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: \textit{TUPLE} \([X, Y, ...]\)
Mathematical model for tuples

- First intuition: 
  \[ \text{TUPLE} [A, B, C] \] represents the cartesian product \( A \times B \times C \)

- But: \( A \times B \times C \) cannot be mapped to a subset of \( A \times B \)!

- Better model:
  - \( \text{TUPLE} \) represents the set of partial functions from \( \mathbb{N} \) (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

- **LIST** \([G]\) (unconstrained): \(G\) represents arbitrary type. May use
  
  \[
  \text{LIST [INTEGER]} \\
  \text{LIST [EMPLOYEE]} \\
  \text{LIST [SHIP]}
  \]

- **SORTABLE_LIST** \([G \rightarrow COMPARABLE]\)
  (constrained by **COMPARABLE**)

- \(G\) represents type descending from **COMPARABLE**
  
  \[
  \text{LIST [INTEGER]} \\
  \text{LIST [T]} \text{ only if } T \text{ descendant of **COMPARABLE}}
  \]
Agent types: Kernel library classes

- **AGENT**
  - ROUTINE*: $[\text{BASE, ARGS} \rightarrow \text{TUPLE}]$
  - PROCEDURE*: $[\text{BASE, ARGS} \rightarrow \text{TUPLE}]$
  - FUNCTION*: $[\text{BASE, ARGS} \rightarrow \text{TUPLE, RES}]$

Inherits from: Deferred

Call:

Item:
Features of routine classes

*call* (values: *ARGS*)

*item* (values: *ARGS*): *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.
Type of closed agent expression

- In class $C$:

  $$f (x_1: T_1; x_2: T_2; x_3: T_3) \text{ is }
  \begin{array}{c}
  \text{do} \\
  \vdots \\
  \text{end}
  \end{array}
  $$

  $$u := \text{agent } f (a_1, a_2, a_3)$$

- In some other class:

  $$a_0 : C$$

  $$v := \text{agent } a_0.f (a_1, a_2, a_3)$$

- Type of both $u$ and $v$:

  $$u, v: \text{PROCEDURE [C, TUPLE]}$$
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:

\[
\begin{align*}
  u & := \text{agent } a0.f(\text{a1, a2, a3}) \\
       & \quad \text{-- All closed (as before)} \\
  w & := \text{agent } a0.f(\text{a1, a2, ?}) \\
  x & := \text{agent } a0.f(\text{a1, ?, a3}) \\
  y & := \text{agent } a0.f(\text{a1, ?, ?}) \\
  z & := \text{agent } a0.f(\text{?, ?, ?})
\end{align*}
\]
Agent types

- Reminder:

\[
\textit{PROCEDURE} \ [\text{BASE, ARGS } \rightarrow \text{TUPLE}]
\]

\(f (x1: T1; x2: T2; x3: T3) \text{ is}
\]

\(\quad \text{-- in class C}
\]

\(\text{do}
\]

\(\quad \ldots
\]

\(\text{end}
\]

\textbf{agent} \(a0.f (a1, a2, a3) \quad \text{PROCEDURE} \ [C, \text{TUPLE}]
\]

\textbf{agent} \(a0.f (a1, a2, ?) \quad \text{PROCEDURE} \ [C, \text{TUPLE} [T3]]
\]

\textbf{agent} \(a0.f (a1, ?, a3) \quad \text{PROCEDURE} \ [C, \text{TUPLE} [T2]]
\]

\textbf{agent} \(a0.f (a1, ?, ?) \quad \text{PROCEDURE} \ [C, \text{TUPLE} [T2, T3]]
\]

\textbf{agent} \(a0.f (?, ?, ?) \quad \text{PROCEDURE} \ [C, \text{TUPLE} [T1, T2, T3]]
\]
Calling an agent

