Lecture 12: More on references and the dynamic model
Purpose of this lecture

Few really new concepts, but gain a better understanding of the tricky matter of references

Reminder on garbage collection and associated concepts
Assignment

Replaces a value by another

\[ p \rightarrow (2, 1) \]

\[
\begin{array}{c|c}
  x & 0 \\
  y & 0 \\
\end{array}
\]

\text{p.set}(2, 1)
Setting fields (in routines of the class)

class
    POSITION
feature - Access
    x: REAL
        -- Horizontal position
    y: REAL
        -- Vertical position
feature - Element change
    set (xval, yval: REAL) is
        -- Set coordinates to (xval, yval).
        require
            x_positive: xval >= 0
            y_positive: yval >= 0
        do
            x := xval
            y := yval
        ensure
            x_set: x = xval
            y_set: y = yval
    end
Effect of an assignment

Reference types: reference assignment

Expanded types: value copy

```v2
class TWO_VALUES feature
  item: INTEGER
  right: TWO_VALUES
  set (n: INTEGER; r: TWO_VALUES) is
    -- Reset both fields
    do
      item := n
      right := r
    end
end
```

```
t: TWO_VALUES
... create t
... t.set (25, Void)
```
Assignment

```plaintext
class METRO_STATION feature
    location: POSITION
    name: STRING
    length: REAL

    set_all (p: POSITION; l: REAL; n: STRING) is
        do
            location := p
            length := l
            name := n
        end
end
```
A linked list of strings: inserting at the end

Haldenegg → Central → Hauptbahnhof

(first_element) → count (4) → last_element

(item) → right

(LINKABLE)

(line)

(item) → right

(LINKABLE)

(line)

(item) → right

(LINKABLE)
Inserting an item at the end

\[
\text{extend} (v: \text{STRING}) \text{ is}
\]
\[
\quad -- \text{Add } v \text{ to end.}
\]
\[
\quad -- \text{Do not move cursor.}
\]
\[
\text{local}
\]
\[
p: \text{LINKABLE [STRING]}
\]
\[
do
\]
\[
\quad \text{create } p.\text{make} (v)
\]
\[
\quad \text{if } \text{is_empty} \text{ then}
\]
\[
\quad \quad \text{first_element} := p
\]
\[
\quad \quad \text{active} := p
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{last_element}.\text{put_right} (p)
\]
\[
\quad \quad \text{if } \text{after} \text{ then active} := p \text{ end}
\]
\[
\quad \text{end}
\]
\[
\quad \text{last_element} := p
\]
\[
\quad \text{count} := \text{count} + 1
\]
\[
\text{end}
\]
Exercise (uses loops)

Reverse a list!

(HALDEN-EGG)

(LINKABLE)

1. Reverse a list using loops.
2. Use appropriate variables like `count`, `last_element`, and `first_element`.
3. Implement the logic to reverse the list.

(HAUPTBAHNHOF)

(LINKED_LIST)

(PARADEPLATZ)
Reversing a list

reverse
local
previous, next: LINKABLE [G]
do
from
next := first; first := Void
invariant
?until next = Void loop
previous := first
first := next
next := next • right
first • put_right (previous)
end
ensure
?end
The key step

Initial items, reversed order

Final items, original order

Loop body:

\[
\begin{align*}
  \text{previous} & := \text{first} \\
  \text{first} & := \text{next} \\
  \text{next} & := \text{next} \cdot \text{right} \\
  \text{first} \cdot \text{put_right} (\text{previous})
\end{align*}
\]
The invariant (informal)

List reachable from \( \textit{first} \)
+ \( \langle \textit{next} \rangle \)
+ List reachable from \( \textit{next}.\textit{right} \)  
  =
  
  \textit{Mirror of original list}

Loop body:
\[
\text{previous} := \textit{first} \\
\textit{first} := \textit{next} \\
\textit{next} := \textit{next}.\textit{right} \\
\textit{first}.\textit{put_right} (\text{previous})
\]
Invariant and postcondition

