

---

# Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 11: Genericità

# What we have seen in the previous lecture

---

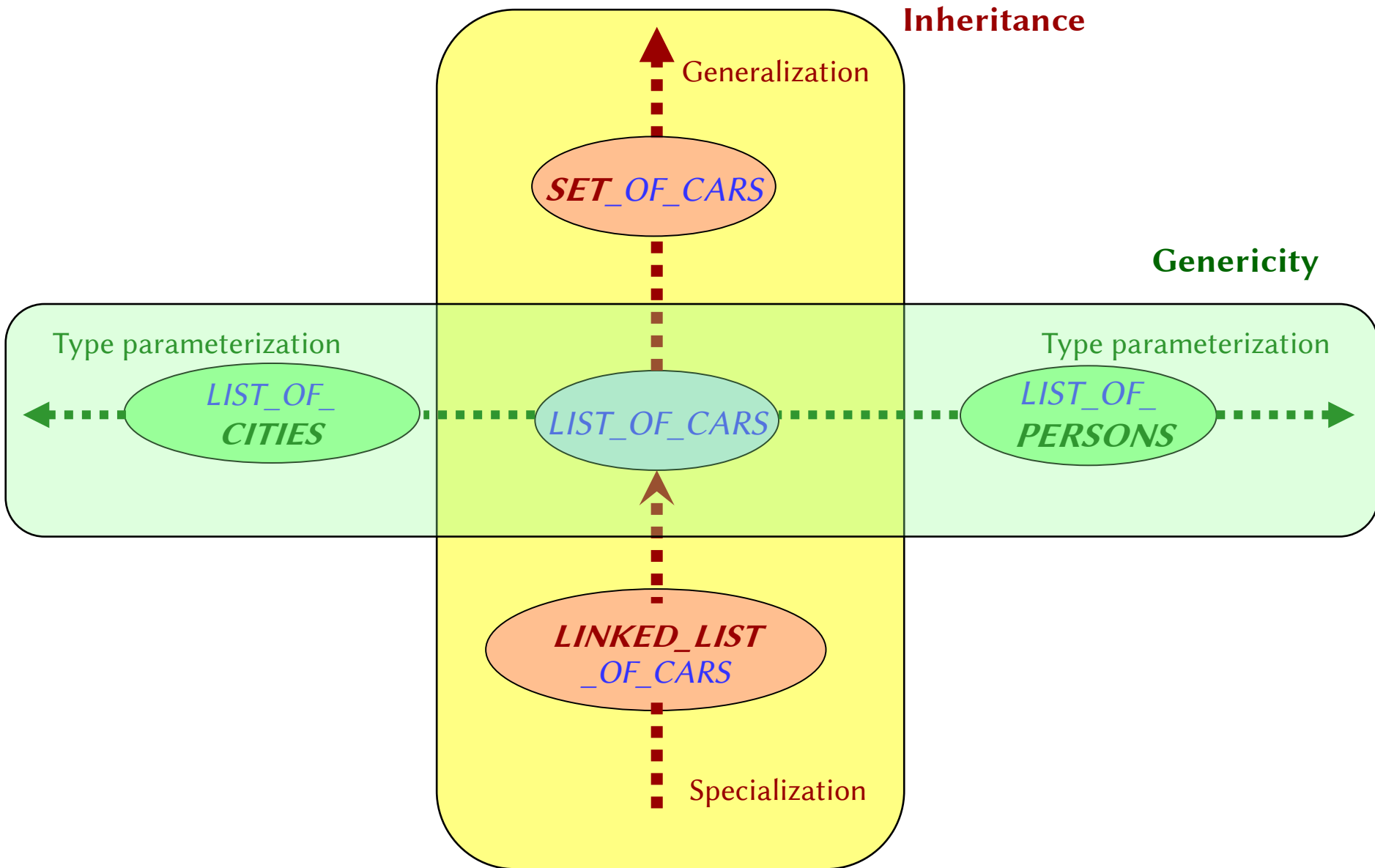
The basics of fundamental O-O mechanisms:

