Type-safe covariance: Competent compilers can catch all catcalls

Mark Howard[1], Éric Bezault[1], Bertrand Meyer[2, 3],
Dominique Colnet[4], Emmanuel Stapf[3], Karine Arnout[2], Markus Keller[2]

[1]Axa Rosenberg, Orinda (California, USA)
[2] Chair of Software Engineering, ETH Zürich (Switzerland)
[3]Eiffel Software, Santa Barbara (California, USA)
[4]LORIA — INRIA Lorraine, Nancy (France)

Work done as part of Eiffel language standardization by the TC39-TG4 committee of ECMA
See www.ecma.ch

Draft — Version of 27 April 2003

Abstract

Goals of expressiveness and flexibility in typed object-oriented programming suggest a "covariant" type policy, where routine redefinitions can change the types of both arguments and results in the same direction as the inheritance hierarchy. Unfortunately, a careless covariant policy, when combined with polymorphism, genericity and dynamic binding — other O-O mechanisms that are just as fundamental — may lead to run-time type violations known as catcalls. We present a new solution to this problem, resulting from recent advances in the Eiffel language, which enables compilers to spot all potential catcalls and forces programmer to resolve them. The resulting language rules statically guarantee type safety; they only require local analysis and are easy to explain to programmers.

1 OVERVIEW

The well-known risk of runtime errors (catcalls) arising from covariant redefinition has led many language designers to prohibit covariance and limit generic mechanisms — avoiding that risk but limiting programmers' power of expression. We present a new set of language mechanisms and rules, introduced into Eiffel as part of standardization at ECMA, which we believe provide a satisfactory solution to the problem, ensuring full type safety without limiting expressiveness.

The solution uses two complementary techniques:

- It takes advantage of the notion of *expanded inheritance*, recently introduced into Eiffel, which permits inheritance with all the associated facilities — renaming, redefinition, undefinition, multiple and repeated inheritance — except polymorphism.

- For the remaining cases, it requires any covariant redefinition to specify a “recast” routine that will process any arguments that, at run time, happen not to match the expectation.

This solution also covers issues caused by generic conformance and by descendant export narrowing. It is described in detail from section 11 onward (to which the reader familiar with the problem may wish to turn first); intervening sections describe the issue, shows how programmers in less expressive languages such as Java address it, examine the pragmatic criteria for acceptability of a solution, and review previous approaches.

2 STATIC TYPING, INHERITANCE, GENERICITY

A language is *statically typed* if its definition includes a set of rules, enforceable through mere analysis of program texts, which guarantee that no program satisfying these rules will ever, during an execution, trigger a *type failure* resulting from an attempt to apply an operation to a value on which it is not defined.

Static typing is good for software quality, since it’s always advantageous to catch an error before rather than during execution, even if that execution is for debugging or testing.

To eliminate type failures, any static typing policy will add some constraints to the programming language, and so will restrict the set of programs that programmers are permitted to write. These restrictions could go too far, as illustrated *ad absurdum* by a trivial way to make any language statically typed: reject all programs. Less extreme policies — ensuring, for example, that even with the constraints the language remains Turing-complete — could still limit the language’s expressiveness in a way that is practically unacceptable, by forcing programmers into lengthy and contorted ways of expressing useful computations. This indicates that even though the notion of static typing has a rigorous definition (refined in section 7 from the first version just given) a discussion of the issue cannot use rigorous criteria alone: it must also rest on an analysis, more subjective by nature, of whether the typing rules are acceptable to programmers. Section 8, “Pragmatic considerations”, will provide this analysis, showing that a good static typing solution must reconcile *safety* with *flexibility*.

Since safety — here, the ability to guarantee the absence of run-time type failures — is the very definition of static typing, we may consider this criterion as an absolute requirement and examine type policies in light of the second criterion: how much flexibility they offer. In the first statically typed languages — such as Pascal — the typing rules were simple:

- Each variable must have a *declaration* that specifies its type.
- Each type permits a clearly defined set of operations on its instances.
- Any operation that the program applies to a variable must be one of those permitted by its type.
- In any assignment $x := e$ of a value to a variable (or argument passing $p\ (e)$ where the formal argument of p is x) the type of e must be exactly the same as the type of the variable x.

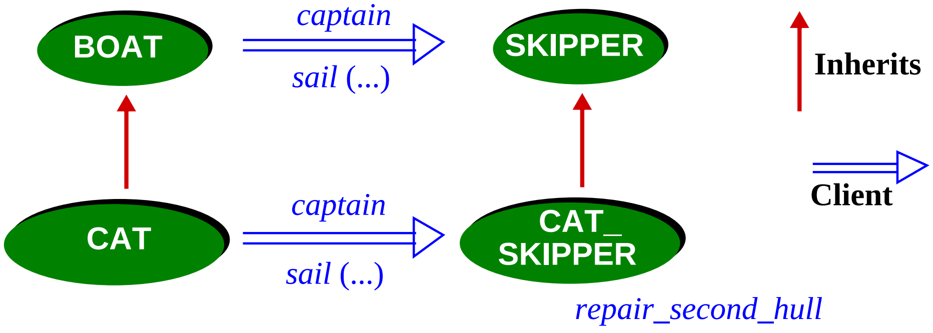
There is only minimal tolerance beyond these rules — to permit the traditional conversions between arithmetic types and, in the case of Pascal, to support record types with variants.

Statically typed object-oriented languages bring a major advance in flexibility by basing their type systems on inheritance. In an assignment (or argument passing, which should have the same semantics), the type of the source may be not just identical to the type of the target, but also a descendant type. If the types are different, the assignment is said to be *polymorphic*.

The type policy of such languages is based on inheritance. In a polymorphic assignment $x := e$ or argument passing $p\ (e)$, the type of the source e must *conform* to the type of the target x. In simple cases this notion of conformance is just inheritance: C conforms to A, both being classes, if C is a descendant (direct or indirect heir) of A. The presence of genericity and other advanced type mechanisms leads to a refinement of this notion; for example if *LINKED_LIST* inherits from *LIST* and both are generic classes, then *LINKED_LIST* [*BOAT*] conforms to *LIST* [*BOAT*] and — subject to the results of the present discussion — *LINKED_LIST* [*CAT*] will conform to *LIST* [*BOAT*] assuming that *CAT* (for “catamaran”, not the animal) conforms to *BOAT*.

3 COVARIANCE

The use of inheritance as the basis for the type system leads to greater flexibility. Of particular interest is the ability to redefine the type of a feature's results and arguments in descendant classes. Typical examples arise with parallel inheritance hierarchies:



A *CAT* is (left side of the picture) a particular kind of *BOAT*, whose *captain* is (top right) a *SKIPPER*; the procedure that assigns the captain to the boat is called *sail*. The class *BOAT* may be declared as

```
class BOAT feature
        captain: SKIPPER
                -- Skipper assigned to this boat

        sail (c: SKIPPER) is
                -- Appoint c as captain of this boat.
            require
                captain_exists: c /= Void
            do
                captain := c
            ensure
                set: captain = c
            end
end
```

Note the precondition (**require**...) included here to specify that the skipper reference passed as argument must not be void, but denote an actual *SKIPPER* object.

In class *CAT* we express that a catamaran must be sailed by someone with the proper qualifications. To this effect we may use the *redefinition* mechanism to give *captain* a new type in *CAT*:

```
class CAT inherit
        BOAT
                redefine captain, sail end
feature
        captain: CAT_SKIPPER
                        -- Catamaran skipper assigned to this catamaran
        ... See next ...
end
```

For clarity the language requires us to announce, in a **redefine** clause, which features we are going to redefine. This type redefinition of *captain* is called **covariant** because it goes in the same direction as inheritance: from general to specific.

If we only ever applied covariant redefinition to the *result type* of attributes and functions, as with *captain*, no particular safety issue would arise from the increased flexibility. But we can't stop here. If catamarans require a more specific kind of skipper, the argument to the procedure that *assigns* a skipper to a boat must change accordingly. So we should use covariant redefinition for routine arguments too:

```
-- (Missing part of CAT above.)

        sail (c: CAT_SKIPPER ) is
                        -- Appoint c as captain of this boat.
                ... do clause as in class BOAT ...
```

For such a "setter" procedure the body of the redefinition would, as shown, simply reproduce the original, unlike routine redefinitions that also change the implementation. Such signature-only redefinitions are common and would cause "redefinition avalanche" with large amounts of duplicated code. This is not acceptable in practice and is avoided by Eiffel's *anchored type* mechanism. It suffices to declare the original version of *sail*, in *BOAT*, as

```
-- (Rewrite of declaration of sail in class SKIPPER.)

        sail (c: like captain ) is
                        -- Appoint c as captain of this boat.
                ... Body same as in original ...
```

This "anchors" the type of the argument *c* to the type of *captain*, so that *c* has type *SKIPPER* in class *BOAT* and, in any descendant that covariantly redefines that type, the new type of *captain*, for example *CAT_SKIPPER* in class *CAT*.

