Concurrent Object-Oriented Programming

Arnaud Bally, Bertrand Meyer and Volkan Arslan


Formal Logics

- Have a syntax defining (legal) terms.
- Allow reasoning by providing:
  - Axioms defining equality on terms, and/or
  - A semantics defining the meaning of terms accompanied by a semantic equivalence.
- Soundness/Completeness properties may relate the two approaches.

Example: Integer Calculus

- Syntax in (Enhanced) Backus-Naur Form.
- Digit ::= 0 | 1 | ... | 9
- Integer ::= <Digit> | <Integer><Digit>
- Term ::= <Term> + <Integer>
  | <Term> - <Integer>
  | <Term> * <Integer>
- Equality is symmetric, transitive, reflexive.
- Operators can be associative, commutative, distributive.
- Allows axiomatic reasoning, by tautology.
- How to define "reasonable" equality?

Formalisms are Useful for...

- Understanding (precise definitions).
- Documentation.
- Static analysis:
  - protocol verification,
  - code optimization (e.g. partial evaluation),
  - code validation.
- Code generation.
- Test case generation.
- Execution monitoring.

Functions and Functional Programming

- Mathematical function:
  - taking (positional) parameters \( x_1, x_2, \ldots, x_n \),
  - potentially calling other functions,
  - returning a result.
- The result depends only on \( x_1, x_2, \ldots, x_n \).
- Calls with same parameters yield same result.
- A function defines an input-output relation.
- Functional programming:
  - Functions are 1st-class entities,
  - Extensional equivalence with termination.
  - Programs are more concrete than functions!
**Lambda-Calculus [Church 1934]**

- Calculus of (sequential) functional programs.
  \[ t := x | \lambda x.t | tt \]
- Two constructions (operators):
  - lambda-abstraction,
  - function application.
- Infinite but enumerable set of variables.
- Lambda acts as a binding operator.
- The scope of a lambda abstraction extends as far to the right as possible.

**Syntactical Aspects (1)**

- Occurrences of a variable under a lambda are **bound**.
- Other occurrences are **free**.
  \[ fv(x) \triangleq \{ x \} \]
  \[ bv(x) \triangleq \emptyset \]
  \[ fv(u) \triangleq fv(t) \cup fv(u) \]
  \[ bv(u) \triangleq bv(t) \cup bv(u) \]
  \[ bv(\lambda x.t) \triangleq bv(fv(t) \setminus \{ x \}) \]

**Syntactical Aspects (2)**

- The substitution of \( x \) by \( y \) in \( t \) is noted \( t[y/x] \):
  \[ x[t/x] \triangleq t \]
  \[ z[t/x] \triangleq z \text{ if } z \neq x \]
  \( (uv)[t/x] \triangleq u[t/x]v[t/x] \)
  \( (\lambda y.u)[t/x] \triangleq \lambda y.u[t/x] \)
- **Alpha-conversion** (renaming) is defined as:
  \( \lambda x.t \equiv_{\alpha} \lambda y.t[y/x] \text{ if } y \notin fv(t) \)
- We shall assume from now that all terms are considered **modulo alpha-conversion**.

**Operational Semantics**

- Given as a set of **rewriting rules**.
- Only 1 rule, **Beta-reduction**:
  \( (\lambda x.t) u \rightarrow t[u/x] \)
- Defines a **transition relation**.
- Repeatedly convert and reduce a term to have its semantics \( \rightarrow \), given by **transitive closure**.
- A reducible part of a term is called a **redex**.
- Functional application associates to the left.
- **Values are non-reducible** terms (terminated computations).

**Encoding Booleans (1)**

- **true** \( \triangleq \lambda t. \lambda f. t \)
- **false** \( \triangleq \lambda t. \lambda f. f \)
  \( true \ v \ w \equiv (\lambda t. \lambda f. t) \ v \ w \)
  \( \rightarrow (\lambda f. v) \ w \)
  \( \rightarrow v \)
  \( false \ v \ w \equiv (\lambda t. \lambda f. f) \ v \ w \)
  \( \rightarrow (\lambda f. f) \ w \)
  \( \rightarrow w \)

**Encoding Booleans (2)**

- **not** \( \triangleq \lambda b. b \) **false** true
- **and** \( \triangleq \lambda b. \lambda c. b \ c \) **false**
- **or** \( \triangleq \lambda b. \lambda c. b \) **true** c
- **if** \( \triangleq \lambda b. \lambda t. \lambda f. b \ t \ f \)
Encoding Pairs (or lists, records...)