- Inheritance
- Polymorphism
- Dynamic binding
- Static typing

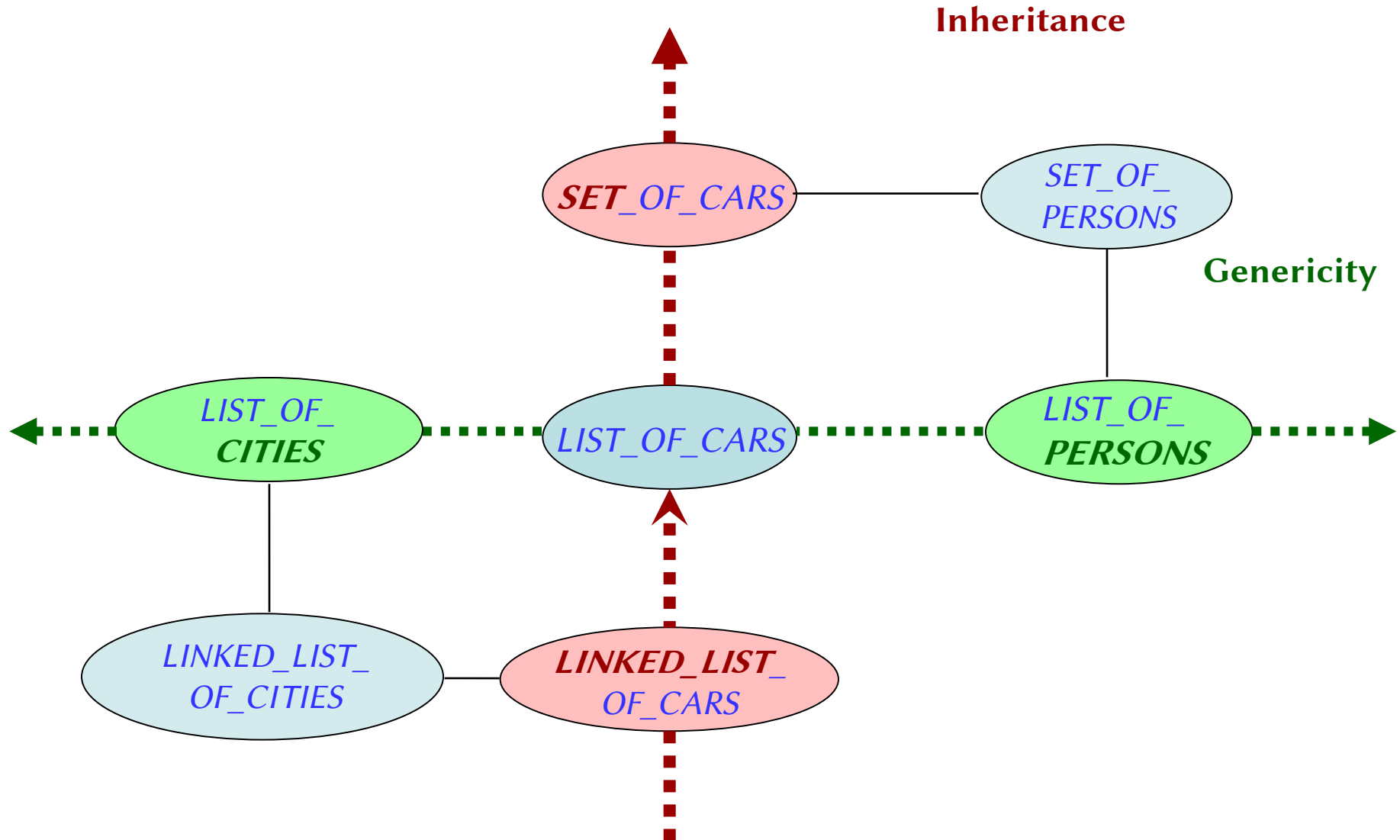
In this lecture we extend them with another key mechanism:

- Genericity

# Extending the basic notion of class



# Extending the basic notion of class



# Ensuring type safety

How could we define consistent “container” data structures, for various types, e.g. list of accounts, list of points?

Something like this:

*c : CITY ; p : PERSON*

*cities : LIST ;*

*people : LIST ;*

-----  
*people.extend ( p )*

*cities.extend ( c )*

**What if arguments  
are wrong?**

*c := cities.last*

*c.some\_city\_operation*

# Possible approaches

---

1. Duplicate code, manually or with help of macro processor
2. Wait until run time; if types don't match, trigger a run-time failure (Smalltalk)
3. Convert (“cast”) all values to a universal type, such as “pointer to void” in C
4. **Parameterize the class**, giving an explicit name **G** to the type of container elements. This is the Eiffel approach, also found in recent versions of Java, .NET and others.

# A generic class

## Definition

```
class LIST [G] feature
  extend (x: G) ...
  last: G ...
end
```

Formal generic parameter

Declaration

**Use:** obtain a generic derivation, e.g.

```
cities: LIST [CITY]
```

Actual generic parameter

Use

# Using generic derivations

*cities* : *LIST* [*CITY*]

*people* : *LIST* [*PERSON*]

*c* : *CITY*

*p* : *PERSON*

...

