Static typing*

Bertrand Meyer
ISE Inc.
270 Storke Road Suite 7
Goleta, CA 93117 (USA)
<meyer@eiffel.com>, http://www.eiffel.com

Every once in a while, in the history of science, there arises a problem whose statement is so deceptively simple as to fit in a few sentences that a curious adolescent can understand, and whose solution baffles the best minds of a generation. It is of such a problem, central to the understanding and use of object-oriented principles, that I propose to present both the statement and a solution.

The simplicity of the problem comes from the simplicity of the object-oriented model of computation, introduced almost thirty years ago by Professors Dahl and Nygaard. If one puts aside the details of an object-oriented language, necessary to write realistic software but auxiliary to the basic model, only one kind of event ever occurs during the execution of an object-oriented system: routine call. In its general form it may be written, using the syntax of Simula and Eiffel, as

$x.f(arg)$

meaning: execute on the object attached to x the operation f, using the argument *arg*, with the understanding that in some cases *arg* stands for several arguments, or no argument at all. Our Smalltalk friends would say "pass to the object x the message f with argument *arg*", and use another syntax, but those are differences of style, not substance. At run time, this is what our systems do: calling features on objects, passing arguments as necessary. That everything relies on this canonical scheme accounts in part for the general feeling of beauty that object-oriented ideas arouse in many people.

From the simplicity of the model follows the simplicity of the typing problem, whose statement mirrors the structure of the Basic Construct:

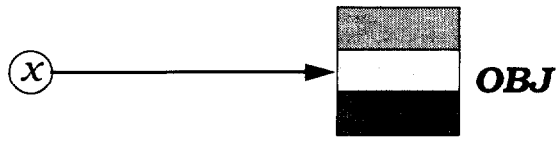
The typing problem

When, and how, do we know that:

1 • There is a feature corresponding to f and applicable to the object?

2 • *arg* is an acceptable argument for that feature?

The policy known as static typing, which I will argue is the only reasonable one for professional software development, states that we should answer the "when" part of the question by "before we ever think of running the system", and the "how" part by "through mere examination of the software text".

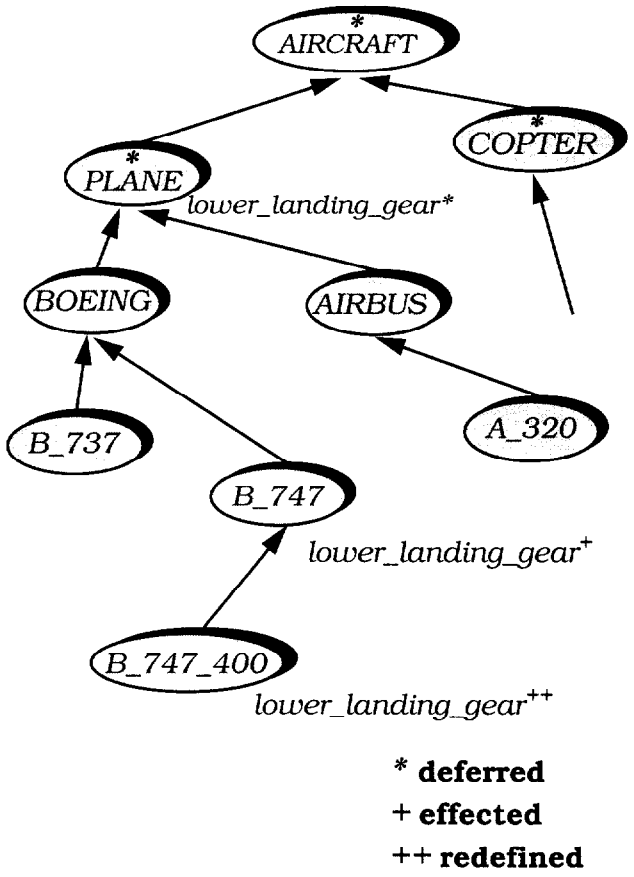


Let us make the terminology precise. x will be called an entity; this is a generalization of the traditional notion of variable. f will, in interesting cases, be a routine; Smalltalk would call it a method, but there is no need for a new term since the older one is well established. At run time the value of x, if not void, will be attached to a certain object, ***OBJ*** on the figure.