Anchored types only make sense with covariance; conversely, it's hard — because of the redefinition avalanche problem — to imagine a covariant type policy that would not provide anchored types.

> Not all cases of argument covariance will, however, use anchored types, because anchoring requires that you can *predict* covariant redefinition at the level of the original declaration, here in *BOAT*. This issue of "ancestor foresight" is discussed further below (page 17).

Anchored types, so important in the practice of covariance, will require no special treatment in this discussion because the language definition treats them as a mere syntactical abbreviation for explicit covariant redefinition. All type rules apply to an "unfolded" form of classes where anchored types have been expanded to their full meaning; for example **like** *captain* will be replaced, in the application of type rules to class *CAT*, by the actual type it denotes in that class: *CAT_SKIPPER*.

4 COVARIANCE IN PRACTICE

The cats-and-boats example is not contrived. It is typical of a common scheme: covariant redefinition of the type of a query (usually an attribute, but possibly a function) such as *captain*. The very first case of inheritance that many people see in elementary presentations tends to be (see e.g. [16]) something like a class *MAMMAL* inheriting from *ANIMAL*; if there is a query *offspring*, then its type should be redefined covariantly throughout, reflecting that the offspring of a mammal are mammals, not just animals.

This is of course just a pedagogical example, but the scheme is just as frequent in practical software development. If we want a general class *NUMERIC* representing numbers with the usual arithmetic operations, then a query such as *minus*, giving *–a* for a number *a*, should yield a real number in the descendant *REAL*, a complex number in *COMPLEX* and so on. Or, if we have various implementations of trees, the query *parent* should be redefined, in each of the corresponding classes, to yield the same kind of tree node.

As soon as we permit such covariant redefinition of query *results*, the types of *arguments* must follow in the associated "setter" procedures, such as *sail* for boats, *engender* for animals, *set_parent* for tree nodes.

Argument covariance also arises in the absence of such queries. The most obvious example is the function to compare two objects, declared in Eiffel as

```
is_equal (other: like Current): BOOLEAN
```

and essential in any O-O model. (*Current* is the current object, also known as “this” or “self”; **like** *Current* is anchored to the type defined by the enclosing class.) Even in the absence of an anchored declaration mechanism, this would call for covariance: in a class *A*, we’ll want to compare instances of *A* with instances of the same type. These observations also apply to a procedure *copy* (*other*: **like** *Current*) which copies the contents of *other* onto the current object.

5 HOW JAVA PROGRAMMERS DO IT

Many typed object-oriented languages, notably Java and C#, avoid the issues discussed in this article by prohibiting signature redefinition, as well as descendant hiding and genericity. (Genericity mechanisms are planned but not released.) It is interesting to try to understand how programmers in these languages get around the issue. Typically, they use a large amount of casting (type conversions).

A good example is provided by the following extract from the Java AWT library (abbreviated from the source at java.sun.com/j2se/1.4.1/download.html). An “interface” (completely deferred class) defines a routine addLayoutComponent:

```
public interface LayoutManager2 extends LayoutManager {...
      void addLayoutComponent(Component comp, Object constraints);
      ...
}
```

Here now is a class that would want to make the routine secret, but cannot because officially there is no descendant hiding:

```
public class BoxLayout implements LayoutManager2 ... {...
      ...
       /** * Not used by this class.
      ...
      */
      public void addLayoutComponent
            (Component comp, Object constraints) { }
      ...
}
```

Note the techniques, highlighted above, to suggest descendant hiding: a comment saying the routine is inapplicable here; and a body that does nothing. Nothing, however, prevents a client from executing the routine, either directly or through a polymorphic call; the result — doing nothing — will hardly be satisfactory, or reveal the error.

The original types for the arguments are Component and Object. As in many practical cases, another descendant requires covariant redefinition. This is not permitted by the language so let us see how the programmer has addressed the issue:

```
public class GridBagLayout implements LayoutManager2 ... {...
        public void addLayoutComponent
                (Component comp, Object constraints) {
        if (constraints instanceof GridBagConstraints) {
                setConstraints(comp, (GridBagConstraints)constraints);
            } else if (constraints != null) {
                throw new IllegalArgumentException(
                        "cannot add to layout: constraints must be
                        a GridBagConstraint");
            }
    }
...
}
```

Since in this class the routine expects the constraints argument to be of type GridBagConstraints rather than the original Object, it has to perform a type test (instanceof), then a cast, to call the appropriate operation setConstraints with valid arguments; if the type doesn't match, all it can do is throw an exception. This kind of processing, made necessary by the inflexible type rules of the language, is not attractive, and may lead to undesirable exceptions at run time.

The same example incidentally shows the consequences of lacking a genericity mechanism. The routine setConstraints is itself written as

```
        public void setConstraints
                        (Component comp, GridBagConstraints constraints) {
                comptable.put(comp, constraints.clone());
         }
```

with

```
/**
   * This hashtable maintains the association between
   * a component and its gridbag constraints.
   * The Keys in <code>comptable</code> are the components and the
   * values are the instances of <code>GridBagConstraints</code>.
 ... */
 protected Hashtable comptable;
```

(A possible name for the programming language mechanism illustrated here is "Genericity Through Comments".) Defining a class Hashtable without genericity will, in the code using hash tables and other common container structure, lead again to numerous casts, and in the end to brittle programs where unplanned run-time cases may cause program failures.

These examples, from a fundamental Java library, seem to suggest that the mechanisms discussed in this article — covariance, descendant hiding, genericity — do address fundamental expressive needs of programming; excluding them leads to a contorted style involving casts and exceptions, and to risks of run-time failure.

The mechanisms have their own potential problems, to be discussed now. It seems appropriate to include the mechanisms and address their problems, rather than stick to the more restrictive style just illustrated.

6 CATCALLS

The flexibility of covariant redefinition leads to the risk of run-time type failures known as *catcalls*. A catcall is an attempt to apply an operation to an object for which the operation is not defined; the precise definition appears in section 7. Catcalls may happen out of three main causes: combination of argument covariance and polymorphism; genericity; descendant hiding.

The first is the conflict between covariant argument redefinition, polymorphism and reference aliasing. With these declarations (used throughout later examples):

```
boat1: BOAT; cat1: CAT
skipper1: SKIPPER; cat_skipper1: CAT_SKIPPER
```

the following *polymorphic* assignment is valid:

```
boat1 := cat1
```

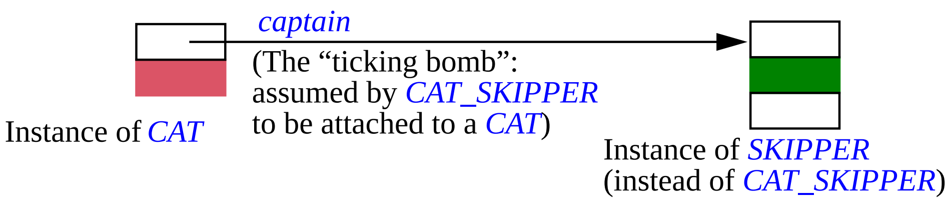
and the following call is always type-valid:

```
boat1.sail (skipper1)
```

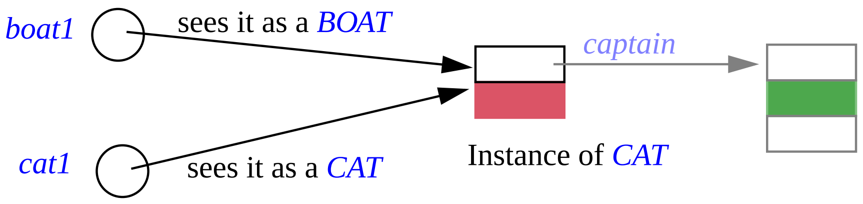
But if the two are executed in sequence — assuming that all the associated objects have been previously created — then the routine *sail* of class *CAT* puts at run time a non-specialist *SKIPPER*, rather than a *CAT_SKIPPER*, in charge of a *CAT*, precisely what the redefinition of the argument of *sail* was intended to prohibit.

