MORE ABOUT SYSTEM-LEVEL VALIDITY IN EIFFEL

Yania Crespo González-Carvajal, Miguel Katrib Mora[1]

Published in *Journal of Object Oriented Programming*, July/August 1998

ABSTRACT

A programming language is referred to as doing *static strong type checking* when it detects, before runtime, all possible errors due to the inconsistent use of types.

Because typed object oriented languages (OOLs) support type hierarchies, an inconsistent use of a type might not be detected directly from entity declarations and from the direct text of message passing constructions (the expression texts where we apply operations to object operands). Now type problems might depend on the thread of a program execution.

Can be detected such potential type problems before execution?. Being less conservative than the class-level static type restriction of C++ and more secure than the dynamic type checking of Smalltalk, Eiffel introduces a new concept named *system-level validity*. Type problems producing flaws in the system-level validity are named *system-level type problems*

The literature on Eiffel reports two trials of problems in the system-level validity ([1] p. 358): the problem of *selective inheritance*, and the problem of *covariant redefinition of routine parameters*. This paper introduces two new kind of problems not previously reported, and intends to formalize, as simply and clearly as possible, the different kinds of problems. Moreover, it explains how a detection algorithm can be implemented in an Eiffel compiler [2].

(Published in *Journal of Object Oriented Programming*, USA July/August 1998)

1 INTRODUCTION

A typed language aids programming in four ways : *design*, *readability*, *reliability* and *efficiency*. *Design*, because it helps to model reality. *Readability*, because it helps the reader of the program text to understand its operation. *Reliability*, because it prevents operands from incorrect operations. *Efficiency*, because a compiler can generate better executable code, using the hardware resources in a more efficient way, and decreasing the runtime controls.

Depending on the moment in which the checking of type problem in a program takes place, we can distinguish two kinds of type controls: *static* (done at compile time) and *dynamic* (done in runtime).

In a type system with dynamic control (as Smalltalk), type problem detection is far more retarded. Type problems in a program remain undiscovered until by chance or redesign the offending case is encountered; sometimes after the system is released. Furthermore, dynamic control implies runtime overhead.

Static control, in contrast, detects type problems by analyzing the program text before execution. It supports more reliability and efficiency. But, static control may be too rigid if it forbids the execution of a program which is, in fact, potentially correct.

Typed OOLs support type hierarchies by making the type system richer but more complex. They can support polymorphism and late binding, allowing the construction of more flexible system. But, can a compiler do static type checking of all potential type problems? How can a compiler detect type problems and at the same time not be too pessimistic by rejecting programs that will run safely ?

---

[1]Computer Science Department, Mathematics and Computer Science Faculty, University of Havana
e-mail: mkatrib@comuh.uh.cu, mkm@orion.ccu.uniovi.es, yania@matcom.uh.cu,.

[2]Like the one being applied in the Eiffel/UH compiler which is under way at the Computer Science Department, University of Havana.

Eiffel tries to give the maximum possible type flexibility, but at the same time it tries to guarantee maximum security by means of static type control. In order to do so, it introduces the notion of *system-level validity*. There are potential *type problems* that can produce flaws in this validity but could not be detected during the compilation of a single class because they might depend on which classes compose a system (program) and how they are connected. They are named *system-level type problems.*

Eiffel goes beyond other OOL typed languages, such as C++, in defining these problems. The first two problems [1-3] reported in the Eiffel literature relate to *selective inheritance*, and to *covariant redefinition of routine parameters*. In the following section, we will illustrate these problems, and further on we will introduce other two problems related to *expanded types and conformance*, and to *instance creation*. We will explain simple and clear notation and definitions so as to define the four categories of problems mentioned above. These definitions will help the development of a detection algorithm that is suggested in the last section.

Problem with selective inheritance

Due to the orthogonality of export rules and inheritance, an heir might restrict the export of a feature that has been broadly exported by its parent, for example:

```
class BIRD
  --....
  fly;
  eat;
  --....
end
```

```
class PIGEON
  inherit BIRD

  --... .
end
```

```
class PENGUIN
  inherit BIRD
    export fly {NONE}
  --....
end
```

During compilation of the above hierarchy, there isn't any kind of type problem. Now let's consider the classes below to belong to a system where PRACTICE is the root class. Due to conformance rules, an object of an heir type (PENGUIN or PIGEON) may be attached to an entity of a parent type (as BIRD). Note that the call p. parachute_test(new_parach,b) may attach an object of the PENGUIN type to parameter b of the BIRD type. But, what will happen when *the bird is asked to fly in a parachute test, and that bird turns out to be a penguin!* ? The BIRD class of the static type of the parameter b exports fly, but the class type of the object dynamically attached to b (PENGUIN in this case) doesn't export fly

```
class PARACHUTE
feature
  --...
  open is boolean
   do
     --...
   end
  --....
end
```

```
class PLANE
  ...
 parachute_test (par: PARACHUTE ; b:BIRD)
is do
     --...
      if not par.open then b.fly;
     --...
  end;
  ...
end
```

```
class PRACTICE
  creation make
feature
  make is
    local p:PLANE;   pi:PIGEON;  pg:PENGUIN;
          b : BIRD;
          new_parach : PARACHUTE;
  do
    !!pg; !!pi; !!p;
    --...
     if cond then b = pi
    else b = pg;
    --...
    p. parachute_test (new_parach,b);
     --...
    end;
  ...
end
```