*cities.extend* (*c*)

*people.extend* (*p*)

*c* := *cities.last*

*c.some\_city\_operation*

## STATIC TYPING

The compiler will reject:

➤ *people.extend* (*c*)

➤ *cities.extend* (*p*)



# Static typing (reminder)

## Type-safe call:

A feature call  $x.f$  such that any object attached to  $x$  during execution has a feature corresponding to  $f$

[Generalizes to calls with arguments,  $x.f(a, b)$ ]

## Static type checker:

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be *type-safe*

## Statically typed language:

A programming language for which it is possible to write a *static type checker*

# Using genericity

---

Not only

*LIST [CITY]*

But also

*LIST [LIST [CITY]]*

And so on...

Many types can be derived from a same class:

a type is no longer exactly the same thing  
as a class!

(But every type remains **based** on a class.)

# What is a type?

(To keep things simple let's assume that a class has zero or one generic parameter)

A **type** is of one of the following two forms:

- $C$ , where  $C$  is the name of a **non-generic class**
- $D[T]$ , where  $D$  is the name of a **generic class** and  $T$  is a **type**

Have you seen the recursion?

# Types (reminder)

We use types to declare entities, as in

*x: SOME\_TYPE*

With the mechanisms defined so far, a type is one of:

- A non-generic class  
e.g. *x: METRO\_STATION*
- A **generic derivation**, i.e. the name of a class followed by a list of ***types***, the **actual generic parameters**, in brackets  
e.g. *x: LIST [METRO\_STATION]*  
*x: LIST [ARRAY [METRO\_STATION]]*

# Genericity: summary

---

Type extension mechanism

Reconciles flexibility with type safety

Enables us to have parameterized classes

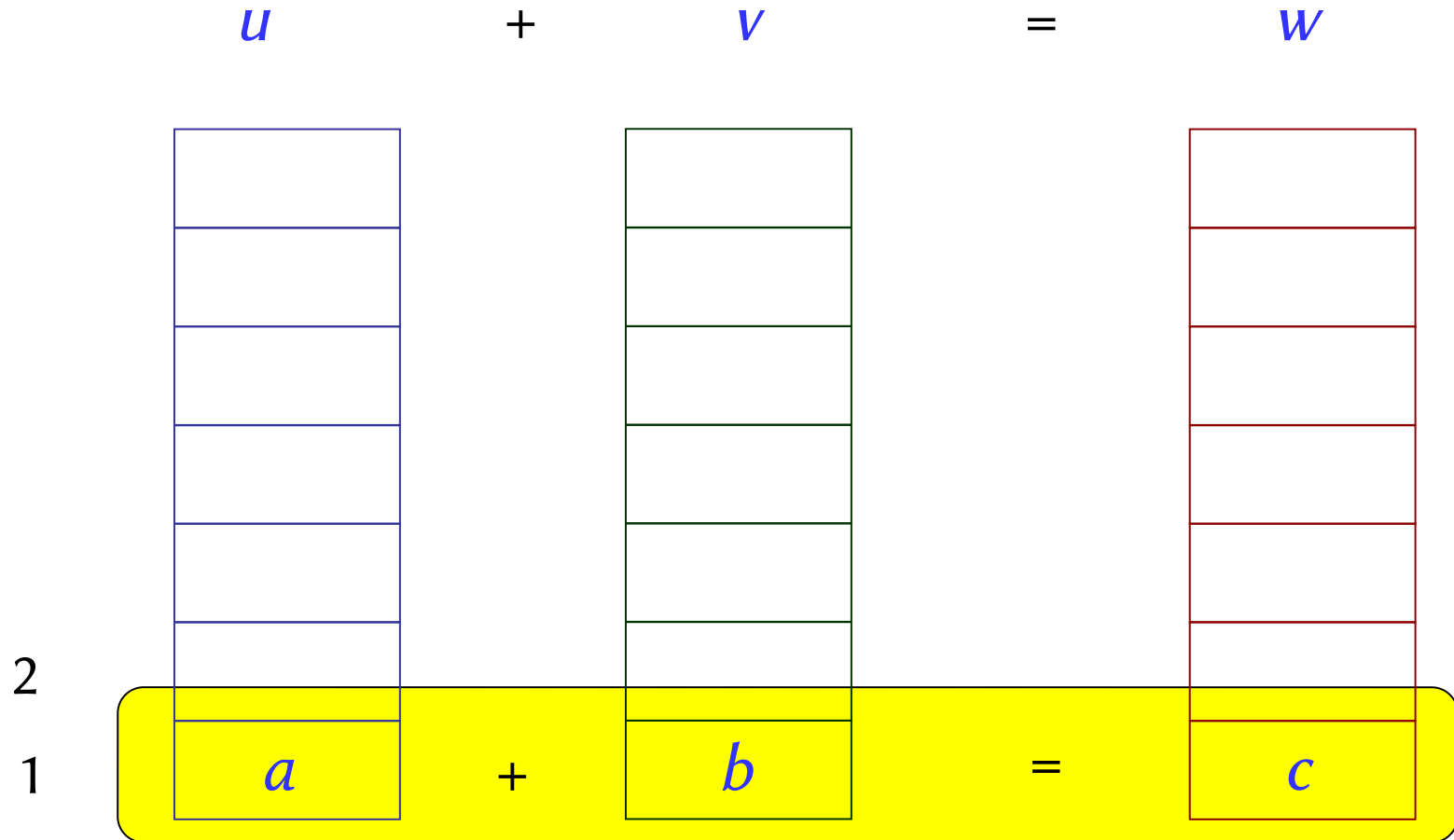
Useful for container data structures: lists, arrays, trees, ...

