Lecture 8: Genericity, Inheritance

Genericity

- Parameterized classes for static typing
- Examples: stacks, arrays
- Constrained genericity (preview)
Genericity

Unconstrained

\text{LIST} [G]
e.g. \text{LIST} [\text{INTEGER}], \text{LIST} [\text{PERSON}]

Constrained

\text{HASH\_TABLE} [G \rightarrow \text{HASHABLE}]
\text{VECTOR} [G \rightarrow \text{NUMERIC}]

Extending the basic notion of class

\text{LIST\_OF\_STOCKS}
\text{SET\_OF\_STOCKS}
\text{LINKED\_LIST\_OF\_STOCKS}
\text{LIST\_OF\_COMPANIES}
\text{LIST\_OF\_PERSONS}
\text{LINKED\_LIST\_OF\_COMPANIES}
\text{SET\_OF\_PERSONS}

Inheritance

Abstraction

Specialization
**Genericity: Ensuring type safety**

How can we define consistent "container" data structures, e.g. list of accounts, list of points?

Dubious use of a container data structure:

```plaintext
c : COMPANY
a : PERSON
companies : LIST ...
people : LIST ...
  companies.extend(c)
  people.extend(a)
  c := people.last
  c.change_recommendation(Buy)
```

**Possible approaches**

Wait until run time; if types don't match, trigger a run-time failure. (Smalltalk)

Cast to a universal type, such as "pointer to void" in C.

Duplicate code, manually or with help of macro processor.

Parameterize the class, giving an explicit name \( G \) to the type of container elements. This is the Eiffel approach.

**A generic class**

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

To use the class: obtain a generic derivation, e.g.

```plaintext
companies : LIST[COMPANY]
```
**Conformance rule**

$B[U]$ conforms to $A[T]$ if and only if $B$ is a descendant of $A$ and $U$ conforms to $T$.

**Using generic derivations**

```plaintext
companies: LIST[COMPANY]
people: LIST[PERSON]
c: COMPANY
p: PERSON
...
companies.extend(c)
people.extend(p)

c := companies.last

c.change_recommendation(Buy)
```

**Genericity and static typing**

Compiler will reject

```plaintext
people.extend(c)
companies.extend(p)
```

To define more flexible data structures (e.g. stack of figures): use inheritance, polymorphism and dynamic binding.
Typing in an O-O context

An object-oriented language is statically typed if and only if it is possible to write a "static checker" which, if it accepts a system, guarantees that at run time, for any execution of a feature call x.f, the object attached to x (if any) will have at least one feature corresponding to f.

Constrained genericity

class VECTOR [G] feature
    infix "+" (other VECTOR [G]): VECTOR [G] is
        -- 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

\[ \begin{align*}
    u + v &= w \\
    a + b &= c
\end{align*} \]
Constrained genericity

Body of infix "+":

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 infix "*", infix "/" and so on.

Improving the solution

Make VECTOR itself a descendant of NUMERIC, effecting the corresponding features:
class VECTOR [G –> NUMERIC] inherit
NUMERIC
feature
... The 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 generic library class: Arrays

Using arrays:

\[ a : \text{ARRAY[REAL]} \]

\[ \text{create a make (1, 300)} \]
\[ \text{a.put (3.5, 25)} \]
\[ x := a \text{ item(i)} \]

-- Alternatively: \( x := a @ i \)
-- Using the function infix "@"

Also: \( \text{ARRAY2[G]} \) etc.

Class ARRAY (1)

\[
\text{class ARRAY[G] create make feature lower, upper: INTEGER count: INTEGER make (min: INTEGER, max: INTEGER) is \quad \text{-- Allocate array with bounds min and max. do end}}\]

Class ARRAY (2)

\[
\text{item, infix "@" (i, INTEGER): G is \quad \text{-- Entry of index i require lower <= i i <= upper do ... end}}\]
\[
\text{put (v \& i INTEGER) is \quad \text{-- Assign the value of v to the entry of index i require lower <= i i <= upper do ... end}}\]

\[
\text{invariant count = upper - lower + 1 end}\]

Complementary material

- OOSC2:
  - Chapter 10: Genericy

What is inheritance?

- Describe a new class as extension or specialization of an existing class. (With MULTIPLE inheritance it can be an extension of several existing classes.)
  - From the module viewpoint: if \(B\) inherits from \(A\), all the services of \(A\) are potentially available in \(B\) (possibly with a different implementation).
  - From the type viewpoint: inheritance is the "is-plus-but-except" relation. If \(B\) inherits from \(A\), whenever an instance of \(A\) is required, an instance of \(B\) will be acceptable.

What is inheritance?

```
class A
  feature a is do
    -- Some Code
  end
end

class B
  feature b is do
    -- Some Code
  end
end

class C
  inherit A
  feature
    c is do
      -- Some Code
    end
end

class D
  inherit A, B
  feature
    c is do -- Some Code end
    a is do -- Some Code end
    b is do -- Some Code end
end
```
### Terminology
- Parent, Child
- Ancestor, Descendant
- The ancestors of $B$ are $B$ itself and the ancestors of its parents.
- Proper ancestor, Proper descendant
- Direct instance, Instance
- The instances of $A$ are the direct instances of its descendants.
- (Other terminology: subclass, superclass, base class)

### Terminology: Deferred classes
A deferred class has at least one deferred feature.
A deferred feature is a feature that has no feature body but only the feature declaration with its signature.

### Terminology: Effective classes and features
An effective class is a class that inherits from a deferred class and implement at least one of the deferred features of the deferred class.
### Terminology: Redefinition of a feature of a parent class

++ redefined

```
class C
  feature f
end
end

class D
  inherit C
  redefine f
end
end
```

C and D are normal classes – they are not deferred and not effective. Here they are used to show how a feature \( f \) from class C can be redefined in class D.

### Example: Inheritance hierarchy

![Inheritance hierarchy diagram](image)

- **OPEN\_FIGURE**
- **CLOSED\_FIGURE**
- **SEGMENT**
- **POLYLINE**
- **POLYGON**
- **ELLIPSE**
- **CIRCLE**
- **RECTANGLE**
- **TRIANGLE**
- **SQUARE**

### Example: POLYGON

```
class POLYGON
create
make
feature vertices: ARRAY [POINT]
vertices_count: INTEGER
perimeter: REAL is
  -- Perimeter length
  do from ... until ... loop
    Result := Result + (vertices \( i \) . distance (vertices \( i + 1 \))
  end
end
invariant
  vertices_count \( \geq \) 3
  vertices_count = vertices_count
end
```

![Polygon example diagram](image)
Example: RECTANGLE by redefining POLYGON

class RECTANGLE
inherit POLYGON
redefine perimeter end
create make
feature
diagonal, side1, side2: REAL
perimeter: REAL is
-- Perimeter length
do
Result := 2 * (side1 + side2)
end
invariant vertices_count = 4
end

End of lecture 8