This case prompts two comments:

- Nothing truly bad happens at the time of the call to *sail*; rather, as illustrated below, the run-time structure now contains a ticking bomb, a reference *captain* to a *SKIPPER* object in a *CAT* object, instead of the expected *CAT_SKIPPER* object. The problem will manifest itself when the execution attempts to access a feature that is only defined for *CAT_SKIPPER* object, for example *captain.repair_second_hull*, where the procedure *repair_second_hull* did not exist in *SKIPPER* (as shown in the figure on page 4). Applied to an ordinary non-catamaran *SKIPPER*, the attempt to call *captain.repair_second_hull* will fail at run time.



- The problem only arises because object models typically rely on references and on *aliasing*, the ability for different references to be attached to the same object. Here the polymorphic assignment causes the same object, of type *CAT*, to be available through two references, *boat1* and *cat1*. The problem is that the corresponding program variables are of different types and hence assume different operations.



The second source of catcalls is the addition of genericity to the preceding mechanisms. It seems desirable, for a generic class such as *LIST* [*G*] — as well as *ARRAY*, *SET*, *BINARY_TREE* and others representing container structures — that *LIST* [*C*] conform to *LIST* [*A*] if *C* conforms to *A*, since a list of catamaran skippers (for example) is a list of boat skippers. But then with

```
skipper_list: LIST [SKIPPER]; cat_skipper_list: LIST [CAT_SKIPPER]
```

and the assignment

```
skipper_list := cat_skipper_list
```

we may add an element to the skipper list:

```
skipper_list.extend (skipper1)
```

where *extend* appends an element at the end of the list; for that element the call uses a *SKIPPER* argument since we have a list of skippers. But that is still the same list as *cat_skipper_list*; so we may also use

```
cat_skipper1 := cat_skipper_list.last
```

where *last* returns the last element of a *LIST*. This puts us in the same ticking-bomb state as in the first case; with *cat_skipper1* we think we have a *CAT_SKIPPER* but it's actually attached to a plain *SKIPPER*, so a seemingly legitimate call *cat_skipper1*. *repair_second_hull* will fail at execution time.

The third cause of catcalls, "descendant hiding" is somewhat different. It arises if we allow a descendant class to restrict the export status of a feature that it inherits. The standard example is a procedure *add_vertex*, exported in class *POLYGON*, but made secret (private) in its descendant *RECTANGLE*. Then with dynamic binding a call of the form *polygon1*.*add_vertex* will fail if *polygon1* is polymorphically attached to an object of type *RECTANGLE*.

The most common response is that such descendant hiding is bad and should be prohibited by the language. But one may argue a more subtle case [10]. Without deciding on the methodological issue for the moment, we simply register the problem as another potential source of catcalls.

In the term "catcall" [9], denoting a run-time type failure arising from one of the causes listed here, "cat" is an abbreviation for a feature's *Changed Availability* (the last case) *or Type* (the other two). The rest of this article develops techniques to defeat catcalls.

7 DEFINITIONS

Defeating begins with defining. The following are more precise statements of the basic concepts and issues used informally so far.

Although the discussion applies to object-oriented programs, we may start in a more general context. We assume that the execution of a program consists of a sequence of operations, and that each operation expects operands of specific types. This is true even at the machine level: a floating-point addition requires its operands to be floating-point numbers, and otherwise will fail.

Here is what we are trying to avoid:

Run-time type failure

A *run-time type failure* is a program's attempt, during execution, to apply an operation to an operand whose type is not acceptable to the operation.

Run-time type failures are damaging not only to software reliability but also potentially to *security*, since a favorite intrusion technique is for an attacker to misuse an operation on operands for which it was not intended.

To fend off run-time type failures, a programming language definition may include special constraints, making up what is also called the "static semantics" of the language:

Validity constraint

A *validity constraint* for a programming language is a boolean condition applicable to any syntactically legal program text in the language.

A typical validity constraint is the usual rule on assignments *target* := *source*, stating that the type of the *source* must be the same as the type of the *target* (or, in an O-O language with inheritance, conform to it). In line with the definition of validity constraints:

- It only applies to *syntactically legal* texts; it is the job of the syntax rules (not the validity constraints) to specify that the assignment symbol must be properly written as :=, that the target must be a variable and the source an expression.
- It is a *boolean condition* on program texts: a syntactically legal assignment may or may not satisfy the constraint.

In this discussion we are interested in a particular kind of validity constraint, for languages where it is possible to associate a type with various program elements such as variables and expressions:

Type rule

A *type rule* is a validity constraint involving the types associated with program elements.

Programs satisfying these rules will be called “valid”:

Valid program

A syntactically legal program written in a programming language whose definition includes a set of type rules is *valid* if it satisfies all these type rules.

Note the gradation in soundness properties: a program may or not be syntactically *legal*; if it is, it may or not be *valid* (a concept that is not defined if the program is not syntactically legal). If we also have a way to define the intended behavior of a program, as with Eiffel’s contracts, a program may or not be *correct* for that specification (a concept that is not defined if the program is not valid, and is also not explored further in this discussion).

These definitions lead to the programming language property that we are trying to achieve:

Statically typed language

A programming language is *statically typed* if its definition includes a set of type rules, guaranteeing that no execution of a valid program will ever produce a run-time type failure.

The power of this notion comes from its *static* nature: the type rules are boolean-valued functions on program texts. If a language is statically typed we can guarantee the absence of run-time type failures by examining program texts prior to any execution. This is far superior to any technique that would involve running tests of the programs.

This examination of program texts will be the responsibility of a software tool:

Static checker

A static type checker (*static checker* for short) for a programming language is a program that can process any syntactically legal program text in that language and determine whether it satisfies the type rules.

Making a language statically typed means adding a set of type rules to its definition, and hence — for the sake of safety of program execution — restricting the set of valid programs. The issue discussed in this article is the conflict between this safety concern and the flexibility resulting from object-oriented mechanisms. We must devise a set of typing rules that make the language statically typed according to this definition, without depriving programmers of the power of expression afforded by the powerful mechanisms of object-oriented development including classes, single and multiple inheritance, polymorphism, dynamic binding and — if possible — covariance.

It's also useful to define "catcall". What's special about catcalls is that they are not straightforward type errors — such as an assignment *cat1 := boat1* where the type of the source would not conform to the type of the target — but plague programs that pass basic static type checking. Let's first define that basic level:

Class-level validity

A system (object-oriented program) is *class-level-valid* if it satisfies the following properties:

1 • In every assignment or argument passing, the type of the source conforms to the type of the target.

2 • In every feature call *f* (...) (unqualified) or *x*.*f* (...) (qualified), *f* is an exported feature of the class on which the type of *x* is based.

Here a type *conforms to* another if its base class is a descendant of the other's base class and, if there are generic parameters, they also conform (as with *LIST* [*CAT*] conforming to *LIST* [*BOAT*]). The name "class-level validity", indicates that this property can be checked class by class, without access to the rest of the system. The above definition sticks to the essentials; for details see chapter 22 of [7].

Catcalls arise when class-level validity does not suffice to ensure validity:

Catcall

A catcall is a feature call that, in a class-level-valid system, can cause a run-time type failure because of the presence in the system of any of:

1 • A covariant argument redefinition.

2 • A routine argument whose type is a generic parameter.

3 • Descendant hiding (export restriction for an inherited feature).

Calls may be *qualified*, as in *x*.*f* (...), or *unqualified*, as in *f* (...) where *f* is a feature of the enclosing class. In Eiffel the export rules only apply to qualified calls; hence:

Export-catcall rule (for an Eiffel-style export policy)

Only qualified calls may cause catcalls.

Thanks to this property we don't need any anti-catcall validity constraint for *secret* features. We'll take advantage of this property when defining our final type rules.

8 PRAGMATIC CONSIDERATIONS

The precise definitions of the previous section are not sufficient to decide whether a proposed set of type rules is satisfactory. We must also consider a number of practical criteria.

Efficiency

The first practical requirement could in fact be added to the basic definitions: since we expect the type checking to be done by a software tool — the *static checker* of the last definition — the type rules must at the very least be *decidable*. In practice that's not even sufficient: we will also want the checking process to be fast enough.

What concretely is a static checker? Since statically typed languages lend themselves to compilation, the static checker is usually integrated with a compiler, which just has type checking as one of its responsibilities along with syntax analysis and checking, code generation, optimization etc.

Alternatively, the static checker may be a separate tool, in the style of *lint* for C and other *static analyzers* that check various properties of programs. With this scheme the compiler performs basic type checking only, leaving aside the more delicate issues, so that catcalls may remain after compilation; the checker will be run at specific milestones. The only argument for doing things this way — other than making up for the type-checking deficiencies of a compiler over which you have no control — is that the static checker may take too long to be integrated in a standard compilation cycle, especially under incremental compilation. Although feasible in principle, this approach carries the obvious risk of forgetting to run the checker before releasing a system to production. In any case it doesn't relieve static checking from efficiency constraints; if checking time adds too much to compilation time, developers are unlikely to use it systematically.

