CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 4.1,4.2

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2021
Reference Cell in the AMST Actor Language

A reference cell can be encoded in the actor language as follows:

\[ cell = \text{rec}( \lambda b. \lambda c. \lambda m. \]

\[ \text{if}( \text{get}(m), \]

\[ \text{seq}( \text{send}(\text{cust}(m), c), \]

\[ \text{ready}(b(c))), \]

\[ \text{if}( \text{set}(m), \]

\[ \text{ready}(b(\text{contents}(m))), \]

\[ \text{ready}(b(c)))))) \]

A client of the cell can be encoded as follows:

\[ \text{let } a = \text{new}(cell(0)) \text{ in seq}( \text{send}(a, \text{mkset}(7)), \]

\[ \text{send}(a, \text{mkset}(2)), \]

\[ \text{send}(a, \text{mkget}(c))) \]

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 4.1, 4.2 – p. 2/21
Reference Cell in the AMST Actor Language

Exercises

1. What value will actor $c$ receive after the example is executed?

2. How would you define the auxiliary $\lambda$ calculus functions:

   \[
   \begin{align*}
   mkget & = \\
   mkset & = \\
   get? & = \\
   set? & = \\
   cust & = \\
   contents & = \\
   \end{align*}
   \]

3. Modify the reference cell example to notify a customer when the cell value is updated (such as is done in the $\pi$ calculus reference cell example).
Reference Cell in the AMST Actor Language

1. A new reference cell actor $a$ is created and three messages are sent to the cell where $mkset$ and $mkget$ create appropriate pairs representing $set$ and $get$ messages.

   - Actor $c$ will receive a message containing either 0, 2, or 7, depending on the order of message processing at $a$.

2. The auxiliary functions can be defined in terms of booleans and the pairing primitives as follows:

   $mkget = \lambda c. pr(true, c)$

   $mkset = \lambda n. pr(false, n)$

   $get? = \lambda m. if(ispr?(m), 1^{st}(m), false)$

   $set? = \lambda m. if(ispr?(m), not(1^{st}(m)), false)$

   $cust = \lambda m. if(ispr?(m), 2^{nd}(m), nil)$

   $contents = \lambda m. if(ispr?(m), 2^{nd}(m), nil)$
Join Continuations

Consider the functional programming code to compute the product of numbers in the leaves of a binary tree:

\[ \text{treeprod} = \text{rec}( \lambda f. \lambda \text{tree}. \]

\[
\text{if(isnat?(tree),}
\text{tree,}
\text{f(left(tree))} \times \text{f(right(tree)))}
\]

- Binary trees are represented as either number values (leaves) or pairs with the left and right sub-trees (non-leaves).
- We can write actor code to compute the left and right branches concurrently.
Tree Product Behavior

\[ t_{prod} = \text{rec}( \lambda b. \lambda m. \text{seq}\left( \text{if}\left( \text{isnat}\left( \text{tree}\left( m \right) \right) \right), \right. \text{send}\left( \text{cust}\left( m \right), \text{tree}\left( m \right) \right), \text{let } new\text{cust} = \text{new}\left( \text{joincont}\left( \text{cust}\left( m \right) \right) \right), \text{lp} = \text{new}\left( t_{prod} \right), \text{rp} = \text{new}\left( t_{prod} \right) \text{in } \text{seq}\left( \text{send}\left( \text{lp}, \text{pr}\left( \text{left}\left( \text{tree}\left( m \right) \right), new\text{cust} \right) \right), \text{send}\left( \text{rp}, \text{pr}\left( \text{right}\left( \text{tree}\left( m \right) \right), new\text{cust} \right) \right) \right)), \text{ready}\left( b \right) \right) \]

\[ \text{joincont} = \lambda customer. \lambda firstnum. \text{ready}\left( \lambda num. \text{seq}\left( \text{send}\left( customer, firstnum \times num \right), \text{ready}\left( \text{sink} \right) \right) \right) \]
Tree Product Behavior

- Messages to the tree product actor are pairs containing the tree and the customer actor to send the result to.
- These are accessible by $tree(m)$ and $cust(m)$ respectively.

