Creol as formal model for distributed, concurrent objects

Martin Steffen

IfI UiO

Flacos, Malta

27. November 2008

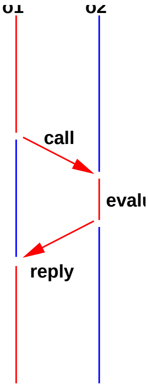
Structure

[0]

Creol: a concurrent object model

- executable oo modelling language concurrent objects
- formal semantics in rewriting logics /Maude
- strongly typed
- method invocations: synchronous or asynchronous
- targets open distributed systems
- recently: concurrent objects by (first-class) futures/promises
- dynamic reprogramming : class definitions may *evolve at runtime*
- the language design should support verification

Object-orientation: remote method calls



RMI / RPC method call model

- Control threads follow call stack
- Derived from sequential setting
- Hides / ignores distribution!
- Tightly synchronized!

***Creol*:**

- Show / exploit distribution!
- Asynchronous method calls
  - more efficient in distributed environments
  - *triggers* of concurrent activity
- Special cases:
  - *Synchronized communication:*
    the caller decides to wait for the reply
  - *Sequential computation:*
    only synchronized computation

Object Communication in *Creol*

- Objects communicate through method invocations *only*
- Methods organized in classes, seen externally via interfaces
- *Different ways to invoke* a method m
- Decided by caller — *not* at method declaration
- **Asynchronous** invocation: $l!o.m(In)$
- **Passive waiting** for method result: **await** $l?$
- **Active waiting** for method result: $l?(Out)$
- **Guarded** invocation: $l!o.m(In); \ldots;$ **await** $l?; l?(Out)$

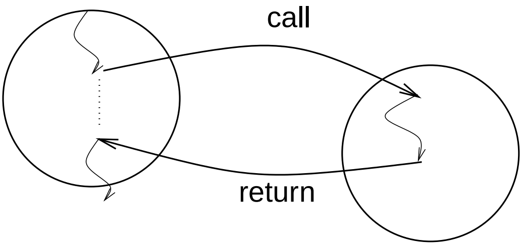
Language Constructs

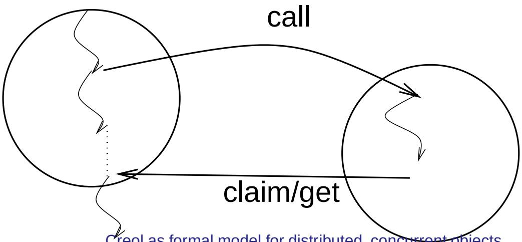
Syntactic categories.	*Definitions*.
l in Label	$g ::= wait \mid \phi \mid l? \mid g_1 \wedge g_2$
g in Guard	$p ::= o.m \mid m$
p in MtdCall	$S ::= s \mid s; S$
S in ComList	$s ::= \textbf{skip} \mid (S) \mid S_1 \Box S_2 \mid S_1 \| \! \| S_2$
s in Com	$\mid x := e \mid x := \textbf{new}\ classname(e)$
x in VarList	$\mid \textbf{if}\ \phi\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
e in ExprList	$\mid\ !p(e) \mid l!p(e) \mid l?(x) \mid p(e; x)$
m in Mtd	$\mid \textbf{await}\ g \mid \textbf{await}\ l?(x) \mid \textbf{await}\ p(e; x)$
o in ObjExpr	
ϕ in BoolExpr	

Futures

- introduced in the concurrent Multilisp language [7] [2]
- originally: transparent concurrency compiler annotation
- `future e`:
  - evaluated potentially in parallel with the rest ⇒ 2 threads (producer and consumer)
  - future variable dynamically generated
  - when evaluated: future identified with value
- wait-by-necessity [3] [4]
- supported by *Oz, Alice, MultiLisp, . . .* (shared state concurrency), Io, Joule, E, and most actor languages (Act1/2/3 . . . , ASP), Java

Async. method calls and futures



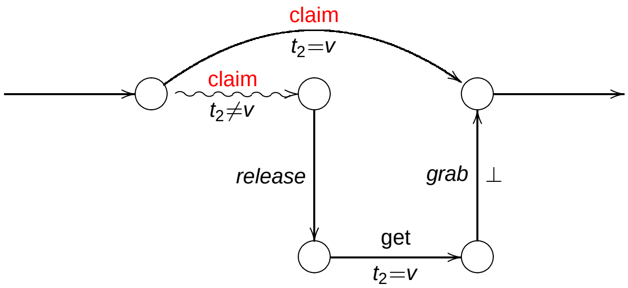


Syntax

- $o@l(\vec{v})$: asynchronous method call, non-blocking
- execution:
  1. create a "placeholder"/reference to the eventual result: future reference ("label")
  2. initiate execution of method body
  3. continue to execute (= non-blocking, asynchronous)

$$e \quad ::= \quad \ldots \mid o@l(v, \ldots, v) \mid \text{claim}@(n, o) \mid \underline{\text{get}@n} \mid \ldots$$