“Type” now a bit more general than “class”

# Adding two vectors (1)

```
class VECTOR [G ]  
  feature  
    plus alias "+" (other: VECTOR [G ]): VECTOR [G ]  
      -- Sum of current vector and other.  
    require  
      lower = other.lower  
      upper = other.upper  
    local  
      a, b, c : G  
    do  
      ... See next ...  
    end  
    ... Other features ...  
end
```

# Adding two vectors (2)



# Adding two vectors (3)

Body of *plus alias* "+" in *VECTOR [G]* :

**create** *Result.make (lower, upper)*

**from**

*i := lower*

**until**

*i > upper*

**loop**

*a := item (i)*

*b := other.item (i)*

*c := a + b* -- Requires "+" operation on G!

**Result.put** (*c, i*)

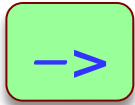
*i := i + 1*

**end**



# The solution

Declare class *VECTOR* as

```
class VECTOR [G  NUMERIC] feature
    ... The rest as before ...
end
```

Class *NUMERIC* (from the Kernel Library) provides features *plus alias* "+", *minus alias* "-" and so on.

# Constrained genericity (1)

---

## Unconstrained

*LIST* [*G*]

e.g. *LIST* [*INTEGER*], *LIST* [*PERSON*]

## Constrained

*HASH\_TABLE* [*G*  $\rightarrow$  *HASHABLE*]

*VECTOR* [*G*  $\rightarrow$  *NUMERIC*]

# Constrained genericity (2)

## Unconstrained

- the generic class  
 $LIST [G]$   
 can be used by substituting G with any class, e.g.:  
 $LIST [INTEGER]$ ,  $LIST [PERSON]$   
 as in:  
 $prices : LIST [INTEGER]$   
 $candidates : LIST [PERSON]$

## Constrained

- the generic **constrained** class  
 $HASH\_TABLE [G \rightarrow HASHABLE]$   
 can be used by substituting G only with a class  
 inheriting from (i.e., a subclass of) **HASHABLE**
- unconstrained genericity is constrained to **ANY**
- multiple constraints can be specified  
 $VECTOR [G \rightarrow \{COMPARABLE, NUMERIC\}]$   
 and then all constraint must be satisfied, e.g.:  
 $prices : VECTOR [INTEGER]$   
 $candidates : VECTOR [STRING]$

Correct!

Wrong!

# Improving the solution

Can we have *VECTOR* of *VECTOR* ... ?

Remember we need operation "+" on single items...

