Abstractions for Concurrency

Lecture 18 Tuesday, April 7, 2015

1 Message passing

Last lecture we looked at a shared memory model of concurrency: different threads could concurrently access the same memory locations. A shared memory model is commonly used in many programming languages. However, shared memory can make it difficult to reason about the interaction between threads.

This lecture we consider a different model of concurrency: message passing. In this model, threads communicate by sending and receiving messages over channels. Channels are first-class values: they can be created at runtime, and used as values, including being passed as messages over channels.

Several languages use message passing, including Erlang, Go, Rust, Racket, X10, Smalltalk, F#, Concurrent ML (CML), and others.

The following grammar describes our language.

$$
\begin{aligned}
& c \in \textbf{ChanId} \\
& e ::= \lambda x{:}\tau.\, e \mid x \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid () \mid \mu f.\, e \\
& \qquad \mid c \mid \mathsf{spawn}\ e \mid \mathsf{newchan}_\tau \mid \mathsf{send}\ e_1\ \mathsf{to}\ e_2 \mid \mathsf{recv\ from}\ e \\
& v ::= n \mid c \mid () \mid \lambda x{:}\tau.\, e \\
& \tau ::= \textbf{int} \mid \textbf{unit} \mid \tau\ \textbf{chan} \mid \tau_1 \to \tau_2
\end{aligned}
$$

The language is a lambda calculus with integers, fixpoints ($\mu f.\ e$), and primitives for creating threads, and creating and using channels. Primitive spawn e creates a new thread, and expression e will execute in the new thread. A new channel can be created with primitive $\mathsf{newchan}_\tau$. The type annotation τ will be used to enforce that the new channel is used to send and receive values of type τ.

Expression send e_1 to e_2 computes e_1 to a value v, computes e_2 to a channel c, and sends v over channel c. The send "blocks" until some other thread executes a recv on the same channel, and then evaluates to the unit value $()$. That is, execution of the send e_1 to e_2 expression doesn't complete until some thread receives the value. Expression recv from e evaluates e to a channel c and then blocks until it receives a value on channel c. The expression evaluates to the value received.

The type system for this language expresses some aspects of the intended behavior of these new primitives.

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{spawn}\ e : \textbf{unit}} \qquad
\frac{}{\Gamma \vdash \mathsf{newchan}_\tau : \tau\ \textbf{chan}} \qquad
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \textbf{chan}}{\Gamma \vdash \mathsf{send}\ e_1\ \mathsf{to}\ e_2 : \textbf{unit}} \qquad
\frac{\Gamma \vdash e : \tau\ \textbf{chan}}{\Gamma \vdash \mathsf{recv\ from}\ e : \tau}
$$

1.1 Operational semantics

A configuration is now a list of expressions, one expression for each thread. That is, configuration $\langle e_1, \ldots, e_n \rangle$ represents the concurrent execution of n threads.

We use two judgements to define the operational semantics, one to indicate how a configuration (i.e., a list of threads) takes a step, and one to indicate how an individual thread takes a step. We use judgment $\langle e_1, \ldots, e_n \rangle \Longrightarrow \langle e'_1, \ldots, e'_m \rangle$ to indicate that configuration $\langle e_1, \ldots, e_n \rangle$ can take a small step to $\langle e'_1, \ldots, e'_m \rangle$ and we write $e \longrightarrow e'$ to indicate that thread e can take a small step to e'. For clarity, we present the entire operational semantics for the language here.

