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: `TUPLE [X, Y, ...]`

Mathematical model for tuples

- First intuition:
  `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:
  - `TUPLE` represents the set of partial functions from \(\mathbb{N}\) (set of integers) to the set of possible values, whose domain includes the interval \([1 \ldots n]\) for some `n`.
  - Example of such a function:
    \(<1, "a">, <2, "a">, <3, "a">\>
  - An element of `TUPLE [A, B, C]` is a function whose domain includes the interval \([1 \ldots 3]\))
  - So it's also an element of `TUPLE [A, B]`: functions whose domain includes interval \([1 \ldots 2]\).

Reminder: constrained genericity

- `LIST [G]` (unconstrained): `G` represents arbitrary type. May use:
  - `LIST [INTEGER]`
  - `LIST [EMPLOYEE]`
  - `LIST [SHIP]`

- `SORTABLE_LIST [G => COMPARABLE]` (constrained by `COMPARABLE`)

- `G` represents type descending from `COMPARABLE`
  - `LIST [INTEGER]`
  - `LIST [T]` only if `T` descendant of `COMPARABLE`
Agent types: Kernel library classes

- **ROUTINE**
  - `BASE, ARGS -> TUPLE`
  - `call`
  - Inherits from `Deferred`

- **PROCEDURE**
  - `BASE, ARGS -> TUPLE`

- **FUNCTION**
  - `BASE, ARGS -> TUPLE, RES`

Features of routine classes

- `call (values: ARGS)`
- `item (values: ARGS): RES`
- `... 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 (x1: T1; x2: T2; x3: T3) is
  do
  ... end
  u := agent f (a1, a2, a3)
  ```
- In some other class:
  ```
  a0: C
  v := agent a0.f (a1, a2, a3)
  ```
- Type of both `u` and `v`:
  ```
  u, v: 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:

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

Agent types

- Reminder:

\[
\begin{align*}
    \text{PROCEDURE [BASE, ARGS -> TUPLE]} & \\
    f(x1: T1; x2: T2; x3: T3) & \text{is} \\
    \begin{cases}
        \text{do} & \\
        \text{end} & \text{...}
    \end{cases} \\
    \text{agent } a0.f(a1, a2, a3) & \text{PROCEDURE [C, TUPLE]} \quad \text{-- All closed} \\
    \text{agent } a0.f(a1, a2, ?) & \text{PROCEDURE [C, TUPLE [T3]]} \\
    \text{agent } a0.f(a1, ?, a3) & \text{PROCEDURE [C, TUPLE [T2]]} \\
    \text{agent } a0.f(a1, ?, ?) & \text{PROCEDURE [C, TUPLE [T2, T3]]} \\
    \text{PROCEDURE [C, TUPLE [T1, T2, T3]]}
\end{align*}
\]

Calling an agent

\[
\begin{align*}
    a0: C; a1: T1; a2: T2; a3: T3 \\
    u := \text{agent } a0.f(a1, a2, a3) & \text{PROCEDURE [C, TUPLE]} \\
    v := \text{agent } a0.f(a1, a2, ?) & \text{PROCEDURE [C, TUPLE [T3]]} \\
    w := \text{agent } a0.f(a1, ?, a3) & \text{PROCEDURE [C, TUPLE [T2]]} \\
    x := \text{agent } a0.f(a1, ?, ?) & \text{PROCEDURE [C, TUPLE [T2, T3]]} \\
    y := \text{agent } a0.f(? , ?, ?) & \text{PROCEDURE [C, TUPLE [T1, T2, T3]]} \\
    \text{PROCEDURE [C, TUPLE [T1, T2, T3]]}
\end{align*}
\]
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: \( s.\text{call}(\{a0, a1, a2, a3\}) \)

Calling an agent: integration example

\[
\int_a^b \text{my\_function}(x) \, dx \\
\int_a^b \text{your\_function}(x, u, v) \, dx
\]

\( \text{my\_integrator}.\text{integral}(\text{agent my\_function}, a, b) \)

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

The integral function

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

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

\[
\text{local} \ x: \text{REAL} \\
\text{i: INTEGER} \\
\text{do} \ from \ x := \text{low} \ \text{until} \ x \geq \text{high} \ \text{loop} \\
\quad \text{Result} := \text{Result} + f.\text{item}(\{x\}) \ast \text{step} \\
\quad i := i + 1 \ast \text{step} \\
\quad x := x + i \ast \text{step} \\
\text{end} \\
\text{end}
\]
### Calling an agent: iterator

```
all_positive :=
  my_integer_list.for_all (agent is_positive ?)

all_married :=
  my_employee_list.for_all (agent {EMPLOYEE}.is_married)
```

### Iterators

- 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
    until off or not Result
    loop
      Result := test.item ([item])
      forth
  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:

```plaintext
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