Recitation 13: Concurrency

15-312: Foundations of Programming Languages

Serena Wang

April 18th, 2018

1 Modeling Concurrency with Programming Languages

We have previously modelling parallelism in programming languages, and we will now discuss concurrency using another language model. Keep in mind that parallelism is not concurrency!

Parallelism	Concurrency
Deterministic behavior	Allows non-determinism
Focus on dependencies	Focus on synchronization
Abstraction	Implementation

Process calculus is a freestanding language model for concurrency and is the central machinery behind Concurrent Algol, or **CA**. Process calculus is defined by processes, which compute things, and events, which let processes communicate with each other. Chapter 39 in the textbook explains process calculus in detail, but the key idea is that processes in process calculus are identified up to structural congruence. This means that we can re-order processes at will, which is very helpful for implementing non-deterministic dynamic rules. By extending Modernized Algol with process calculus, we get **CA**!.

2 Concurrent Algol

Concurrent Algol, or **CA**, incorporates the typing system and modal distinction from **MA** with the mechanisms of process calculus for dealing with concurrent computation. For simplicity, we don't have assignables anymore in **CA**, but we can actually still define free assignables in **CA** using processes as cells (see the end of Chapter 40 for details).

The syntax for **CA** is derived from **MA**, but without assignables and with a new syntactic level of processes for representing the global state of a program:

Typ	τ	::=	$\texttt{cmd}$	command
Exp	e	::=	$\texttt{cmd}(m)$	encapsulation
Cmd	m	::=	$\texttt{ret}\, e$	return
			$\texttt{bnd}\, x \leftarrow e\, ;\, m$	sequence
Proc	p	::=	$\texttt{stop}$	idle
			$\texttt{run}(m)$	atomic
			$p_1 \otimes p_2$	concurrent
			$\nu\, a \sim \tau . p$	creation of new channels
Signature	Σ	::=	$\cdot$	empty
			$\Sigma, a \sim \tau$	type of values in channel a
Action	α	::=	ϵ	none
			$a\, !\, e$	send
			$a\, ?\, e$	receive

All of the **MA** constructs remain, but we now also have the ability to define and run commands that declare variables and assign to them. The modal separation ensures that expressions do not directly cause the effects of variable state update, but commands when they are executed may do so. **CA** thus introduces new concepts such as processes, messages, actions, events, and dynamic classfication in combination with the previously seen separation between pure and impure computation.

In **CA**, processes are the outermost level of computation. You an execute commands in processes and have concurrent subprocesses. You can also create new channels within processes with $\nu\, a \sim \tau . p$ such that the new channel a can used only within process p.

3 Shared Skeleton

There are different approaches that we can take to construct Concurrent Algol depending on the type of communication we want in our concurrent system. However, they all extend this shared skeleton to provide different types of communication. In the shared skeleton, actions then refer to sending and receiving messages at runtime.

3.1 Statics

For the statics of **CA**, we have four judgments:

$\Gamma \vdash_\Sigma e : \tau$	expression typing
$\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau$	command typing
$\Gamma \vdash_\Sigma p\ \mathsf{proc}$	process formation
$\Gamma \vdash_\Sigma \alpha\ \mathsf{action}$	action formation

We have these rules for process formation. Again, processes are identified up to structural congruence.

$$\frac{}{\Gamma \vdash_\Sigma \texttt{stop}\ \mathsf{proc}} \qquad \frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{run}(m)\ \mathsf{proc}} \qquad \frac{\Gamma \vdash_\Sigma p_1\ \mathsf{proc} \quad \Gamma \vdash_\Sigma p_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma p_1 \otimes p_2\ \mathsf{proc}} \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} p\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a \sim \tau . p\ \mathsf{proc}}$$

We have these rules for action formation. Notice that messages don't necessarily need to contain values.

$$\frac{}{\Gamma \vdash_\Sigma \epsilon \text{ action}} \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau} a \,!\, e \text{ action}} \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau} a \,?\, e \text{ action}}$$

The statics for the expressions and commands stay the same as in **MA**.

3.2 Dynamics

For the dynamics of **CA**, since we can now step expressions, step processes, and spawn processes in commands, we have three judgments. Notice that processes and commands can produce actions but expressioncs cannot.

$$\begin{array}{cc} e \underset{\Sigma}{\longmapsto} e' & \text{expression stepping} \\ p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p' & \text{process stepping} \\ m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m' & \text{command stepping} \end{array}$$

We have these rules for stepping processes.