Note that the problem cannot be located in a single class, although the problem reaches its climax in the instruction b.fly (which will result in killing the penguin). The problem is caused by the combination of the classes BIRD, PENGUIN, PLANE, PARACHUTE and PRACTICE in the same system, and by the potential existence of an execution thread where a *penguin* may be attached to a *bird* and later the *bird* may be asked to *fly*. It's important to remark that even the class PLANE, where the crash will arise, could have been compiled before the existence of the class PENGUIN, and we will try to detect this problem without recompiling the class PLANE.

Selective inheritance can be obtained in C++ if a derived class privately inherits from a base class, and then specifies again as public the others public features of the base class. Fortunately, death of the penguin cannot happen in C++ because C++ prohibits transfer of a reference or a pointer from a PENGUIN to a BIRD parameter (if PENGUIN derives privately from BIRD). But, C++ applies this policy even if there is not an instruction like b.fly. This is a *far too pessimistic* philosophy [4] because a C++ compiler will also refuse an application such as the following (including a class ZOO and not the class PLANE):

```
class ZOO
  ...
  food_test (b:BIRD) is do
    --...
     b.eat;
    --...
  end;
  ...
end
```

```
class PRACTICE
  ...
 z:ZOO;   pi:PIGEON;  pg:PENGUIN;
 ...
  if cond then z.food_test(pi)
  else z.food_test(pg)
 ...
end
```

The goal of Eiffel in relation to the definition and detection of system-level type problems, as our former example, is to be less pessimistic, rejecting only those systems in which a potential wrong combination exists --without delaying, in a dangerous highly optimistic way, the problem detection until execution--, but not rejecting type safe systems such as the *zoo* example.

Problem with Covariance

The other problem reported in the Eiffel literature is known as the *problem with covariance and polymorphic attachment.* A routine may be redefined by an heir class, and each formal parameter type can be changed by a subtype (i.e. a type *conforming* to the corresponding parameter type in the parent class). This is known as *covariant redefinition*[3].

Examples justifying covariance come from parallel hierarchies of classes, as the following for patients and surgeons:

```
class PATIENT
  --...
end
```

```
class SURGEON
  operate_on(p:PATIENT) is do
    --if the patient is operated on with anesthesia, then test if
    --patient tolerates anesthesia
  end
  --...
end
```

```
class NORMAL_P
  inherit PATIENT
  --tolerant to
  --anesthesia but not
  --verified as
  --hypnotizabele
  --...
end
```

```
class ALERGIC_P
  inherit PATIENT
  -- not tolerant to
  --anesthesia verified
  --as hypnotizable
--...
end
```

```
class CARDIO_SURGEON
  inherit SURGEON
    redefine operate_on
  --...
    operate_on(np:NORMAL_P)is
   do
    -- no test is done, assume
    -- patient tolerates anesthesia
  end
  --...
end
```

```
class EXPERIMENTAL_SURGEON
  inherit SURGEON
    redefine operate_on
  ...
    operate_on(hp:ALERGIC_P)is
    do
      --operate_on using hypnosis
      --but without anesthesia
      ....
     end
  ....
end
```

Suppose the following program segment in some root class:

[3]Fans are divided between the covariant redefinition and contravariant redefinition. A discussion about this matter goes beyond this paper.

s: SURGEON;
es: EXPERIMENTAL_SURGEON;
np:NORMAL_P;
--...
!!es; !!np;
--a normal patient who is not hypnotizable is
--created
--...
s:= es;
--an experimental surgeon replaces a general surgeon
...--...
s.operate_on(np);
	The construction s.operate_on(np) is correct according to the static type of s. It is statically interpreted as a call to operate_on of SURGEON which accepts a NORMAL_P as parameter because it conforms with PATIENT. Nevertheless, the operate_on of EXPERIMENTAL_SURGEON which expects a parameter of the type ALERGIC_P is going to be applied (because of dynamic binding)

In this example we see that an EXPERIMENTAL_SURGEON could be provisionally operating (maybe because of an emergency) in the operating room of a general surgeon. In a case like this, it is necessary to be careful with a normal patient and not to send him to be operated on in this room (s.operate_on(np)). Due to polymorphism, that *patient* will be operated on by an *experimental surgeon* (supposing his patient is hypnotizable) who will try to operate *without anesthesia!*.

We are dealing with another system-level type problem that cannot be detected during the compilation of each single class because it depends on the coexistence of the above classes in the same system (application program) and on the existence of statements like s:=es and s.operate_on(np).