Claiming a future



Futures and promises

- terminology is not so clear
- relation to handled futures
- promises [9], I-structures [1]

⇒ 2 aspects of future var:
  - write = value of *e* "stored" to future
  - read by the clients
- *promises*: separating the creation of future-reference from attaching code to it[1]
- good for delegation

[1] as in for async. calls

Syntax (promise)

- instead of $o@l(\vec{v})$
- split into
  1. create a promise[2]
  2. fulfill the promise = bind code to it.

$$e \quad ::= \quad \ldots \mid \text{promise } T \mid \text{bind}\, o.l(\vec{v}) : T \hookrightarrow n \mid \ldots$$

[2] or a handle to the future.

$$n'\langle \text{let}\, x{:}T' = \text{promise}\, T \;\text{in}\; t\rangle \rightsquigarrow \nu(n{:}T').(n'\langle \text{let}\, x{:}T' = n \;\text{in}\; t\rangle) \qquad \textsc{Prom}$$

$$\begin{aligned} \ldots n_1\langle \text{let}\, x{:}T = \text{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2 \;\text{in}\; t_1\rangle &\xrightarrow{\tau} \\ \ldots n_1\langle \text{let}\, x{:}T = n_2 \;\text{in}\; t_1\rangle & \qquad \textsc{Bind}_i \\ \parallel (n_2\langle \text{let}\, x{:}T_2 = \text{grab}(o); M.l(o)(\vec{v}) \;\text{in release}(o); x\rangle) & \end{aligned}$$

Interface description: Task

- characterize possible interface behavior
- possible = adhering to the restriction of the language
  - well-typed
- basis of a trace logic / interface description
- abstraction process:
  - not $C \stackrel{t}{\Longrightarrow} \acute{C}$?
  - rather: consider C in a context / environment

$$C \parallel E \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$$

for some environment E

$\Rightarrow$ open semantics

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash C : \acute{\Theta}$$

- assumptions Δ abstracts environments E

One step further: legal traces

- open sesmantics

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash C : \acute{\Theta}$$

abstracts the environment

- existential abstraction of component, as well:
- characterization of *principally possible* interface behavior

$$C \parallel E \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$$

for some component C + some environment E

$\Rightarrow$ legal trace

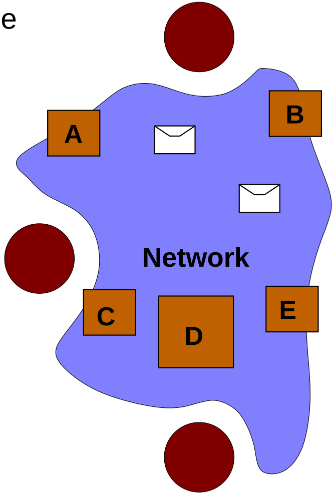
$$\Delta \vdash t : trace :: \Theta$$

Behavioral interface description

- type system for futures, especially resource aware (linear) type system for promises
- standard soundness results (subject reduction, ...)
- formulation of an open semantics plus characterization of possible interface behavior by abstracting the environment
- soundness of the abstractions
- basis for testing Creol objects/components

Dynamic Classes in Creol

- Dynamic classes: *modular* OO upgrade mechanism
- asynchronous upgrades propagate through the dist. system
- Modify class definitions at runtime
- Class upgrade affects:
  - All future instances of the class and its subclasses
  - All existing instances of the class and its subclasses

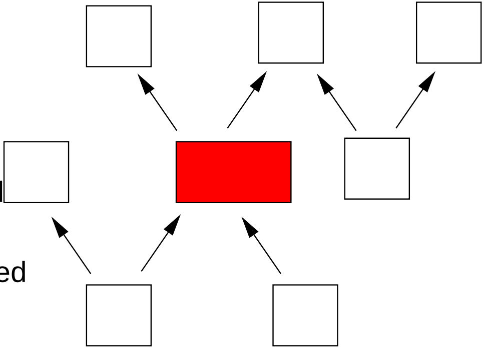


A Dynamic Class Mechanism

General case: Modify a class in a class hierarchy

Type correctness: Method binding should still succeed!