Expressiveness

As has been noted, any static typing policy (as defined by a set of type rules) restricts the set of valid programs. Some computations that would be expressible without the type rules will no longer be permitted.

We may exclude over-restrictive policies by requiring for example (as suggested earlier) that the resulting language still be Turing-complete. This has not been included in the definitions of static typing because the real criterion is stronger: the typing rules must still leave to the programmer a sufficient power of expression. How much expressiveness is sufficient remains partly a subjective judgment; for example, Java programmers appear to survive without genericity, covariance or multiple inheritance, and some of them may not realize what they are missing if they have not had access to these mechanisms in other languages. On the other hand, examination of typical Java code shows (see Viega *et al*. [14]) that they resort to various unnatural tricks to emulate them.

In any case we must recognize that almost any static type policy will be *pessimistic*: to disallow *every* program that *might* cause run-time type failures, it may disallow *some* programs that *might not* create type failures. An elementary example is the prohibition, in statically typed languages, of the assignment of a floating-point expression to an integer variable, on the grounds that it would usually cause a loss of information, and that the programmer should specify an explicit behavior such as rounding up, rounding down or truncation. But occasionally this does not apply, for example in *my_integer* := *my_real* where *my_real* has value *0.0*, representable as an integer without any loss. The assumption behind static typing is that excluding such cases does not place an undue restriction on programmers.

Proponents of dynamic typing, as represented among O-O languages by Smalltalk, disagree: rejecting any kind of static limitation, they are willing to take the risk of run-time type failures rather than limit type combinations in any way. (More precisely, they generally assert that in real development programmers make few type errors, and that any that do occur are easily caught during testing.) At the other extreme we find what some people deride as "*bondage-and-discipline* languages", whose type rules are too strict for the needs of practical programming. (No examples will be cited here.)

These observations indicate that we cannot pretend to full objectivity while discussing the issue of a proper type system for an O-O language; rather, the result must be an engineering decision of how much expressiveness programmers are entitled to enjoy.

Not requiring ancestor foresight

Another criterion, often neglected in theoretical discussions of covariance, is critical in practice: how much foresight is required of the author of a class whose descendants may redefine a routine covariantly.

In the scheme that we have seen so far, an ancestor can explicitly specify an argument or result as covariant by declaring it anchored, as **like** *anchor* for some *anchor* that descendants may redefine, automatically causing the implicit redefinition of all elements anchored to it. But such foresight is not required: a descendant may want to redefine argument and result types even if the ancestor made no mention of the possibility.

Assuming foresight — for example by limiting covariant redefinition to the anchoring scheme — slightly simplifies the issue. The problem is that such an approach conflicts with the *open-closed principle* of object-oriented design, which suggests [10] producing classes that are both directly usable (*closed*) and amenable to variation (*open*) through inheritance; as a result, ancestor classes should not have to know about their descendants, and should not limit the descendants' power of extension other than by specifying contracts — the class invariant, the routines' preconditions and postconditions — that bind descendant redefinitions.

Covariance fits well with this principle, but not if it requires ancestor foresight. We should look for an approach that permits covariant redefinition even if the ancestor has not prepared it.

Several of the previous approaches described in section 9 work well theoretically but fail to meet this criterion, implying that any new case of covariant redefinition in a descendant requires going back to the original ancestor to update it. This is usually not acceptable in the practice of object-oriented development.

Scope of the solution

A typing policy is only targeted at avoiding a certain kind of failures as defined earlier: *type failures*. Other kinds of run-time failures will subsist; the most common is the application of a feature to a non-existing object by trying to follow a void (null) reference. In the Eiffel framework where *Void* is of a special type *NONE* that conforms to all other class types and exports no features, void feature calls could in theory be treated as catcalls, but this view does not appear very useful in practice and other techniques have to be devised, specific to void calls (see for example [11]).

Other kinds of run-time failures, not specific to the object-oriented framework, are still possible, for example floating-point exceptions.

Handling catcalls

A category of solutions exists that follows the letter of the preceding definitions but may not be satisfactory in practice. Since the official purpose of a typing policy is to guarantee that *no run-time type failures* will occur, we might take advantage of the last observation to catch any such failure at run time just before it occurs, trigger a failure of some non-type kind — for example a void call — and declare victory.

An example of such a policy, which meets the definition of static typing, is to define the language semantics as requiring that any call, at run time, must check every argument for conformance to the statically declared type and, if it doesn't conform, trigger a non-type failure. The compiler — which in this scheme is also the static checker — will for every call generate code to check argument types.

Although we may consider such a solution as cheating with the definitions, it does present a practical advantage: if the concern is *security* rather than reliability, turning type failures into other kinds of failures rules out illegal access to objects, achieving a significant security benefit. (The goals of reliability and security are not always concomitant: for the reliability engineer, a program that stops by crashing is a disaster; for the security administrator, it's a disaster avoided.)

Indeed we cannot dismiss the idea of *handling* catcalls dynamically rather than *preventing* them statically. But pursuing it requires two key improvements:

- First, there is an efficiency issue. It's absurd to start from a statically typed language and still force the compiler-generated code to check type consistency at run time for every call and every argument — all for a case (catcalls) that arises only exceptionally. (In the authors' collective Eiffel experience, ranging over a period up to eighteen years and over programs of up to two million lines of code, a catcall might occur, in a given project, once every few months — although this doesn't make it less painful.) If run-time type checking code is to be used, it should only be generated for the covariant redefinition cases that might cause catcalls.
- Second, if the generated code detects a catcall, it is not sufficient to terminate the program abruptly. In production programming, we can't deliver programs that may in some cases just mysteriously crash, even if it's not a security risk.

Rather than termination we could trigger an exception; but that's not enough either, because there is no simple and effective way to force the programmer to process the exception according to a predefined scheme.

What we will do — in the solution finally retained below, which is a variation on this policy — is to limit possible catcalls to the strict minimum through purely static means, then require that programmers make any remaining cases explicit and provide a systematic way of handling them through associated routines.

Compatibility

A practical criterion to be taken into account is the effect of any catcall-preventing solution on existing software compiled with a more tolerant policy. Clearly, any catcall risk unearthed by the new policy signals a dormant flaw in the existing software, and should be corrected. But it would be improper to make the code suddenly uncompilable. If the new typing rules are enforced by the compiler, a temporary compatibility option should be available to allow, with the appropriate warnings, continued use of software that was written prior to the new rules — and leave time for the correction of potential catcalls.

Clarity, fragility, incrementality

A final requirement, not necessarily prominent the first time one looks at a proposed theoretical solution, is essential to its practical acceptance: whether violations of the type rules can be reported clearly to the programmer, with a message identifying the error locally and incrementally. Given the possible complexity of catcall schemes this may be delicate. Consider the earlier call

```
boat1.sail (skipper1)
```

which is catcall-prone if *boat1* may polymorphically become attached to an object of type *CAT*. Earlier, the fateful assignment, *boat1 := cat1*, was in the same program unit. But an identical catcall situation will occur as a result of instructions

```
boat1 := boat2
boat1.sail (skipper1)
```

appearing in a routine

```
r ( boat2: BOAT )
```

where, locally, everything seems fine, since all variables are of type *BOAT*. Assume the above code is in a class *A*. A different class *C* may have a call to *r*, of the form

```
x.r (cat1 )
```

creating a catcall situation. In theory, this is the same scheme as before. But if we have a mechanism to detect catcalls, phrasing the error message is more delicate: how do we explain to the author of class *C* what's wrong? This is even harder if class *A* — seemingly innocuous by itself, but now causing a catcall — is a library class, not controlled by the author of the client *C*, and possibly with source code not even available. The level of indirection is potentially unbounded: instead of *cat1* the call could be using *boat3*, itself an argument of the enclosing routine in *C*, for which a caller may at any time use an actual argument of type *CAT*.

We may draw three lessons from this analysis:

- *Clarity*: since the cause of a catcall can be a combination of elements scattered throughout the text of a system and the libraries it uses, any strategy for detecting catcalls must be able to produce error messages that clearly identify this combination, enabling the client programmer to understand what's wrong.

- *Fragility*: adding a simple assignment or routine call to a system that has been proved catcall-free can introduce a catcall.

- *Incrementality*: a detection policy should not only produce locally relevant messages but also be applicable after a change to the system, without forcing a new analysis of the entire text.

9 PREVIOUS SOLUTIONS

The issue of covariance has been widely discussed, and language solutions to the catcall problems proposed by many authors. This section briefly examines some of the existing approaches.