We first present the inference rules for the thread judgment $e \longrightarrow e'$.

$$
E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E \mid \mathsf{send}\ E\ \mathsf{to}\ e \mid \mathsf{send}\ v\ \mathsf{to}\ E \mid \mathsf{recv\ from}\ E
$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}} \qquad \frac{}{n_1 + n_2 \longrightarrow m}\ m = n_1 + n_2 \qquad \frac{}{\mathsf{newchan}_\tau \longrightarrow c}\ c \text{ is fresh}$$

We now present the rules for judgment $\langle e_1, \ldots, e_n \rangle \Longrightarrow \langle e'_1, \ldots, e'_m \rangle$. As a notational convenience, we write $\langle e_1, \ldots, e_n \rangle_{i \mapsto e'}$ as shorthand for the configuration $\langle e_1, \ldots, e_{i-1}, e', e_{i+1}, \ldots, e_n \rangle$, i.e., where the ith thread is replaced with expression e'.

$$\frac{e_i \longrightarrow e'_i}{\langle e_1, \ldots, e_n \rangle \Longrightarrow \langle e_1, \ldots, e_n \rangle_{i \mapsto e'_i}} \qquad \frac{e_i = E[\mathsf{spawn}\ e]}{\langle e_1, \ldots, e_n \rangle \Longrightarrow \langle e_1, \ldots, e_n, e \rangle_{i \mapsto E[()]}}$$

$$\frac{e_i = E_i[\mathsf{send}\ v\ \mathsf{to}\ c] \quad e_j = E_j[\mathsf{recv\ from}\ c]}{\langle e_1, \ldots, e_n \rangle \Longrightarrow \langle e_1, \ldots, e_n \rangle_{i \mapsto E_i[()], j \mapsto E_j[v]}}$$

The first rule for configurations states allows a thread of the configuration to take a step. This rule is nondeterministic: any thread that can take a step is allowed to. That is, we are not modeling any scheduler that restricts the order in which threads may execute.

The next rule describes spawning a new thread (spawn e), which is added to the end of the list of threads. The last rule handles a matching send and receive on the same channel. The thread evaluating the receive term evaluates to the value sent by the other thread. This semantics enforces blocking, in that the thread sending cannot complete the send operation until some other thread executes a receive, and vice versa.

1.2 Example

For convenience, we write $e_1; e_2$ as shorthand for let $x = e_1$ in e_2 where $x \notin FV(e_2)$. (This notation is only useful in a language with side-effects. Why?)

The follow program creates a new channel, spawns a thread that sends an integer value on the channel, and the original thread receives the value and adds 7 to it.

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{newchan}_{\mathbf{int}}\ \mathsf{in} \\ \mathsf{spawn}\ (\mathsf{send}\ 35\ \mathsf{to}\ c); \\ \mathsf{recv\ from}\ c + 7 \end{array}$$

The final configuration for this program will be $\langle 42, () \rangle$. Make sure you understand how this configuration is derived from an initial configuration that contains a single thread that executes the program above.

What about the following program? What does it do?

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{newchan}_{\mathbf{int}}\ \mathsf{in} \\ \mathsf{spawn}\ (\mu f.\ (\mathsf{send}\ 35\ \mathsf{to}\ c; f); \\ \mathsf{recv\ from}\ c + \mathsf{recv\ from}\ c + \mathsf{recv\ from}\ c \end{array}$$

Note that even though there is no shared memory, the language is still non-deterministic. Consider the following program, where 2 threads are executing a receive instruction, and one thread is executing a send, all on the same channel. What are the possible final configurations?

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{newchan}_{\mathbf{int}}\ \mathsf{in} \\ \mathsf{spawn}\ (3 + \mathsf{recv\ from}\ c); \\ \mathsf{spawn}\ (5 + \mathsf{recv\ from}\ c); \\ \mathsf{send}\ 15\ \mathsf{to}\ c \end{array}$$

2 First-class synchronization

Sending and receiving on channels as described above is synchronous. This is often desirable, but sometimes we would like *asynchronous* sends and receives. For example, suppose we wanted to know when a message was received on either channel c or channel d; it can be awkward to achieve this when sends and receives are synchronous.

