Tuples and Agents

Christian Estler
ETH Zurich
christian.estler@inf.ethz.ch

Distributed and Outsourced Software Engineering - ETH course, Fall 2012
Motivation for Tuples

Imagine the following scenario:

Need to store click-coordinates on a chess-board

**letter**: value of a .. h
**number**: value of 1 .. 8

We want to store a coordinate as a single object.
Motivation for Tuples

Default approach to storing coordinates → write a small class

class COORDINATE

create make

feature {NONE} -- Initialization

make (a_letter: CHARACTER; a_number: INTEGER)
  -- Creation procedure
  do
    letter := a_letter
    number := a_number
  end

feature {ANY} -- Attributes

  letter: CHARACTER
  number: INTEGER

invariant
  number_valid: number >= 1 and number <= 8
  letter_valid: letter >= 'a' and letter <= 'h'
end
Tuples-Motivation

Writing a full fledged class can feel “too heavy”

Eiffel offers an alternative with TUPLE

TUPLE is not a real class, but is a type that represents an infinite number of classes

TUPLE can have an arbitrary number of generic arguments, e.g.

TUPLE
TUPLE [A]
TUPLE [A, B]
TUPLE [A, B, C]

...
Using a tuple to store chess-board coordinates

foo
local
  coord: TUPLE [CHARACTER, INTEGER]
do
  coord := ['a', 1] -- direct assignment
  -- an assignment using create
  create coord
  coord.put ('a', 1)
  coord.put (1, 2)
end

value, index

Type of value is checked at runtime, not compile-time; could put anything
Tuples and Labels

A tuple can also have labels (easier to access that way)

TUPLE [author: STRING; year: INTEGER; title: STRING]

A labeled tuple type denotes the same type as its unlabeled form, here

TUPLE [STRING, INTEGER, STRING]

but facilitates access to individual elements

Denoting a particular tuple (labeled or not) remains the same:

[“Tolstoi”, 1865, “War and Peace”]

To access tuple elements: use e.g. \( t.\text{year} \)
Tuples and Inheritance

Inheritance structure

**Generic types A, A’ must conform to each other, otherwise no subtype relationship**

**Remember conforms:**

Y conforms to X if Y inherits from X
tuple_conformance

local
t0: TUPLE
t2: TUPLE [INTEGER, INTEGER]
do
create t2
t2 := [10, 20]
t0 := t2
print (t0.item (1).out + "%N")
print (t0.item (3).out)
end

Not necessary in this case
Implicit creation
Runtime error, but will compile
Agents
Assignment in Eiffel (other languages)

\[ x: \text{MY\_CLASS} \]

\[ \text{-- declaration of } x \]

\[ \ldots \]

\[ x := \text{create MY\_CLASS.make} \]

\[ \text{-- assigning a value to } x \]

\( x \) is a reference to an object of type \( \text{MY\_CLASS} \)
Motivation for Agents

By default

- OO-design encapsulates **data** into objects
- Operations are **not** treated as objects

\[ r := \text{my\_operation} \]

\[
\text{-- assigning an operation to } r
\]

But, sometimes we would like to represent operations as objects

- Could include operations in object structures (e.g. LIST)
- Traverse the structure at some later point
- Execute the operations

Concrete examples → next slide
Motivation for Agents

Examples where we could use operations as objects

• GUI programming
  • Event occurs, e.g. a mouse click on some button
  • Button holds a reference to an operation object that shall be executed

• Iteration on data structures
  • Introduce general-purpose routine \texttt{do\_all} that applies an arbitrary operation to all elements of the structure
  • Can provide operation object to routine \texttt{do\_all}
Eiffel supports such operation objects, they are called **Agents**

Same concept in other languages:

- C and C++: “function pointers”
- C#: delegates
- Functional languages: closures
Creating an Agent

Given a routine

```pascal
my_printer (i, j, k: INTEGER)
-- this is a printing routine
do
   print("Value of i: " + i.out + "%N");
   print("Value of j: " + j.out + "%N");
   print("Value of k: " + k.out + "%N");
end
```

we can create an operation object for my_printer as follows

```pascal
r := agent my_printer(?,?,?)
```

But what’s the type of r???

agent keyword wraps operation into an object

Routine expects 3 arguments which we don’t know yet
An Agent’s Type

An agent creates an object (that wraps an operation)

\[ r := \text{agent} \ my\_printer \ (?,?,?,?) \]

What is the type of that object?

- Either the object represents a **PROCEDURE** or
- The object represents a **FUNCTION**

Thus, the type of \( r \) would be **PROCEDURE**

\[ r: \text{PROCEDURE} \ [\text{ANY}, \text{TUPLE}[[\text{INTEGER}, \text{INTEGER}, \text{INTEGER}]]) \]

Let’s have a closer look what those generic arguments are…

Official terminology is “agent definition” but you can think of it as a **create** for operation objects
An Agent’s Type

Given an agent declaration for a procedure

\[ r: \text{PROCEDURE} \ [\text{ANY, TUPLE}[\text{INTEGER, INTEGER, INTEGER}]] \]

1\textsuperscript{st} argument represents the class (type) to which \( r \) belong

In practice, we always put \text{ANY}, as every class is of type \text{ANY}

2\textsuperscript{nd} argument represents the type of the arguments of \( r \)
class
  AGENT_DEMO

feature

r: PROCEDURE [ANY, TUPLE[INTEGER, INTEGER, INTEGER]]
   -- declaration of the agent

foo
   -- some routine, where the agent is created
   do
      r := agent my_printer (?,?,?,?)
   end

my_printer (i, j, k: INTEGER)
   -- this is a printing routine
   do
      print("Value of i: " + i.out + "%N");
      print("Value of j: " + j.out + "%N");
      print("Value of k: " + k.out + "%N");
   end
end
More on Agent Types

How to declare an agent for a Function rather than a Procedure?