*This article reproduces the text of the author's OOPSLA keynote address. That it is the transcript of a verbal presentation explains that it does not have all the usual trappings of a scientific article and makes generous use of the first person singular. The final part of the speech (general comments about the respective roles of technical and methodological work, about OOPSLA, and about the current state of development of Eiffel) has been removed but is available from the author. See also the Web page.

Listeners already familiar with the issues of typing will perhaps appreciate a preview of the conclusion. It can be expressed in reference to a conjecture that Pierre America from Philips Research Laboratories expressed in a panel on the topic at TOOLS EUROPE a few years ago. America stated that three properties are desirable: static typing, substitutivity, and covariance; his conjecture is that one can achieve at most two of them. The aim of the present work is to disprove the America conjecture and show that we can enjoy static typing and covariance while preserving substitutivity when it is needed and safe.

If you do not completely understand these terms do not worry; they will be explained shortly. Starting now with the basic concepts we must first make sure to avoid any misunderstanding. Beginners with a Smalltalk background sometimes confuse static *typing* with static *binding*. Two separate questions are involved. Typing, as noted, determines when to check that the requested routine will be applicable to the requested object; binding determines what version of the feature to apply if there is more than one candidate.



For example in the hypothetical inheritance hierarchy shown on the preceding figure, we might define the feature *lower_landing_gear* only at the level of *PLANE*, not at the general level of *AIRCRAFT*. Then for a call of the form

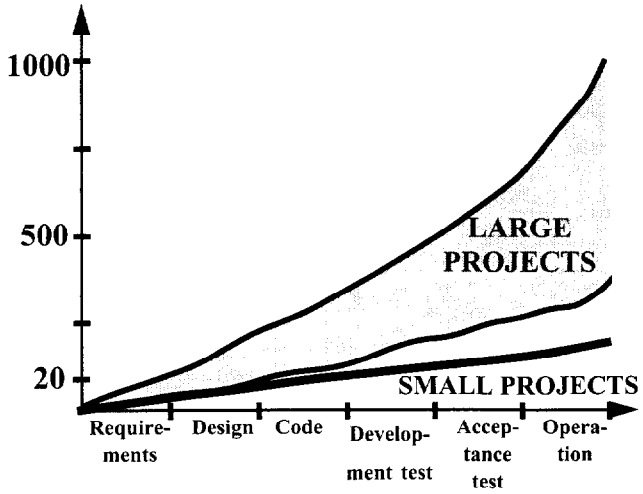
my_aircraft.lower_landing_gear

two separate questions arise: **when** to ascertain that there will be a feature *lower_landing_gear* applicable to the object; and, if there is more than one, **which one** to choose. The first question is the typing question, the second is the binding question. Both answers can be dynamic, meaning at execution time, or static, meaning before execution. For safety, flexibility and efficiency, the proper combination is, I believe, the Eiffel one: static typing and dynamic binding.

Static binding, which if I understand properly is the default C++ policy, would mean that we disregard the object type and believe the entity declaration, leading us for example to apply to a Boeing 747-400 the version of a feature, such as *lower_landing_gear*, that has been defined for the standard Boeing 747 planes. This does not seem right, so we should choose dynamic binding, defined simply as applying **the right feature**. Dynamic binding, as many of you know, is crucial in ensuring the decentralized, evolutionary system architectures made possible by the object-oriented method. Except when it has the same semantics as dynamic binding — and that is a property best left for an optimizing compiler to ascertain — static binding is always wrong; dynamic binding, as in Eiffel and Smalltalk is always the proper policy. (C++ developers can obtain dynamic binding for specific features, but only by declaring them explicitly as "virtual".) The cost of dynamic binding can be extremely small with a good compiler and a supporting language design (in which, incidentally, static typing will help tremendously, as it does for compiling Eiffel).

Dynamic *binding* does not imply dynamic *typing*. Typing is about something else: when to determine that there will be **at least one feature** applicable to the object. *Dynamic* typing, as in Smalltalk, means waiting until the execution of the feature call to make that determination. *Static* typing, as in Eiffel, means performing the check before any execution of the software; typically, this will be part of the verifications made by a compiler.