Make *VECTOR* itself a descendant of *NUMERIC*,  
effecting the corresponding features:

```
class VECTOR [G -> NUMERIC] inherit  
    NUMERIC
```

```
feature
```

```
    ... Rest as before, including infix "+"...
```

```
end
```

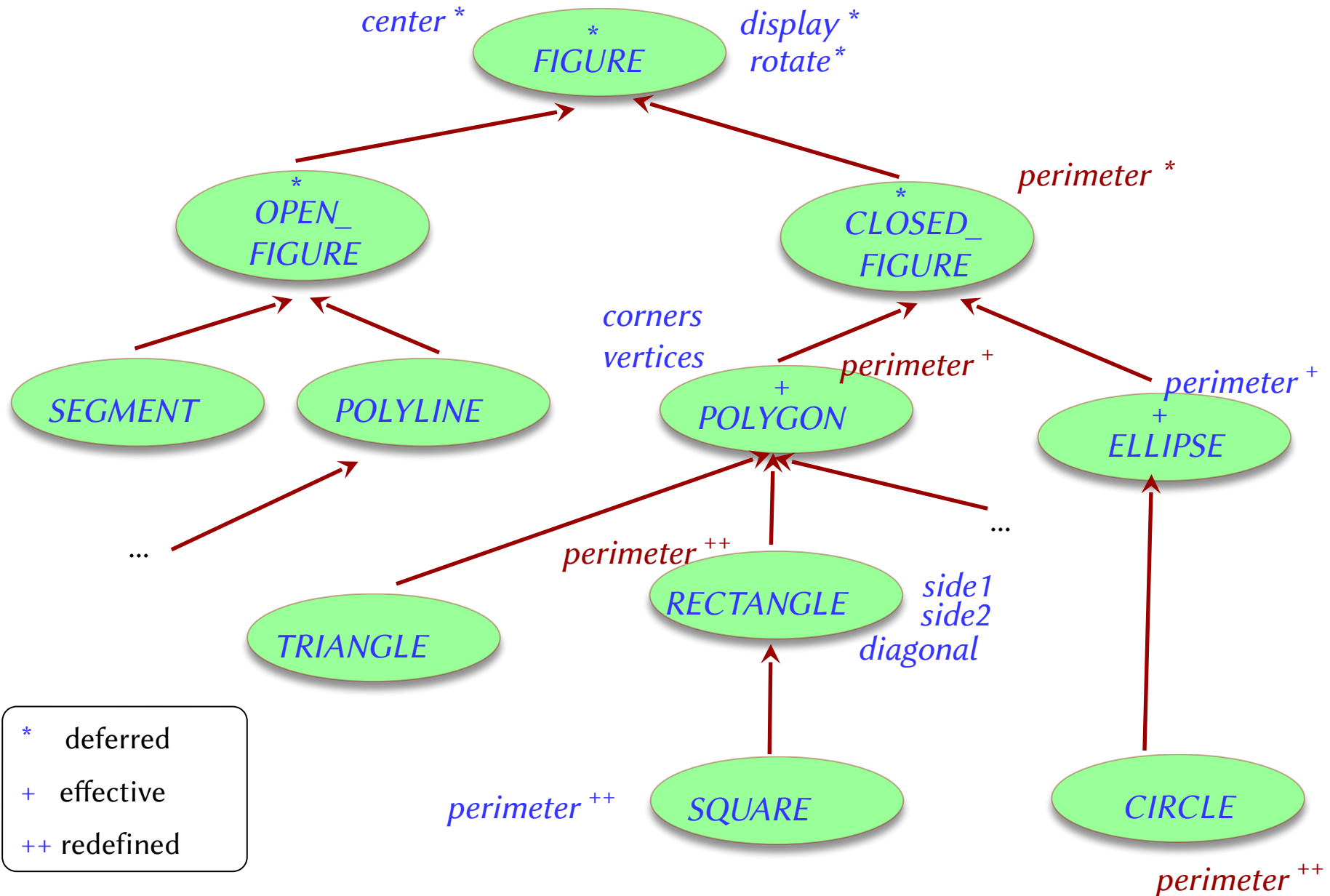
Then it is possible to define

```
v: VECTOR [INTEGER]
```

```
vv: VECTOR [VECTOR [INTEGER]]
```

```
vvv: VECTOR [VECTOR [VECTOR [INTEGER]]]
```

# A more realistic example of inheritance hierarchy



# Polymorphic data structures

*figs: LIST [FIGURE]*

*p1, p2: POLYGON*

*c1, c2: CIRCLE*

*e: ELLIPSE*

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

**feature**

*extend* (*v: G*)

