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

Carlos Varela
Rensselaer Polytechnic Institute

November 7, 2014
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.
\( \lambda \)-Calculus as a Model for Sequential Computation

Syntax

\[
e  ::=  \ v  \quad \text{value}  \\
    |  \ \lambda v. e  \quad \text{functional abstraction}  \\
    |  (\ e\ e )  \quad \text{application}
\]

Example of beta-reduction:

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

- \textbf{pr}(x,y) \quad \textit{returns a pair containing } x \& y

- \textbf{ispr}(x) \quad \textit{returns } t \textit{ if } x \textit{ is a pair; } f \textit{ otherwise}

- \textbf{1}^{\text{st}}(\text{pr}(x,y)) = x \quad \textit{returns the first value of a pair}

- \textbf{2}^{\text{nd}}(\text{pr}(x,y)) = y \quad \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

\[ b_5 = \text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x,5), \text{ready}(y))) \]
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:
\[
\text{send(new(b5), a)}
\]

A sink, an actor that disregards all messages:
\[
sink = \text{rec}(\lambda b. \lambda m. \text{ready}(b))
\]
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:

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

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}) \)), \]
\[ \text{ready}(b)) \]
Tree Product (continued)

\[ B_{\text{joincont}} = \]

\[ \lambda \text{cust}. \text{ready}( \]

\[ \lambda \text{firstnum}. \text{ready}( \lambda \text{num}. \]

\[ \text{seq}(\text{send}(\text{cust}, \text{firstnum} \ast \text{num}), \]

\[ \text{ready}(\text{sink}))) \]
Sample Execution

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

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

$\text{JC}$

$cust$

$\text{JC}$

$cust$

$\text{JC}$

$cust$
Sample Execution

\[ f(\text{left(tree)}, \text{JC}) \]
Sample Execution

Cust

num

Cust

firstnum

JC

(f)

Cust

(firstnum * num)

(e)
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 $\langle a \leq v \rangle$ in $\mu$, then $\{a\} \cup \text{fv}(v)$ is a subset of $A$. 
Labeled Transition Relation

\[
\begin{align*}
\frac{e \rightarrow_{\lambda} e'}{
\alpha, [R \triangleright e \leftarrow]_a \parallel \mu \quad \xrightarrow{[\text{fun:a}]} \quad \alpha, [R \triangleright e' \leftarrow]_a \parallel \mu
}
\end{align*}
\]

\[
\begin{align*}
\alpha, [R \triangleright \text{new}(b) \leftarrow]_a \parallel \mu \quad \xrightarrow{[\text{new:a,a'}]} \quad \alpha, [R \triangleright a' \leftarrow]_a, [\text{ready}(b)]_{a'} \parallel \mu
\end{align*}
\]

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

\[
\alpha, [R \triangleright \text{ready}(b) \leftarrow]_a \parallel \{ \langle a \leftarrow v \rangle \} \uplus \mu \quad \xrightarrow{[\text{rcv:a,v}]} \quad \alpha, [b(v)]_a \parallel \mu
\]
Exercises

70. Write
   get?
cust
set?
contents
mkset
mkget
to complete the reference cell example in the AMST actor language.

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

72. PDCS Exercise 4.6.6 (page 77).

73. PDCS Exercise 4.6.7 (page 78).