CS442
Module 9: Concurrent Programming

University of Waterloo

Winter 2025

> "Concurrency is the most extreme form of programming. It's like white-water rafting without the raft or sky diving without the parachute."
>
> — Peter Buhr

1 Concurrent Programming

We've developed two core calculi and looked at four exemplar programming languages, but we're as yet missing one of the most important characteristics of modern computing: *concurrency*. First, let's clarify some terms, because the concurrent programming community are quite picky about these terms:

- *Parallelism* is the *phenomenon* of two or more things—presumably, two computations—actually happening at the same time.
- *Concurrency* is the *experience* of two or more things—such as applications, tasks, or threads—appearing to happen at the same time, whether or not they actually are.

For instance, an average computer in the 1990s had only one CPU and one core, and so had no parallelism, but could still run multiple applications "at the same time", and so still had concurrency. In this case, concurrency is achieved by quickly switching which task the CPU is concerned with between the various running programs. Most home computers of the 1980s had neither: we could only run one program at a time, and we liked it; it builds character! It is also possible for a system to be parallel without being concurrent: for instance, a compiler may optimize an imperative loop into a specialized parallel operation on a particular CPU, but this is only visible to the programmer as a speed boost, so the programmer's experience still lacks concurrency. And, of course, a system can have both: on a modern, multi-core CPU, one uses concurrency to take advantage of the parallelism, by running multiple programs or threads on their multiple cores. The concurrent tasks appear to happen at the same time because they *do* happen at the same time.

In this module, we concern ourselves with concurrency, not parallelism. Whether two things *appear* to happen at the same time has relevance to our formal semantic rules. Whether they *actually* happen at the same time is just an implementation detail.

The domain of concurrent programming has changed dramatically over time, in response to growing parallelism. Early exploration into concurrency was theoretical, and then became practical with networks. When multi-core CPUs started becoming the norm, concurrency rose in importance again, because it is often impossible to take advantage of real parallelism without concurrency.

The core idea of a concurrent system is that there are multiple tasks which can communicate with each other, and computation can proceed in any of them at any rate. This is contrary to the behavior of the λ-calculus or the Simple Imperative Language, and contrary to all of our exemplar languages (at least, as far as we've investigated them), so we will need both a new formal language and a new exemplar language for concurrency. In practice, of

course, in the same way that many languages which are not fundamentally object oriented have picked up object-oriented features, most languages which are not fundamentally concurrent have picked up at least some kind of concurrent features.

In a formal model of concurrency, we need a way of expressing multiple simultaneous tasks, and, as usual, a way of taking a step. Unlike our previous formal semantics, we *want* this semantics to be non-deterministic: the fact that any of multiple tasks may proceed is the essence of concurrency. However, we will not define our semantics as truly parallel: a step will involve only one task taking a step, with the non-determinism only affecting *which* task that is.

Another issue to be addressed is *models* of concurrency, i.e., how a concurrent language presents multiple tasks. Models of concurrency are defined across two dimensions: how one *specifies* multiple tasks, and how those multiple tasks *communicate*. Specification comes down to what structures a language provides for a program to create tasks, and often, to specify real parallelism as well. Options include threads of various sorts, actors, processes, and many others, but all of these terms are imprecise and ambiguous. We will put little focus on the specification of concurrency beyond how it's done in our formal model and exemplar language, and instead will focus, as usual, on semantics.

The other dimension is communication, which falls largely into two forms: *shared-memory* concurrency and *message-passing* concurrency. In shared-memory concurrency, either the entire heap or some portion of the heap is shared between multiple tasks. In our terms, all tasks have the same Σ, and if they share any labels, they can see changes made by other tasks. However, because multiple tasks can compute at the same time, there may not be any guarantee that one task's write to a location in Σ occurs before another task's read. Other mechanisms, such as locks, are required to guarantee ordering.