Motivated by this need for asynchrony, we introduce *events*, which are abstract representations of synchronous operations. By having appropriate primitive operations on events, we can express asynchronous communication, and also other interesting synchronization patterns. Events, and the operations on them, come from Concurrent ML (CML), developed by John Reppy in the 80s and 90s. Events can be thought of as decoupling the description of a synchronous operation (e.g., "send value v to channel c") from the act of synchronizing on the operation.

We extend our concurrent language above with a new type τ **event**, which represents events. Intuitively, a value of type τ **event** is a synchronous operation that will yield a value of type τ when it is synchronized upon.

Given a channel c of type τ **chan**, we can construct an event of type τ **event** for the primitive synchronous operators recvEvt and sendEvt. Intuitively, if event v is constructed using recvEvt from c, then v will have a value of type τ available once a receive event of c occurs, i.e., once we have received a value on c. Before receiving a value on c, event v does not have a value of type τ available. The primitive operation sync v synchronizes on event v: it will block until event v has a value available, and will then evaluate to the received value.

Similarly, if event w is constructed using sendEvt a to c, then w will have a value of type **unit** available once value a has been successfully sent over channel c. And, sync w will block until the send has occurred (and will then evaluate to the unit value $()$).

Finally, given two events v and w, both of type τ **event**, primitive operation choose v w produces a new event that has a value available when either v or w has a value available. For example, synchronizing on choose (recvEvt from c) (recvEvt from d) will block until a value is received on either channel c or on channel d.

The syntax of expressions and types is extended as follows.

$$e ::= \cdots \mid \text{recvEvt from } e \mid \text{sendEvt } e_1 \text{ to } e_2 \mid \text{sync } e \mid \text{choose } e_1\ e_2$$
$$\tau ::= \cdots \mid \tau\ \textbf{event}$$

The additional typing rules for events are as follows.

$$\frac{\Gamma \vdash e : \tau\ \textbf{chan}}{\Gamma \vdash \text{recvEvt from } e : \tau\ \textbf{event}} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \textbf{chan}}{\Gamma \vdash \text{sendEvt } e_1 \text{ to } e_2 : \textbf{unit event}}$$

$$\frac{\Gamma \vdash e : \tau\ \textbf{event}}{\Gamma \vdash \text{sync } e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \textbf{event} \qquad \Gamma \vdash e_2 : \tau\ \textbf{event}}{\Gamma \vdash \text{choose } e_1\ e_2 : \tau\ \textbf{event}}$$

Before we see the formal semantics for events, let's consider some examples.

The following program sends 42 on a channel, does some other stuff, and then synchronizes on the event.

```
let c = newchan_int in
spawn (do some long computation; recv from c);
let w = sendEvt 42 to c in
do some other computation;
sync w;
do more stuff after the send finishes
```

Here's a more interesting example, where the main thread spawns three of threads that search (very inefficiently!) for a multiple of 42, and then the main thread blocks until one of the spawned threads succeeds.

$$
\begin{array}{l}
\mathsf{let}\ f = \lambda c{:}\,\textbf{int chan}.\ \mu g.\ \lambda x : \textbf{int}.\ \mathsf{if}\ ((x \bmod 42) = 0)\ (\mathsf{send}\ x\ \mathsf{to}\ c)\ (g\ (x+3))\ \mathsf{in} \\
\mathsf{let}\ c1 = \mathsf{newchan}_{\textbf{int}}\ \mathsf{in} \\
\mathsf{let}\ c2 = \mathsf{newchan}_{\textbf{int}}\ \mathsf{in} \\
\mathsf{let}\ c3 = \mathsf{newchan}_{\textbf{int}}\ \mathsf{in} \\
\mathsf{spawn}\ (f\ c1\ 43); \\
\mathsf{spawn}\ (f\ c2\ 44); \\
\mathsf{spawn}\ (f\ c3\ 45); \\
\mathsf{let}\ d = \mathsf{choose}\ (\mathsf{recvEvt\ from}\ c1)\ (\mathsf{choose}\ (\mathsf{recvEvt\ from}\ c2)\ (\mathsf{recvEvt\ from}\ c3))\ \mathsf{in} \\
\mathsf{sync}\ d
\end{array}
$$

The choose operation allows the main thread to synchronize on any of the three threads, i.e., find the first solution to the search problem.