Novariance

In many O-O languages, there is no covariant redefinition, no descendant hiding, and even no genericity. This is the case in Java and C#; C++ has templates for genericity, and permits covariant redefinition of results but not of arguments.

In these languages the problem obviously doesn't arise, but one may argue that the loss of flexibility penalizes programmers and forces them into programming idioms that may create the same kind of run-time crashes that might arise from catcalls in a covariant language. The Java library examples seen in section 5 support this view.

Contravariance

One possibility explored by a number of theoretical papers is to allow covariant redefinition for results (new type conforms to old type, as discussed in this presentation) and *contravariant* redefinition (old type conforms to new type) for arguments. To our knowledge the only O-O programming language that has supported contravariance is Sather [13].

In this approach no catcalls are possible, and the mathematical models are somewhat simpler. The problem is that contravariance doesn't seem useful in practice. The earlier discussions, and everyday examples, suggest that "the world is covariant". In particular allowing covariant results but disallowing covariant arguments mean that we can't associate setter procedures (such as *sail* or *engender*) with queries, or write object comparison functions such as *is_equal* with proper type signatures. Regrettably in light of its mathematical elegance, this scheme seems to have little practical applicability.

Runtime exceptions

Another approach, mentioned above as "handling catcalls", has been implemented by some compilers: in situations that can cause catcalls, such as a covariant argument redefinition, generate code that checks types and produces an exception if they don't match.

The disadvantage of this scheme is that it doesn't explicitly make client programmers aware of a risk of exception, so that they have no incentive to provide an exception handler.

Another drawback is the efficiency penalty, hard to justify in a language that is statically typed, or at least "almost" typed with the exception of covariance.

More fundamentally, it's not right to let the programmers who cause potential catcalls, for example through a covariant argument redefinition, to wash their hands off the problem by just triggering an exception in case of a type mismatch.

The approach described below does use a form of runtime detection, but only for catcall-prone cases, and with explicit programmer-specified handling.

Genericity-based solution

Franz Weber observed [15] that covariance is a form of type parameterization, which could be handled by the parameterization mechanism present in Eiffel and some other object-oriented languages: genericity.

Indeed we can make a class such as *BOAT* generic, *BOAT* [*S*] where the formal generic parameter *S* represents the kind of skipper sailing the boat. Then *captain* and the argument of *sail* are simply declared of type *S* within the class. Variables representing ordinary boats will be declared of type *BOAT* [*SKIPPER*]; *CAT* will inherit from *BOAT* [*CAT_SKIPPER*].

This approach works nicely for examples with only one covariant redefinition, but adds a generic parameter for every argument type that may undergo covariant redefinition; the extra parameters, which become part of the class specification, are hard to justify to a casual user of the class. For class designers, they assume perfect foresight of all covariant redefinitions and hence defeat one of our criteria. In addition the approach requires a set of new type rules that appear difficult to learn

F-bounded polymorphism

An abundant literature on type theory, starting from the work of Cardelli [1] [2], has proposed mathematical models for the type systems of object-oriented languages, building in particular on the notion of F-bounded polymorphism proposed in [4] and developed in many subsequent articles. A more immediately understandable name for F-bounded polymorphism is "types defined by recursive constraints". A concise type definition for *BOAT* is

$$\boxed{BOAT = \{captain: SKIPPER;\ set\_captain: SKIPPER \rightarrow BOAT\}}$$

where $A \rightarrow B$ is the type whose elements are functions from *A* to *B*; in line with the work on F-bounded polymorphism, which has mostly been using functional languages as examples, *set_captain* appears here as a function returning a new boat (rather than a procedure modifying the current boat). This style of definition characterizes a class type as a kind of record type $\{a_1: T_1;\ ...\ a_n: T_n\}$ with named components, some of which may represent attributes, like *captain*, and others functions, like *set_captain*. The definition of *BOAT* above is recursive and so meaningless without further conventions. In the F-bounded interpretation, we may consider that, taking *BOAT* to denote the type *BOAT* and all its descendants such as *CAT*, the definition means

$$\boxed{[D1]\quad BOAT \leq F\_BOAT\ [BOAT, SKIPPER]}$$

where $\leq$ denotes type conformance (based on the inheritance relation) and *F_BOAT*, a two-argument function from types to types, is defined as

$$\boxed{[D2]\quad F\_BOAT\ [B, S] = \{captain: S;\ set\_captain: S \rightarrow B\}}$$

With this typing, the definition of *CAT* (including its own descendants) is now

$$\boxed{[D3]\quad CAT \leq F\_BOAT\ [CAT, CAT\_SKIPPER]}$$

implying that *CAT* also satisfies [D1] and hence that we can consider *CAT* as a descendant of *BOAT*. From [D3] the signature of *set_captain* in *CAT* is $CAT\_SKIPPER \rightarrow CAT$, as desired.

For the practicing programmer, this approach means essentially the same as Weber’s genericity-based solution. We may implement it in Eiffel using the *constrained genericity* mechanism which combines genericity and inheritance; class *BOAT* will become

```
class BOAT [ S –> SKIPPER ] feature
        captain: S
        sail (c: S ) is .. As before ....
end
```

where the arrow in *S –> SKIPPER* indicates that the valid types to be used as actual generic parameters for *S* must be descendants of the “generic constraint” *SKIPPER*. So *CAT* may be defined as inheriting from *BOAT* [*CAT_SKIPPER*].

This solution removes the catcall issue for covariance, but suffers from the same practical problems as the previous one: need for total foresight, addition of spurious generic parameters to the class interface, and sophisticated type rules.

Covariant redefinition as overloading

An ingenious approach devised by Giuseppe Castagna [3] considers that a covariant redefinition — as for *sail* in *CAT* — does not override the original (here from *BOAT*) but coexists with it. This leads to a dynamic form of *routine overloading* where any particular call will at run time use one version or the other depending on the type of the object passed as actual argument.

In our example, the call *boat1*.*sail* (*skipper1*), with *boat1* dynamically attached to a *CAT* object, can no longer cause a run-time type failure:

- If *skipper1* is attached to a *CAT_SKIPPER* object, the call executes the version of *sail* redefined in *CAT* (as should all solutions since this is not a catcall).
- If *skipper1* is attached to a *BOAT* object, the call will simply use the *BOAT* version of the procedure.

This approach has the advantage of removing all risks of catcalls, and also of providing a form of “multi-methods” (routine selection based on arguments, not just the target of a call).

The overloading technique doesn’t go well, however, with the object-oriented style of programming. If we redefine a routine in a new class, it’s because we want to apply the new version to instances of that class. This is particularly important if the redefinition is not just for type adaptation (as in the examples so far) but also changes the implementation and possibly the contracts. Then the overloading convention can be very confusing: a client programmer who writes code using class

BOAT will want to know, from the description of the class interface, what its features are and what each expects and provides. For example the new version of a redefined routine may have a new contract (weakened precondition, strengthened postcondition). With redefinition treated as overloading, the client can no longer trust what a call with known target and argument types will do; the result may be to trigger any number of overloaded variant of a routine, each with its own contracts.

Castagna's view is that covariance and dynamic binding are exclusive techniques; indeed, as we have seen, combining them without precaution leads to catcalls. But we should still look for a solution that accepts covariance without renouncing the basic rules of object-oriented programming.

The solution described later in this article does retain an element of Castagna's approach, as it requires examining, for a CAT routine, the actual argument type at the time of any given call. But it stays away from overloading, always calling the redefined version; the idea will be to "recast" any argument of a wrong type, using a programmer-specified "recast function" that turns it into an object of the expected type, subject to the contracts of the redefined version. This approach accepts covariance while continuing to apply dynamic binding.

Match types

Colnet and Liquori proposed [5] a notion of "match type", syntactically similar to anchored (**like**) declarations, but with stricter conformance rules removing covariance-related catcalls. The rules seem, however, too constraining in comparison of the flexibility provided by anchored types and other current covariance mechanisms.

System validity

One solution, described in the earlier Eiffel reference [7], is to rely on global system analysis. For every variable *e* of an entire program (a system), the static checker will compute a set of types, the *dynamic type set* of *e*, that includes the types of all objects to which *e* may become attached during any execution. Then to typecheck every call *x*.*r* (*a*, ...) it will consider that *x* and the arguments may take on not only their declared types but also every type of their dynamic type sets.

The dynamic type set of *e* doesn't have to be the *exact* set of run-time object types for *e*, which would be impossible or very hard to compute; it has to *include* all these types. System validity, then, takes a possibly pessimistic approach, in line with the earlier observation that static typing is by nature pessimistic. To avoid performing complicated control flow analysis, it considers that any assignment *e* := *f* in the system text causes the dynamic type set of *e* to contain all the types in the dynamic type set of *f*.