It is hardly necessary to emphasize the importance of static typing. Anyone concerned with software reliability knows how much more expensive it is to detect errors late in the lifecycle. This is confirmed quantitatively by Barry Boehm's well-known studies (see *Software Engineering Economics*, Prentice Hall, 1981):



This insistence that the language should permit static type checking is one of the major differences between Eiffel and Smalltalk, and the Smalltalk policy is one of the reasons why I am always skeptical about using Smalltalk for serious industrial developments. After all, run time is a little late to find out whether you have a landing gear.

For static type checking to be possible, the language must be designed accordingly. Here are the basic rules as they exist in Eiffel, which at first sight seem to allow a compiler, global or incremental, to ascertain type safety.

Simplifying a little, there are three rules; one applies to declarations, the second to feature calls, and the third to attachments, that is to say assignments and argument passing.

First, we require that every entity be declared with a certain type:

Declaration rule

Every entity must be declared as being of a certain type.

For example:

x: *AIRCRAFT* ;

n: *INTEGER* ;

bal: *BANK_ACCOUNT*

Only to the inexperienced will this appear to be constraining. Seasoned software engineers know that software written once is read and rewritten many times, and that the small effort of declaring the type of every entity is generously repaid by the readability that such declarations bring to the software text.

Next, for what we have called the basic operation of object-oriented computing, we require that any feature call use a feature that exists in the base class of the target *x*:

Call rule

If a class *C* contains the call

$x.f(...)$

there must be a feature of name *f* in the base class of the type of *x*, and that feature must be available (exported) to *C*.

This is easy to determine thanks to the preceding rule, which causes the type of *x* to be known clearly and unambiguously to anyone who reads the class text.

The "base class" of a type defines the applicable features. A non-generic class is both a type and its base class, but the distinction is needed when we consider generic classes: *LIST* [*INTEGER*] is a type, with *LIST* as its base class.

Finally, we have a rule regarding attachment: for any assignment of an expression *y* to an entity *x*, the type of *y* must be compatible with the type of *x*.

Attachment rule

In an assignment *x* := *y*, or the corresponding argument passing, the base class of the type of *y* must be a descendant of that of *x*.

In a classical language such as Pascal or Ada, we would require identical types. Thanks to inheritance the compatibility requirement is more flexible here:

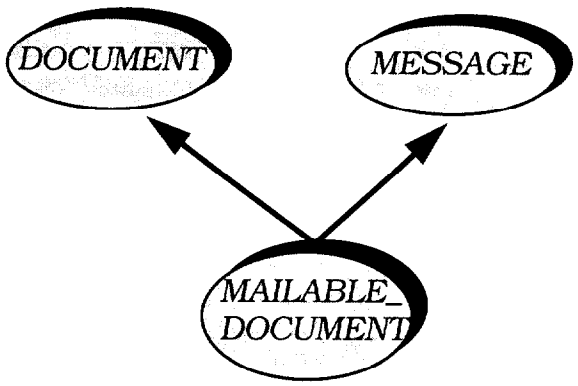
the type of y may be any descendant, in the sense of inheritance between the base classes, of the type of x. The same rule applies to the case in which x is a formal argument to a routine and y is the corresponding actual argument in a call. The term "attachment" covers both cases — assignment and actual-to-formal association.

In a simple world these rules would suffice. They are easy for a software developer to understand, and easy for a compiler to implement. In particular, the compiler can check them incrementally. One of the achievements of ISE's Eiffel compiler development, known as the **Melting Ice** technology, has been to show that it is possible to guarantee type checking and efficient code generation, as in compiled environments, while avoiding the long edit-compile-link-execute cycles traditionally required in such environments; using Eiffel, one can get the fast turnaround that people have come to associate with Lisp and Smalltalk while preserving efficient code generation and static typing. Even for systems of hundreds of thousands of lines, the time to recompile after changing a few classes is typically a few seconds, including the time for full type checking.