In summary, $tree = left = 1^{st}$, and $cust = right = 2^{nd}$.

- If the tree is a leaf, then the number is returned to the customer.
- Otherwise,
  - two new tree product actors are created,
  - and a new (*join continuation*) customer, in charge of:
    - getting the partial results for the left and right sub-trees,
    - composing the final result for the original customer.
Join Continuations

The new customer is called a *join continuation*, since its behavior is to wait for two computations executing concurrently and perform an action when they both complete.

- The join continuation actor has an initial state containing a *customer* to be notified of completion.
- On reception of the *first number* containing the result of the first computation completed,
  - it becomes ready to receive the *second number* and then
    - notifies the *customer* with the product of the two sub-computations, and
    - becomes a *sink* since its goals have been met.
Operational Semantics for AMST Actor Language

The operational semantics of the AMST actor language is defined as:

- a set of labelled transition rules from \textit{actor configurations} to actor configurations specifying valid computations.

Concurrent systems’ evolution over time can be followed by applying the rules specified in the operational semantics in a manner consistent with fairness.
Actor Configurations

Actor configurations model concurrent system components as viewed by an idealized observer, frozen in time. An actor configuration is composed of:

- a set of individually named actors, and
- messages “en-route”.

An actor configuration, $\kappa$, denoted as:

$$\alpha \parallel \mu$$

contains

- an actor map, $\alpha$, which is a function mapping actor names to actor expressions, and
- a multi-set of messages, $\mu$. A message is denoted as $\langle a \leftarrow v \rangle$. 
Syntactic Restrictions on Actor Configurations

There are two syntactic restrictions that valid actor configurations must conform to:

1. If \( a \in \text{dom}(\alpha) \), then \( \text{fv}(\alpha(a)) \subseteq \text{dom}(\alpha) \).
2. If \( \langle a \leftarrow v \rangle \in \mu \), then \( \{a\} \cup \text{fv}(v) \subseteq \text{dom}(\alpha) \).

- The first restriction ensures that free variables in an actor expression—representing its state—refer to valid actor names in the actor configuration.
- The second restriction does the same for free variables in messages.
- The only two variable binders in actor expressions are \( \lambda x \) . . . and \( \text{letrec } x = \ldots \) (i.e., they bind occurrences of variable \( x \) in the expression.) All other variable occurrences are said to be free.
Reduction Contexts and Redexes

**Lemma 1:** An actor expression, \( e \), is either a value \( v \), or otherwise it can be uniquely decomposed into a *reduction context*, \( R \), filled with a *redex*, \( r \), denoted as \( e = R \triangleright r \lhd \).

For example, the actor expression \( e = \text{send}(\text{new}(b5), a) \) can be decomposed into reduction context \( R = \text{send}(\Box, a) \) filled with redex \( r = \text{new}(b5) \). We denote this decomposition as:

\[
\text{send}(\text{new}(b5), a) = \text{send}(\Box, a) \triangleright \text{new}(b5) \lhd
\]

- The redex \( r \) represents the next sub-expression to evaluate in a standard 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.
Actor Language Operational Semantics

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

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

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

\[ \alpha, [R \triangleright \text{nil} \triangleright]_a \parallel \mu \uplus \{ \langle a' \leftarrow v \rangle \} \]

\[ \alpha, [R \triangleright \text{ready}(b) \triangleright]_a \parallel \{ \langle a \leftarrow v \rangle \} \uplus \mu \quad \frac{}{[\text{rcv}:a,v]} \quad \alpha, [b(v)]_a \parallel \mu \]
Actor Language Operational Semantics

The transition rules are of the form $\kappa_1 \xrightarrow{l} \kappa_2$, where

- $\kappa_1$ is the initial configuration,
- $\kappa_2$ is the final configuration, and
- $l$ is the transition label.

