Eiffel: Analysis, Design and Programming

Bertrand Meyer
(Nadia Polikarpova)
- 7 -

Genericity
Overview

- Motivation
- Syntax
- Type safety
- The notion of type
What’s wrong with this code?

class LIST_OF_CARS feature
  extend (v: CAR) is ...
  remove (v: CAR) is ...
  item: CAR is ...
end

class LIST_OF_CITIES feature
  extend (v: CITY) is ...
  remove (v: CITY) is ...
  item: CITY is ...
end
What’s wrong with this code?

```ruby
class LIST_OF_CARS feature
  append (other: LIST_OF_CARS)
    do
      from other.start until other.after loop
        extend (other.item)
      end
    end
end

class LIST_OF_CITIES feature
  append (other: LIST_OF_CITIES)
    do
      from other.start until other.after loop
        extend (other.item)
      end
    end
end
```

**DRY Principle: Don’t Repeat Yourself**
Possible approaches for containers

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, now also found in Java (1.5), .NET (2.0) and others.
Genericity solution to LIST_OF...

class LIST\([G]\) feature
    append (other: LIST\([G]\) ) do
        do
            from other.start until other.after loop
                extend (other.item)
            end
        end
    end
end

city_list: LIST\([CITY]\]
car_list: LIST\([CAR]\)
A generic class

class \textit{LIST}[\textbf{G}] \ feature
\begin{align*}
  \text{extend} \ (x: \textit{G}) & \ \ldots \\
  \text{last}: \textit{G} & \ \ldots
\end{align*}
end

To use the class: obtain a \textit{generic derivation}, e.g.

cities: \textit{LIST}[\textbf{CITY}]
Genericity: ensuring type safety

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

Without genericity, something like this:

```plaintext
C : CITY ; p : PERSON
city : LIST ...
people : LIST ...
people.extend (p)
city.extend (c)
c := city.last
C.add_road
```

What if wrong?
Static typing

**Type-safe call** (during execution):

A feature call `x.f` such that the object attached to `x` 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 generic derivations

`cities : LIST[CITY]`
`people : LIST[PERSON]`
`c : CITY`
`p : PERSON`
...
`cities.extend (c)`
`people.extend (p)`

\[ c := cities.last \]
\[ c.add_road \]

**STATIC TYPING**

The compiler will reject:

- `people.extend (c)`
- `cities.extend (p)`
Types

\textit{LIST [CITY]}
\textit{LIST [LIST [CITY]]}
...

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

(But every type remains \texttt{based} on a class.)
Definition: Type

We use types to declare entities, as in

\[ x : \text{SOME\_TYPE} \]

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

- A non-generic class  
  e.g.  \text{METRO\_STATION}

- A generic derivation, i.e. the name of a class followed by a list of \textit{types}, the actual generic parameters, in brackets (also recursive)  
  e.g.  \text{LIST[ARRAY[METRO\_STATION]]}  
  \text{LIST[LIST[CITY]]}  
  \text{TABLE[STRING, INTEGER]}
So, how many types can I possibly get?

Two answers, depending on what we are talking about:

- **Static types**
  Static types are the types that we use while writing Eiffel code to declare types for entities (arguments, locals, return values)

- **Dynamic types**
  Dynamic types on the other hand are created at runtime. Whenever a new object is created, it gets assigned to be of some type.
### Static types

```plaintext
class EMPLOYEE
  feature
    name: STRING
    birthday: DATE
  end

class DEPARTMENT
  feature
    staff: LIST[EMPLOYEE]
  end
```

bound by the program text:

- `EMPLOYEE`  
- `STRING`  
- `DATE`  
- `DEPARTMENT`  
- `LIST[G]`  
- becomes `LIST[EMPLOYEE]`
Object creation, static and dynamic types

class TEST_DYNAMIC_CREATION
feature
    ref_a: A; ref_b: B
    -- Suppose B, with creation feature make_b,
    -- inherits from A, with creation feature make_a
    do_something
        do
            create ref_a.make_a
            -- Static and dynamic type is A
            create {B} ref_a.make_b
            -- Static type is A, dynamic type is B
        end
    end
end
Dynamic types: another example

class SET[G] feature
  powerset: SET[SET[G]] is
    do
      create Result
      -- More computation...
    end

  i_th_power (i: INTEGER): SET[ANY]
    require i >= 0
    local n: INTEGER
    do
      Result := Current
      from n := 1 until n > i loop
        Result := Result.powerset
        n := n + 1
      end
    end
end

Dynamic types from i_th_power:

SET[ANY]
SET[SET[ANY]]
SET[SET[SET[ANY]]]

...
Generics: 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”