# Type Problems in Eiffel

Allowable changes which can be made in subclasses:

1. Add new features (instance vbles or routines).

2. Instance variables may be given a new type which is a subclass of the original.

3. In redefining routines, may replace parameter and result types by types which are subclasses of originals.

This may be done automatically if type defined in terms of "like Current" or similar.

Notice that "like Current" solves problems with clone and equals from Java's Object class.
In fact return type of clone and parameter type of equals will change automatically!

More flexible than Object Pascal, Java or C++
(C++ now allows changing result types)
but leads to problems:

Biggest problem - identification of class with type.

C' is a subclass (or heir) of C if C' inherits from C.

When redefine methods in subclass may replace class of arguments and answer by subclasses.

E.g.

If m(a:A):B in C then can redefine m(a:A'):B' in subclass C' if A' inherits from A and B' inherits from B.

Unfortunately, leads to holes in typing system!

Recall A' is subtype of A if element of type A' can be used in context expecting element of type A.

Eiffel allows programmer to use elt of subclass anywhere it expects elt of its superclass.

Therefore distinction between static and dynamic class!

Unfortunately subtype ≠ subclass when make changes to types of parameters and instance vbles.

Recall:

```
class LINKABLE [G]

feature

   item: G;

   right: like Current;   -- Right neighbor

   put_right (other: like Current) is
         -- Put `other' to right of current cell.
      do
         right := other
   ensure
      chained: right = other
   end;

end -- class LINKABLE
```

and

```
class BI_LINKABLE [G] inherit

   LINKABLE [G]
   redefine
         put_right
      end

feature -- Access

   left: like Current;    -- Left neighbor
```

```
   put_right (other: like Current) is
        -- Put `other' to the right of current cell.
      do
         right := other;
         if (other /= Void) then
         other.simple_put_left (Current)
        end
      end;
   …
end -- class BI_LINKABLE
```

So far so good.

But now suppose have following routine

```
trouble(p, q : LINKABLE [RATIONAL] ) is
   do
      p.put_right(q);
      ....
   end
```

and suppose have `s_node : LINKABLE [RATIONAL]` and
`bi_node: BI_LINKABLE [RATIONAL]`.
What happens if write:
`trouble(bi_node,s_node)`

If `BI_LINKABLE [RATIONAL]` is subtype of
`LINKABLE [RATIONAL]`, then this should work, i

Instead, BANG!!!!! - system crash.

Problem is `s_node.put_right` takes parameter of type LINKABLE [RATIONAL]
while `bi_node.put_right` takes parameter of type BI_LINKABLE [RATIONAL]
and these are not subtypes:

A' -> B' subtype of A -> B iff
B' subtype of B and A subtype of A'
note reversal!

Subclass in Eiffel not always give legal subtype!

Hence get holes in type system.

Allowable to export method from superclass, but not from subclass.

Breaks system if send message to object of subclass which is not visible.

E.g., define

```
hide_n_break(a:A) is
   do
      a.meth ....
   end
```

and then write `hide_n_break(a')` where `a' : A'`, and `A'` is subclass of `A` which does not export `meth`.

Earlier versions of Eiffel allowed user to break the type system in these ways.

Eiffel 3.0 attempts to compensate by running a global check of all classes used in a system to make sure that above situation could not occur (class-level check vs system-level check).
Involves dataflow analysis of program to determine if certain calls could occur. [Conservative check]

Consequence is that a system could work fine,but addition of new (separately compiled) class could break a previously defined class.

No Eiffel compilers implement system validity check.

In Fall, '95, Bertrand Meyer announced solution to "covariant typing problem" at OOPSLA '95.

No "polymorphic cat-calls":
Essentially can't send a method with a covariant change to any object where don't know exact type.

Originally had errors (omitted check to see if instance variable types changes).
Not know if guarantees type safety.
Like system validity check, never implemented.

I suspect both rule out too many useful programs.

Most statically typed object-oriented languages either
1. Type unsafe like Eiffel
2. Too rigid with types (like C++ or Object Pascal) or
3. Insert dynamic checks where unsafe (Java, Beta).

Trellis/Owl (by DEC) avoids the problem by only allowing subclasses which are also subtypes, but still pretty restrictive - rules out above LINKABLE class.