There are four rules, all of which apply to an actor $a$, which we call in focus.
The first one, labelled fun specifies sequential progress within the actor.

\[
\frac{e \rightarrow_{\lambda} e'}{
\alpha, [R \triangleright e \triangleleft]_a \parallel \mu \quad \alpha, [R \triangleright e' \triangleleft]_a \parallel \mu}
\]

- The fun rule subsumes functional computation using the \( \lambda \) calculus reduction rules (extended with pairs, primitive types, and conditionals.)
new Rule

The other three rules specify creation of and communication with other actors, and apply respectively to actor redxes: \texttt{new}(b), \texttt{send}(a', v), and \texttt{ready}(b).

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

- The rule labelled \texttt{new} specifies actor creation, which applies when the focus actor \(a\)'s redex is \texttt{new}(b): actor \(a\) creates a new actor \(a'\).
- The behavior of \(a'\) is set to the value \texttt{ready}(b), the actor \(a\)'s redex is replaced by the new actor’s name \(a'\).
- \(a'\) must be fresh, that is, \(a' \notin \text{dom}(\alpha) \cup \{a\}\).
**snd Rule**

\[ \alpha, [\mathbb{R} \triangleright \text{send}(a', v) \triangleleft]_a \parallel \mu \xrightarrow{[\text{snd}:a]} \]

\[ \alpha, [\mathbb{R} \triangleright \text{nil} \triangleleft]_a \parallel \mu \uplus \{\langle a' \Leftarrow v \rangle\} \]

- The rule labelled \texttt{snd} specifies asynchronous message sending, which applies when the focus actor \(a\)'s redex is \text{send}(a', v):
  - actor \(a\) sends a message containing value \(v\) to its acquaintance \(a'\).
  - Actor \(a\) continues execution and the network \(\mu\) is extended with a new message \(\langle a' \Leftarrow v \rangle\).
**rcv Rule**

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

- The rule labelled *rcv* specifies message reception, which applies when
  - the focus actor \(a\)’s redex is \text{ready}(b), and
  - there is a message in \(\mu\) directed to \(a\), e.g., \(\langle a \leftarrow v \rangle\).
- The actor \(a\)’s new state becomes \(b(v)\), that is, its behavior \(b\) is applied to the incoming message value \(v\).
- Notice that the reduction context \(R\) is discarded.
Computation Sequences and Paths

If $\kappa$ is a configuration, then the computation tree $\tau(\kappa)$ is the set of all finite sequences of labelled transitions $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$ for some $n \in \mathbb{N}$, with $\kappa = \kappa_0$.

- Such sequences are called *computation sequences*.
- These sequences are partially ordered by the initial segment relation.

A *computation path* from $\kappa$ is a maximal linearly ordered set of computation sequences in the computation tree, $\tau(\kappa)$.

- We denote a computation path by its maximal sequence.
- $\tau^\infty(\kappa)$ denotes the set of all (possibly infinite) paths from $\kappa$. 
Fair Computation Paths

Since transition labels have sufficient information to determine computation paths, we can refer to a computation path $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \infty]$ as

$$\pi = \kappa_i; l_i, l_{i+1}, \ldots$$

where $\kappa_j \xrightarrow{l_j} \kappa_{j+1}$, for $j \geq i$.

- Not all computation paths are admissible.
- Unfair computation paths, where infinitely-often enabled transitions never happen are ruled out.
  - A transition labelled $l$ is *enabled* in a configuration $\kappa$, if and only if there is a configuration $\kappa'$ such that $\kappa \xrightarrow{l} \kappa'$. 
Fair Computation Paths

A path \( \pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \infty] \) in the computation tree \( \tau^\infty(\kappa) \) is \textit{fair} if each enabled transition eventually happens or becomes permanently disabled.

- A transition with label of the form \([\text{rcv} : a, v]\) becomes permanently disabled if the actor \(a\) starts processing another message and never again becomes ready to accept a new message.

For a configuration \(\kappa\), we define \(\mathcal{F}(\kappa)\) to be the subset of paths in \(\tau^\infty(\kappa)\) that are fair.

- Notice that finite computation paths are fair by maximality, since all enabled transitions must have happened.