notation «`pussy,doggy`» corresponds to a 2-element array whose type is ARRAY[QUADRUPEDE] . Indeed, QUADRUPEDE is the smallest type to which `pussy` and `doggy` conform. So type ARRAY[QUADRUPEDE] is also a living type.

The living operations of types ARRAY[CAT] and ARRAY[QUADRUPEDE] are copied and adapted separately in each of these 2 types. For example, function `item`, which returns an element, has two possible adaptations: one in ARRAY[CAT] and the other in ARRAY[QUADRUPEDE] . This function's static type is CAT in ARRAY[CAT] and QUADRUPEDE in ARRAY[QUADRUPEDE] . In this way, the code for instruction `(1)` given in figure 11 is a direct call to procedure `cry` of type CAT, since the static type of the target (which is the result of function `item`) is CAT. For instruction `(2)`, the static type of function `item`'s result is QUADRUPEDE. Using a switch is thus mandatory, because CAT and DOG are both living types that conform to QUADRUPEDE.

5 Results and Conclusion

5.1 Results Using Eiffel Source Code of SmallEiffel

boost	check	require	Number of ...
17666	19736	23039	direct calls without any test of target existence
0	7190	8289	direct calls with the target's existence test
3379	5138	5899	switched calls
194	203	211	defined switches
83	319	258	functions which do not use the target
27	82	36	procedures which do not use the target
1434	1436	1698	functions which do use the target
1010	1589	1758	procedures which do use the target
1010	0	0	inlined functions
266	0	0	inlined procedures

Fig. 12. Compiling SmallEiffel with itself (45000 lines / 250 classes).

The results we present here are obtained when compiling SmallEiffel compiler's root itself. Obviously, such a root is a significant benchmark, with about 250 classes and 45000 lines of Eiffel source code. Figure 12 gives a general survey of our results. We used 3 different compilation modes :
boost : Compilation mode which includes all optimizations. There is no target's existence test, and no system-level validity checking. Some routines are inlined, and switches are simplified. There is no assertion check.

check : Compilation mode in which no Eiffel assertion is checked. The target's existence test is performed. Some code is generated for the system-level validity checking, and to produce an execution trace. There is no inlining and no assertion check.
require : Compilation mode in which Eiffel preconditions are checked. The generated code is similar to the previous one, but also includes code to test preconditions (`require` clause).

Results are extremely positive concerning the number of calls without switch compared to the total number of calls. For example, in boost mode, the total number of calls is 21045, including only 3379 switched calls. This means that 84% of the calls are direct, fast calls.
In the 2 other modes (check and require), this ratio is similar (respectively 83% and 84%) if one admits that a call with only a target's existence test may be considered as a direct call. The ratio of routines (procedures and functions) that do not use the target is 5% in boost mode, 13% in check mode and 8% in require mode. This relatively low proportion in all modes comes from the fact that, generally, a routine which is put into a class is designed to operate on instances (targets) of this class. This is a basic principle of object-oriented programming. Figure 13 gives a survey of results obtained about the size of the generated code

	SmallEiffel	Eiffel/S 1.3
1 Size (boost / -O)	622 Kb	884 Kb
2 Size (check)	2.3 Mb	3.3 Mb
3 Size (require)	2.7 Mb	3.3 Mb

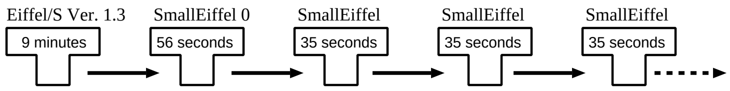

Fig. 13. Executable size and compilation times: SmallEiffel vs Eiffel/S 1.3.

as well as the compilation time to obtain C code from Eiffel. As before, the code of SmallEiffel is used as a benchmark. All the results shown in figure 13 were obtained from tests realized on the same machine (HP 9000/887), with the same C compiler and the same options (`cc` -O). Since we used the Eiffel/S compiler (Release 1.3, from SiG Computer GmbH) to initiate SmallEiffel's bootstrap, we compare our results with this compiler. The size of the optimized code given in line 1 corresponds to SmallEiffel's boost mode and option -O of Eiffel/S compiler. The size comparisons of lines 2 and 3 are given for information only, since we should compare without including the code used to trace execution errors. This code is different from one compiler to an other, and varies with the level of detail of the trace. We can nonetheless note that the code size of Eiffel/S is constant, whereas the size of SmallEiffel's code increases. The latter point is due to the fact that SmallEiffel produces *only* the code which is strictly necessary for a specific compilation mode.