In message-passing concurrency, two new primitive operations are introduced: *sending* a message and *waiting* for a message. A task which is waiting for a message cannot proceed until another task sends it a message. A task may send a message at any time, but to do so, must have a way of communicating with the target task. Thus, message-passing concurrency requires some form of *message channels*, by which two tasks can arrange to exchange messages. Ordering is guaranteed by the directionality of message passing; a waiting task will not proceed until a sending task sends a message.

> **Aside:** Shared-memory and message-passing concurrency are equally powerful, and in fact, either can be rewritten in terms of the other. However, this rewriting is fairly unsatisfying: shared memory can be rewritten in terms of message passing by imagining Σ itself as a "task" that expects messages instructing it to read and write certain memory locations. Equivalently, we can consider a modern CPU as sending messages to the memory bus, rather than simply reading and writing memory. As a practical matter, message-passing implementations tend to be slower than shared-memory implementations of the same algorithm.

This course is not intended to compete with CS343 (Concurrent and Parallel Programming), so will take a very language-focused view of concurrency. That course uses concurrency to solve problems; in this course, concurrency is only a cause of problems.

2 π-Calculus

In the λ-calculus, we built a surprising amount of functionality around abstractions: with only three constructs in the language (abstractions, applications, and variables), we could represent numbers, conditionals, and ultimately, anything computable. π-calculus (The Pi Calculus) takes a similar approach to message-passing concurrency. The only structures are concurrency, communication, replication, and names, but these will be sufficient to build any computation. Notably lacking is functions (or abstractions). π-calculus itself was developed by Robin Milner[1], Joachim Parrow, and David Walker in *A Calculus of Mobile Processes*[1], but it was the culmination of a long line of development of calculi of communicating processes, to which Uffe Engberg and Mogens Nielsen also made significant contributions.

We will look at the behavior of π-calculus, but will not discuss how to encode complex computation into π-

[1] The same Milner of Hindley-Milner type inference.

calculus concurrency. π-calculus is Turing-complete, but like the λ-calculus, it's more common to layer other semantics on top of it than to take advantage of its own computational power.

A π-calculus program consists of any number of concurrent tasks, called *processes*, separated by pipes (|). Those tasks can be grouped by parentheses. Each process can *create* a channel, *receive* a message on a channel, *send* a message on a channel, *replicate* processes, or *terminate*. The only values in π-calculus are channels, so any encoding of useful information also needs to be done with channels, and the only value you can send over a channel is another channel. Channels are also called "names", because they are simply named by variables; two processes must agree on a name in order to communicate (as well as another restriction which we will discuss soon).

The syntax of π-calculus is as follows, presented in BNF, with $\langle program \rangle$ as the starting non-terminal:

$$
\begin{aligned}
\langle program \rangle &::= \langle program \rangle \text{ “|” } \langle program \rangle \\
&\mid \langle receive \rangle \\
&\mid \langle send \rangle \\
&\mid \langle restrict \rangle \\
&\mid \langle replicate \rangle \\
&\mid \langle terminate \rangle \\
\langle receive \rangle &::= \langle var \rangle \, (\langle var \rangle). \, \langle program \rangle \\
\langle send \rangle &::= \overline{\langle var \rangle} \text{ “⟨” } \langle var \rangle \text{ “⟩” } . \, \langle program \rangle \\
\langle restrict \rangle &::= (\nu \, \langle var \rangle) \, \langle program \rangle \\
\langle replicate \rangle &::= ! \, \langle program \rangle \\
\langle terminate \rangle &::= 0 \\
\langle var \rangle &::= a|b|c| \cdots
\end{aligned}
$$

Note that ν is the Greek letter nu, not the Latin/English letter 'v', because somebody decided that using confusing, ambiguous Greek letters was acceptable; we will avoid using v as a variable for this reason. Like in the λ-calculus, we will actually be more lax in our use of variable names than this BNF suggests, for clarity. Like in the Simple Imperative Language, this is assumed to be an abstract syntax, and we will add parentheses to disambiguate as necessary. The pipe (|) in $\langle program \rangle$ has the lowest precedence, so, for instance, $x(y).0|z(a).0$ is read as $(x(y).0)|(z(a).0)$, not $x(y).(0|z(a).0)$.

