Introduction to Programming
Bertrand Meyer

Lecture 11:
The dynamic model

Last revised 5 December 2005

Reversing a list

\[
\text{reverse}\ 
\text{local} \ 
\text{previous, next: LINKABLE}[G] \ 
\text{do} \ 
\text{from} \ 
\text{next := first ; first := Void} \ 
\text{invariant} \ 
\text{first} \cdot \text{right} \cdot \text{mirror} + \text{next} \cdot \text{right} \cdot \text{tail} = \text{old model} \ 
\text{until} \ 
\text{next == Void loop} \ 
\text{previous := first} \ 
\text{first := next} \ 
\text{next := next} \cdot \text{right} \ 
\text{first} \cdot \text{put_right}(\text{previous}) \ 
\text{end} \ 
\text{ensure} \ 
\text{model} \cdot \text{mirror = old model} \ 
\text{end}
\]

Notation

\[ a^x \]
where \( a \) is an attribute (e.g. \( \text{right} \))

List reversal

\[ \text{reverse}\ 
\text{local} \ 
\text{previous, next: LINKABLE}[G] \ 
\text{do} \ 
\text{from} \ 
\text{next := first ; first := Void} \ 
\text{invariant} \ 
\text{first} \cdot \text{right} \cdot \text{mirror} + \text{next} \cdot \text{right} \cdot \text{tail} = \text{old model} \ 
\text{until} \ 
\text{next == Void loop} \ 
\text{previous := first} \ 
\text{first := next} \ 
\text{next := next} \cdot \text{right} \ 
\text{first} \cdot \text{put_right}(\text{previous}) \ 
\text{end} \ 
\text{ensure} \ 
\text{model} \cdot \text{mirror = old model} \ 
\text{end}
\]

The key step

We get:

\[ \text{first} \cdot \text{right}^x \cdot \text{mirror} + \text{next} \rightarrow \text{tail} \]

The invariant was:

\[ \text{first} \cdot \text{right}^x \cdot \text{mirror} + \text{next} \cdot \text{right}^x \cdot \text{tail} = \text{old model} \]

List reversal

\[ \text{reverse}\ 
\text{local} \ 
\text{previous, next: LINKABLE}[G] \ 
\text{do} \ 
\text{from} \ 
\text{next := first ; first := Void} \ 
\text{invariant} \ 
\text{first} \cdot \text{right} \cdot \text{mirror} + \text{next} \cdot \text{right} \cdot \text{tail} = \text{old model} \ 
\text{until} \ 
\text{next == Void loop} \ 
\text{previous := first} \ 
\text{first := next} \ 
\text{next := next} \cdot \text{right} \ 
\text{first} \cdot \text{put_right}(\text{previous}) \ 
\text{end} \ 
\text{ensure} \ 
\text{model} \cdot \text{mirror = old model} \ 
\text{end}
\]

The trouble with reference assignment

A comfortable mode of reasoning:

- Here SOME_PROPERTY holds of \( a \)
- "Apply SOME_OPERATION to \( a \)"
- Here SOME_PROPERTY still holds of \( a \)

This applies to "expanded" values, e.g. integers

- Here \( P(a) \) holds
- \( \text{OP}(b) \)
- Here \( P(a) \) still holds of \( a \)
On the other hand...

-- I heard that the boss's cousin earns less
-- than 50,000 francs a year

"Raise Caroline's salary by 1 franc"
-- ??????

Metaphors:
- "The beautiful daughter of Leda"
- "Menelaus's spouse"
- "Paris's lover"

Practical advice

- Reference assignment is useful
- It's also potentially tricky
- As much as possible, leave it to specialized libraries of
general data structures

Variants of assignment and copy

- Reference assignment ($a$ and $b$ of reference types):
  $b := a$
- Object duplication (shallow):
  $c := a.twin$
- Object duplication (deep):
  $d := a.deep_twin$
- Also: shallow field-by-field copy (no new object is
created):
  $e.copy(d)$

Shallow and deep cloning

Initial situation:

Result of:

- $b := a$
- $c := a.twin$
- $d := a.deep_twin$

Where do these mechanisms come from?

- Class ANY in the Eiffel "Kernel Library"
- Every class that doesn't explicitly inherit from another
  is considered to inherit from ANY
- As a result, every class is a descendant of ANY.
Completing the inheritance structure

A related mechanism: Persistence

\begin{align*}
  &a.\text{store} \,(\text{file}) \\
  &\quad \Rightarrow b\neq \text{retrieved} \,(\text{file}) \\
  &\text{Storage is automatic.} \\
  &\text{Persistent objects identified individually by keys.} \\
  &\text{These features come from the library class \text{STORABLE}.}
\end{align*}

Objects and references

What to do with unreachable objects

- States of a reference:
  - create \( p \)
  - \( p := q \) (where \( q \) is attached)
  - \( p := \text{Void} \)

- Operations on references:
  - create \( p \)
  - \( p := q \)
  - \( p := \text{Void} \)
  - if \( p = \text{Void} \) then ...

- Reference assignments may make some objects useless.

  - Two possible approaches:
    - Manual "free" (C++).
    - Automatic garbage collection (Eiffel, Oberon, Java, .NET)

The C programmer's view

Arguments for automatic collection

- Manual reclamation is dangerous for reliability.
  - Wrong "frees" are among the most difficult bugs to detect and correct.
- Manual reclamation is tedious.
- Modern garbage collectors have acceptable performance overhead.
- GC is tunable: disabling, activation, parameterization...
Properties of a garbage collector (GC)

- Consistency (never reclaim a reachable object).
- Completeness (reclaim every unreachable object eventually).
- Consistency (also called safety) is an absolute requirement. Better no GC than an unsafe GC.
- But: safe automatic garbage collection is hard in C-based languages.

The object-oriented form of call

- `some_target.some_feature(some_arguments)`

For example:

```java
Paris.display
Line6.extend(Station_Parade_Platz)
```

```java
x := a.plus(b)
```

Infix and prefix operators

- In `a - b`
  - the `-` operator is "infix" (written between operands)
- In `- b`
  - the `-` operator is "prefix" (written before the operand)

Operator features

```java
expanded class INTEGER feature
  plus alias *+ (other: INTEGER): INTEGER is
    do ...
  end

product alias */ (other: INTEGER): INTEGER is
    do ...
  end

unary minus alias - (other: INTEGER): INTEGER is
    do ...
  end
```

Calls such as `i.plus(j)` can now be written `i + j`

End of lecture 11