A presumptuous programmer will argue that he will never commit such an error. But again, Eiffel's interest is not to please vanity but to guarantee reliable software with as few restrictions as possible[4].

These type problems in Eiffel are reported in [1-3]. The reader can found more about type safety in OOP in [5-6]. In the following sections we will propose a more complete and comprehensible definition for the above problems, and we will also submit to your consideration other types problems not considered so far.

2 STATIC AND DYNAMIC TYPE

An *entity* in Eiffel is a language construction that may be a left part of an assignment or a left part of an object attachment such as an attribute of the class, a routine local variable (including the special variable Result in a function routine), or a formal routine parameter.

A simple expression is either an entity, or a single function call. The *static type of this expression* is the type with which the entity is declared, or the type of the function definition.[5]

Recursively, the static type of an expression Expr.x (where x is an attribute or a function with no parameters) or an expression like Expr.x(...) (where x is a function with parameters) is the static type of the entity x in the class corresponding to the static type of Expr.

To simplify, although Eiffel allows infix operator and operator overload, we are not including here expressions like E_1 *op* E_2 because they can be considered as function calls like $E_1.op(E_2)$

So, the static type of any expression appearing in a class can be determined during the compilation of the class. Then, any attempt to apply a feature x to an expression Expr will be reported as an error if there is no definition for the x feature in the class of the static type of Expr , or if that class doesn't export x to the class where the expression Expr.x appears. This can be detected when we compile this later class.

Nevertheless, because of polymorphism, the runtime object attached to an entity x can be of a different type than its static type (due to conformance rules [1], it must be a subtype, i.e., a type corresponding to a descendant class). As the origin of this attachment might have begun in a class other than the class in which

[4]Ask this programmer if he would like to be operated on in a hospital where that kind of controls is not being applied.
[5] To simplify we will not mention a constant expression. Manifest constants are too simple and will not be considered, Eiffel implements other constants as *once* functions.

the entity was declared (by means of parameter transference), it is not possible to detect all types of problems that could occur in a class during its compilation; we need all classes participating in a system (application program).

So, at different times of a program execution, or at different executions of the same program, an expression can be dynamically associated to objects of types which are different from the expression's static type. The types of all objects potentially associated to an expression *E* at any execution of a system *S* will be called *dynamic type (DT) of an expression in S*, and, using object notation, it will be denoted as *E.DT*.

In the example we gave regarding the problem of selective exportation, the instruction x.fly is applied inside the class PLANE. In this case the expression x has static type BIRD, and dynamic types PENGUIN and PIGEON.

Thus, it would be convenient to formalize the ways by means of which in a system *S*, an expression can have a dynamic type different from its static type.

Assignable

> An expression Expr is said to be *assignable* to an entity x inside a routine r (x must be a local variable of the routine r including the special variable Result, or an attribute of the class to which that routine belongs), if the assignment x := Expr appears in the body of r. [6]

Bindable

> An expression Expr is said to be *bindable* to an entity x (being x a formal parameter of a routine r in a class C), if a call like Expr1.r(Expr) exists somewhere in the classes involved in the system *S* and the dynamic type of Expr1 (Expr1.DT) includes C (note that the call may be only r(Expr) if it's inside the class C)

Createable

> An entity x in a class C (local variable of a routine r of C or attribute of C) is said to be *createable* with type T, if in the body of some routine in C, one of the following creation sentences appears:
>
> !!x or !!x.init and the static type of x is T
>
> or
>
> !T!x or !T!x.init;
>
> where T is the static type of x or a type which conforms with the static type of x and init is a creation routine of the type T

If the entity is declared expanded then it's *createable* and its dynamic type always will be equal to its static type.

Now we can formally define the dynamic type of an expression.

Dynamic types of an expression in a class C of a system S

a) If the expression is an attribute x of the class C, the dynamic types of x consist of all the types with which x is *createable*, plus the dynamic types of all expressions *assignable* to x through some routine of the class C.

b) If the expression is a local variable x of a routine r, the dynamic types of x consist of all the types with which x is *createable* (in the body of r), plus the dynamic types of all expressions *assignable* to x inside the routine r.

---

[6] Remember, in Eiffel the attributes of a class are read only when they are visible. Because of this, it is not possible for assignments like a.x:=y to exist.

c) If the expression is a call f(....) to a local member function, its dynamic types involve the dynamic types of the local variable Result in the definition body of the function f in the class C

d) If the expression is a formal parameter x of a routine of class C, its dynamic types are the dynamic types of all the expressions *bindable* to x through the classes involved in the system S.

e) If the expression is of the form $Expr_1.Expr_2$, where $Expr_2$ is either (a) or (c), its dynamic types are the union of all the dynamic types of $Expr_2$ for each class in the dynamic types of $Expr_1$ and the recursive application of cases (a),...,(e).

Note that the dynamic type of an expanded entity is the same as its static type.