Unfortunately, π-calculus uses several of the symbols which we use in BNF as well, which we've surrounded in quotes to separate them from BNF. π-calculus also uses an overline, which is shown in the BNF above, but is at the very least unusual. Here are two small examples to clarify the syntax. The following snippet receives a message on the channel x, into the variable y, before proceeding with the process P:

$$x(y).P$$

The following snippet sends y on the channel x, before proceeding with the process P:

$$\overline{x} \, \langle y \rangle \, .P$$

As discussed, a program consists of a number of processes, separated by pipes. Each process is itself a program, so the distinction is just usage. We will use the term "process" to refer to any construction *other* than the composition of multiple programs with a pipe, so that a program can be read as a list of processes.

A program proceeds through its processes sending and receiving messages on channels until they terminate. For instance, this program consists of two processes, of which the first sends the message h to the second, and the second then attempts to pass that h along on another channel:

$$\overline{x} \, \langle h \rangle \, .0 | x(y).\overline{z} \, \langle y \rangle \, .0$$

The first process is $\overline{x} \, \langle h \rangle \, .0$, which consists of a send of h over the channel x, and then termination of the process (0). The second process is $x(y).\overline{z} \, \langle y \rangle \, .0$, which consists of a receive of y from the channel x, then a send of y over

the channel z, then termination. Programs in π-calculus proceed by sending and receiving messages; in this case, the program can proceed, because a process is trying to send on x, and another process is trying to receive on x. After sending that message, the program looks like this:

$$0|\overline{z}\,\langle h\rangle\,.0$$

Like in λ-calculus applications, message receipt works by substitution, so the y was substituted for the received message, h. We will not formally define substitution for π-calculus; it is fairly straightforward. A terminating process can simply be removed, so the next step is as follows:

$$\overline{z}\,\langle h\rangle\,.0$$

This program cannot proceed, because no process is prepared to receive a message on the channel z.

Consider this similar program:

$$\overline{x}\,\langle h\rangle\,.z(a).0|x(y).\overline{z}\,\langle y\rangle\,.0|x(y).0$$

This time, we have three processes. The first sends the message h over the channel x, then receives a message on the channel z, then terminates. The second is identical to our original second process: it receives a message on x, then sends it back on z. The third message receives a message on x, then immediately terminates. This program can proceed, because a process is sending on x and a process is prepared to receive on x. But, which process receives the message? π-calculus is non-deterministic, so the answer is that *either* process may receive the message. Both ways for the program to proceed are valid. In this case, these two reductions are both valid:

$\Rightarrow z(a).0|\overline{z}\,\langle h\rangle\,.0|x(y).0$

$\Rightarrow 0|0|x(y).0$

$\Rightarrow 0|x(y).0$

$\Rightarrow x(y).0$

$\Rightarrow z(a).0|x(y).\overline{z}\,\langle y\rangle\,.0|0$

$\Rightarrow z(a).0|x(y).\overline{z}\,\langle y\rangle\,.0$

The first sequence of reductions occurs if the message is received by the second process, and the third occurs if the message is received by the third process. The first sequence may seem more complete, since two of the three processes terminated, but both are valid. We are using $\Rightarrow$ informally for "takes a step", because we haven't yet formally defined $\rightarrow$, and we will discover that there's an extra complication to the definition of steps in π-calculus in Section 2.2.