- Attributes may be added (no restrictions)
- Methods may be added (no restrictions)
- Methods may be redefined (subtyping discipline)
- Superclasses may be added
- Formal class parameters may *not* be modified



Theorem. Dynamic class extensions are type-safe in Creol's type system!

Example of a Class Upgrade: The Good Bank Customer (1)

```
class BankAccount implements Account          --- Version 1
begin var bal : Int = 0
with Any
  op deposit (in sum : Nat) == bal := bal+sum
  op transfer (in sum : Nat, acc : Account) ==
    await bal ≥ sum ; bal := bal−sum; acc.deposit(sum)
end
upgrade class BankAccount
begin var overdraft : Nat = 0
with Any
  op transfer (in sum : Nat, acc : Account) ==
    await bal ≥ (sum−overdraft); bal := bal−sum;
      acc.deposit(sum)
with Banker
  op overdraft_open (in max : Nat) == overdraft := max
end
```

Example of a Class Upgrade: The Good Bank Customer (2)

class BankAccount **implements** *Account* - - - *Version 2*
begin var bal : Int = 0, overdraft : Nat = 0
with *Any*
  op deposit (**in** sum : Nat) == bal := bal$+$sum
  op transfer (**in** sum : Nat, acc : Account) ==
    await bal $\geq$ (sum$-$overdraft); bal := bal$-$sum;
      acc.deposit(sum)
with *Banker*
  op overdraft_open (**in** max : Nat) == overdraft := max
end

Substitutability and subtype polymorphism

Problem:

When can some expression e_1 replace some other expression e_2?

classical answer: subtyping

Example 1: Assignment

x := e

$$\frac{\Gamma \vdash e : T \qquad T \leq \Gamma(x)}{\Gamma \vdash \mathbf{x\ :=\ e} : \mathbf{ok}}$$

Example 2: Method Calls

x := m(e)

$\mathtt{m:}\quad \mathbf{T}_1 \rightarrow \mathbf{T}_2$

$\mathbf{T}_1 \leq \mathbf{T}_1' \qquad \Downarrow \qquad \Uparrow \qquad \mathbf{T}_2' \leq \mathbf{T}_2$

(contravariance) $\mathtt{m':}\quad \mathbf{T}_1' \rightarrow \mathbf{T}_2'$ **(covariar**

Want: **m(e)**

Get: **m′(e)**

Behavioral subtyping

Extend subtyping to behavioral properties:

"any property proved about supertype objects
also holds for subtype objects" [Liskow & Wing 94]

Consider an assertion language on local state variables,
a programming language, and some program logic.

Assertions $p_1, p_2, q_1, q_2, \ldots$ used for pre- and postconditions

When can we replace e_1 by e_2?

$\{p_1\}$ e1 $\{q_1\}$ *Applicability:* $p_1 \Rightarrow p_2$ *(ref. contravariance)*

$\{p_2\}$ e2 $\{q_2\}$ *Predictability:* $q_2 \Rightarrow q_1$ *(ref. covariance)*

Late Binding of Method Calls

Object-oriented programming

- incremental program development
- *Substitutability* is exploited to organize programs by means of *inheritance*
  - *object substitutability*:
    a subclass object may be bound to a superclass variable
  - *method substitutability* (late binding):
    subclass methods may be selected instead of superclass methods

Late binding of method calls

- code bound to a call depends on the actual class of the object
- decided at runtime
- Not statically decidable

Example

```
class C {
  m() {...}
  n() {...; m(); ...}
}

class D extends C {
  m() {...}
}
```

- the binding of `m()` depends on the *actual class of the object*
- Incremental development*: the class `D` may be added later*
- *late binding and incremental development pose a challenge for program verification*

Verifying late-bound method calls

- two main approaches in the literature
- **Open world** [America 91, Liskow & Wing 94, Leavens & Naumann 06, . . .]
  - Behavioral subtyping: supports incremental reasoning
  - Subtyping constraints: too restrictive in practice
- **Closed world** [Pierik & de Boer 05, . . .]
  - Complete reasoning method
  - Breaks incremental reasoning
- **Lazy behavioral subtyping** [6]
  - supports incremental reasoning
  - less restrictive than behavioral subtyping