Proper solution is to separate subtype and inheritance hierarchies (originally proposed by researchers in ABEL group at HP Labs)

Inheritance hierarchy has only to do with implementation.

Subtype hierarchy has only to do with interface.

Therefore class ≠ type.

Bonus: Can have multiple classes generating objects of same type.

E.g., cartesian and polar points with same external interface.

Even though don't necessarily care if subclasses turn out to be subtypes, still need restrictions on redefinitions to avoid breaking other inherited methods.

Ex.:

```
method1(...) = ... p.method2(..)....
method2(...) = .....
```

If redefine method2 with different type, how do know will continue to be type-safe when used in method1 (if method1 is inherited & not changed).

Saw last time that sufficient to require new type of method2 to be subtype of original type.

Can set up type-checking rules for determining legal subclasses and subtypes and be guaranteed that can't break the typing system.

This is extremely important, since one of goals of object-oriented programming languages is to provide reusable libraries of components, much like that found with FORTRAN for numerical routines or Modula-2 for data structures.

Major advantage would be ability to make minor modifications to allow user to customize classes.

Sale of libraries is expected to become a major software industry. However, if selling library will typically only sell compiled version, not source code (though provide something like definition module).

If user doesn't see source code of superclass, how can s/he be confident that will get no type errors. Need the kind of guarantees claimed above.

Work here on TOOPLE, TOIL, PolyTOIL and LOOM (theses by Rob van Gent '93, Angie Schuett '94, Leaf Petersen '96, Hilary Browne '97, and Joe Vanderwaart '99, and support from J. Rosenberg & S. Calvo '96)

Resulted in object-oriented language which is type-safe and only requires classes and methods to be type-checked once (don't have to repeat when inherit methods).

LOOM: Uses MyType as type of "self" ("this) - similar to "like Current", but type-checking rules (provably) guarantee type-safety.

Replace subtyping by notion of "matching", written <#.
BILINKABLE[T] <# LINKABLE[T], but not subtypes!

Matching corresponds to extension, but when use "MyType" for parameter type or instance vble type, extension NOT correspond to subtyping.

If write x:T, then any value in "x" must have exactly type T, while x:#T means value can come from any type matching T.

If wish to use "slippery types" (like subtypes in ordinary OO languages) then must use #T as type.

Only restriction is cannot send "binary" messages to #-types.

E.g., if x:#LINKABLE[T] for some T, then can write
x.next() or x.right(),
but can't write
x.putRight(other)

Supports "match-bounded" polymorphism
F-bounded extension needed only very rarely.

Jon Burstein '98 implemented extension of Java based on similar principles.

## F-bounded vs "MyType" based languages

Both kinds of languages allow expression of all bad examples cited earlier - clone, equals, ColorCircle

F-bounded:
- Not preserved under subclass.
- Encoding requires extra type parameter in interface

Matching:
- Preserved under subclass
- Must distinguish exact vs. #-types.
  Can't send binary message to #-type.
- If no occurrences of MyType then #-types give exactly same effect as subtyping.

Which is better seems to depend on taste
- more experience necessary.

## Evaluation of OOL's.

Pro's (e.g., with Eiffel and Java)

1. Good use of information hiding. Objects can hide their state.

2. Good support for reusability. Supports generics like Ada, run-time creation of objects (unlike Ada)

3. Support for inheritance and subtyping provides for reusability of code.

Con's

1. Loss of locality.

2. Type-checking too rigid, unsafe, or requires link time global analysis. Others require run-time checks.

3. Semantics of inheritance is very complex. Small changes in methods may make major changes in semantics of subclass. It appears you must know definition of methods in superclass in order to predict impact on changes in subclass. Makes provision of libraries more complex.

4. Weak or non-existent support of modules.

Eiffel also provides support for number of features of modern software engineering - e.g., assertions.

What will be impact of OOL's on programmers and computer science?

Large number of powerful players jumped on the bandwagon without careful assessment of consequences. Current reaction against C++.

Many OO programmers don't really understand paradigm, esp. if use OO add-on to older language.

Many of the advantages claimed by proponents could be realized in Clu, Modula-2, or Ada (all available decade or more ago).