3 SYSTEM-LEVEL TYPE PROBLEMS

3.1 Problem with selective exports

In a system S, there exists a System-level Type Problem with Exports if there is a class C with an expression such as Expr.f where Expr has some dynamic type T and T does not export the feature f to the class C.

The first example considered above corresponds to this case. The expression b.fly occurs in the class PLANE (in the body of the routine parachute_test), here PENGUIN is a dynamic type of b (because pg is *createable* with the type PENGUIN and pg is *bindable* to b) and, the class PENGUIN does not export fly to the class PLANE.

3.2 Problem with covariance redefinition

There exists a System-level Type Problem with Covariance redefinition, when we have a statement like $Expr_1.f(Expr_2)$ where T is a dynamic type of $Expr_2$ and there exists a dynamic type of $Expr_1$ which redefines f and changes the type of its parameter to T´ and T does not conform with T´.

The problem will occur because we are attempting to call a routine which is expecting an argument of type T´ and we are trying to bind it with a real parameter of type T (which does not conform with T´).

Let's now consider the statement s.operate_on(np), of the second example in section 1. The formal parameter p of operate_on (in the class SURGEON static type of s) has type PATIENT. Accordingly it may be called with an actual parameter as np of type NORMAL_P conforming to PATIENT. However, s can have dynamic type EXPERIMENTAL_ SURGEON who redefines operate_on expecting a parameter of type ALERGIC_P, but NORMAL_P doesn't conform to ALERGIC _P.

3.3 Problem with conformance

Let's examine new system-level type problems.

There is a System-level Type Problem with Conformance when the source expression in an association (assignment or parameter transference) has a dynamic type which does not conform with the static type of the target entity of the association (the left part of the assignment or the formal parameter in the routine).

Three particular cases make up the above general statement. The first of these 3 sub-cases was reported by the authors in a previous article [7]. Having now generalized the definition we subsume all 3 sub-cases under the moniker of the 3rd system-level type problem.

3.3.1 Problem in the Conformance with expanded

Reference objects coexist with expanded objects. In order to manage this coexistence, Eiffel defines a way to associate expanded to non-expanded objects and vice versa as shown in the following table (let a class DERIVATIVE which inherits from a class BASE).

```
b    : BASE;
e_b: expanded BASE
...
!!b;
e_b:=b;  --Correct, because b and e_b have
         --the same static type BASE. A bit copy
         --is done.
```

```
d    : DERIVATIVE;
e_b: expanded BASE
...
!!d;
e_b:=d;     -- Wrong. The type DERIVATIVE does not
            -- conform with expanded BASE
```

There exists a system-level type problem 3.1 or ***Problem in the Conformance with expanded types*** *when there is an attempt to associate an entity* x *having static type* expanded T *with an expression having a dynamic type different from* T *or* expanded T.

The example below illustrates this problem

```
class A
  --...
  x:expanded BASE;
  f(y:BASE) is do
   x:=y;
  --...
  end;
  --...
end
```

```
class B
  a:A;
  --...
  g(z:BASE) is do
   a.f(z)
  --...
  end;
  --...
end
```

```
class C
  ...b:B; d:DERIVATIVE;
  --...
    b.g(d);
  --...
end
```

The static type of x in x:=y is expanded BASE, but the dynamic type of y includes the type DERIVATIVE.

If expanded types were included in the language in order to achieve efficiency, then this efficiency should have been achieved by the compiler and not by the programmer. If expanded types were included for design goals (such as being able to express non-shared objects, or objects without identity) this should have been achieved by means of better language features. Anyway, the problem above could be avoided if Eiffel forbid association between reference and expanded types. What design objective could have been required this kind of association?.

A discussion between expanded and reference types goes beyond the scope of this paper (see for example [8]). For the present, we will consider this type problem in our algorithm.

3.3.2 Problem in the Conformance with BIT type.

The BIT type is one of the predefined types in Eiffel, so it is legal to declare entities like x : BIT N, where N is a constant specifying the number of bits used to represent x.

Let's use examples to illustrate the conformance rules related to the BIT type.

```
a : BIT N1;
b : BIT N2;
...
a:=b; -- Correct if N2≤N1
```

```
a : BIT N;
b : T;
...
a:=b; -- Correct if the number of bits needed to
      -- represent T is ≤ N
```

Denoting by bitsT the constant which represents the bit size of the type T, we find a system type problem if:

There exists a constant N such that, bitsBASE≤N≤bitsDERIVATE and

```
b       : BASE;
d       : DERIVATIVE;
N_bits : BIT N;
...
!!d;
b:=d;          -- associate entity d to entity b
...
N_bits:=b;  --Attempting to assign to entity N_bits the entity b dynamically
            -- associated with entity d.
```

Now we are able to formalize this kind of problem.

There is a system-level type problem 3.2, or ***Problem in the conformance with BIT type,*** *if there is an attempt to associate to an entity of type* BIT N*, an expression including a dynamic type* T *where* bitsT > N.