- Type of an agent for a procedure (we’ve already seen)
  
  \[
  \text{PROCEDURE } [T, \text{ARGS}]
  \]

- Type of an agent for a function
  
  \[
  \text{FUNCTION } [T, \text{ARGS}, \text{RES}]
  \]
Agent for a Function

class
AGENT_FUNCTION_DEMO

feature

f: FUNCTION [ANY, TUPLE[INTEGER], INTEGER]
   -- declaration of the agent

foo
   -- some routine, where the agent is created
   do
      f := agent square (?)
   end

square (a_number: INTEGER): INTEGER
   -- this returns the square of `a_number`
   do
      Result := a_number * a_number
   end
end
Executing an Agent

So far, we’ve declared and created agents.

How about running them?

✓ If `a` represents a **procedure**, `a.call (argument_tuple)`
calls the procedure

✓ If `a` represents a **function**, `a.item (argument_tuple)`
calls the function and returns its result

Notice the brackets; we provide a TUPLE
class
    AGENT_DEMO

feature

r: PROCEDURE [ANY, TUPLE[INTEGER, INTEGER, INTEGER]]
    -- declaration of the agent

foo
    -- some routine, where the agent is created
    do
        r := agent my_printer (?,?,?)
        r.call ([1, 2, 3])
    end

my_printer (i, j, k: INTEGER)
    -- this is a printing routine
    do
        print("Value of i: " + i.out + "%N");
        print("Value of j: " + j.out + "%N");
        print("Value of k: " + k.out + "%N");
    end
class
AGENT_FUNCTION_DEMO

feature

f: FUNCTION [ANY, TUPLE[INTEGER], INTEGER]
   -- declaration of the agent

foo
   -- some routine, where the agent is created
   do
      f := agent square (?)
      print ((f.item ([3])).out)
   end

square (a_number: INTEGER): INTEGER
   -- this returns the square of `a_number'
   do
      Result := a_number * a_number
   end
end
Classes representing agents

Instead of using `item`, we can use `call` and get the last result using `last_result`
Open and Closed Agent Arguments

Up to now, we have provided all arguments once we call the agent

```plaintext
r := agent my_printer (?,?,?)
r.call ([1, 2, 3])
```

What if we’d like to **fix** the arguments at the time we create the agent? We can do that:

```plaintext
r := agent my_printer (1,2,3)
r.call ([][])
```
Open and Closed Agent Arguments

Closed arguments are set at agent definition time. Open arguments are set at agent call time.

We can also mix open and closed arguments

\[
\begin{align*}
  u & := \text{agent } a_0.f (a_1, a_2, a_3) \quad \text{-- All closed} \\
  w & := \text{agent } a_0.f (a_1, a_2, ?) \\
  x & := \text{agent } a_0.f (a_1, ?, a_3) \\
  y & := \text{agent } a_0.f (a_1, ?, ?) \\
  z & := \text{agent } a_0.f (?, ?, ?) \quad \text{-- All open}
\end{align*}
\]
The agent’s type must reflect the number of open arguments

Example 1:

```pascal
r: PROCEDURE [ANY, TUPLE[INTEGER, INTEGER, INTEGER]]
r := agent my_printer (?,?,?)
r.call ([1, 2, 3])
```

Example 2:

```pascal
r: PROCEDURE [ANY, TUPLE[INTEGER]]
r := agent my_printer (1,2,?)
r.call ([3])
```
Agents with open Target

All examples seen so far were based on routines of the enclosing class. This is not required.

```plaintext
class APPLICATION

feature

printer: AGENTPROCEDURE -- class from previous slide
my_agent: PROCEDURE [ANY, TUPLE[INTEGER]]

foo
   -- some routine, where the agent is created
do
   create printer
   my_agent := agent printer.my_printer (1, ?, 3)
   my_agent.call ([2])
end
end
```

Calls my_printer of object printer
Inline Agents

So far, we assumed that there already exists some routine that we wish to represent with an agent.

Sometimes the only usage of such a routine could be as an agent. We can use **inline agents**, i.e. write a routine in the agent declaration:

```plaintext
demo_list.do_all (agent (i: INTEGER))
    do
        print ("Value: "+i.out+"%N")
    end
```

10/22/2012
Doing something to a list

Given a simple ARRAY [G] class, with only the features `count' and `at', implement a feature which will take an agent and perform it on every element of the array.

\[
\text{do\_all} \ (\text{do\_this} : \text{PROCEDURE}[\text{ANY}, \text{TUPLE}[G]]) \\
\text{local} \\
\quad i : \text{INTEGER} \\
\text{do} \\
\quad \text{do} \\
\quad \text{from} \\
\quad \quad i := 1 \\
\quad \text{until} \\
\quad \quad i > \text{count} \\
\quad \text{loop} \\
\quad \quad \text{do\_this}\text{.call} ([\text{at} (i)]) \\
\quad \quad i := i + 1 \\
\quad \text{end} \\
\text{end}
\]
for_all (pred: PREDICATE [ANY, TUPLE[G]]): BOOLEAN
local
   i: INTEGER
do
   Result := True
from
   i := 1
until
   i > count or not Result
loop
   Result := pred.item ([at (i)])
i := i + 1
end
end