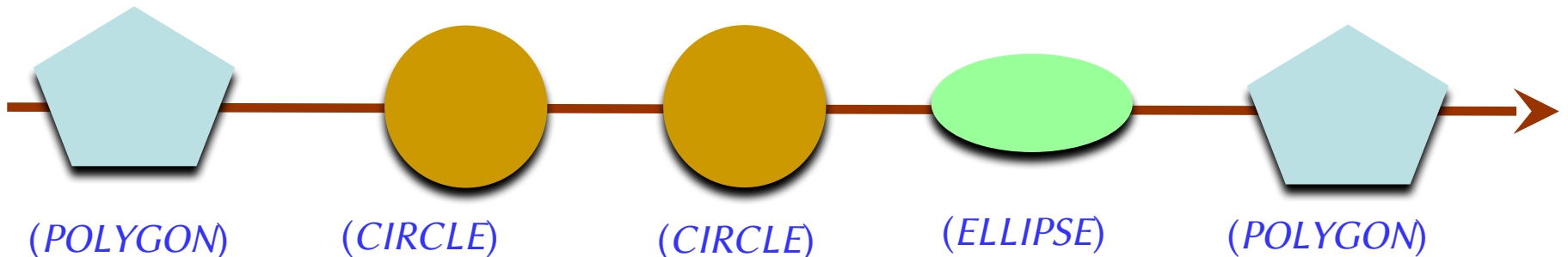
**do ... end**

...

**end**

*figs.extend* (*p1*) ; *figs.extend* (*c1*) ; *figs.extend* (*c2*)

*figs.extend* (*e*) ; *figs.extend* (*p2*)



# Working with polymorphic data structures

*figs: LIST [FIGURE]*

...

**across** *figs* **as** *c* **loop**

*c.item.display*

**end**

**from** *figs.start*

**until** *figs.after*

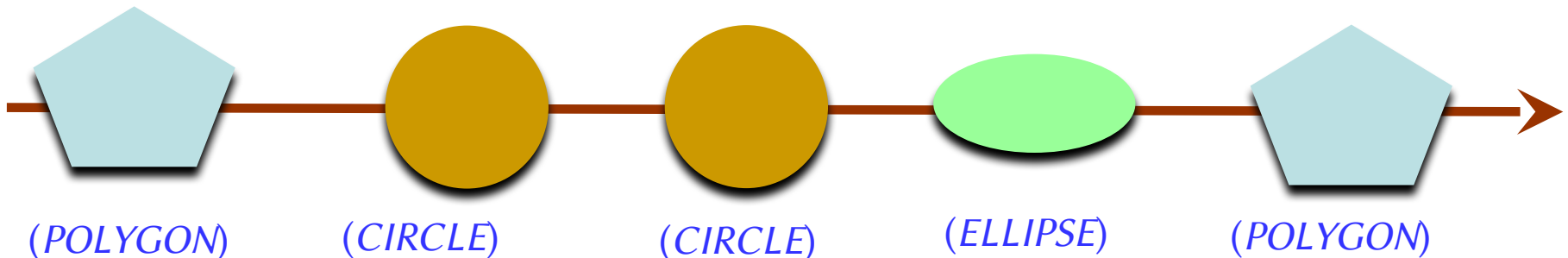
**loop**

*figs.item.display*

*figs.forth*

**end**

Dynamic binding



# Polymorphic data structures (one more example)

*fleet: LIST [VEHICLE]*

*a\_cab: TAXI*

*a\_tram: TRAM*

*a\_house: HOUSE*

Not a subclass of  
*VEHICLE*

*fleet.extend(a\_tram)*

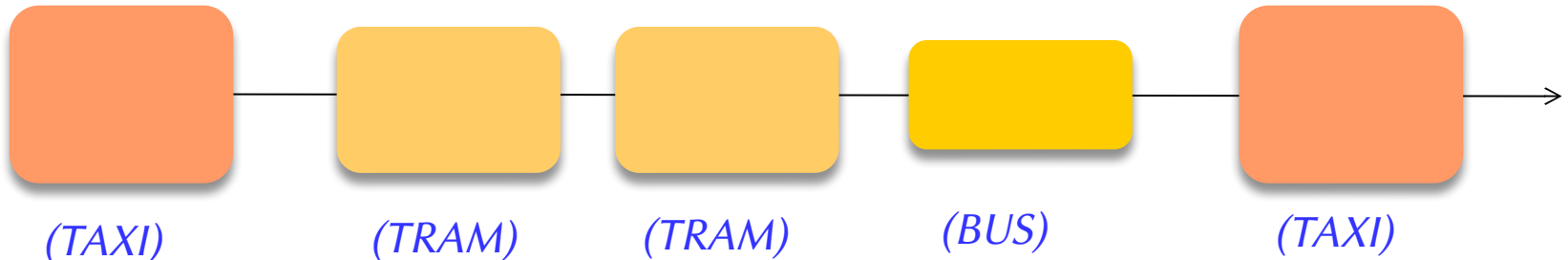
*fleet.extend(a\_cab)*

*fleet.extend(a\_house)*

Allowed?