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m'}{\texttt{run}(m) \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \texttt{run}(m')} \qquad \frac{m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\Sigma'\{m' || p\}}{\texttt{run}(m) \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\Sigma'\{\texttt{run}(m') \otimes p\}} \qquad \frac{e \; \texttt{val}_\Sigma}{\texttt{run}(\texttt{ret}(e)) \overset{\epsilon}{\underset{\Sigma}{\longmapsto}} \texttt{stop}} \qquad \frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1'}{p_1 \otimes p_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \otimes p_2}$$

$$\frac{p_1 \overset{a!e}{\underset{\Sigma}{\longmapsto}} p_1' \quad p_2 \overset{a?e}{\underset{\Sigma}{\longmapsto}} p_2'}{p_1 \otimes p_2 \overset{\epsilon}{\underset{\Sigma}{\longmapsto}} p_1' \otimes p_2'} \qquad \frac{p \overset{\alpha}{\underset{\Sigma, a\sim\tau}{\longmapsto}} p' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a \sim \tau . p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\, a \sim \tau . p'}$$

Notice in the second rule that the command m created a new channel contained in the signature Σ' and also spawned a new process p. We then still run the process for m' and run the new process p in parallel.

In the third rule, we see p_1 send a message that p_2 receives. Because of this rule, we have synchronous communication in this system. We need p_1 and p_2 to be on the same page before we can send or receive a message.

In the last rule, we step the process p within the signature where we can use the channel a. However, we check that the action α does not use channel a to prevent stealing of messages. Any message that gets leaked through α can only use channels that already exist in the external signature Σ. Since a is only bound within p, the only process that can use the channel a is p. Thus, if p has some external action (sending or receiving a message), that action should not use channel a.

4 Dynamic Classification for Messages

Messages are values that are sent or received by processes. A message consists of a channel, which is its class and a payload, which is a value of the type associated with the channel. Messages thus have the type `clsfd`, and channels are classes of messages. An event is the type of combined channels.

Typ	τ	$::=$	$\texttt{clsfd}$	type of values in messages
			$\texttt{cls}(\tau)$	type of channels
Exp	e	$::=$	$\&a$	channel
			$e_1 \cdot e_2$	message
			$\texttt{isof}(e_0; e_2; x.e_2; e_3)$	check message

We now have expressions for dynamically creating classes and classfied values of type `clsfd`. Using a fresh channel a created by $\nu\, a \sim \tau.p$, we can use the $\&\, a$ expression to refer to that channel, and we can use $\&a \cdot e$ to attach the tag a with the expression e. $\texttt{isof}(e_0; e_2; x.e_2; e_3)$ checks whether e_0 is tagged with the class given by e_1, evaluating e_2 if so and e_3 otherwise. We can thus use $e_1 \cdot e_2$ to create payloads for messages and use $\texttt{isof}(e_0; e_2; x.e_2; e_3)$ to check message payloads. Refer to Chapter 33 for full statics rules for dynamic classification in general.

5 Synchronous Broadcast Communication

In addition to all of the constructs defined above, we also add some new commands for sending and receiving messages to get synchronous broadcast communication.

Cmd	m	$::=$	$\texttt{spawn}(e)$	spawn
			$\texttt{emit}(e)$	emit message
			$\texttt{acc}$	accept message
			$\texttt{newch}\{\tau\}$	new channel

To have synchronous communication, we use commands to send messages. In asynchronous communication, we would use processes to send messages. The key point is that a command can send or receive messages on a channel if and only if they have access to that class of messages. By dynamically generating tags for message payloads, we create a secret-sharing system by wrapping and unwrapping expressions inside message payloads.

5.1 Statics

Refer to Chapter 33 in the textbook for full statics rules for dynamic classification in general. The typing rules for expressions in the synchronous broadcast version of **CA** follows the statics rules for dynamic classification exactly.

The statics rules for commands in this programming language are as follows:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\texttt{unit})}{\Gamma \vdash_\Sigma \texttt{spawn}(e) \mathrel{\dot\sim} \texttt{unit}} \qquad \frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma \texttt{emit}(e) \mathrel{\dot\sim} \texttt{unit}} \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{acc} \mathrel{\dot\sim} \texttt{clsfd}} \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{newch}\{\tau\} \mathrel{\dot\sim} \texttt{cls}(\tau)}$$

Note that since all messages have type `clsfd`, you can only send and receive messages of type `clsfd`.

