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

Carlos Varela
Rensselaer Polytechnic Institute

October 8, 2019
Advantages of concurrent programs

• Reactive programming
  – User can interact with applications while tasks are running, *e.g.*, stopping the transfer of a large 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 hardware resources in 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**
  - Concurrency 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 main 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, C++)

• 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 (call-by-value $\lambda$ calculus + if's and pair's) 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) uses synchronous communication.

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

• In the $\pi$-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 (stateful) entities with an explicit identity used for communication.
Fairness

• The actor model theory assumes fair computations:
  1. Message delivery is guaranteed.
  2. Infinitely-often enabled computations must eventually happen.

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

Syntax:
\[
e \ ::= \begin{array}{l}
v \\ | \lambda\, v. \, e \\ | \, e(e)
\end{array}
\]

\textit{variable} \hspace{1cm} \textit{function} \hspace{1cm} \textit{application}

Example of beta-reduction:
\[
\lambda\, x.\, x^2(3) \quad \rightarrow \quad x^2\{3/x\}
\]
λ-Calculus extended with pairs

- \( \text{pr}(x, y) \) returns a pair containing \( x \& y \)
- \( \text{ispr}(x) \) returns \( t \) if \( x \) is a pair; \( f \) otherwise
- \( 1^{st}(\text{pr}(x, y)) = x \) returns the first value of a pair
- \( 2^{nd}(\text{pr}(x, y)) = y \) returns the 2nd 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 $\lambda$-calculus functional 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 \( y \) \((b5)\).

Sample usage:
\[ \text{send(new}(b5),\ a) \]

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)))

C. Varela
Join Continuations

Consider:

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

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

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

\[ B_{\text{treeprod}} = \]
\[ \text{rec}(\lambda b. \lambda m. \]
\[ \quad \text{seq}(\text{if}(\text{isnat}(\text{tree}(m)), \]
\[ \quad \quad \text{send}(\text{cust}(m), \text{tree}(m)), \]
\[ \quad \quad \text{let newcust}=\text{new}(B_{\text{joincont}}(\text{cust}(m))), \]
\[ \quad \quad \quad \text{lp} = \text{new}(B_{\text{treeprod}}), \]
\[ \quad \quad \quad \text{rp} = \text{new}(B_{\text{treeprod}}) \text{ in} \]
\[ \quad \quad \text{seq}(\text{send}(\text{lp}, \]
\[ \quad \quad \quad \text{pr}(\text{left}(\text{tree}(m)), \text{newcust})), \]
\[ \quad \quad \quad \text{send}(\text{rp}, \]
\[ \quad \quad \quad \quad \text{pr}(\text{right}(\text{tree}(m)), \text{newcust}))), \]
\[ \quad \quad \text{ready}(b)) \]
Tree Product (continued)

\[ B_{\text{joincont}} = \lambda \text{cust}.\lambda \text{firstnum}.\text{ready}(\lambda \text{num}.\]
\[ \quad \text{seq}(\text{send}(\text{cust}, \text{firstnum} \times \text{num}),\]
\[ \quad \quad \text{ready}(\text{sink})) \]
Sample Execution

\[ f(\text{tree}, \text{cust}) \]

(a)

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

(b)

C. Varela
Sample Execution

\( f(\text{left}(\text{tree}), \text{JC}) \)
Sample Execution
Operational Semantics of AMST Actor Language

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

\[ k_1 \xrightarrow{\text{[label]}} k_2 \]

- Actor configurations model open system components:
  - Set of individually named actors
  - Messages “en-route”
Operational Semantics of AMST Actor Language

- Operational semantics of actor language 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 $fv(\alpha(a))$ is a subset of $A$.

• If $<a \leq v>$ in $\mu$, then $\{a\} \cup fv(v)$ is a subset of $A$. 
Reduction contexts and redexes

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 *continuation*) is represented as the surrounding expression with a *hole* replacing the redex.

\[
\begin{align*}
\text{send}(\text{new}(b5), a) &= \text{send}(\Box, a) \triangleright \text{new}(b5)\blacktriangleright \\
e &= R \triangleright r \blacktriangleright \text{ where} \\
R &= \text{send}(\Box, a) \\
r &= \text{new}(b5)
\end{align*}
\]
Labeled Transition Relation

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

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

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

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

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