2.1 Translational semantics

We define the semantics of the new events and primitives by a translation to our calculus with blocking sends and receives. The translation is type directed, as we need to know the types of expressions in order to perform the translation correctly (specifically, in order to add the correct type annotations). We write $\mathcal{T}[\![e{:}\,\tau]\!]$ to indicate that we are translating expression e of type τ.

The idea behind the translation is as follows. An event of type τ **event** is translated to a value with type (τ **chan**) **chan**, i.e., a channel that we will send a τ **chan** value over to it. The guarantee is that when the event occurs (i.e., when the value of type τ is available), the value will be sent over the channel that was passed in. Thus, we translate the sync e operation to create a new channel of τ, and pass that to the event, and then wait to receive a value over that channel. For a choose primitive, we create a channel that will accept a τ-channel, and send that τ-channel to both of the events.

For convenience we use $e_1; e_2$ as shorthand for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ where $x \notin FV(e_2)$.

$$
\begin{array}{rl}
\mathcal{T}[\![\mathsf{recvEvt\ from}\ e{:}\,\tau\ \textbf{event}]\!] = & \mathsf{let}\ c = \mathcal{T}[\![e{:}\,\tau\ \textbf{chan}]\!]\ \mathsf{in} \\
& \mathsf{let}\ d = \mathsf{newchan}_{\tau\ \textbf{chan}}\ \mathsf{in} \\
& \mathsf{spawn}\ (\mathsf{let}\ x = \mathsf{recv\ from}\ c\ \mathsf{in\ let}\ y = \mathsf{recv\ from}\ d\ \mathsf{in\ send}\ x\ \mathsf{to}\ y); \\
& d \\
\\
\mathcal{T}[\![\mathsf{sendEvt}\ e_1\ \mathsf{to}\ e_2{:}\,\tau\ \textbf{event}]\!] = & \mathsf{let}\ v = \mathcal{T}[\![e_1{:}\,\tau]\!]\ \mathsf{in} \\
& \mathsf{let}\ c = \mathcal{T}[\![e_2{:}\,\tau\ \textbf{chan}]\!]\ \mathsf{in} \\
& \mathsf{let}\ d = \mathsf{newchan}_{\textbf{unit chan}}\ \mathsf{in} \\
& \mathsf{spawn}\ (\mathsf{let}\ x = \mathsf{send}\ v\ \mathsf{to}\ c\ \mathsf{in\ let}\ y = \mathsf{recv\ from}\ d\ \mathsf{in\ send}\ x\ \mathsf{to}\ y); \\
& d \\
\\
\mathcal{T}[\![\mathsf{sync}\ e{:}\,\tau]\!] = & \mathsf{let}\ d = \mathcal{T}[\![e{:}\,\tau\ \textbf{event}]\!]\ \mathsf{in} \\
& \mathsf{let}\ c = \mathsf{newchan}_{\tau}\ \mathsf{in} \\
& \mathsf{send}\ c\ \mathsf{to}\ d; \\
& \mathsf{recv\ from}\ c \\
\\
\mathcal{T}[\![\mathsf{choose}\ e_1\ e_2{:}\,\tau\ \textbf{event}]\!] = & \mathsf{let}\ c = \mathcal{T}[\![e_1{:}\,\tau\ \textbf{event}]\!]\ \mathsf{in}
\end{array}
$$

let $d = \mathcal{T}[\![e_2 : \tau$ **event**$]\!]$ in
let $x =$ newchan$_{\tau\ \textbf{chan}}$ in
spawn $($let $y =$ recv from x in
spawn $($send y to $c)$;
send y to $d)$;
x