Programming in the large

Bertrand Meyer

Lecture 14: Agents

Behind agents: tuples

- Purposes:
  - Allow manipulation of sequences of values with arbitrary number of elements, with simple structure
  - Support "anonymous classes"
  - To allow for function that return multiple results
  - Permit type-safe agents

Tuple classes

- Syntax: \text{TUPLE} [X, Y, ...]

Mathematical model for tuples

- First intuition: \text{TUPLE} [A, B, C] represents the cartesian product A x B x C
- But: A x B x C cannot be mapped to a subset of A x B!
- Better model:
  - \text{TUPLE} represents the set of partial functions from \text{N} (set of integers) to the set of possible values, whose domain includes the interval [1 .. n] for some n.
  - Example of such a function: \{<1, "a">, <2, "a">, <3, "a">\}
  - An element of \text{TUPLE} [A, B, C] is a function whose domain includes the interval [1 .. 3])
  - So it’s also an element of \text{TUPLE} [A, B]: functions whose domain includes interval [1 .. 2].

Reminder: constrained genericity

- \text{LIST} [G] (unconstrained): G represents arbitrary type. May use
  \text{LIST} [\text{INTEGER}]
  \text{LIST} [\text{EMPLOYEE}]
  \text{LIST} [\text{SHIP}]
- \text{SORTABLE\_LIST} [G \rightarrow \text{COMPARABLE}] (constrained by \text{COMPARABLE})
- G represents type descending from \text{COMPARABLE}
  \text{LIST} [\text{INTEGER}]
  \text{LIST} [T] only if T descendant of \text{COMPARABLE}
Agent types: Kernel library classes

- **ROUTINE**
  - [BASE, ARGS $\rightarrow$ TUPLE]
  - Inherits from BASE

- **PROCEDURE**
  - [BASE, ARGS $\rightarrow$ TUPLE]

- **FUNCTION**
  - [BASE, ARGS $\rightarrow$ TUPLE, RES]

Features of routine classes

- **call** (values: ARGS)
- **item** (values: ARG, RES)

  In FUNCTION only

  - ... target, arguments, set_target, set_arguments...

- Introspection features (in progress):
  - **precondition**: FUNCTION [BASE, ARGUMENTS, BOOLEAN]
  - **postcondition**: type: TYPE

Features of class TYPE: heirs, parents, routines etc.

In some other class:

- Reminder:
  - PROCEDURE [BASE, ARGS $\rightarrow$ TUPLE]

  - $f(x1: T1; x2: T2; x3: T3)$
    - do
      - end
    - $u := \text{agent} f(a1, a2, a3)$

- In some other class:
  - $a0: C$
    - $v := \text{agent} a0.f(a1, a2, a3)$
    - Type of both $u$ and $v$:
      - $u, v: \text{PROCEDURE} [C, TUPLE]$

Agent types

- $\text{agent} a0.f (a1, a2, a3)\text{PROCEDURE} [C, TUPLE]$
- $\text{agent} a0.f (a1, a2, ?)\text{PROCEDURE} [C, TUPLE]$ (as before)
- $\text{agent} a0.f (a1, ?, a3)\text{PROCEDURE} [C, TUPLE]$ (as before)
- $\text{agent} a0.f (?, a2, a3)\text{PROCEDURE} [C, TUPLE]$ (as before)

Keeping arguments open

- An agent can have both "closed" and "open" arguments.
- Closed arguments set at time of agent definition; open arguments set at time of each call.
- To keep an argument open, just replace it by a question mark:
  - $u := \text{agent} a0.f (a1, a2, a3)$
    - $\text{-- All closed (as before)}$
  - $w := \text{agent} a0.f (a1, a2, ?)$
  - $x := \text{agent} a0.f (a1, ?, a3)$
  - $y := \text{agent} a0.f (a1, ?, ?)$
  - $z := \text{agent} a0.f (?, ?, ?)$