5.2 Dynamics

Since commands can now spawn processes, we have a new judgment for when the command m transitions to the command m' while creating new channels Σ' and new processes p'.

$$m \underset{\Sigma}{\overset{\alpha}{\Rightarrow}} \nu\,\Sigma'\{m' \otimes p'\}$$

We have these rules for stepping commands in the synchronous broadcast version of **CA**.

$$\frac{m \underset{\Sigma}{\overset{\alpha}{\Rightarrow}} m'}{\texttt{spawn}(\texttt{cmd}(m)) \underset{\Sigma}{\overset{\epsilon}{\Rightarrow}} \nu\,\cdot\,\{\texttt{ret}\,\langle\rangle \otimes \texttt{run}(m)\}} \quad \frac{e \longmapsto e'}{\texttt{spawn}(e) \underset{\Sigma}{\overset{\epsilon}{\Rightarrow}} \texttt{spawn}(e')} \quad \frac{}{\texttt{emit}(a \cdot e) \xRightarrow[\Sigma,a\sim\tau]{a!e} \texttt{ret}\,\langle\rangle}$$

$$\frac{a \cdot e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\texttt{acc} \xRightarrow[\Sigma,a\sim\tau]{a?e} \texttt{ret}\,a \cdot e} \quad \frac{}{\texttt{newch}\{\tau\} \underset{\Sigma}{\overset{\epsilon}{\Rightarrow}} \nu\,a{\sim}\tau\{\texttt{ret}\,\&a \otimes \texttt{stop}\}}$$

Note that once we get a command inside `spawn`, we create a new process for running that encapsulated lazy computation. `spawn`(`cmd`(m)) doesn't create new any channels, however, so there are no new channels to put in a signature.

`ret` $\langle\rangle$ means that we stop executing the command and return unit. Thus, we step to this once we spawn a new process or finish sending a message.

We also see in the rules for `emit`($a \cdot e$) and `acc` that you must have a in your signature (i.e. know the channel a) to be able to send or receive a message on channel a. Since only commands within processes that have access to a can send messages on a, we are able to maintain an integrity invariant for that channel. Since only commands within processes with access to a can listen to that channel, we also have a confidentiality property for the channel a.

5.3 Safety

The preservation theorem for **CA** ensures that well-typed programs remain well-typed during execution. The actual proof for this theorem also requires a lemma about command execution, which you can see in Chapter 40 of the textbook, and some strengthening to deal with actions.

Theorem 40.2 (Preservation). If $\vdash_\Sigma p\ \mathsf{proc}$ and $p \underset{\Sigma}{\overset{\alpha}{\longmapsto}} p'$, then $\vdash_\Sigma p'\ \mathsf{proc}$.

However, our preservation theorem does not guarantee progress if there's a process waiting to receive a message on an empty channel or if there's a process trying to send a message on a channel no one is listening to. Thus, our progress theorem states that the only way to "get stuck" is to wait for another process to have a complementary action on some external channel.

Theorem 40.2 (Progress). If $\vdash_\Sigma p\ \mathsf{proc}$, then either $p \equiv \texttt{stop}$ or $p \equiv \nu\,\Sigma'\{p'\}$ where $p' \xmapsto[\Sigma,\Sigma']{\alpha} p''$ for some $\vdash_{\Sigma,\Sigma'} p''\ \mathsf{proc}$ and some $\vdash_{\Sigma,\Sigma'} \alpha\ \mathsf{action}$.

6 Asynchronous Broadcast Communication

To allow asynchronous communication in **CA**, we simply add a way to send messages in processes.

$$\mathsf{Proc}\quad p\quad ::=\quad \texttt{send}[a](e)\quad \text{send message}$$

We still have everything previously mentioned in the shared skeleton and synchronous broadcast version of **CA**. The main change is that because we allow asynchronous communication, we can now step a process that sends a message without waiting for a complementary action for receiving that message.

6.1 Statics

$$\frac{\vdash_{\Sigma,a\sim\tau} e : \tau}{\vdash_{\Sigma,a\sim\tau} \texttt{send}[a](e)\ \mathsf{proc}}$$

Again, you can only send messages on channel a if you have access to channel a (i.e. having a in Σ).