No!

Using genericity you can provide an implementation of type safe containers. This helps avoiding object-tests





# Definition: polymorphism (adapted)

---

An **attachment** (assignment or argument passing) is **polymorphic** if its target entity and source expression have different types.

An **entity** or **expression** is **polymorphic** if – as a result of polymorphic attachments – it may at runtime become attached to objects of different types.

A **container data structure** is **polymorphic** if it may contain references to objects of different types.

**Polymorphism** is the existence of these possibilities.

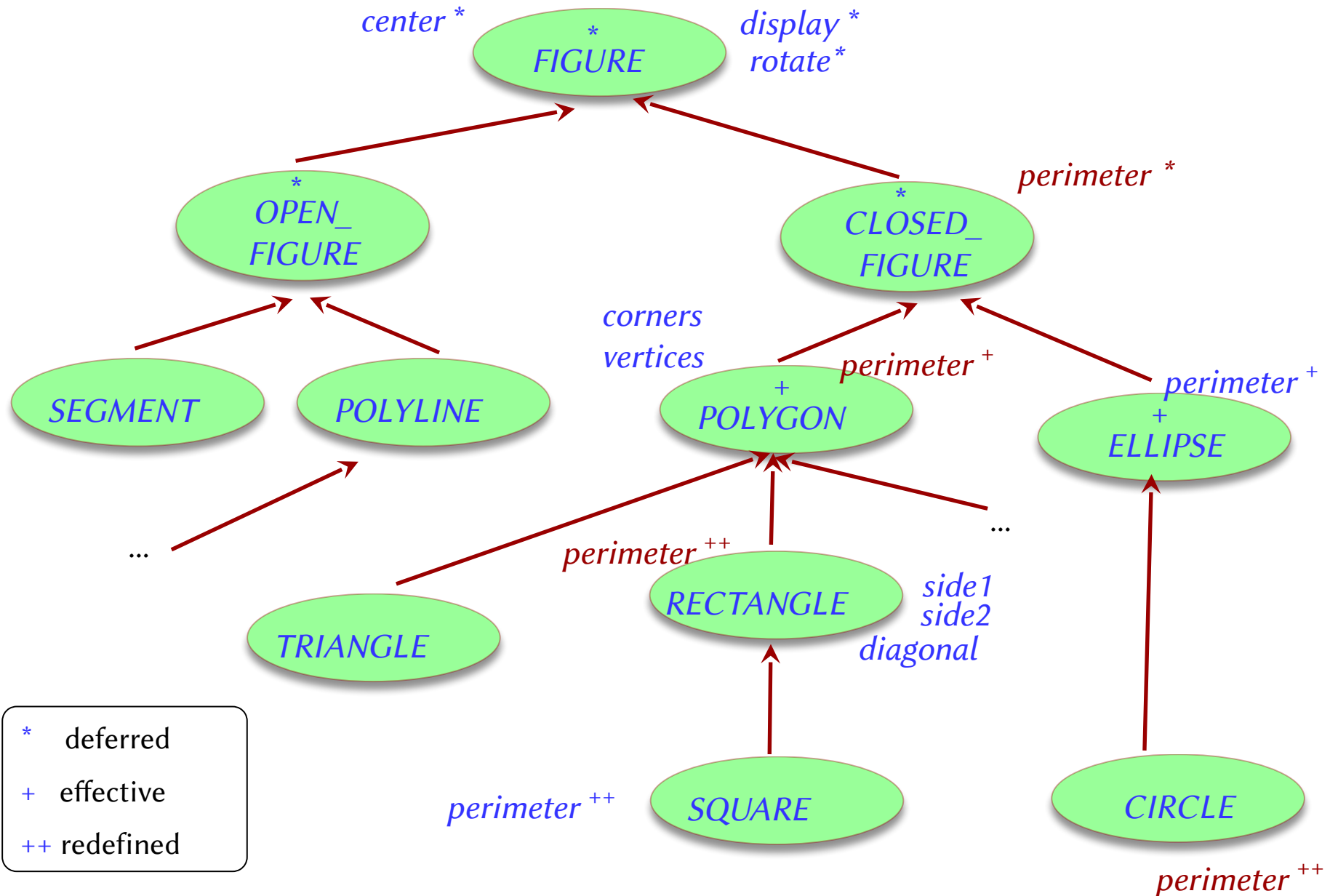
# Conformance: full definition

An expanded type conforms only to itself.