Because name uniqueness is so important to communication, π-calculus also has a mechanism to *restrict* names. For instance, consider this rewrite of the above program:

$$(\nu x)(\overline{x}\,\langle h\rangle\,.z(a).0|x(y).\overline{z}\,\langle y\rangle\,.0)|x(y).0$$

It is identical to the previous program, except that the first two processes are nested inside of a *restriction*: (νx). A restriction is like a variable binding, in that it scopes the variable to its context: the x inside of the restriction is not the same as the x outside of the restriction. Unlike variable binding, however, it doesn't bind it to any particular value—remember, names are all there are in π-calculus, so there's nothing else it could be bound *to*—it merely makes for two distinct interpretations of the name. That's why it's called a restriction; it restricts the meaning of, in this case, x, within the restriction expression. Now, this program can only proceed like so:

$$\Rightarrow (\nu x)(z(a).0|\overline{z}\,\langle h\rangle\,.0)|x(y).0$$
$$\Rightarrow (\nu x)(0|0)|x(y).0$$
$$\Rightarrow (\nu x)(0)|x(y).0$$
$$\Rightarrow x(y).0$$

In the last step, we can remove a restriction when it is no longer restricting anything (i.e., when its contained process terminates). When a restriction is at the top level like this, it can always be rewritten by renaming variables to new, fresh names, so the above program is equivalent to the following program:

$$\overline{x'}\,\langle h\rangle\,.z(a).0|x'(y).\overline{z}\,\langle y\rangle\,.0|x(y).0$$

Aside: If your eyes are glazing over from π-calculus syntax, don't worry, you're not the only one. Something about π-calculus's use of overlines and ν and pipes makes it semantically dense and difficult to read. I have no advice to alleviate this; just be careful of the pipes and parentheses.

A restriction only applies to the variable it names, so processes within restrictions are still allowed to communicate with processes outside of restrictions:

$$\begin{aligned} & (\nu x)\overline{z}\left\langle h\right\rangle .0|z(y).\overline{x}\left\langle y\right\rangle .0 \\ \Rightarrow\ & (\nu x)0|\overline{x}\left\langle h\right\rangle .0 \\ \Rightarrow\ & \overline{x}\left\langle h\right\rangle .0 \end{aligned}$$

Substitution must be aware of restriction, because a restricted variable is distinct from the same name in the surrounding code. For instance:

$$(x(y).(\nu x).x(y).0)[z/x] = (z(y).(\nu x).x(y).0)$$

This exception is the same as is introduced by λ-abstractions in substitution for the λ-calculus.

The only messages that processes can send are names. Names are also the channels by which processes send messages. As a consequence, processes can send channels over channels. For instance, consider this program:

$$\overline{x}\left\langle z\right\rangle .0|x(y).\overline{y}\left\langle h\right\rangle .0|z(a).0$$

The first process will send the name z over the channel x. The second process is waiting to receive a message on the channel x. The third process is waiting to receive a message on the channel z, but there is no send on the channel z in the entire program. The third process cannot possibly proceed, but the first and second can, like so:

$$\Rightarrow 0|\overline{z}\left\langle h\right\rangle .0|z(a).0 \Rightarrow \overline{z}\left\langle h\right\rangle .0|z(a).0$$

The second process received a z on the channel x, as the variable y. But, it then proceeds to *send* on the channel y. Because message receipt works by substitution, that y has been substituted for z. This program can now proceed, by sending h to the third process (at which point both the second and third process terminate).

Restriction has an unusual interaction with sending channels. For instance, consider this program:

$$(\nu x)\overline{z}\left\langle x\right\rangle .x(y).0|z(a).\overline{a}\left\langle x\right\rangle .0$$

The first process is under a restriction for x, and the second process is not. But, the behavior of the first process is to send x over the channel z, and the second process is waiting to receive a message on the channel z. It's then going to send x back, but that x isn't the same x as the first process's x, because of the restriction. So, what happens if we send a restricted channel outside of its own restriction? How do we deal with these two conflicting x's? The answer is made clear by our statement that a restriction can always be rewritten by simply using a new name. In this case, this program can be rewritten like so:

$$\overline{z}\left\langle x'\right\rangle .x'(y).0|z(a).\overline{a}\left\langle x\right\rangle .0$$

From this state, the steps are clear:

$$\begin{aligned} & \Rightarrow x'(y)0|\overline{x'}\left\langle x\right\rangle .0 \\ & \Rightarrow 0|0 \\ & \Rightarrow 0 \end{aligned}$$