6.2 Dynamics

$$\frac{}{\texttt{send}[a](e) \overset{a!e}{\underset{\Sigma}{\longmapsto}} \texttt{stop}} \qquad \frac{a \cdot e\ \mathsf{val}_\Sigma}{\texttt{emit}(a\cdot e) \overset{\epsilon}{\underset{\Sigma}{\Rightarrow}} \nu \cdot \{\texttt{ret}\ \langle\rangle \otimes \texttt{send}[a](e)\}}$$

The first rule above tells us that sending messages is now asynchronous. $\texttt{send}[a](e)$ can step to a stopped process without waiting for another process to receive a message on the same channel a.

What does the second dynamics rule tell us? Instead of the `emit` command resulting in an action explicitly, the `emit` command now spawns a concurrent process that sends the message.

7 Synchronous Selective Communication

Broadcast communication does not give us a way to receive messages of just a particular class. If we want an abiliity to wait for specific events, we need to add some new constructs to **CA**. Specifically, we'll take out our receive commands and replace them with receive expressions. This version of **CA** matches the programming language that you'll work with and implement in Assignment 6 almost exactly. The version of **CA** in Assignment 6 also includes channel references, whose rules you can see in the writeup.

$$\begin{array}{llll}
\mathsf{Typ} & \tau & ::= & \texttt{event}(\tau) & \text{type of combined channels} \\
\mathsf{Exp} & e & ::= & \texttt{rcv}[a] & \text{receive event} \\
 & & & \texttt{never}\{\tau\} & \text{null event} \\
 & & & \texttt{or}(e_1; e_2) & \text{binary disjunction event} \\
\mathsf{Cmd} & m & ::= & \texttt{sync}(e) & \text{wait for event}
\end{array}$$

$\texttt{rcv}[a]$ allows us to receive messages on channel a, while $\texttt{or}(e_1;e_2)$ allows us to listen to multiple channels if the messages on those channels have the same type. $\texttt{never}\{\tau\}$ is for when you don't listen to any channel. This can technically be left out so that you're always listening to at least one channel for an event, and it won't be part of the language for Assignment 6.

7.1 Statics

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \mathrel{\dot{\sim}} \tau} \qquad \frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{rcv}[a] : \mathtt{event}(\tau)} \qquad \frac{}{\Gamma \vdash_\Sigma \mathtt{never}\{\tau\} : \mathtt{event}(\tau)}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{event}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{or}(e_1; e_2)}$$

Note that the event type for $\mathtt{rcv}[a]$ depends on the type of the channel a in the signature.

7.2 Dynamics

$$\frac{}{\mathtt{rcv}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \qquad \frac{}{\mathtt{never}\{\tau\}\ \mathsf{val}_\Sigma} \qquad \frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{val}_\Sigma}{\mathtt{or}(e_1; e_2)\ \mathsf{val}_\Sigma} \qquad \frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\longmapsto} \mathtt{or}(e_1'; e_2)}$$

Note that for $\mathtt{rcv}[a]$ to be a value, the channel a must be in the signature.

$$\frac{e\ \mathsf{val}_{\Sigma, a \sim \tau} \quad \vdash_{\Sigma, a \sim \tau} e : \tau}{\mathtt{sync}(\mathtt{rcv}[a]) \xRightarrow[\Sigma]{a?e} \mathtt{ret}(e)} \qquad \frac{\mathtt{sync}(e_1) \xRightarrow[\Sigma]{\alpha} m_1}{\mathtt{sync}(\mathtt{or}(e_1; e_2)) \xRightarrow[\Sigma]{\alpha} m_1} \qquad \frac{\mathtt{sync}(e_2) \xRightarrow[\Sigma]{\alpha} m_2}{\mathtt{sync}(\mathtt{or}(e_1; e_2)) \xRightarrow[\Sigma]{\alpha} m_2}$$

The first rule tells us that an acceptance on channel a may synchronize only with messages classified by a. We shouldn't just receive any message on any channel. The synchronization steps only to expressions tagged with the class a (i.e. expressions sent on the channel a).

The second and third rules tells us that either event between the two choices can lead to an action. The synchronization is satisfied if we receive a message in either of the two choices.