Introduction to Programming

Bertrand Meyer

Lecture 11:
The dynamic model

Reversing a list

reverse local
previous, next: LINKABLE
from
next := first; first := Void
invariant
?
until next = Void loop
previous := first
first := next
next := next • right
first • put_right(previous)
end
ensure
?
end

Notation

\( a^x \)

where \( a \) is an attribute (e.g. right)
The key step

We get:
\[\text{first} \cdot \text{right}^k \cdot \text{mirror} \]
\[+ \]
\[\text{next} \cdot \text{right}^k \cdot \text{tail} \]

The invariant was:
\[\text{first} \cdot \text{right}^k \cdot \text{mirror} + \text{next} \cdot \text{right}^k \cdot \text{old model} \]

List reversal

reverse
local
previous, next: LINKABLE[6]
do
from
next := first; first := Void
invariant
\[\text{first} \cdot \text{right}^k \cdot \text{mirror} = \text{next} \cdot \text{right}^k \cdot \text{old model} \]
until next = Void loop
previous := first
first := next
next := next \cdot right
first \cdot put_right(previous)
end
ensure
\[\text{model} \cdot \text{mirror} = \text{old model} \]
end

The trouble with reference assignment

A comfortable mode of reasoning:

-- Here SOME_PROPERTY holds of a

"Apply SOME_OPERATION to it"

-- Here SOME_PROPERTY still holds of a

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

-- Here P(a) holds

OP(b)

-- Here P(a) still holds of a
Dynamic aliasing

\begin{itemize}
  \item \texttt{a, b: LINKABLE [STRING]}
  \item \texttt{create a...}
  \item \texttt{\quad b := a}
  \item \texttt{\quad a.put("Haldenegg")}
  \item \texttt{\quad -- Here \texttt{a.item} has value "Haldenegg"}
  \item \texttt{\quad a.put("Paradeplatz")}
  \item \texttt{\quad -- Here \texttt{a.item} has value ????}
\end{itemize}

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"

-- ????

\textbf{Metaphors:}
  \begin{itemize}
    \item "The beautiful daughter of Leda"
    \item "Menelaus's spouse"
    \item "Paris's lover"
  \end{itemize}

= Helen of Troy

Practical advice

\begin{itemize}
  \item Reference assignment is useful
  \item It's also potentially tricky
  \item As much as possible, leave it to specialized libraries of general data structures
\end{itemize}
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):
  \[ e.copy(a) \]

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

Object and references

States of a reference:

Operations on references:

create \( p \)

\( p := q \) (where \( q \) is attached)
What to do with unreachable objects

- 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

Newsgroup posting by Ian Stephenson, 1993 (as cited in Object-Oriented Software Construction, 2nd edition):

*I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers — if you allocate an object you should be responsible for it, and free it when it's finished with. (Didn't your mother always tell you to put your toys away when you'd finished with them?)*

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:

```plaintext
Paris.display
Line6.extend(Station.Parade_Platz)

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

---

Infix and prefix operators

- In
  
  ```plaintext
  a - b
  ```
  
  the `-` operator is "infix" (written between operands)

- In
  
  ```plaintext
  - b
  ```
  
  the `-` operator is "prefix" (written before the operand)
Operator features

expanded class INTEGER feature

```plaintext
plus a as ** (other INTEGER; INTEGER is
  ... Sum with other
  do ... end
  plus a as ** (other INTEGER; INTEGER is
  ... Product by other
  do ... end
  plus a as ** (other INTEGER is
  ... Unary minus
  do ... end

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

---

End of lecture 11