For example the presence of an assignment *boat1* := *cat1* implies that the dynamic type set of *boat1* contains the type *CAT*.

This rule is pessimistic since a finer analysis might reveal that when *f* is of some specific type *T* the assignment *e* := *f* will never be executed. System validity, in its simplest form, does not consider such properties, assuming instead that every assignment or argument passing may be executed.

The basic scheme for computing dynamic type sets is simple:

- Starting from dynamic type sets that are all empty, consider that every creation instruction — **create** *e* or **create** {*EXPLICIT_TYPE*} *e* in Eiffel, *e* := **new** *EXPLICIT_TYPE* (...) in some other languages, adds the given type to the dynamic type set of *e*.
- Then, for every assignment *e* := *f*, or argument passing, add to the dynamic type set of *e* all the elements of the dynamic type set of *f*.
- Repeat the previous step until no dynamic type set changes any more (this is guaranteed to terminate after a finite number of iterations).

In spite of its pessimistic bias this approach has some advantages, in particular that as a side product it yields information that the compiler can use for optimization. It is also not too bad with respect to the *clarity* criterion, since the analysis can report the exact combination of constructs that produces a potential catcall situation.

The main drawback of the system validity solution is its need to analyze the entire source of a system. This obviously defeats the efficiency criterion, as well as incrementality. In addition the solution requires significant extensions to handle systems relying on libraries whose source code is not available. Experience with the SmartEiffel compiler, which offers an option to check system validity, suggests that the checks flag too many programs as invalid, and has trouble generating clear error messages.

Catcall detection

An incremental version of system validity is possible if we accept an even more pessimistic policy. In this version [9][10], the static typing policy disallows any call *x*•*r* (*a*, ...) if *x* is polymorphic and *r* is a “CAT” routine. with the following definitions:

- *x* is polymorphic if it is a formal argument of the enclosing routine, or appears as target of an assignment *x* := *y*, where *y* is of a different type from *x* or is itself (recursively) polymorphic according to these same criteria.
- A routine *r* of a class *C* is a CAT routine (Changing Availability or Type) if a descendant of *C* ever redefines it with a covariant argument, or restricts its export status.

An advantage of this policy is that the determination of polymorphism is local to a class; no global information is required. To spot CAT routines it suffices to make sure that the compilation of every routine in a class records whether the routine has been redefined or export-restricted; this is possible even without retaining the source code.

There are, however, two drawbacks. One is the added pessimism; in particular, treating any formal argument as polymorphic, although required to permit incrementality, may trigger many false alarms. But the more serious obstacle is fragility. Adding a single assignment to a system that previously passed the type checks may cause a catcall situation for reasons that are buried in the existing code.

10 SUPPORTING LANGUAGE ADVANCES

Language constructs recently introduced into Eiffel — generally for different purposes — considerably help the search for a solution. They include tuples and expanded inheritance.

Tuples

A tuple is a sequence of values of specified types. For example the Eiffel type *TUPLE* [*X*, *Y*, *Z*] describes sequences ("tuples") of three or more values, of which the first is of type *X* or conforming, the second of type *Y* or conforming, the third of type *Z* or conforming. An example of such a tuple value is [*x1*, *y1*, *z1*], with *x1* of type *X* and so on.

The number of parameters, three in this example, is arbitrary: tuple types include *TUPLE* (without parameters), *TUPLE* [*X*] for any *X*, *TUPLE* [*X*, *Y*] for any *X* and *Y* etc. Because the definition of *TUPLE* [*X1*, *X2*, ..., *Xn*] is that it covers sequences of *at least* *n* elements of which the first has a type conforming to *X1* etc., the conformance rules let *TUPLE* [*X*] conform to *TUPLE*, *TUPLE* [*X*, *Y*] conform to *TUPLE* [*X*] and so on.

Tuples will help our solution by letting us describe the type of a set of arguments to a routine, rather than of just one argument.

Expanded inheritance

Even more directly relevant is the notion of "expanded inheritance", also called non-conforming inheritance and implementation inheritance.

Expanded inheritance follows from the observation that inheritance combines two purposes: subtyping (which justifies polymorphism) and reuse, and that sometimes only the second is relevant. In this second role, inheritance simply acts as a generalized module inclusion facility, enabling a class to take advantage of features defined in others. Usually this mixes well with the subtyping role, but sometimes we want reuse and no subtyping.

The reader has probably been warned against "implementation inheritance", perhaps as part of reading indictments of multiple inheritance. Although such criticisms are valid when they target abuses of inheritance, they often assume incomplete or flawed inheritance mechanisms. "Implementation inheritance", better called reuse inheritance, has its place as a legitimate reuse mechanism. See [10] for a more extensive discussion.

Accepting the need for such a facility, one might think of achieving it through language constructs completely distinct from inheritance. But it requires the same mechanisms as subtyping inheritance: feature redefinition, renaming, undefinition and others. As a result, reuse inheritance has been introduced into Eiffel as a mere syntactical variant of usual inheritance, through the addition of the keyword **expanded**. (This keyword, which explains the name "expanded inheritance", also serves to specify that a type is used through its actual values, not through references.)

The syntax difference is very simple. Using standard subtyping inheritance (not expanded) we express that a class *B* inherits from a class *A* by declaring it as

```
class B inherit
        A
feature
        ...
end
```

or, to take advantage of mechanisms such as redefinition, undefinition, renaming:

```
class B inherit
    A
        rename
            f1 as g1,  f2 as g2
        redefine
            f1
        end
feature
    ...
end
```

This default form permits polymorphism, as in an assignment *a1 := b1* with *a1* of type *A* and *b1* of type *B*, and hence serves for subtyping.

If you do not need polymorphism and subtyping, you may use expanded inheritance instead, by just adding the keyword **expanded** to the class being inherited:

```
class B inherit
    expanded A                -- Everything else unchanged:
        rename
            f1 as g1,  f2 as g2
        redefine
            f1
        end
feature
    ...
end
```

The only difference with this form is that it doesn't permit polymorphic attachments (assignments or argument passing) from *B* to *A*.

The relevance of this new technique to the catcall issue is that — as shown by the discussion of the issues in section 6 — covariant argument redeclarations, as well as export restrictions, are harmless in the absence of polymorphism. So the first step in a cleanup of existing code is to perform a systematic examination of inheritance links, and add an "**expanded**" qualification whenever a link doesn't require subtyping. This will considerably decrease the number of catcall risks.

Secret features

Along with these new language developments, our anti-catcall policy will use a long-present property that most earlier approaches didn't mention explicitly: the Export-catcall rule (page 14), enabling us to limit our attention to non-secret features.

11 THE NEW COVARIANCE SOLUTION

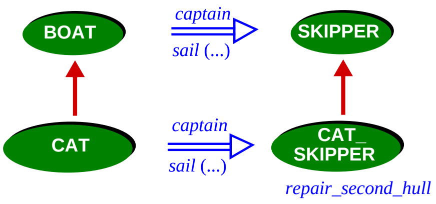
We now examine the rules that will remove catcalls in remaining cases, starting with the covariant redefinition case (the following two sections will examine genericity and export restriction).

Handling a covariant argument

Since the use of expanded inheritance avoids catcalls, we could altogether prohibit covariant argument redefinition in non-expanded (polymorphic) inheritance. We have rejected this solution as too restrictive; Eiffel programmers are attached to the flexibility of covariant redefinition, even with polymorphism.

We must, however, force the programmer who uses a covariant redefinition to realize that for a non-secret routine this causes a risk of catcall, and to handle the risk situation. The requirement will be, in such a case, to provide a **recast function**. It can be illustrated through the earlier covariant redefinition of *sail*. We had

```
class BOAT feature
        captain: SKIPPER
        sail (s: SKIPPER) is ... do ... end
        ...
end
```



and

```
class CAT inherit
        BOAT redefine captain, sail end
feature
        captain: CAT_SKIPPER
        sail (s: CAT_SKIPPER ) is    -- Not valid as given, see next
                ... do ... end
end
```

Because this is not expanded inheritance, the new solution makes the redefinition of *sail* invalid as given. Making it valid requires including a **recast** clause as follows:

```
sail (c: CAT_SKIPPER) is
        -- Appoint c as captain of this boat.
    recast
        trained_as_cat_skipper
    do
        captain := c
    end
```

where **recast** is a new keyword of the language. The **recast** clause cites a feature, here *trained_as_cat_skipper*, which must be a function of the enclosing class, taking an argument of the previous argument type, *SKIPPER*, and returning a result of the new argument type, *CAT_SKIPPER*:

```
trained_as_cat_skipper (s: SKIPPER): CAT_SKIPPER is
        ... See an example implementation below ...
```

The semantic rules specify that the execution of a call to such a covariantly redefined routine starts by checking the argument type and, if it is not of the newly expected type *CAT_SKIPPER* — in other words, there’s a potential catcall — applying the recast function, here *trained_as_cat_skipper*, to produce a value of that type. Then it executes the routine as usual, but with that “recast” value instead of the value as passed by the caller.