Example: Closed World Approach

```
class C {
  m(): (p1, q1) {...}                     Commitment (declaration sit
  n() {...; {p}m(){q}; ...}               Requirement (call site)
}                                         PO: p ⇒ p1 ∧ p2, q1 ∨ q2 ⇒

class D extends C {
  m(): (p2, q2) {...}                     Commitment (declaration sit
}
```

PO: $p \Rightarrow p_1 \wedge p_2,\ q_1 \vee q_2 \Rightarrow$

Closed world approach

- Assumes all commitments of a method known at reasoning time
- Sufficiently expressive: *complete reasoning system*
- *redo* *proofs if a new class is added to the program*
- *breaks with incremental development principle (proof reuse)*

Example: Open World Approach

class C {	
m(): (p_1, q_1) {...}	*Commitment (declaration site)*
n() {...; $\{p\}$m()$\{q\}$; ...}	*Requirement (call site)*
}	*PO:* $p \Rightarrow p_1$, $q_1 \Rightarrow q$
class D **extends** C {	
m(): (p_2, q_2) {...}	*Commitment (declaration site)*
}	*PO:* $p_1 \Rightarrow p_2$, $q_2 \Rightarrow q_1$

Behavioral subtyping

- (p_1, q_1) acts as a commitment (contract) for declarations of `m`
- redefinitions relate to the contract, not to the call site
- incremental: Proof reuse when the program is extended
- restriction: (p_1, q_1) too strong requirement for redefinitions

Example: Lazy Behavioral Subtyping

class C {	
m(): (p_1, q_1) {...}	*Commitment (declaration site)*
n() {...; $\{p\}$m()$\{q\}$; ...}	*Requirement (call site)*
}	*PO:* $p \Rightarrow p_1$, $q_1 \Rightarrow q$
class D **extends** C {	
m(): (p_2, q_2) {...}	*Commitment (declaration site)*
}	*PO:* $p \Rightarrow p_2$, $q_2 \Rightarrow q$

Lazy behavioral subtyping

- POs depend on *requirements*, not on commitments (contracts)
- irrelevant parts of old commitments may be ignored
- more flexible than behavioral subtyping approach
- incremental: proof reuse when program is extended

Lazy Behavioral Subtyping

- Distinguish method use and method declarations
- track call site requirements and declaration site commitments
- Proof reuse: Impose these requirements on method overridings in new subclasses to ensure that old proofs remain valid
- declaration site proof obligations wrt. superclass' requirements
  - Many, but weaker POs than with behavioral subtyping

  for superclass declarations
- Formalize how commitments and requirements propagate as subclasses and proof outlines are added
  - Proof environment tracks commitments and requirements
  - Syntax-driven inference system for program analysis
  - Independent of a particular program logic

Conclusion and prospect

- testing Creol-components
- FP7 prject HATS "highly-adaptable and trusworthy software"
  - software evolution
  - software families

References I

[1] Arvind, R. S. Nikhil, and K. K. Pingali.
I-structures: Data-structures for parallel computing.
ACM Transactions on Programming Languages and Systems, 11(4):598–632, 1989.

[2] H. Baker and C. Hewitt.
The incremental garbage collection of processes.
ACM Sigplan Notices, 12:55–59, 1977.

[3] D. Caromel.
Service, asynchrony and wait-by-necessity.
Journal of Object-Oriented Programming, 2(4):12–22, Nov. 1990.

[4] D. Caromel.
Towards a method of object-oriented concurrent programming.
Communications of the ACM, 36(9):90–102, Sept. 1993.

[5] D. Clarke, E. B. Johnsen, and O. Owe.
Concurrent objects à la carte.
In D. Dams, U. Hannemann, and M. Steffen, editors, *Correctness, Concurrency, and Compositionality*, 2008.

[6] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen.
Lazy behavioral subtyping.
In *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 2008.

[7] R. H. Halstead, Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, Oct. 1985.

[8] E. B. Johnsen, O. Owe, and I. C. Yu.
Creol: A type-safe object-oriented model for distributed concurrent systems.
Theoretical Computer Science, 365(1–2):23–66, Nov. 2006.

References II

[9] B. Liskov and L. Shrira.
Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems.
SIGPLAN Notices, 23(7):260–267, 1988.

[10] O. Owe, G. Schneider, and M. Steffen.
Components, objects, and contracts.
In *Sixths International Workshop on Specification and Verification of Component-Based Systems, Sept. 3–4, 2007, Catvat, Croatia*, pages 95–98, Aug. 2007.

[11] C. Prisacariu and G. Schneider.
A formal language for electronic language.
In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS '07*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, June 2007.