Strangely enough, one encounters objections to the static typing approach. These objections do not hold on further examination, but they do highlight the set of properties that must be satisfied by a realistic use of static typing.

First, the type system must have **no loopholes**. There is no such thing as "a little bit statically typed" (as in the famous "a little bit pregnant"). Either the language is statically typed or it is not. The C++ approach, where you can still *cast* — that is to say convert — a value into just about any type, defeats in my view the principle of static typing. For one thing, it makes garbage collection, a required component of serious object-oriented computing, very difficult if not impossible.

Second, a statically typed language requires **multiple inheritance**. The objection against typing often heard from people with a Smalltalk background is that it prevents one from looking at objects in different ways. For example an object of type *DOCUMENT* might need to be transmitted over a network, and so will need the features associated with objects of type *MESSAGE*.



But this is a problem only in languages such as Smalltalk that do not permit multiple inheritance. Multiple inheritance, of course, must be handled properly, with mechanisms as in Eiffel for taking care of name clashes, conflicting redefinitions, and potential ambiguities in repeated inheritance. I mention this because one still encounters people who have been told that multiple inheritance is tricky or dangerous; such views are usually promoted by programmers using languages that do not permit multiple inheritance, and are about as convincing as opinions against sex emanating from the Papal nuncio.

Next, static typing requires **genericity**, so that we can define flexible yet type-wise safe container data structures. For example a list class will be defined as

class *LIST* [*G*] ...

Genericity in some cases, needs to be **constrained**, allowing us to apply certain operations to entities of a generic type. For example if a generic class *VECTOR* has an addition operation, it requires an addition also to be available on entities of type *G*, the generic parameter. This is achieved by associating with *G* a generic constraint *NUMERIC*:

class *VECTOR* [*G* –> *NUMERIC*] ...

meaning that any actual generic parameter used for *VECTOR* must be a descendant of this class *NUMERIC*, which has the required operations such as "plus" and "minus".

We also need a mechanism of **assignment attempt**. This makes it possible to check that a certain object, usually obtained from the outside world, for example a database or a network, has the expected type. The assignment attempt x ?= y will assign to x the value of

y if it is of a compatible type, but otherwise will make *x* void. This instruction (which has been made available to other languages under the name *type-safe downcasting*) is one of the inventions of which the Eiffel community can be proudest.

Also necessary are **assertions**, associated, as part of the idea of Design by Contract, with classes and features. Assertions, directly associated with object-oriented constructs in the form of preconditions, postconditions and class invariants, make it possible to describe the semantic constraints which cannot be captured by type specifications.

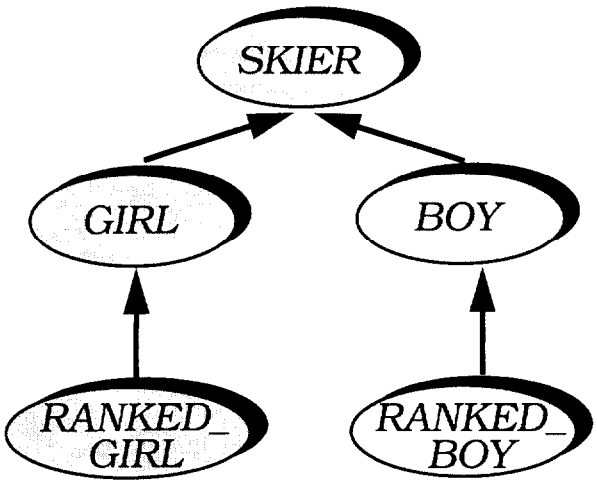
Finally, a realistic object-oriented type system will require two mechanisms that will be described in more detail shortly: **covariance**, which governs how we can redefine the signatures of routines, and **anchored declarations**, of the form

y: ***like*** *x*

which avoid endless type redeclarations.

Ideally, a presentation of typing should stop here. Unfortunately, the combination of static typing with other requirements of the object-oriented method makes the issues more difficult than they appear at first.