3.3.3 Problem with Conformance and Attributes Redefinition

An heir class in Eiffel may redefine the type of an attribute, inherited from a parent class, to a conforming type. Related to this property a new case of system type problem appears (this problem was indirectly suggested in [9] but not completely formalized). Suppose a class A with heirs A1 and A2, and a system *S* that also includes the following classes:

class C
--... .
a: A;
f(par: A) is do
a :=par;
--...
end;
end;
	Non apparent problems in this assignment.	Class C1
		inherit C
		redefine a
		--...
		a:A1
		--...
		end;
			class D
			a2:A2;
			c1:C1; c:C;
			--...
			c:=c1;
			--...
			c.f(a2)
			--...
			end;

Apparently there are no problems with this call because a2 conforms to the static type (A) of par in C. However, the current object, which is the entity c (attribute of D), may have a dynamic type C1, and in C1 the type of a is redefined as A1. Then there is a problem in the assignment a:=par because the dynamic type of par (A2) will not conform to A1.

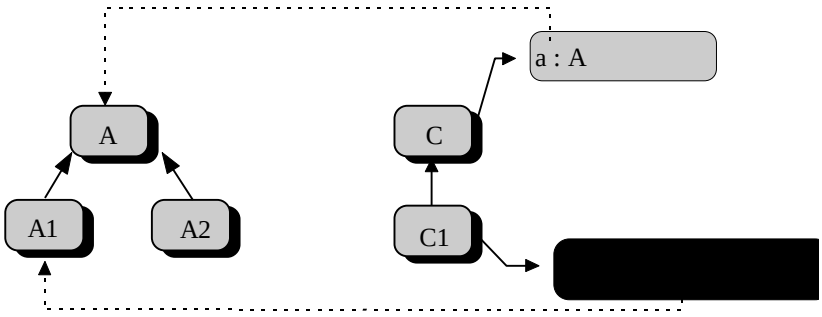

Figure 1

There exists a system-level type problem 3.3, or ***Problem with Conformance and Attributes Redefinition****, if there is a routine involving an assignment to an attribute* (a:=par in the routine f of the class C) *that is redefined in a heir class* (a is redefined with type A1 in the heir class C1)*, and the right part of the*

assignment (par) *has a dynamic type which does not conform with the redefined type of the attribute* (par has A2 in the dynamic type, and A2 doesn't conform with A1).

3.4 Problem With creation of instances

There is a problem that it is no clear that can be inferred from the creation rules and system-level creation validity ([1] p. 286-288). So, finally this paper introduces a fourth category of problems related to attribute redefinition and instance creation. The table below shows one of the two cases of the problem:

<table>
<tr>
<td>class A
--...
creation
create_a
--...
end;</td>
<td>class A1
inherit A
creation
create_a1
--...
end;</td>
<td>class B
--...
a: A;
--...
f is do
!!a.create_a;
--...
end;
--...
end;</td>
<td>class B1
inherit B
redefine a
--...
a: A1; --correct because A1
--inherits from A
--...
end;</td>
</tr>
<tr>
<td colspan="2">class APPLICATION
--...
b1:B1;
!!b1;
--...
b1.f;
--...
end</td>
<td colspan="2">Calling f through b1 attempts to create the attribute a (with type A1 in b1) using a creation routine create_a which is not a proper creation routine for objects of type A1.</td>
</tr>
</table>

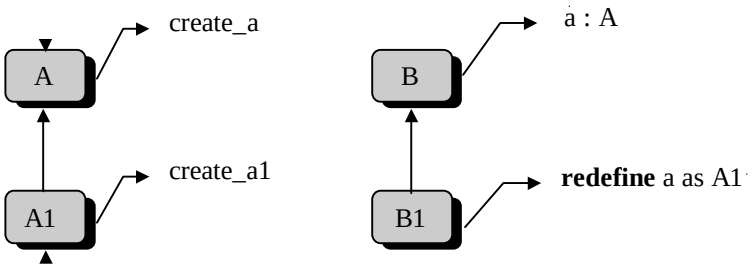

Figure 2

There exists a system-level type problem 4.1 or Problem with creation of redefined attributes, if there is a routine f in a class B where a creation of an attribute of type A occurs (using a creation routine of A). This attribute is redefined (by a class B1, an heir of B) and at the same time a call to .f occurs through an expression including B1 in its dynamic type. This leads to an attempt to create the attribute redefined in B1 by using a creation routine which is no longer a proper creation routine for objects of the new type.

If Eiffel would consider creation routines as class routines instead of object routines, then the problem above could be expressed in a style as follows:

class A --... creation create_a --... end;	class A1 inherit A creation create_a1 --... end;	class B --... a: A; --... f is do **a:= !!A.create_a;** --... end; --... end;	class B1 inherit B redefine a --... a: A1; --... end;

where the **!!A.create_a** would be a new form of creation[7], then the problem 4.1 would be a special case of the problem 3.3.