This approach forces the author of any covariant redefinition, which with polymorphic inheritance opens the risk of catcalls, to provide a guard against catcalls through a conversion function declared in the **recast** clause. In the absence of a **recast** clause, the redefinition will be rejected as invalid (unless of course it appears in an expanded inheritance branch).

A recast function

What might such a recast function look like? You might let it just trigger an exception, but that’s not very helpful. A better approach is to try to produce an object of the new type that approximates the old object. Taking our example seriously we might declare

```
trained_as_cat_skipper (s: SKIPPER): CAT_SKIPPER is
            -- A version of s born again as a catamaran skipper
      require
            exists: s /= Void
      do
            s.train_for_cats
            Result := s.cat_skipper_equivalent
      end
```

where we assume that it's possible to train an ordinary *SKIPPER* into catamaran sailing through the procedure *train_for_cats*. This procedure doesn't give us a *CAT_SKIPPER* — the skills may be right but the type is wrong — so we also need a conversion function called *cat_skipper_equivalent* above, which will look like

```
cat_skipper_equivalent: CAT_SKIPPER is
            -- Catamaran skipper version of this skipper, if properly trained
      require
            trained: trained_for_cats
                        -- Procedure train_for_cats ensures the postcondition
                        -- trained_for_cats.
      do
            create Result
            ... Initialize Result from fields of current object ...
      ensure
            result_exists: Result /= Void
      end
```

This example illustrates the general scheme: define a recast function that will return, in potential catcall cases, a result of the newly expected type. This of course implies extra work for the programmer, but is justified by the risk of catcalls. For "don't care" cases, it's always possible to define a recast function that returns a *Void* result — so that it will cause a later void call or other fatal event — or directly raises an exception.

Multiple-argument routines

The case described so far involved a routine with just one argument. We may have covariant redefinition for a routine with more than one argument

```
r (t1: T; u1: U; v1: V) ... is ... do ... end
```

redefined covariantly in a descendant as

```
r (t1: T’; u1: U’; v1: V’) ... is ...
        recast
                r_transform
        ...
        do
                ...
        end
```

where the types *T’*, *U’*, *V’* respectively conform to *T*, *U* and *V*. It’s rare in practice to redefine more than one of the arguments, but the possibility is there.

In this case too you will have to write a single **recast** clause specifying just one routine, called *r_transform* in this example; the routine must take arguments corresponding to the original signature — here three arguments of respective types *T*, *U*, *V* — and return a tuple, here of type *TUPLE* [*T’*, *U’*, *V’*], corresponding to the new signature. It will be declared:

```
r_transform (t1: T; u1: U; v1: V): TUPLE [T’, U’, V’] ... is ... do ... end
```

Note the role of tuple types in enabling the declaration of the function’s result. As in the one-argument case, any call will start by checking the argument types for a possible mismatch and, if it detects one, calling the recast function *r_transform* to obtain a tuple of argument values recast to the correct type. Then it will call the redefined version of *r* with these values as replacement arguments.

Handling all arguments through a single recast function has two advantages. First, it obviously limits the amount of “recasting” code to write. But it also makes it possible, in advanced cases, to use information from all the arguments to produce the new values needed for any one of them that is being redefined covariantly.

The rule

The new validity rule for covariant argument redefinition may be stated as follows:

Covariance rule

Any redeclaration of a non-secret routine *r* that changes the argument signature to *S’*, from a previous one *S*, must include a **recast** clause listing a function *f* of the enclosing class with the following signature:

- If *S* is a single argument type, *f* has a single argument of type *S* and a result of type *S’*.
- If *S* includes multiple argument types, *f* has the same arguments as the previous *r* and a result of the tuple type corresponding to *S’*.

The rule applies only to *non-secret* routines since secret ones cannot cause catcall trouble (Export-catcall rule, page 14). It applies not only to redefinition but to *redeclaration*, which covers both redefinition (overriding a previously implemented feature) and effecting (implementing a routine that was previously deferred).

Since anchored types (**like** *anchor*) as treated as an abbreviation for explicit covariant redefinition, there is a companion rule for this case:

Anchoring rule

Any non-secret routine *r* that has at least one argument of an anchored type must include a **recast** clause listing a function *f* of the enclosing class, with the same properties as in the Covariance rule, taking *S'* to be the argument signature of *r*, and *S* the signature obtained from *S'* by replacing any **like** *anchor* argument type by the base type of *anchor*.

(The "base type" of *anchor* is normally just the type of *anchor* but, if *anchor* is itself of type **like** *other_anchor*, it is recursively the base type of *other_anchor*. The language rules permit this but ensure that there are no anchor cycles, so the base type is always defined.)

12 THE NEW GENERICITY SOLUTION

If we want to permit generic conformance — *C* [*U*] conforms to *C* [*T*] whenever *U* conforms to *T* — then a generic class *C* [*G*] raises a risk of catcalls as soon as a routine has an argument of type *G*. This was illustrated earlier by an example:

```
skipper_list: LIST [SKIPPER]; cat_skipper_list: LIST [CAT_SKIPPER]
...
skipper_list := cat_skipper_list
skipper_list.extend (skipper1)
...
cat_skipper1 := cat_skipper_list.last
```

The procedure *extend*, in class *LIST* [*G*] and its descendants, adds an element at the end of a list, and hence takes an argument of type *G*, here representing *SKIPPER*. It causes exactly the same problems as a covariant redefinition. As a result, the rule is the same, even in the absence of any redefinition:

Genericity rule

In a generic class, any non-secret routine that takes at least one argument whose type is one of the generic parameters of the class must include a **recast** clause satisfying the same properties as in the Covariance rule.

This requirement appears rather drastic at first, but is needed to make genericity safe and preserve the possibility of generic conformance. Studies are in progress at ETH and LORIA to determine what the impact will be on existing libraries such as EiffelBase, which uses genericity extensively.

13 THE NEW DESCENDANT HIDING SOLUTION

The final cause of catcalls was export restrictions in descendants, as in

```
class RECTANGLE inherit
    expanded POLYGON            -- See below why now "expanded"
        export {NONE} add_vertex end
feature
    ...
end
```

which makes *add_vertex* secret (exported to *NONE*) in the new class.

Using expanded inheritance

Without any new language rule, class *RECTANGLE* is valid in the form given: the **expanded** keyword specifies expanded (non-conforming) inheritance, so no polymorphism is possible and hence no catcalls.

This scheme actually takes care of most common uses of descendant hiding. The "marriage of convenience" form of inheritance [10], used widely in the EiffelBase libraries [8], lets a class inherit its interface from one parent and its implementation from another, as with *ARRAYED_STACK* inheriting from *STACK* and *ARRAY*.

We must mention here that many people view the second inheritance link as improper; but closer analysis suggests it is better than the replacements they typically suggest, such as making *ARRAYED_STACK* a client rather than heir or *STACK*. See the references just cited.

In such an example many of the features of the implementation parent, *ARRAY*, should be secret in the descendant, since clients of *ARRAYED_STACK* should not be permitted directly to access and update arbitrary array elements. Expanded inheritance is directly applicable:

```
class ARRAYED_STACK [G] inherit
      STACK [G]
      expanded ARRAY [G]
            export {NONE} all end
feature
      ...
end
```

The **all** keyword applies the **export** {*NONE*} status to all features inherited from *ARRAY*. This may be overridden for specific features.

This form disallows polymorphic assignments from arrayed stacks to stacks, which seems appropriate since there is no evidence that such assignments would be useful, and clear evidence that they are catcall-prone. Based on our experience with Eiffel libraries it appears that most existing cases of descendant hiding can and should be rewritten to use expanded inheritance, removing the catcall risk.

Playing with the precondition

The question remains of whether we can have a catcall-free form of descendant hiding with polymorphism, as would be the case with *RECTANGLE* in the absence of the **expanded** keyword. Is there a way to specify both that one may assign a *RECTANGLE* value to a *POLYGON* variable and that class *RECTANGLE* doesn't export *add_vertex*?