The principal problem is what happens when we redefine a feature's type. To accompany the discussion it will be convenient to use the example hierarchy shown here, applying to a high-school ski team preparing for a trip to a minor-league championship. For brevity and simplicity we use the class names *GIRL* as an abbreviation for "member of the girls' ski team" and *BOY* as an abbreviation for "member of the boys' ski team". Some skiers in each team are ranked, that is to say have already recorded good results in earlier championships. This is an important notion: ranked skiers will start first in a slalom, giving them a considerable advantage over the others; after too many competitors have used it, the run is much harder to negotiate. (This rule that ranked skiers go first is a way to privilege the already privileged, and may explain why skiing holds such a fascination over the minds of many people: that it serves as an apt metaphor for life itself.) This yields two new classes, *RANKED_GIRL* and *RANKED_BOY*.



To assign the rooms we may use a parallel hierarchy; some rooms will be reserved for boys only, girls only, or ranked girls only.

Here is an outline of class *SKIER*:

```
class SKIER feature

    roommate: SKIER;
            -- This skier's roommate

    share (other: SKIER) is
            -- Choose other as roommate.
        require
            other /= Void
        do
            roommate := other
        end
    ...
end -- class SKIER
```

We have two features of interest: the attribute *roommate*, shown in blue; and the procedure *share*, which makes it possible to assign a certain skier as roommate to the current skier. Note the use of Eiffel's assertions: the ***require*** clause introduces a precondition stating that the argument must be attached to an object.

A typical call, as in

```
s1, s2: SKIER;
...
s1.share (s2)
```

will enable us to assign a certain roommate to a certain skier.

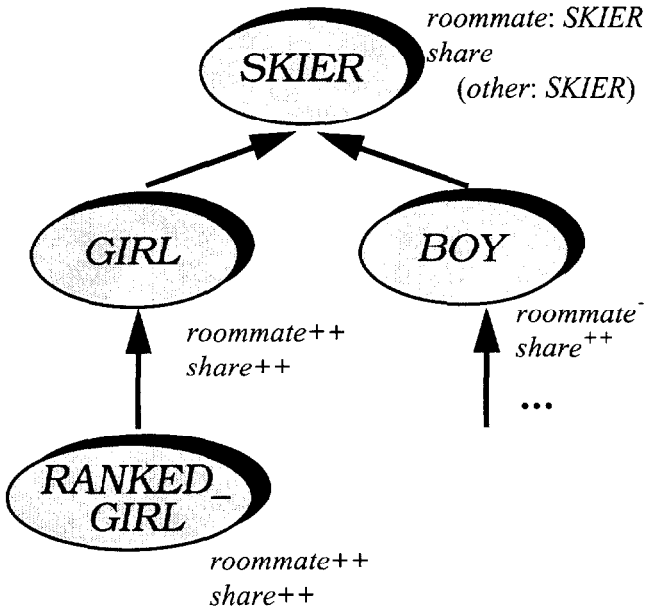
How does inheritance get into the picture? Assume we want girls to share rooms only with girls, and ranked girls only with other ranked girls. We will redefine the type of feature *roommate*, as shown below (in this class text and the next, the redefined elements appear underlined).

```
class GIRL inherit

    SKIER
        redefine roommate end

feature

    roommate: GIRL;
            -- This skier's roommate.

    ...

end -- class GIRL
```

We should correspondingly redefine the argument to procedure *share*, so that a more complete version of the class text is:

```
class GIRL inherit
    SKIER
        redefine roommate, share end

feature
    roommate: GIRL;
            -- This skier's roommate.

    share (other: GIRL) is
            -- Choose other as roommate.
        require
            other /= Void
        do
            roommate := other
        end

    ...

end -- class GIRL
```

The general picture is this:



++, a B.O.N. notation (see Kim Waldén and Jean-Marc Nerson, *Seamless Object-Oriented Software Architecture*, Prentice Hall, 1994), means "redefined". Since inheritance is specialization, the type rules naturally require that if we redefine the result of a feature, here *roommate*, the new type must always be a descendant of the original one. We should correspondingly redefine the type of the argument *other* of routine *share*. This is the policy known as covariance, where the "co" indicates that the argument and result vary together. The reverse policy is termed contravariance. I believe that this terminology was introduced by Luca Cardelli.