Calling an agent

- $a0: C; a1: T1; a2: T2; a3: T3$
- $u := \text{agent} a0.f (a1, a2, a3)\text{PROCEDURE} [C, TUPLE]$
  - $u.call([])$
- $v := \text{agent} a0.f (a1, a2, ?)\text{PROCEDURE} [C, TUPLE]$ (as before)
  - $v.call([a3])$
- $w := \text{agent} a0.f (a1, ?, a3)\text{PROCEDURE} [C, TUPLE]$ (as before)
  - $w.call([a2])$
- $x := \text{agent} a0.f (a1, ?, ?)\text{PROCEDURE} [C, TUPLE]$ (as before)
  - $x.call([a2, a3])$
- $y := \text{agent} a0.f (?, ?, ?)\text{PROCEDURE} [C, TUPLE]$ (as before)
  - $y.call([a1, a2, a3])$
Keeping the target open

\[ r := \text{agent}(T0).f(a1, a2, a3) \]
--- Target open, arguments closed

Type is: \( \text{PROCEDURE}[T0, \text{TUPLE}[T0]] \)
Example call: \( r.\text{call}([a0]) \)

\[ s := \text{agent}(T0).f(? , ? , ?) \]
--- Open on all operands
--- Can also be written as just:
\( \text{agent}(T0).f \)

Type is: \( \text{PROCEDURE}[T0, \text{TUPLE}[T0, T1, T2, T3]] \)
Example call: \( s.\text{call}([a0, a1, a2, a3]) \)

The integral function

\[ \text{integral}(f: \text{FUNCTION}[\text{ANY}, \text{TUPLE}[\text{REAL}], \text{REAL}]; \]
\[ \text{low, high}: \text{REAL}) \text{ is} \]
--- Integral of \( f \) over the interval \([\text{low}, \text{high}]\)

local
\[ x: \text{REAL} \]
\[ i: \text{INTEGER} \]
do
from \( x := \text{low} \) until \( x > \text{high} \)
loop
\[ \text{Result} := \text{Result} + f.\text{item}([x]) \times \text{step} \]
\[ i := i + 1 \]
\[ x := x + i \times \text{step} \]
end
end

Calling an agent: integration example

\[ b \]
\[ \int my\_function(x) \, dx \]
\[ a \]
\[ b \]
\[ \int your\_function(x, u, v) \, dx \]
\[ a \]

\( my\_integrator.\text{integral}(\text{agent my\_function}, a, b) \)
\( my\_integrator.\text{integral}(\text{agent your\_function}(?, u, v), a, b) \)

Calling an agent: iterator

all_positive :=
\( my\_integer\_list.\text{for\_all}(\text{agent is\_positive}()) \)
all_married :=
\( my\_employee\_list.\text{for\_all}(\text{agent EMPLOYEE}.\text{is\_married}) \)

Iterators

- In class \( \text{LINEAR}[G] \), ancestor to all classes representing lists, sequences etc.

\[ \text{for\_all(test: \text{FUNCTION}[\text{ANY}, \text{TUPLE}[G], \text{BOOLEAN}]) is} \]
--- Is there no item in structure that doesn't
--- satisfy test?

do
from \( \text{start} \) until \( \text{off or not Result} \)
loop
\[ \text{Result} := \text{test.\text{item}([item])} \]
forth
end

Iterators (cont’d)

\[ \text{for\_all} \]
\[ \text{there\_exists} \]
\[ \text{do\_all} \]
\[ \text{do\_if} \]
\[ \text{do\_while} \]
\[ \text{do\_until} \]
Command classes

- Undo-redo design pattern
- Support for undo, redo
- Class `COMMAND` provides a procedure `execute` and its heir `UNDOABLE_COMMAND` adds `undo`.
- Write a new class for every kind of undoable command.

Command classes

- Traditionally: one command class (descendant of `COMMAND`) for every kind of command.
- Now, can transform any procedure into command:
  ```
  operation: PROCEDURE [CONTEXT, TUPLE]
  -- Operation to be applied by current command
  make (p: like operation) is
    -- Make p the current command's operation.
    require
      p_not_void: p /= Void
    do
      operation := p
    ensure
      operation_set: operation = p
  end
  ```

Complementary material

- Eiffel: The Language, 3rd edition (draft), chapter 25
  → Available online at
  (User name: Talkitover; password: etl3)
- Paper on Event-driven programming
  → Available online at
  http://www.inf.ethz.ch/~meyer/ongoing/events.pdf

End of lecture 14