Actors (PDCS 4)
AMST actor language syntax, semantics, join
continuations

Carlos Varela
Rennselaer Polytechnic Institute

October 4, 2016
Advantages of concurrent programs

- **Reactive programming**
  - User can interact with applications while tasks are running, e.g., stopping the transfer of a big file in a web browser.

- **Availability of services**
  - Long-running tasks need not delay short-running ones, e.g., a web server can serve an entry page while at the same time processing a complex query.

- **Parallelism**
  - Complex programs can make better use of multiple resources in new multi-core processor architectures, SMPs, LANs, WANs, grids, and clouds, e.g., scientific/engineering applications, simulations, games, etc.

- **Controllability**
  - Tasks requiring certain preconditions can suspend and wait until the preconditions hold, then resume execution transparently.
Disadvantages of concurrent programs

- **Safety**
  - « *Nothing bad ever happens* »
  - Concurrent tasks should not corrupt consistent state of program.
- **Liveness**
  - « *Anything ever happens at all* »
  - Tasks should not suspend and indefinitely wait for each other (deadlock).
- **Non-determinism**
  - Mastering exponential number of interleavings due to different schedules.
- **Resource consumption**
  - Threads can be expensive. Overhead of scheduling, context-switching, and synchronization.
  - Concurrent programs can run *slower* than their sequential counterparts even with multiple CPUs!
Overview of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the most difficult, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, otherwise use actors and message passing.
Actors/SALSA

• Actor Model
  – A reasoning framework to model concurrent computations
  – Programming abstractions for distributed open systems

• SALSA
  – Simple Actor Language System and Architecture
  – An actor-oriented language for mobile and internet computing
  – Programming abstractions for internet-based concurrency, distribution, mobility, and coordination
1. Extend a functional language ($\lambda$-calculus + ifs and pairs) with actor primitives.

2. Define an operational semantics for actor configurations.

3. Study various notions of equivalence of actor expressions and configurations.

4. Assume fairness:
   - Guaranteed message delivery.
   - Individual actor progress.
Open Distributed Systems

• Addition of new components

• Replacement of existing components

• Changes in interconnections
Synchronous vs. Asynchronous Communication

- The $\pi$-calculus (and other process algebras such as CCS, CSP) take synchronous communication as a primitive.

- The actor model assumes asynchronous communication is the most primitive interaction mechanism.
Communication Medium

• In the π-calculus, channels are explicitly modeled. Multiple processes can share a channel, potentially causing interference.

• In the actor model, the communication medium is not explicit. Actors (active objects) are first-class, history-sensitive entities with an explicit identity used for communication.
Fairness

• The actor model theory assumes fair computations:
  1. Message delivery is guaranteed.
  2. Individual actor computations are guaranteed to progress.

Fairness is very useful for reasoning about equivalences of actor programs but can be hard/expensive to guarantee; in particular when distribution and failures are considered.
λ-Calculus as a Model for Sequential Computation

Syntax

\[ e ::= v \quad \text{value} \]

\[ \mid \lambda v.e \quad \text{functional abstraction} \]

\[ \mid (e \ e) \quad \text{application} \]

Example of beta-reduction:

\[ (\lambda x.x^2 \ 2) \]

\[ \longrightarrow x^2\{2/x\} \]
\( \lambda \)-Calculus extended with pairs

- \( pr(x,y) \)  \textit{returns a pair containing} \( x \) \& \( y \)
- \( ispr(x) \)  \textit{returns} \( t \) \textit{if} \( x \) \textit{is a pair}; \( f \) \textit{otherwise}
- \( 1^{st}(pr(x,y)) = x \)  \textit{returns the first value of a pair}
- \( 2^{nd}(pr(x,y)) = y \)  \textit{returns the second value of a pair}
Actor Primitives

- **send**(a, v)
  - Sends value v to actor a.

- **new**(b)
  - Creates a new actor with behavior b (a λ-calculus abstraction) and returns the identity/name of the newly created actor.

- **ready**(b)
  - Becomes ready to receive a new message with behavior b.
AMST Actor Language

Examples

\( b5 = \text{rec}(\lambda y. \ \lambda x. \text{seq}(\text{send}(x,5), \text{ready}(y))) \)
receives an actor name \( x \) and sends the number 5 to that actor, then it becomes ready to process new messages with the same behavior \( y \).

Sample usage:

\( \text{send}(\text{new}(b5), \ a) \)

A \textit{sink}, an actor that disregards all messages:

\( \text{sink} = \text{rec}(\lambda b. \ \lambda m. \text{ready}(b)) \)
Reference Cell

cell = rec(\lambda b. \lambda c. \lambda m.
        if ( get?(m),
            seq( send(cust(m), c),
                 ready(b(c)))
        if ( set?(m),
            ready(b(contents(m))),
            ready(b(c)))))

Using the cell:
let a = new(cell(0)) in seq( send(a, mkset(7)),
                               send(a, mkset(2)),
                               send(a, mkget(c)))
Join Continuations

Consider:

\[
\text{treeprod} = \text{rec}(\lambda f. \lambda \text{tree}. \\
\text{if(isnat(tree),} \\
\text{tree,} \\
\text{f(left(tree))}*f(right(tree))))
\]

which multiplies all leaves of a tree, which are numbers.