Strangely enough, some workers in the field have been advocating a contravariant policy. Here it would mean that if we go for example to class *RANKED_GIRL*, where the result of *roommate* is naturally redefined to be of type *RANKED_GIRL*, we may for the argument of routine *share* use type *GIRL*, or, rather scaringly, *SKIER* of the most general kind. One type that is **never** permitted in this case is *RANKED_GIRL*! Here is what, under various mathematical excuses, some professors have been promoting. No wonder teenage pregnancies are on the rise.

As far as I understand, by the way, the C++ policy is to bar any type redefinition — novariance as it is sometimes called. If this is indeed the rule I do not think that it is appropriate.

Covariance, of course, is not without its problems. Before looking at them we should examine a fundamental simplification. It is extremely tedious to have to redefine *share* the way we did in class *GIRL*. This redefinition only changes the type of the argument *other*; the rest of the routine, assertions and body, is just replicated. Anchored declarations, another Eiffel original, address this problem. In class *SKIER* we can prepare for such redefinitions by declaring *other* as being of type ***like*** *roommate*. This is the only difference with the previous version:

```
class SKIER feature

    roommate: SKIER;
            -- This skier's roommate.

    share (other: like roommate) is
            -- Choose other as roommate.
        require
            other /= Void
        do
            roommate := other
        end

    ...
end -- class SKIER
```

Such a ***like*** declaration, known as an anchored declaration, means that *other* is treated in the class itself as having the same type as the anchor, here *SKIER*; but in any descendant that redefines *roommate* then *other* will be considered to have been redefined too.

One can say without fear of exaggeration that without anchored redeclarations it would be impossible to write realistic typed object-oriented software.

But what about the problems of covariance? They are caused by the clash between this concept and polymorphism. Polymorphism is what makes it possible to attach to an entity an object of a different type. It is made possible by the attachment rule introduced earlier: in the assignment $x := y$, the type of y may be a descendant of that of x. But with covariance this may get us into trouble. Assume we have entities *s1* of type *SKIER*, *b1* of type *BOY* and *g1* of type *GIRL*; the names should be mnemonic enough:

```
s1: SKIER ; b1: BOY ; g1: GIRL
```

In creation instructions, marked with double exclamation marks !!, we create two objects of types *BOY* and *GIRL* and attach them to *b1* and *g1* respectively:

```
!! b1 ; !! g1;
```

Polymorphism allows us to let *s1* represent the same object as *b1*:

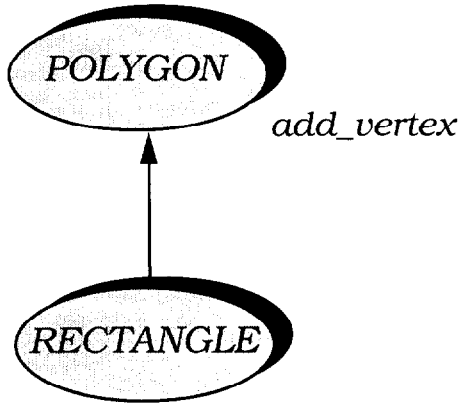
```
s1 := b1
```

Then the feature call

```
s1.share (g1)
```

achieves what all of us boys always dreamed of in high school, and what all parents fear.

A similar problem arises out of a very important inheritance mechanism: descendant hiding, the ability for a class not to export a feature that was exported by one of its parent.



A typical example is a feature *add_vertex*, which class *POLYGON* exports but its descendant *RECTANGLE* hides, because it would violate the invariant of the class:

```
class RECTANGLE inherit
    POLYGON
        export
            {NONE} add_vertex
        end

feature
    ...
invariant
    vertex_count = 4
end
```

The invariant is expressed here in Eiffel syntax as *vertex_count* = *4*. Another well-known example, more academic in nature, is a class *OSTRICH* that inherits from a class *BIRD* that was equipped with a routine *fly*. Clearly *OSTRICH* should not export that routine.