Suppose a class A with heirs A1 and A2 , then the following table depicts the second kind of problem:

class B --... a: A; --... f is do !A2!a;--correct because A2 -- inherits from A --... end; --... end;	class B1 inherit B redefine a --... a: A1; --correct because A1 --inherits from A ... end;	class APPLICATION --... b1:B1; --!!b1; --... b1.f; --... end

Calling f through b1 attempts to create the attribute a (with type A1 in b1) by using a polymorphic creation !A2!a, but A2 does not conform to the type A1.

There exists a system-level type problem 4.2 or Problem with creation of redefined attributes, if there is a routine f in a class B where a creation of an attribute of type A occurs (using a polymorphic creation as !A2!a where A2 is heir of A). This attribute is redefined as A1(by a class B1, an heir of B) and at the same time a call to f occurs through an expression including B1 in its dynamic type. This leads to an attempt to create the attribute redefined in B1 as an object of type A2 which does not conform to A1.

As in the previous case, if we had a construction as a:= !!A2 then the above problem 4.2 would be special case of the problem 3.3.

4 IMPLEMENTATION

The rich Eiffel syntactic and semantic model doesn't allow a compiler to work in a simple one-pass mode. Thus, from the point of view of compiling techniques, it is advisable to include an intermediate representation for classes. In Eiffel/UH all information on an Eiffel class is encapsulated in a single object. This object is referred to as EU (Eiffel Unit). It is persistent and it doesn't depend on projects (Ace files). It depends only on the relations between classes. When the source text of a class is compiled (as an isolated compilation of a single class[8] or as part of the compilation of an entire Eiffel application system), an EU will be constructed for each compiled class. With *Smart Compiling* techniques (see [10]) not all classes in the system will be recompiled as a system is extended or modified.

[7] Authors' opinion is that this would be a better form of creation. In this way anonymous objects could be created in a more functional notation without using local variables. It could be very useful for the declarative notation of assertions.

[8] Unfortunately in several commercial Eiffel compilers you cannot easy compile a single class from scratch. It must be included in a project that you must define trough an Ace file. We think this limitation undermines the *component culture* that Object Oriented Programming tries to encourage.

As a result of the compilation of a class, its EU holds:

- All left part entities of the related source class (attributes, local variables, formal parameters)
- All expressions (i.e. constructions calculating an object)
- All procedure (non function routine) calls

In other words, if in a source text of the class, there is a cascade of messages *E1.E2.…..En* where *E1* is an entity or function call, each *Ei 1< i < n* is an attribute or function call, and *En* is either an attribute, or a function call, or a procedure call, then the EU holds all the subexpressions *E1, E1.E2, E1.E2.E3* and so on.[9]. Together with those subexpressions, the EU holds their lengths: i.e, the length of the subexpression *E1.E2…..Ek* is k.

Each entity or expression (subexpression) *E* in the EU will have an associated set to hold its potential dynamic types (*E.DT*). After class compilation (class level), only the dynamic types of such *createable* entities (attributes and local routines variables) have been computed.

When, before the execution of a system, the compiler arranges all EUs (to be executed by an interpreter either to generate C or other target code) the entire set of the dynamic types for each entity and expression will be computed. At this level (*system-level*), due to the *Bindable* definition (see section 2) the other classes (EUs) of the system may participate in the construction of the dynamic types.
Note that the client-graph of system classes may have cycles. To avoid an infinite algorithm a fixed point iteration must be applied. A general algorithm, in informal notation, follows: an expression, local variable, or formal parameter *E*, inside a routine *r* of a class *C* will be denoted as (*C, r, E*), an attribute *a* of the class *C* will be denoted as (*C, _, a*).

```
-- During class compilation the DTs of all attributes and local variables were initialized to
-- its corresponding createables, the DTs of other expressions were initialized to empty
loop
until no E.DT  increases its size
do for all classes C in the system
   do for all routines r in C
      do
        -- update assignables. Add to the dynamic types of x the dynamic types of E
        for all assignment sentences x:= E of the routine r of C
        do  if x has static expanded type then check problem 3.1with E
              else (C, r, x).DT := (C, r, x).DT + (C, r, E).DT;

       -- update bindables
        for all sentence or expression E.f(E1, …..,En) of the routine r of C
        do for all corresponding class CT of a type T in (C, r, E).DT
           do for all   formal parameter p of  f  in CT
                    and Ei the corresponding actual parameter in E.f(E1, …En)
             do if p  has static expanded type then check problem 3.1with Ei
                  else (CT, f, p).DT:=(CT, f, p).DT + (C, r, Ei).DT;

      --update dynamic types of single expressions or expressions with length 1
        for all single function calls f (with or without parameters) in a routine r of C
        do  (C, r, f).DT := (C, f, Result).DT;

      --update dynamic types of complex expressions or expressions with length >1
        for all expressions E with length n = 1 to max_length-1 in the routine r of C
        do for all expressions E.f or E.f(….) of length n+1 in the routine r of C
           do for all corresponding class CT of a type T in (C, r, E).DT
              do if f is attribute then (C, r, E.f).DT:= (C, r, E.f).DT + (CT,_, f).DT
```

[9] Don't worry about the cost of such representations, in any case a compiler always does something similar when the parser constructs internal syntax trees.

```
            else – f is a function
                (C, r, E.f).DT:= (C, r, E.f).DT + (CT, f, Result).DT
        end – all routines r in C
    end – all classes C
end –general fixed point loop
```

Note that the last section of the algorithm avoids unnecessary loops because it isn't recursive, it begins with the smallest expressions ending with the largest ones.