In order to evaluate the benefit that directly comes from our processing of late binding, we may examine more closely SmallEiffel's bootstrap process. Bottom of figure 13 shows how SmallEiffel is obtained, through a succession of compilations on the same source code (SmallEiffel's Eiffel code). After each compilation, the compiler produced is used for the next compilation, and so on... The first compilation (9 minutes) corresponds to an execution of the Eiffel/S compiler. The second one (56 seconds) uses SmallEiffel's compilation algorithm, but late binding still relies on Eiffel/S indirection algorithm. The third compilation (35 seconds) is the first that uses our implementation of late binding by switches. The fourth compilation and the following show that the process has stabilized [3]. The actual benefit when compiling the SmallEiffel compiler code is given by the ratio 56/35. SmallEiffel runs 1.6 times as fast as Eiffel/S for this big benchmark.

5.2 Comparison with C++ and Other Eiffel Compilers

Results presented in this section are available in `comp.lang.eiffel` on the Internet. This comparison was done by Dietmar Wolz. See archives files available at `http://www.cm.cf.ac.uk` for details. The Eiffel program consists of 13 classes where one (dynamic arrays, inheriting from ARRAY[G]) was adapted to the different compilers for performance optimization. The C++ program uses the same algorithm, a similar structure and is based on the Standard Template Library. The test was realized on the same machine with the same C compiler. The following compilers have been tested: `(a)`-gnu g++ 2.7.2 with STL from gnu libg++2.7.1 `(b)`-gnu g++ 2.7.2 with commercial STL from Object Space `(c)`-ISE ebench 3.3.7 finalize, no garbage collection `(d)`-ISE ebench 3.3.7 finalize, with garbage collection `(e)`-SmallEiffel, no garbage collection `(f)`-SmallEiffel, with Boehm-Demers-Weiser garbage collector `(g)`-Sig Eiffel 1.3S no garbage collection `(h)`-Sig Eiffel 1.3S with garbage collection `(i)`-Tower Eiffel 1.5.1 no garbage collection.
Results of figure 14 include run time, compilation time, memory used in MByte and code size of the executable file in kByte. According to Dietmar Wolz's test, SmallEiffel is really the most efficient Eiffel compiler. The C++ program does memory management by hand, but there are some leaks. Thus one cannot compare the memory usage between Eiffel and C++, but only between the different Eiffel compilers.

5.3 Conclusion and Work in Progress

Each time it is possible, the dynamic dispatch is used into the SmallEiffel source code for various kinds of expressions as well as for various kinds of instructions, various kinds of names and so on. We were ourselves very surprised by the excellent score of 84%. Obviously, the 100% limit cannot be reached for all programs whatever the quality of the inference algorithm used : a simple ARRAY[ANIMAL]