You can do the “left” and “right” computations concurrently.
Tree Product Behavior

\[ B_{\text{treeprod}} = \]
\[
\text{rec}(\lambda b. \lambda m. \]
\[
\text{seq}(\text{if}(\text{isnat}(\text{tree}(m)), \]
\]
\[
\text{send}(\text{cust}(m), \text{tree}(m)), \]
\]
\[
\text{let newcust} = \text{new}(B_{\text{joincont}}(\text{cust}(m))), \]
\[
\text{lp} = \text{new}(B_{\text{treeprod}}), \]
\[
\text{rp} = \text{new}(B_{\text{treeprod}}) \text{ in} \]
\[
\text{seq}(\text{send}(\text{lp}, \]
\]
\[
\text{pr}(\text{left}(\text{tree}(m)), \text{newcust})), \]
\]
\[
\text{send}(\text{rp}, \]
\[
\text{pr}(\text{right}(\text{tree}(m)), \text{newcust}))))), \]
\[
\text{ready}(b) ))
\]
\[ B_{\text{joincont}} = \]
\[ \lambda \text{cust.} \lambda \text{firstnum.} . \text{ready} (\lambda \text{num.} . \]
\[ \text{seq} (\text{send} (\text{cust, firstnum*num}), \]
\[ \text{ready} (\text{sink})) \]
Sample Execution

(a) $f(\text{tree}, \text{cust})$

(b) $f(\text{left(tree)}, \text{JC})$

$\text{JC}$

$\text{JC}$

$\text{JC}$

$\text{cust}$

$\text{cust}$

$\text{cust}$
Sample Execution

(e)

(f)

C. Varela
Operational Semantics for AMST Actor Language

- Operational semantics of actor model as a labeled transition relationship between actor configurations.

- Actor configurations model open system components:
  - Set of individually named actors
  - Messages “en-route”
Actor Configurations

\[ k = \alpha \parallel \mu \]

\(\alpha\) is a function mapping actor names (represented as free variables) to actor states.

\(\mu\) is a multi-set of messages “en-route.”
Syntactic restrictions on configurations

Given $A = \text{Dom}(\alpha)$:

- If $a \in A$, then $\text{fv}(\alpha(a))$ is a subset of $A$.

- If $<a \leq v>$ in $\mu$, then $\{a\} \cup \text{fv}(v)$ is a subset of $A$. 
Consider the expression:
\[ e = \text{send}(\text{new}(b5), a) \]

- The redex \( r \) represents the next sub-expression to evaluate in a left-first call-by-value evaluation strategy.
- The reduction context \( R \) (or \textit{continuation}) is represented as the surrounding expression with a \textit{hole} replacing the redex.

\[
\text{send}(\text{new}(b5), a) = \text{send}(\square, a) \triangleleft \text{new}(b5) \triangleright
\]
\[ e = R \triangleright r \triangleleft \text{ where } \]
\[ R = \text{send}(\square, a) \]
\[ r = \text{new}(b5) \]
Labeled Transition Relation

\[
\frac{e \rightarrow_{\lambda} e'}{\alpha, [R \triangleright e \triangleright]_a \parallel \mu \quad [\text{fun:a}] \quad \alpha, [R \triangleright e' \triangleright]_a \parallel \mu}
\]

\[
\alpha, [R \triangleright \text{new}(b) \triangleright]_a \parallel \mu \quad [\text{new:a,a'}] \quad \alpha, [R \triangleright a' \triangleright]_a, [\text{ready}(b)]_{a'} \parallel \mu
\]

\[
\alpha', \text{fresh}
\]

\[
\alpha, [R \triangleright \text{send}(a', v) \triangleright]_a \parallel \mu \quad [\text{snd:a}] \quad \alpha, [R \triangleright \text{nil} \triangleright]_a \parallel \mu \uplus \{\langle a' \leftarrow v \rangle\}
\]

\[
\alpha, [R \triangleright \text{ready}(b) \triangleright]_a \parallel \{\langle a \leftarrow v \rangle\} \uplus \mu \quad [\text{rcv:a,v}] \quad \alpha, [b(v)]_a \parallel \mu
\]
37. Write
   get?
cust
set?
contents
mkset
mkget
to complete the reference cell example in the AMST actor language.

38. Modify the cell behavior to notify a customer when the cell value has been updated.

39. PDCS Exercise 4.6.6 (page 77).

40. PDCS Exercise 4.6.7 (page 78).