I should note in passing that some people criticize such practices as incompatible with a good use of inheritance. They are wrong. It is a sign of the limitations of the human ability to comprehend the world — similar perhaps to undecidability results in mathematics and uncertainty results in modern physics — that we cannot come up with operationally useful classifications without keeping room for some exceptions. Descendant hiding is the crucial tool providing such flexibility. Hiding *add_vertex* from *RECTANGLE* or *fly* from *OSTRICH* is not a sign of sloppy design; it is the recognition that other inheritance hierarchies that would not require descendant hiding would most likely be more complex and less useful.

Like covariance, then, descendant hiding is made necessary by the modeling requirements of the object-oriented method. But like with covariance this modeling power causes a conflict with the tricks made possible by polymorphism. An example is trivial to build:

p: *POLYGON*; *r*: *RECTANGLE* ; ...

!! *r* ; ...

p := *r* ; ...

p.*add_vertex* (...)

The simplicity of these examples makes up what we may call the static typing paradox. A student can make up a counter-example showing a problem with covariance or descendant hiding in a few minutes; yet Eiffel users universally report that they almost never run into such problems in real software development. This is certainly confirmed by ISE's own practice, even though the ISE Eiffel environment represents about half a million lines of Eiffel and about 4000 classes. But of course this does not relieve us from the need to find a theoretical and practical solution.

The problem was first spotted by several people in 1988 and has been discussed several times in the literature. William Cook described it in a paper at ECOOP 1989. At TOOLS EUROPE 1992 in Dortmund, Franz Weber proposed a solution based on adding a generic parameter for each problematic type.

In chapter 22 of the book *Eiffel: The Language* (Prentice Hall, 1992, the current Eiffel language reference), a solution was described which is based on determining the set of all possible dynamic types for an entity. So we would for example find out that *s1* can have *BOY* among its dynamic types and hence disallow the call *s1*.*share* (*g1*).

This approach is theoretically correct but has not been implemented since it requires access to the entire system; so rather than a mechanism to be added to an incremental compiler it is a kind of *lint* that should be applied to a finished system. Incremental algorithms seem possible (an ISE report described one a few years ago), but they have not been fully demonstrated.

The new approach that I think is the right one is paradoxical in that it is *more* pessimistic than the earlier one. In general, typing is pessimistic. To avoid some possibly failed computations, you disallow some possibly successful computations. In Pascal, for example, assigning 0.0 to an integer variable *n* would always work; assigning 1.0 would probably work; assigning 3.67 would probably not work; but assigning 3.67 – 3.67 would actually work. Pascal cuts to the essentials by disallowing, once and for all, *any* assignment of a floating-point value to an integer variable. This is a pessimistic but safe solution.

The question is how pessimistic we can afford to be. For example we can have a guaranteeably safe language by disallowing everything, but this is not very useful. What we need is a pragmatic assessment of whether the type rules disallow anything that is indispensable to real programs.

In other words a set of typing rules should be *sound* and *useful*. "Sound" means that every permitted text is safe. "Useful" means that every desirable computation can still be expressed (reasonably simply) without a violation of the type rules. I believe that the rules which follow satisfy these two properties of soundness and usefulness.

I would need a little more time to explain the details of the rules here but I will try to give you the gist. The full rules may be found on the Web at **http://www.eiffel.com**. Follow the link to "Full type rules for the OOPSLA talk". Workshops to discuss the underlying issues will be held at forthcoming TOOLS conferences (in particular at TOOLS EUROPE in Paris, 26-29 February 1996), and the complete discussion will be in the second edition of the book *Object-Oriented Software Construction.*

The two basic notions are "polymorphic entity" and "catcall".

Definition: Polymorphic entity

An entity *x* is polymorphic if it satisfies one of the following properties:

- It appears in an assignment *x* := *y* where *y* is of a different type or (recursively) polymorphic.
- It is a formal routine argument.
- It is an external function.

Informally, an entity is polymorphic if it can be attached to objects of more than one type. The basic case, the first in the definition, is when the entity appears as target of an assignment whose source is of a different type or, recursively, polymorphic. We also consider — this is the second case in the definition, and very important although very pessimistic — that any routine argument is polymorphic, because we have no control over the actual arguments in possible calls; this rule is closely tied to the reusability goal of object-oriented software construction, where an Eiffel class is intended, eventually, to be included in a library where any client software will be able to call it.