Exit condition:

\[ \text{next} = \text{Void} \]

Invariant:

List reachable from \( \text{first} \) + <\text{next}\>

List reachable from \( \text{next.right} \) = Mirror of original list

Postcondition:

List is the mirror of original
Reversing a list

```plaintext
reverse
  local
    previous, next: LINKABLE [G]
  do
    from
      next := first ; first := Void
    invariant
      ?
    until next = Void loop
      previous := first
      first := next
      next := next • right
      first • put_right (previous)
    end
  ensure
    ?
end
```
Can we make this formal?

\[ a^x \]

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

\[
\text{first} \cdot \text{right} \times \text{mirror} \\
\text{next} \cdot \text{right} \times \text{tail}
\]

Invariant:
\[
\text{first} \cdot \text{right} \times \text{mirror} + \\
\text{next} \cdot \text{right} \times \text{tail} = \text{old model}
\]
A comfortable mode of reasoning:

-- Here \textit{SOME\_PROPERTY} holds of \textit{a}

"Apply \textit{SOME\_OPERATION} to \textit{b}"

-- Here \textit{SOME\_PROPERTY} still holds of \textit{a}

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

-- Here \textit{P(a)} holds

\textit{OP(b)}

-- Here \textit{P(a)} still holds of \textit{a}
Dynamic aliasing

\(a, b: \text{LINKABLE [STRING]}\)

create \(a\)....

\(b := a\)

\(a\. put ("Haldenegg")\)

-- Here \(a\. item\) has value "Haldenegg"

\(b\. put ("Paradeplatz")\)

-- Here \(a\. item\) has value ??????
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”
- “Menelas’s spouse”
- “Paris’s lover”

= Helen of Troy
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):
\[ e.copy(a) \]
Shallow and deep cloning

Initial situation:

Result of:

\[ b := a \]

\[ c := a.twin \]

\[ d := a.deep_twin \]
Where do these mechanisms come from?

Class \textit{ANY} in the Eiffel “Kernel Library”

Every class that doesn’t explicitly inherit from another is considered to inherit from \textit{ANY}

As a result, every class is a descendant of \textit{ANY}. 
Completing the inheritance structure

A

B

C

D

E

ANY

NONE

Inherits from
A related mechanism: Persistence

\[
a.\text{store (file)}
\]

\[
\text{...}
\]

\[
b \equiv \text{retrieved (file)}
\]

Storage is automatic.
Persistent objects identified individually by keys.

These features come from the library class \textit{STORABLE}.
Objects and references

States of a reference:

- **VOID**
- **ATTACHED**

Operations on references:

- create \( p \)
- \( p := q \) (where \( q \) is attached)
- \( p := \text{Void} \)
- \( p := q \) (where \( q \) is void)

if \( p = \text{Void} \) then ...
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)
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 its 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

\[ \text{some_target.some_feature(some_arguments)} \]

For example:

\[ \text{Paris.display} \]
\[ \text{Line6.extend(Station_Parade_Platz)} \]

\[ x := a.plus(b) \]
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

expanded class INTEGER feature

\textbf{plus} alias "+" (other: INTEGER): INTEGER is
\begin{verbatim}
  -- Sum with other
  do ... end
\end{verbatim}

\textbf{times} alias "*" (other: INTEGER): INTEGER is
\begin{verbatim}
  -- Product by other
  do ... end
\end{verbatim}

\textbf{minus} alias "-" : INTEGER is
\begin{verbatim}
  -- Unary minus
  do ... end
\end{verbatim}

\begin{verbatim}
...
\end{verbatim}

end

Calls such as \texttt{i.plus (j)} can now be written \texttt{i + j}
What we have seen

- Playing with references:
  - list reversal
- Dynamic aliasing and the difficulties of pointers & references
- Overall inheritance structure
- Copy, clone and storage operations
- Persistence closure
- Infix & prefix operators
End of lecture 12