Once all the dynamic type set have been computed we can detect system-level type problems checking if one of the problems 3.1 to 3.4 arises.

The application of this algorithm to the ancient example of *selective inheritance* is illustrated in the appendix.

5 CONCLUSIONS

Three important goals have been achieved. First the notion of *system-level type problem* has been clarified. Second, new kinds of problems have been detected and explained. Third, a real, and easy-to-implement algorithm to detect those problems has been proposed.

To the best of our knowledge the commercial Eiffel compilers don't perform system-level type problem static checking. Users may be worried because the above problem detection algorithm may be too expensive. A real Eiffel environment could offer the option to do or not to do system-level type problem checking. Then, the risk of an abnormal execution of a system, due to inconsistent use of types, will be a user's explicit decision not a language flaw.

Our proposed algorithm requires additional processing; on the other hand it enjoys as simple as possible expression. The checking might be enhanced introducing the notion of *potential execution thread* of a system. (i.e., starting at the root class and at the creation routine to be applied we can try to construct a tree of all possible called routines in the system. Then the above dynamic type set should be computed only for expressions in those routines.). This could reduce the number of potential problems but overload the algorithm

To apply a more restrictive and pessimistic policy, to avoid the complexity of the compiler construction , as C++ does, would be a too easy option. To promote a too optimist policy, doing all type checking in runtime as Smalltalk does, would be a too risky option. The Eiffel language follows a better approach with the notion of system-level validity checking. That current commercial compiler do not offer an efficient detection algorithm is an implementation limitation.

Are all these efforts justified by the design and programming benefits introduced by the language resources that generate the problems? This a interesting question that might require more discussion, but it goes beyond the scope of this paper. Covariance vs. Contravariance have both followers and opponents. The inclusion of expanded types (meaning value types) and their coexistence with reference types introduce problems, not only like the one discussed, but compromises the semantics of the language. What are the semantics of associating a reference entity to a value entity and vice versa? Is it justified from the design point of view? Would other language constructions be better?

There are entities whose dynamic types may be modified by the action of an external method. To apply the present approach in external methods is not practicable because we should construct a new compiler for each of the external languages. Therefore, in the current implementation of the present work, whenever a potential external association to an entity occurs, a check mark is inserted in the *assignables* of the entity. This mark is propagated to the dynamic types of the expressions associated with the entity. Finally a warning will be reported to the user, who must take the final decision. Classes which their "external" methods are implemented by the compiler, such as ARRAY, ANY, STRING, might be excluded from this consideration.

The system-level type problem checking algorithm becomes more complicated if it would be introduced the notion of persistence. The main question here is: *How can we compute the Dynamic Types of an expression that is the result of the load of a persistent object from some object repository (object data base, file, stream)?*. The current reverse assignment mixed with the Eiffel class *STORABLE* is not a complete solution to persistence, that is why we have not included its analysis in the present paper. One solution could be that the repository will be static typed with some root type (not necessarily *ANY*). This root type could serve as base class of the type hierarchy for all objects that could be stored in the repository, then the detection algorithm might consider all the descendent types of that type. A real persistence should allow new descendent types to be defined, and objects of that type could be stored in the repository, even beyond the compilation of the system in which we apply the above static detection algorithm. Can we do that without the recompilation of the system? This is an open problem to be solved.

Acknowledgments

This paper was prepared with the support of DGICYT, Education and Science Ministry of Spain, SAB 95-0038. We thank: Tomás Couso Alvarez and Cristina Cuenca López (University of Havana). The Alma Mater grant 94 of the University of Havana supported partially the practical proofs of this work.

REFERENCES

[1] Meyer, B. *EIFFEL: THE LANGUAGE*, Prentice Hall,1992.

[2] Meyer, B. *Static typing for Eiffel* An Eiffel Collection, ISE Eiffel, January 1989.

[3] Hart, F. and Hillegas, A. *Bending the rules* JOOP, Vol. 5, No.8, January 1994.

[4] Katrib, M. *Object Oriented Programming C++ vs Eiffel*, Lectures Notes of the 5th International School in Selected Computer Science, México 1994.

[5] Palsberg, J. and Schwartzbch, M.I. *Object-Oriented Type Systems*, John Wiley & Sons,1994.