A call is polymorphic if its target is polymorphic.

Next, a routine is a CAT (Changing Availability or Type) if in a descendant class some redefinition of the routine makes a change of one of the two kinds we have seen as potentially troublesome: retyping an argument (covariantly), or hiding a previously exported feature. A call is a catcall if some redefinition of the called routine would make the invalid because of such a change.

Definition: Catcall

A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments.

A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.

If we look back at our examples we see that they involve catcalls on polymorphic entities, also known as polymorphic catcalls, marked ** below:

```
s1: SKIER ; b1: BOY ; g1: GIRL ; ...

!! b1 ; !! g1 ; ...

s1 := b1 ;

s1.share (g1) -- **
```

```
p: POLYGON ; r: RECTANGLE ; ...

!! r ; ...
p := r ; ...

p.add_vertex (...) -- **
```

Polymorphic calls are of course permissible; they represent some of the most powerful mechanisms of the object-oriented method. Catcalls are also desirable; they are, as we saw, necessary to obtain the flexibility and modeling power that can be expected from the approach.

But we cannot have both. If a call is polymorphic, it must not be a catcall; if it is a catcall, it must not be polymorphic. Polymorphic catcalls will be flagged as invalid.

The new type rule

Polymorphic catcalls are invalid

If you remember the America conjecture, we of course do not sacrifice static typing; we do not sacrifice covariance; and we do not sacrifice substitutivity, that is to say polymorphic assignments of a more specialized value to a more general entity, except in cases in which they would clash with the other rules. As evidenced by the practical Eiffel experience that was mentioned earlier, such clashes are very rare; they are signs of bad programming practices and should be banned.

So this is the type rule: a prohibition of polymorphic catcalls.

This rule is similar to the one in *Eiffel: The Language* but much, much simpler because it is more pessimistic. It is checkable incrementally: a violation will be detected either when an invalid call is added or when an invalid redefinition is made. It is also checkable in the presence of precompiled libraries whose source is not available to users.

As a complement it is useful to show the robustness of this solution by giving a technique which will answer a common problem. Assume that we have two lists of skiers, where the second list includes the roommate choice of each skier at the corresponding position in the first list. We want to perform the corresponding *share* operations, but only if they are permitted by the type rules, that is to say girls with girls, ranked girls with ranked girls and so on. This problem or similar ones will undoubtedly arise often.

An elegant solution, based on the preceding discussion and assignment attempt, is possible. This solution can be implemented in Eiffel right now; it does not require any language change. We will propose to the Nonprofit International Consortium for Eiffel (NICE), the body responsible for Eiffel standardization, to add to class *GENERAL* a new function *fitted*. *GENERAL* is a part of the Eiffel Library Kernel Standard, the officially approved interoperability basis; every Eiffel class is a descendant of *GENERAL*. Here is the function *fitted* (the name might change).

```
fitted (other: GENERAL): like other is
        -- Current if other is attached to
        -- an object of exactly the same
        -- type; void otherwise.
    do
        if other /= Void
            and then same_type (other) then
                Result ?= Current
        end
    end
```

Function *fitted* returns the current object, but known through an entity of a type anchored to the argument; if this is not possible it returns void. Note the role of assignment attempt.

A companion function which examines the types for conformance rather than identity, is easy to write.

Function *fitted* gives us a simple solution to our problem of matching skiers without violating type rules. Here is the necessary routine *match*:

```
match (s1, s2: SKIER) is
        -- Assign s1 to same room as s2
        --if permissible.
    local
        gender_ascertained_s2: like s1
    do
        gender_ascertained_s2 :=
                                s2.fitted (s1);

        if gender_ascertained_s2 /= Void then
            s1.share (gender_ascertained_s2)
        else
            "Report that matching is
                impossible for s1 and s2"
        end
    end
```

For a skier *s2* we define a version *gender_ascertained_s2* which has a type anchored to *s1*. I find this technique very elegant and I hope you will too. And of course parents concerned with what happens during the school trip should breathe a sigh of relief.