My advice: Specify carefully meaning of methods, avoid long inheritance chains, and be careful of interactions of methods.

When implement F-bounded polymorphism, Java could be a very successful compromise between flexibility and usefulness.

## Survey semantics specification methods.

1. Operational

2. Axiomatic

3. Denotational

Operational Semantics

May have originated with idea that definition of language be an actual implementation. E.g. FORTRAN on IBM 704.

Can be too dependent on features of actual hardware.

Define abstract machine, and give translation of language onto abstract machine.

Interpret abstract machine on actual machine to get implementation.

Ex: Interpreters for PCF.

Transformed a program into a "normal form" program (can't be further reduced).

Expressions reduce to pair (v,s), Commands reduce to new state, s.
E.g.

$$\frac{(e1, r, s) \quad (m, s') \qquad (e2, r, s') \quad (n, s'')}{(e1 + e2, r, s) \quad (m+n, s'')}$$

$$\frac{(X, r, s) \quad (Loc, s') \qquad (M, r, s') \quad (v, s'')}{(X := M, r, s) \quad (r, s''[v/X])}$$

$$(fun(X).M, r, s) \quad (< fun(X).M, r >, s)$$

$$\frac{(f, r, s) \quad (<\text{fun}(X).M, r'>, s') \qquad (N, r, s') \quad (v, s''), \qquad (M, r'[v/X], s'') \quad (v', s''')}{(f(N), r, s) \quad (v', s''')}$$

Type of semantics called "big-step" op. semantics.

Text describes small-step operational semantics.

$$\frac{(M, r, s) \quad (M', r', s')}{(X := M, r, s) \quad (X := M', r', s')}$$

$$(X := n, r, s) \quad s'[n/X] \quad \text{for } n \text{ an integer.}$$

Major difference is reduce expressions one step at a time. Can be shown to be equivalent.

Either way: Meaning of program is sequence of states that machine goes through in executing it. (trace of execution)

Essentially an interpreter for language.

Reasonable to ask about complexity of computation.

Very useful for compiler writers since very low-level description.

Abstract machine is simple enough that it is impossible to misunderstand its operation.

Axiomatic Semantics

No model of execution.

Definition tells what may be proved about programs. Associate axiom with each construct of language. Rules for composing pieces into more complex programs.

Meaning of construct is given in terms of assertions about computation state before and after execution.

General form:

{P} statement {Q}

where P and Q are assertions.

If P is true before execution and it terminates, then Q must be true after termination.

Assignment axiom:

{P [expression / id]} id := expression {P}

e.g.

{a+17 > 0} x := a+17 {x > 0}

or

{x > 1} x := x - 1 {x > 0}

Surprisingly, this axiom fails if allow arrays and subscripts. Also fails if allow side-effects or aliasing. See homework.

Early rules were often not sound!

While rule:

If {P & B} stats {P}, then

{P} while B do stats {P & ¬B}

E.g. if P is an invariant of stats, then after execution of the loop, P will still be true but B will have failed.

Composition:

If {P} S1 {Q}, {R} S2 {T}, and Q R, then

{P} S1; S2 {T}

Conditional:

If {P & B} S1 {Q}, {P & not B} S2 {Q}, then

{P} if B then S1 else S2 {Q}

Consequence:

If P Q, R T, and {Q} S {R}, then {P} S {T}

Prove program correct if show

{Precondition} Prog {PostCondition}

Often easiest to work backwards from Postcondition to Precondition.

```
Ex:       {Precondition: exponent0 ≥ 0}
    base <- base0
    exp <- exp0
    ans <- 1
    while exp > 0 do
        {assert:  ans * (base ** exp) = base0 ** exp0}
        {           exp ≥ 0}
        if odd(exp) then
                ans<- ans*base
                exp <- exp - 1
            else
                base <- base * base
                exp <- exp div 2
        end if
    end while
    {Postcondition:exp = 0}
    {                       ans = base0 ** exp0}
```

Axiomatic semantics due to Floyd & Hoare, Dijkstra also major contributor. Used to define semantics of Pascal [Hoare & Wirth, 1973]

Too high level to be of much use to compiler writers.
Perfect if proving programs correct.