[6] Coen-Porisini, A., Larazza, L. and Zicari, R. *Assuring type safety of object-oriented languages*, JOOP,Vol. 5 No. 9, February 1994.

[7] Crespo, Y., Katrib, M., and Surribas, E. *A third problem in Eiffel requiring type checking to system-level*, Eiffel Outlook, Vol 4, Num 5, September/October 1995.

[8] Kent, S. and Howse, J. *Value Types in Eiffel*, Proceedings of TOOLS Europe 96, Prentice Hall 1996.

[9] Ruckert, M. *Extensible subobjects in C++* , JOOP Vol. 9, No. 4, July/August 1996.

[10] Couso, M., Katrib, M. and Mateo, M. *Smart Compiling in Eiffel*, to be published.

APPENDIX

Let's examine the creation of the described sets and the computation of the dynamic type set applied to our former example.

Initializing dynamic types of expressions at compilation time of each class

---

Class PARACHUTE

Initialized the createables. Result is BOOLEAN, an expanded type, so the creation is implicit
(PARACHUTE, open, **Result**).DT={BOOLEAN}

Class PLANE

Initialize the dynamic type set of the formal parameters
(PLANE, parachute_test, **par**).DT = { }
(PLANE, parachute_test, **b**).DT = { }

Initialize the dynamic type of the expressions
(PLANE, parachute_test, **par.open**) = { }

Class PRACTICE

Initialize the createables
(PRACTICE, make, **pi**).DT={PIGEON}
(PRACTICE, make, **pg**).DT={PENGUIN}
(PRACTICE, make, **new_parach**).DT={PARACHUTE}
(PRACTICE, make, **p**).DT={PLANE}

Initialize the dynamic type set of expressions
(PRACTICE, make, **b**).DT={ }
(PLANE, parachute_test, **par.open**).DT = { }

Trace at system assemble time:

DT := 0;

Fixed Point Loop Begin

-- First iteration

- Old_DT := DT;
- update assignables

  (PRACTICE, make, **b**).DT is { }
  b:= pi in PRACTICE class implies
  (PRACTICE, make, **b**).DT :=(PRACTICE, make, **b**).DT + (PRACTICE, make, **pi**).DT
  { } + { PIGEON }
  then increments Dynamic_Types, DT = 1.

  (PRACTICE, make, **b**).DT is { PIGEON }
  b:= pg in PRACTICE implies
  (PRACTICE, make, **b**).DT :=(PRACTICE, make, **b**).DT + (PRACTICE, make, **pg**).DT
  { PIGEON } + { PENGUIN }
  then increments Dynamic_Types, DT = 2.

- update bindables

  (PLANE, parachute_test, **par**).DT is { }

  The call p.parachute_test(new_parach,b) in the routine make in class PRACTICE implies

  (PLANE, parachute_test, **par**).DT := (PLANE, parachute_test, **par**).DT + (PRACTICE, make, **new_parach**).DT
  { } + { PARACHUTE }

  then increments Dynamic Types, DT= 3.

  (PLANE, parachute_test, **b**).DT = { }

  The call p.parachute_test(new_parach,b) in the routine make in class PRACTICE implies

  (PLANE, parachute_test, **b**).DT := (PLANE, parachute_test, **b**).DT + (PRACTICE, make, **b**).DT
  { } + { PIGEON, PENGUIN }

  then increments Dynamic Types, DT= 4.

- update dynamic types of single expressions or expressions with length 1

  Nothing occurs in this step

- update dynamic types of complex expressions or expressions with length >1 (in incremental order)

  Compute the dynamic type set of the expression par.open

  (PLANE, parachute_test, **par**).DT is { PARACHUTE } then

  (PLANE, parachute_test, **par.open**).DT := (PLANE, parachute_test, **par.open**).DT + (PARACHUTE, open, **Result**).DT
  { } + { BOOLEAN }

  then increments the Dynamic Type, DT = 5.

DT = 5 and Old_DT = 0, then repeat the loop

Old_DT := DT;

-- Second iteration

--... update the dynamic types

DT = 5 and Old_DT = 5 then

Fixed point loop finish

The computed dynamic type sets are:

(PARACHUTE, open, **Result**).DT={BOOLEAN}
(PLANE, parachute_test, **par**).DT = { PARACHUTE}
(PLANE, parachute_test, **b**).DT = { PIGEON, PENGUIN }
(PLANE, parachute_test, **par.open**).DT = {BOOLEAN}
(PRACTICE, make, **pi**).DT={PIGEON}
(PRACTICE, make, **pg**).DT={PENGUIN}
(PRACTICE, make, **new_parach**).DT={PARACHUTE}
(PRACTICE, make, **p**).DT={PLANE}
(PRACTICE, make, **b**).DT={PIGEON,PENGUIN}

A type problem 1 will occur because the sentence b.fly appears in the routine parachute_test of the class PLANE, b has DT { PIGEON, PENGUIN } and PENGUIN doesn't export fly.