- A pair encloses two items \( f \) and \( s \).
- There exist two projection functions \( \pi_1 \) and \( \pi_2 \).
- \( \pi_1 \): \( \text{pair} \rightarrow f \) and \( \pi_2 \): \( \text{pair} \rightarrow s \).
- \( \text{pair}(\pi_1 p, \pi_2 p) \rightarrow p \) where \( p \equiv \text{pair} f s \)

\[
\begin{align*}
\text{pair} & \triangleq \lambda f.\lambda s.\lambda b. b f s \\
\pi_1 & \triangleq \lambda p. p \text{ true} \\
\pi_2 & \triangleq \lambda p. p \text{ false}
\end{align*}
\]

Encoding Naturals (1)

A natural \( n \) just applies \( n \) times a given function \( s \) to an argument \( z \)

\[
\begin{align*}
0 & \equiv \lambda s. \lambda z. z \\
1 & \equiv \lambda s. \lambda z. s \ z \\
2 & \equiv \lambda s. \lambda z. s \ (s \ z) \\
3 & \equiv \lambda s. \lambda z. s \ (s \ (s \ z)) \\
& \ldots
\end{align*}
\]

Encoding Naturals (2)

Successor: \( \text{succ} \triangleq \lambda s. \lambda z. \lambda n. s \ (n \ s \ z) \)

Addition: \( \text{add} \triangleq \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z) \)

Multiplication: \( \text{mult} \triangleq \lambda m. \lambda n. \text{add} \ n \) \( 0 \)

Zero test: \( \text{is \ _ \ zero} \triangleq \lambda n. \lambda t. \lambda f. n \ (\lambda x. f) \ t \)

What Can You Do?

- Reason exactly about naturals.
- Encode objects! (almost with what we have seen)
- Write any sequential program (Church-Turing)
  - Haskell, ML, Lisp, Scheme, etc.
- Prove properties?

Imperative Programming

- Problem: encoding a state.
- Similar to semantic function \( \rightarrow \) for lambda terms.
- Operational semantics:
  - give a function from states to states.
  - Repeat it as long as possible.
- Computation finished when:
  - semantic function not applicable, or
  - Function yields always the same state (fixpoint).
- The result is contained in variable Result.
- In practice: \( \lambda x. t \) becomes \( \lambda x. \lambda s. t' \) where \( s \) associates variables to values.

Example of State Encoding

\[
\begin{align*}
\text{function fact}(n) \text{ is} \\
\text{int result} := 1; \\
\text{while } n > 1 \text{ do} \\
\quad \text{result} := \text{result} \ast n; \\
\quad n := n - 1; \\
\text{end; return result;}
\end{align*}
\]

\[
\begin{align*}
\text{function fact-rec}(n) \text{ is return fact-rec}(n, 1) \\
\text{function fact-rec}(n, 0) \text{ is}
\quad \text{if } (n > 1) \text{ then return fact-rec}(n - 1, n \ast \text{result}); \\
\quad \text{else return result;}
\end{align*}
\]
Modeling Concurrency

- The `choice` operator can be used for expansion:
  - `(x := y + 1 \| y := x + 1)`
  - `(x := y + 1; y := x + 1)`
  - `(y := x + 1; x := y + 1)`
  - `(z := y + 1; y := x + 1; x := z)`

Non-Determinism in Lambda

\[
\begin{align*}
\frac{t \to t'}{\mu t \to \mu t'} \\
\frac{u \to u'}{\nu u \to \nu u'} \\
\frac{t \to t'}{\lambda x.t \to \lambda x.t'}
\end{align*}
\]

Encoding Non-Determinism?

- **It does not work!**
- Because of one property named either `diamond`, `Church-Rosser`, or `confluence`.
- In short: the order of reductions does not matter.

Example

- Although a lambda-term can have more than one redex:
  - `add(3, mult(2, 6))`
  - `3 + mult(2, 6)`
  - `add(3, 12)`

  - One can hardly say this models concurrency!

Calculi for Concurrency

- Have (internal) non-deterministic operators:
  - CCS,
  - CSP,
  - Pi-Calculus,
  - Ambients...
  - Next Time!