\(a_0: C; a_1: T_1; a_2: T_2; a_3: T_3\)

\[\begin{align*}
u & := \text{agent } a_0.f(a_1, a_2, a_3) & \text{PROCEDURE } [C, \text{TUPLE}] \\
\text{u.call } & [[]] & \\
v & := \text{agent } a_0.f(a_1, a_2, ?) & \text{PROCEDURE } [C, \text{TUPLE }[T_3]] \\
\text{v.call } & [[a_3]] & \\
w & := \text{agent } a_0.f(a_1, ?, a_3) & \text{PROCEDURE } [C, \text{TUPLE }[T_2]] \\
\text{w.call } & [[a_2]] & \\
x & := \text{agent } a_0.f(a_1, ?, ?) & \text{PROCEDURE } [C, \text{TUPLE }[T_2, T_3]] \\
\text{x.call } & [[a_2, a_3]] & \\
y & := \text{agent } a_0.f(?, ?, ?) & \text{PROCEDURE } [C, \text{TUPLE }[T_1, T_2, T_3]] \\
\text{y.call } & [[a_1, a_2, a_3]] &
\end{align*}\]
Keeping the target open

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

Type is: \( \text{PROCEDURE} [T0, \text{TUPLE} [T0]] \)

Example call: \( r.call ([a0]) \)

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

Type is: \( \text{PROCEDURE} [T0, \text{TUPLE} [T0, T1, T2, T3]] \)

Example call: \( s.call ([a0, a1, a2, a3]) \)
Calling an agent: integration example

\[ \int_{a}^{b} my\_function(x) \, dx \]

\[ \int_{a}^{b} your\_function(x, u, v) \, dx \]

my_integrator.integral(agent my_function, a, b)

my_integrator.integral(agent your_function (?, u, v), a, b)
The integral function

\[
\text{integral } (f: \text{FUNCTION [ANY, TUPLE [REAL], REAL]};
\text{low, high: REAL}: \text{REAL is}
\]

\[-- \text{ Integral of } f \text{ over the interval } [\text{low, high}]\]

\[
\text{local}
\]

\[
x: \text{REAL} \\
\text{i: INTEGER}
\]

\[
do
\]

\[\text{from } x := \text{low until } x > \text{high loop}
\]

\[
\text{Result} := \text{Result} + f.\text{item} ([x]) \ast \text{step}
\]

\[
i := i + 1
\]

\[
x := x + i \ast \text{step}
\]

\[
end
\]

\[
edo
\]

\[
\text{end}
\]

\[
\]
Calling an agent: iterator

\[
\text{all\_positive} := \\
\text{my\_integer\_list.\texttt{for\_all}} \text{ (agent is\_positive (?))}
\]

\[
\text{all\_married} := \\
\text{my\_employee\_list.\texttt{for\_all}} \text{ (agent \{EMPLOYEE\}.is\_married)}
\]


- In class `LINEAR [G]`, ancestor to all classes representing lists, sequences etc.

```
for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]) is
  -- Is there no item in structure that doesn't
  -- satisfy test?
  do
    from
      start
      Result := True
    until
      off or not Result
    loop
      Result := test.item ([item])
    forth
  end
end
```
Iterators (cont’d)

for_all
there_exists
do_all
do_if
do_while
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:

  \[
  \text{operation}: \text{PROCEDURE}[\text{CONTEXT}, \text{TUPLE}]
  \]
  -- Operation to be applied by current command

  \[
  \text{make } (p: \text{like operation}) \text{ is}
  \]
  -- Make \( p \) the current command’s operation.

  \[
  \text{require}
  \]
  \[
  p\text{-not-void}: p \neq \text{Void}
  \]

  \[
  \text{do}
  \]
  \[
  \text{operation} := p
  \]

  \[
  \text{ensure}
  \]
  \[
  \text{operation\_set}: \text{operation} = p
  \]

end
Complementary material

- **Eiffel: The Language, 3rd edition (draft), chapter 25**
    (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