We must note here that many methodology discussions would dismiss this scheme as conceptually wrong. (As noted, even expanded inheritance, which precludes polymorphism, doesn't always pass muster.) A common reaction is that the inheritance hierarchy should be changed to distinguish between "extendible" polygons, which have *add_vertex*, and those which don't. But this may conflict with other classification criteria and lead to unnecessarily complex inheritance structures. There are, in addition, arguments for permitting the extra flexibility of export restrictions in polymorphic inheritance, covering cases of "taxonomy exceptions" such as the example — famous in artificial intelligence — of treating ostriches as birds even if birds have a "fly" operation. See [10] for a more extensive discussion.

Still without any language changes, there exists a conceptually correct technique for achieving polymorphic descendant hiding of a routine, at least if we disregard the "No ancestor foresight" criterion. The idea is to give the routine a precondition which evaluates to true (or a specific condition) and is redefined in the descendant to yield always *False*, making the feature inapplicable.

This scheme requires strengthening the precondition in the descendant. The rules of Design by Contract do not permit this (they allow precondition weakening and postcondition strengthening [10]), but what matters is the contract as seen by clients, so we can achieve the intended result through the *abstract precondition* technique, again developed in [10]: by making the precondition rely on a function, we leave the precondition as such unchanged, but redefine the function. In the example the original version of *add_vertex* will now have a precondition:

```
class POLYGON feature
        add_vertex (...) is
                require
                        extendible
                do ... end

        extendible: BOOLEAN is
                        -- Is it permitted to add vertices? (By default: yes)
                do
                        Result := True
                end
end
```

In *RECTANGLE*, we leave *add_vertex* untouched and redefine its precondition to yield *False* in all cases:

```
class RECTANGLE inherit
        POLYGON
                redefine extendible end
feature
        extendible: BOOLEAN is
                        -- Is it permitted to add vertices? (Here: no!)
                do
                        Result := False
                end
        ...
end
```

From a theoretical perspective this approach achieves the desired goal. Note in particular that, whatever the inherited *add_vertex* procedure does, it is correct in class *RECTANGLE*, since the correctness of a routine *r* means

$$\{INV \textbf{ and } PRE_r\}\ BODY_r\ \{INV \textbf{ and } POST_r\}$$

where *INV* is the class invariant; since PRE_{add_vertex} now evaluates to *False*, this property is always trivially true.

This scheme suffers, however, from three limitations:

- It requires ancestor foresight. If *add_vertex* comes from an existing class *POLYGON* where it didn't have a precondition, we have to go back to the class and update it.
- In the absence of correctness proofs, we can't trust clients to ensure the postcondition in all cases, even though it is part of the specification so that any call that doesn't test for *extendible* is incorrect.
- If we disallow descendant hiding, the interface of the descendant class ("contract form" in Eiffel) will show the undesirable feature, here *add_vertex*. That is unpleasant since it will advertise the feature to authors of clients of *RECTANGLE*, who cannot and should not do anything with it.

Permitting explicit descendant hiding?

In the current state of discussions, the Eiffel standardization group is leaving the descendant hiding matter exactly as discussed so far: descendant hiding is permitted for expanded inheritance and in that case only.

An argument can be made for a more flexible policy working as follows: a class would be permitted to restrict the export status of an inherited feature, even with polymorphic inheritance, as long as it *redeclares* the feature (overrides the previous implementation if there was one, or introduces a first implementation if the feature was deferred). This is more in line with the "**recast**" solution adopted in other cases, which confronts the possibility of catcalls head-on by providing an explicit processing mechanism in catcall cases. Here too the class author would be required to address the problem, by providing an explicit redeclaration.

In most practical cases such a redeclared version could only trigger an exception or, if it is OK to ignore the call, do nothing (but adapt the precondition or postcondition, in line with the earlier discussion, to preserve correctness).

Although this approach has the support of a few of the authors, a majority considers it too risky and not needed in common cases, preferring to prohibit polymorphic descendant hiding, while of course permitting descendant hiding for expanded inheritance. This is the policy currently retained; there is no new language mechanism, only a new validity rule:

Descendant hiding rule

In an inheritance link, the export status of an inherited feature may only be more restrictive in the heir than in the parent if the link specifies expanded inheritance.

14 CONCLUSION

In the absence of a full mathematical model for the language, there is no proof that the policy just described will guarantee type safety. Examination of known catcall schemes seems to suggest, however, that it rules them all out.

A key property of this policy is that it puts the burden of a solution on the class author who, by taking advantage of the flexibility of covariance and genericity, may introduce a risk of catcall. The rules force the programmer to recognize this risk and handle it at the very point of its introducion. Previous solutions tended to give him a free hand, then to penalize future users of the class. This fairer assignment of responsibilities — "*If you can break it, you must fix it*" — is one of the arguments suggesting that the solution is in the right direction.

Work is in progress to apply this solution to existing libraries and application systems, including very large ones in production. No final standardization decision will be made until the results of these studies are in. On the basis of the concepts and arguments examined so far, we feel that the techniques we have described will provide the basis for a satisfactory and durable solution to the fundamental typing issue: how to provide an ironclad guarantee of run-time safety while letting programmers express their unbounded creative power.

References

[1] Luca Cardelli: *Basic Polymorphic Typechecking*, AT&T Bell Laboratories Computing Science Technical Report, 1984. Revised version in *Science of Computer Programming*, vol. 8, no. 2, 1987.

[2] Luca Cardelli: *A Semantics of Multiple Inheritance*, in *Semantics of Data Types Symposium*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984, pages 51-66; revised version in *Information and Computation*, vol. 76, no. 2/3, February-March 1988, pages 138-164.

[3] Giuseppe Castagna: *Covariance and Contravariance*: *Conflict without a Cause*, in *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 3, 1995, pages 431-447.

[4] Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C.Mitchell: *F-Bounded Polymorphism for Object-Oriented Programming*, in FPCA 89 (Proceedings of Functional Programming Languages and Computer Architecture), ACM, 1989, pages 273-280.

[5] Dominique Colnet and Luigi Liquori: *Match-O, a Dialect of Eiffel with Match-Types*, in *TOOLS 35* (35th conference on Technology of Object-Oriented Languages and Systems), Sydney, Australia, November 2000, IEEE Computer Society Press, 2000, pages 190-201.

[6] William R. Cook: *A Proposal for Making Eiffel Type-Safe*, in *ECOOP 89* (Proceedings of 1989 European Conference on Object-Oriented Programming, Nottingham, U.K., 10-14 July 1989), ed. Stephen Cook, Cambridge University Press, 1989, pages 57-70.

[7] Bertrand Meyer: *Eiffel*: *The Language*, Prentice Hall Object-Oriented Series, 1991; second revised printing, 1992.

[8] Bertrand Meyer: *Reusable Software*: *The Base Object-Oriented Libraries*, Prentice Hall, 1994.

[9] Bertrand Meyer: *Static Typing*, in *Object Technologies for Advanced Software*, eds. Kokichi Futatsugi and Satoshi Matsuoka, Springer Lecture Notes in Computer Science 1049, Springer Verlag, Berlin, 1996, pages 57-75.

[10] Bertrand Meyer: *Object-Oriented Software Construction, second edition*, Prentice Hall, 1997.

[11] Bertrand Meyer: *Prelude to a Theory of Void*, in *JOOP* (*Journal of Object-Oriented Programming*), vol. 11, no. 7, 1998, pages 36-48. Also at www.inf.ethz.ch/~meyer/publications/joop/void.pdf.

[12] Bertrand Meyer: *Eiffel*: *The Language*, *third edition*, work in progress at www.inf.ethz.ch/~meyer/ongoing/etl/, user name *Talkitover*, password *etl3*, consulted March 2003.

[13] Sather documentation at www.icsi.berkeley.edu/~sather.

[14] John Viega, Paul Reynolds, Reimer Behrends: *Automating Delegation in Class-Based Language*, in Proceedings of TOOLS USA 2000, August 2000.

[15] Franz Weber: *Getting Class Correctness and System Correctness Equivalent — How to Get Covariance Right*, in *TOOLS 8* (8th conference on Technology of Object-Oriented Languages and Systems, Dortmund, 1992), eds. Raimund Ege, Madhu Singh, and Bertrand Meyer, Prentice Hall, Englewood Cliffs (N.J.) 1992, pages 199-213.

[16] Wikipedia (online encyclopedia of computer science): *Inheritance* entry at www.wikipedia.org/wiki/Inheritance_(computer_science), consulted March 2003.

Acknowledgments

We thank the other members of the ECMA-TC39-TG4 committee for their input and support. The introduction of non-conforming inheritance into Eiffel was largely the result to the persistent lobbying of Richard Mitchell.