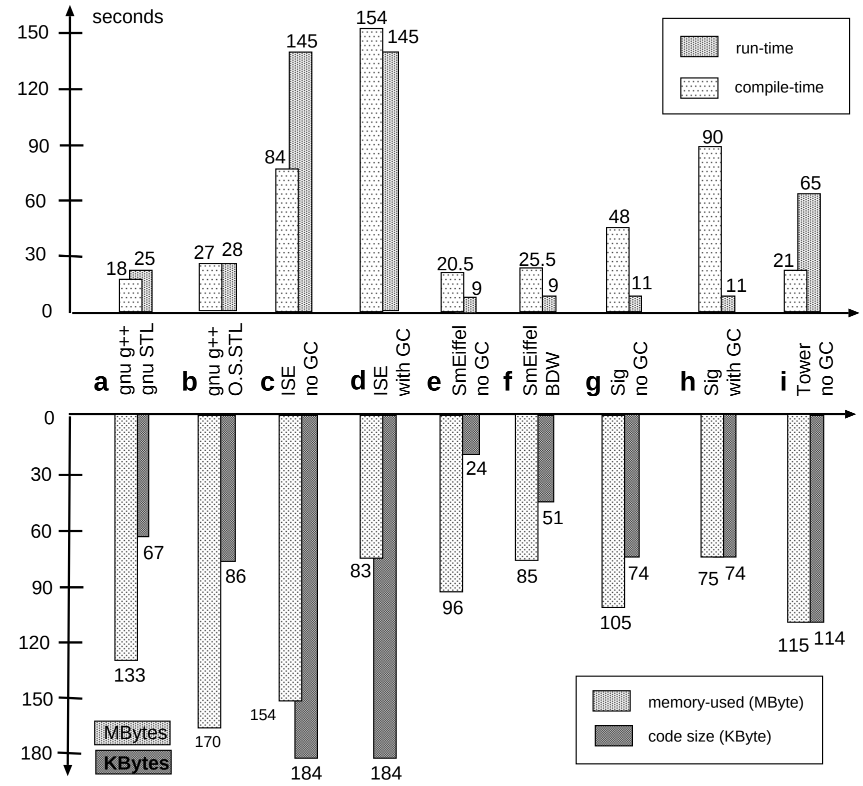

Fig. 14. Results between different compilers.

with mixed DOG and CAT entered at the keyboard is enough to break down any type inference system. Adding flow sensitivity to our compiler will increase the actual score [4]. This is a work in progress. SmallEiffel — as all other Eiffel compilers — is currently unable to check the system-level validity rules of Eiffel. This issue is also our present research theme.

Appendix: Eiffel Source Code

```
deferred class ANIMAL feature
   cry is deferred end;
   imitation(other: ANIMAL): ANIMAL is do Result := other; end;
   choose(other: QUADRUPEDE): ANIMAL is do Result := other; end;
   chameleon(other: ANIMAL) is
     do Current.cry; io.put_string(" imitates "); other.cry;
         io.put_string(" = "); Current.imitation(other).cry; end;
   name: STRING;
   set_name(my_name: STRING) is do name := my_name; end;
   introduction is
     do io.put_string("My name is:"); io.put_string(name); end;
```

```
end -- ANIMAL
deferred class QUADRUPEDE inherit ANIMAL redefine imitation feature
   imitation(a: ANIMAL): ANIMAL is do Result := a.choose(Current); end;
end -- QUADRUPEDE
class DOG inherit QUADRUPEDE feature
   cry is do io.put_string("BARK"); end;
end -- DOG
class CAT inherit QUADRUPEDE redefine choose feature
   choose(quadrupede: CAT): CAT is do Result := Current; end;
   cry is do io.put_string("MEOW"); end;
end -- CAT
class CENTIPEDE inherit ANIMAL feature
   cry is do io.put_string("SCOLO"); end;
end -- CENTIPEDE
```

References

1. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, 1995.
2. Ole Agesen. *Concrete Type Inference : Delivering Object-Oriented Applications.* PhD thesis, Department of Computer Science of Standford University, Published by Sun Microsystem Laboratories (SMLI TR-96-52), 1996.
3. A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, 1977.
4. Diane Corney and John Gough. Type Test Elimination using Typeflow Analysis. In *PLSA 1994 International Conference, Zurich. Volume 782 of Lecture Notes in Computer Sciences, Springer-Verlag*, pages 137–150, 1994.
5. J. Graver and R. Johnson. A Type System for Smalltalk. In *Proceedings of POPL*, pages 139–150, 1990.
6. G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Object Oriented Languages.* Academic Press Limited, London, 1991.
7. B. Meyer. *Eiffel, The Language.* Prentice Hall, 1994.
8. R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, pages 348–375, 1978.
9. Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of 6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'91)*, pages 146–161, 1991.
10. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Series in Computer Science, 1986.
11. N. Suzuki. Inferring Types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, 1981.
12. N. Suzuki and M. Terada. Creating Efficient System for Object-Oriented Languages. In *Eleventh Annual ACM Symposium on the Principles of Programming Languages*, pages 290–296, 1984.
13. J. VitekN, N. Horspool, and J.S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *International Conference on Compiler Construction. Volume 641 of Lecture Notes in Computer Sciences, Springer-Verlag*, pages 237–250, 1992.