- 10.3 -

Covariance
&
anchored types
Covariance?

Within the type system of a programming language, a typing rule or a type conversion operator is*:

- **covariant** if it preserves the ordering, ≤, of types, which orders types from more specific to more generic:
  
  ```
  animal := dog (animal: ANIMAL, dog: DOG)
  list := string_list (list: LIST [ ANY ], string_list: LIST [ STRING ])
  ```

- **contravariant** if it reverses this ordering, which orders types from more generic to more specific
  
  ```
  dog := animal
  string_list := list
  ```

- **novariant** if neither of these apply.

* http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
The need for covariance

- **CUSTOMER**
  - drink : BEVERAGE
  - serve (b : BEVERAGE)
    - do
      - drink := b
    - ensure
      - drink = b
  - end

- **MINOR**
  - drink : SOFT_DRINK
  - serve (b : SOFT_DRINK)
    - do drink := b end

- **BEVERAGE**

- **SOFT_DRINK**

- **ALCOHOL**

- **Client**
  - Inherit
Defeating covariance (1)

CUSTOMER → BEVERAGE

\[
drink : BEVERAGE
serve (b : BEVERAGE)
do drink := b end
\]

MINOR

SOFT_DRINK → ALCOHOL

\[
drink : SOFT_DRINK
serve (b : SOFT_DRINK)
do drink := b end
\]

Pimms : ALCOHOL

\[
c : CUSTOMER
Shiloh : MINOR
\]

\[
b : BEVERAGE
b := Pimms
\]

\[
c := Shiloh
c.\text{serve} (Pimms)
\]
Defeating covariance (2)

CUSTOMER

BEVERAGE

MINOR

SOFT_DRINK

ALCOHOL

Client

Inherit

drink : BEVERAGE
serve (b : BEVERAGE)
  do drink := b end

drink : SOFT_DRINK
serve (b : SOFT_DRINK)
  do drink := b end

Pimms : ALCOHOL

c : CUSTOMER

Shiloh : MINOR

bus : LIST [CUSTOMER]
school_bus : LIST [MINOR]

bus := school_bus

bus.item.serve (Pimms)

Generic conformance
**Terminology**

**Catcall**: incorrect application of a feature to an object as a result of

- Incorrect argument type
- Information hiding violation

*(CAT: Changed Availability or Type)*

**Diagram**

- **BIRD**
- **OSTRICH**

fly
Status

Problem known since late 80s
“Covariance” term coined by Luca Cardelli
Criticism of initial Eiffel design by W. Cook et al., 1989*
Numerous proposals since then

*A Proposal for Making Eiffel Type-Safe, ECOOP 1989
Mitigating mechanism 1: non-conforming inheritance

class
    C
inherit {NONE}
    B
feature
    ...
end

No polymorphism permitted:

\[ b1 := c1 \quad -- \text{Invalid} \]
Mitigating mechanism 2: anchored types

In practice: anchored types

```
class CUSTOMER feature
  drink : BEVERAGE
  serve (b : BEVERAGE)
  do
    drink := b
  end
end

class MINOR inherit CUSTOMER
  redefine drink, serve
feature
  drink : SOFT_DRINK
  serve (b : SOFT_DRINK)
  do
    drink := b
  end
end
```

```
class CUSTOMER feature
  drink : BEVERAGE
  serve (b : like drink)
  do
    drink := b
  end
end

class MINOR inherit CUSTOMER
  redefine drink
feature
  drink : SOFT_DRINK
end
```
Anchoring to \textbf{Current}

Also possible, in a class $C$:

$x$: like \textbf{Current}

In any descendant $D$ of $C$ (including $C$ itself), $x$ has type $D$
Mitigating mechanism 3: flat type checking

An assignment

\[ x := y \]

may be valid in a routine \( r \) of a class \( B \) but not necessarily in a descendant \( C \) which only redefines \( x \) (covariantly)

The Eiffel type system now specifies that every class must be independently valid

“Flat type checking”
Unrealistic approach: contravariance

CUSTOMER \rightarrow BEVERAGE

\textit{drink}: BEVERAGE
\textit{serve}(b: BEVERAGE)
\textbf{do} \textit{drink} := b \textbf{end}

MINOR \rightarrow BEVERAGE

\textit{drink}: ????
\textit{serve}(b: ????)
\textbf{do} \textit{drink} := b \textbf{end}

SOFT\_DRINK \rightarrow BEVERAGE

Results specialized, arguments generalized
Solves the problem
Hard to reconcile with practice. The world does seem to be covariant.
Novariance (argument passing)

C++, Java, .NET languages

Eliminates the problem (obviously)

Forces programmers to do the adaptations themselves

May result in brittle/unnecessary code
Previous solutions: genericity*

class CUSTOMER [DRINK_TYPE]

*Franz Weber: Getting Class Correctness and System Correctness Equivalent (How to get covariance right), TOOLS EUROPE 1992
Previous solutions: system-level validity*

Considering all assignments, compute **dynamic type set (DTS)** of any variable $x$. If there is an assignment $x := y$, or argument passing, all elements of DTS of $y$ are also in the DTS of $x$.

**Advantages:**
- No attempt at control flow analysis
- Fixpoint algorithm
- Helps with optimization

**Disadvantages:**
- Pessimistic
- Not incremental
- Difficulty of giving precise diagnostics

*B. Meyer: *Eiffel: The Language*, Prentice Hall, 1991*
Type intervals

Pimms : ALCOHOL
Shiloh : MINOR
c : CUSTOMER..MINOR

c := Shiloh
c.serve (Pimms)

-- Now invalid
Type intervals

Rule: a call $x.f(a)$, with $x : T..U$, must be valid when $x$ is given any type in $T..U$

Abbreviations:
- $x : T$ means $x : T..NONE$
- $LIST[T]$ means $LIST[T..T]$
The retained solution: a simplified version

Rule: a call \( x.f(a) \), with \( x: T \) not frozen, must be valid when \( x \) is given any descendant type of \( T \)
Genericity rule

**Rule:**

- An assignment with a different actual generic parameter requires the “variant” mark.
- The *variant* mark precludes the use of a routine with an argument of a formal generic parameter type.
By the way!

`bus : LIST[CUSTOMER]`
`vbus : LIST[variant CUSTOMER]`
`school_bus : LIST[MINOR]`

`vbus := school_bus` **OK**

`vbus.extend (Shiloh)` **Invalid**

`bus := school_bus` **Invalid**

`bus.extend (Shiloh)` **OK**

`bus.item.serve (Pimms)` ???

Invalid since, in `LIST[G]`, `item : G` is not `frozen`
Anchored ("like") declarations

New results:

- “Flat type checking” guarantees that like Current declarations are safe.

- $b : \text{like } a$, with $a$ of type $T$, may be considered an abbreviation not for $b : T$ as now, but for $b : \text{frozen } T$.

Then only explicit (non-anchored) covariance remains!
Status

Implemented and currently being experimented

Criterion: must not break code unless there is a real catcall risk

Need for formal proof

Proposal for a more flexible mechanism ("forget")
Covariance: summary

Reconcile modeling power with safety