A reference type  $U$  **conforms** to a reference type  $T$  if either:

- Both have no generic parameters, and  $U$  is a descendant of  $T$ .
- They are both generic derivations with the same number of actual generic parameters, and
  - the base class of  $U$  is a descendant of the base class of  $T$ ,
  - and every actual parameter of  $U$  (recursively) **conforms** to the corresponding actual parameter of  $T$ .

# A more realistic example of inheritance hierarchy



# Reminder: the list of figures

```

class
    LIST [G]

feature
    ...
    last : G do ...
    extend (x : G) do ...

```

end

```

figs : LIST [FIGURE]
r : RECTANGLE
s : SQUARE
t : TRIANGLE
p : POLYGON

```

```

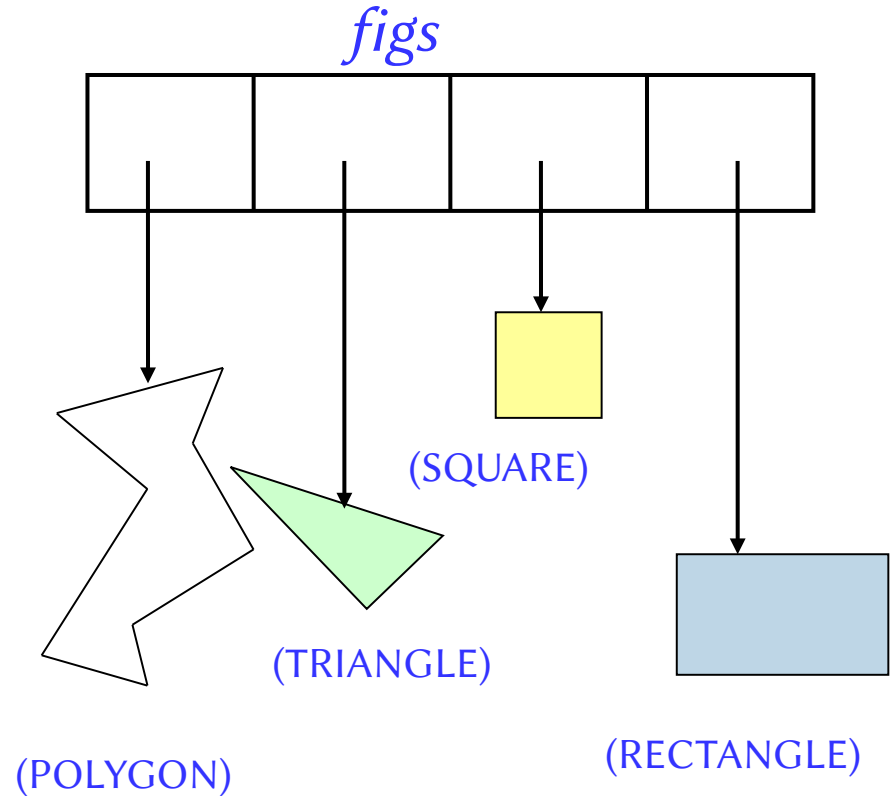
figs.extend (p); figs.extend (t); figs.extend (s); figs.extend (r)

```

```

figs.last.display

```



# Enforcing a type: the problem

*figs.store* ("FN") -- Save on disk (from class *STORABLE*)

...

-- Two years later:

*figs := retrieved* ("FN") -- Read from disk

...

*x := figs.last* -- [1]

*print (x.diagonal)* -- [2]

Is there anything wrong with this?

Which is the type of *x*?

- If *x* is declared of type *FIGURE*, [2] is invalid.
- If *x* is declared of type *RECTANGLE*, [1] is invalid.

# Enforcing a type: the Object Test

Expression to be tested

Object-Test Local

if attached {*RECTANGLE*} *figs.retrieved.last("FN")* as *r* then

*print(r.diagonal)*

...Do anything else with *r*, guaranteed

...to be non void and of dynamic type *RECTANGLE*

else

*print("Too bad.")*

end

**SCOPE** of the Object-Test Local

# Earlier mechanism: assignment attempt

*f: FIGURE*

*r: RECTANGLE*

...

*figs.retrieve* ("FN")

*f := figs.last*

*r ?= f*

**if** *r* /= **Void** **then**

*print* (*r.diagonal*)

**else**

*print* ("Too bad.")

**end**

# Assignment attempt

$$x \text{ ?} = y$$

with

$$x : A$$

Semantics:

- If  $y$  is attached to an object whose type conforms to  $A$ , perform normal reference assignment.
- Otherwise, make  $x$  void.