Finally, π-calculus supports process creation: a process may create more processes. There are actually two mechanisms of process creation. First, processes may simply be nested. For instance, consider this program:

$$x(y).(\overline{y}\left\langle h\right\rangle .0|\overline{y}\left\langle m\right\rangle .0)|f(a).\overline{z}\left\langle a\right\rangle .0|f(b).\overline{z}\left\langle b\right\rangle .0|\overline{x}\left\langle f\right\rangle .0$$

Note in particular the position of the parentheses: this program has *four* processes, not five! The first process is $x(y).(\overline{y}\langle h\rangle.0|\overline{y}\langle k\rangle.0)$. Although this process has the pipe which separates multiple processes within it, those are not two independent processes until this process has received a message on the channel x. Essentially, as soon as this process receives a message, it will split into two processes. This program can proceed as follows (this is not the only possible sequence):

$$\begin{aligned}
&\Rightarrow (\overline{f}\langle h\rangle.0|\overline{f}\langle m\rangle.0)|f(a).\overline{z}\langle a\rangle.0|f(b).\overline{z}\langle b\rangle.0\\
&\Rightarrow \overline{f}\langle h\rangle.0|\overline{f}\langle m\rangle.0|f(a).\overline{z}\langle a\rangle.0|f(b).\overline{z}\langle b\rangle.0\\
&\Rightarrow 0|\overline{f}\langle m\rangle.0|\overline{z}\langle h\rangle.0|f(b).\overline{z}\langle b\rangle.0\\
&\Rightarrow \overline{f}\langle m\rangle.0|\overline{z}\langle h\rangle.0|f(b).\overline{z}\langle b\rangle.0\\
&\Rightarrow 0|\overline{z}\langle h\rangle.0|\overline{z}\langle m\rangle.0\\
&\Rightarrow \overline{z}\langle h\rangle.0|\overline{z}\langle m\rangle.0
\end{aligned}$$

Exercise 1. Give another possible sequence for this example.

The other mechanism of process creation is *replication*. The ! operator creates an endless sequence of identical processes. For instance, consider the following program:

$$!x(a).0|\overline{x}\langle b\rangle.0|\overline{x}\langle c\rangle.0|\overline{x}\langle d\rangle.0$$

There are three processes trying to send on the channel x, but only one process with a receive on the channel x. However, the ! creates any number of the same process, so all three of the sending processes can proceed, in any order. This is one possible sequence:

$$\begin{aligned}
&\Rightarrow x(a).0|!x(a).0|\overline{x}\langle b\rangle.0|\overline{x}\langle c\rangle.0|\overline{x}\langle d\rangle.0\\
&\Rightarrow 0|!x(a).0|\overline{x}\langle b\rangle.0|0|\overline{x}\langle d\rangle.0\\
&\Rightarrow !x(a).0|\overline{x}\langle b\rangle.0|0|\overline{x}\langle d\rangle.0\\
&\Rightarrow !x(a).0|\overline{x}\langle b\rangle.0|\overline{x}\langle d\rangle.0\\
&\Rightarrow x(a).0|!x(a).0|\overline{x}\langle b\rangle.0|\overline{x}\langle d\rangle.0\\
&\Rightarrow 0|!x(a).0|0|\overline{x}\langle d\rangle.0\\
&\Rightarrow !x(a).0|0|\overline{x}\langle d\rangle.0\\
&\Rightarrow !x(a).0|\overline{x}\langle d\rangle.0\\
&\Rightarrow x(a).0|!x(a).0|\overline{x}\langle d\rangle.0\\
&\Rightarrow 0|!x(a).0|0\\
&\Rightarrow !x(a).0|0\\
&\Rightarrow !x(a).0
\end{aligned}$$

2.1 Concurrency vs. Parallelism

The astute reader may have noticed that we have described sequences of steps, with no true parallelism. For instance, consider the following program:

$$\overline{x}\langle a\rangle.0|\overline{y}\langle b\rangle.0|x(z).0|y(z).0$$

There are two possible steps this program can take—it can send a message on x or y—but we describe it as taking one or the other, not both at the same time. The concurrency comes from the lack of prioritization, and non-determinism: each of these two options is equally valid, and to consider how this program proceeds, we need to consider both possibilities. But, the concurrency is restricted by the nature of messages: only pairs of matching sends and receives can actually proceed.

Because concurrency models the *appearance* of multiple tasks happening simultaneously, in most cases, it is not necessary to model true parallelism. The most complex[2] formal models in the domain of concurrency and parallelism are models of parallel shared-memory architectures, and even they are formally descriptions of concurrency rather than parallelism, in that they model parallel action as a non-deterministic ordering.

2.2 Structural Congruence

Because processes may proceed in any order in π-calculus, $P|Q$ is not meaningfully distinct from $Q|P$. Similarly, $(\nu x)P|Q$ is not meaningfully distinct from $P[y/x]|Q$, where y is a new name, and α-equivalent programs are also indistinct.

In the λ-calculus, these equivalences mostly gave us a baseline for comparing things. In π-calculus, it would be difficult or impossible to define reduction without this equivalence, because of the non-deterministic ordering of steps.

This equivalence is defined formally as *structural congruence*, written as $\equiv$. That is, $P \equiv Q$ means that P is structurally congruent to Q. Structural congruence is reflexive, symmetric, and transitive.

The formal rules for structural congruence follow. Note that different presentations of π-calculus present slightly different but equivalent rules of structural congruence, so this may not exactly match other materials on the same topic.

Definition 1. (Structural congruence)

Let the metavariables P, Q, and R range over programs, and x and y range over names. Then the following rules describe structural congruence of π-calculus programs:

$$\text{C\_Alpha}\ \frac{P =_\alpha Q}{P \equiv Q} \qquad \text{C\_Order}\ \frac{}{P|Q \equiv Q|P} \qquad \text{C\_Nest}\ \frac{P \equiv P'}{P|Q \equiv P'|Q}$$

$$\text{C\_Paren}\ \frac{}{(P|Q)|R \equiv P|(Q|R) \equiv P|Q|R} \qquad \text{C\_Termination}\ \frac{}{0|P \equiv P}$$

$$\text{C\_Restriction}\ \frac{y \text{ is a fresh variable}}{(\nu x)P \equiv P[y/x]} \qquad \text{C\_Replication}\ \frac{}{!P \equiv P|!P}$$

The C_Alpha rule specifies that α-equivalence implies structural congruence, i.e., two α-equivalent programs are also structurally congruent. The C_Order and C_Nest rules allow us to reorder programs and apply structural congruence to subprograms. The C_Paren rule specifies that all concurrent processes are equivalent, and different placements of parentheses do not affect their composition, so we can remove parentheses at the top level of a program. The C_Termination rule describes termination in terms of equivalence: rather than termination being a step, we can describe a program with a terminating process as equivalent to a program without that process. The C_Restriction rule makes explicit our description of restriction as creating a fresh variable. Finally, the C_Replication rule makes replication a property of structural congruence, rather than a step: a replicating process is simply equivalent to a version with a replica, and thus, by the transitive property, equivalent to a version with any number of replicas.

Because of C_Termination, C_Restriction, and C_Replication, only sending and receiving messages is described as an actual step of computation. Everything else is structural congruence.

Note that C_Restriction does *not* allow us to remove all restrictions from a program, because restrictions may be nested inside of other constructs, and structural equivalence does not allow us to enter any other constructs. For instance, the program $x(y).(\nu z).0$ has no structural equivalent (except for α-renaming), because the restriction of z is nested inside of a receipt on the channel y.

[2] In this author's opinion.

2.3 Formal Semantics

With structural congruence, we may now describe the formal semantics of π-calculus.

Definition 2. (Formal semantics of π-calculus)

Let the metavariables P, Q, and R range over programs, and x, y, and z range over names. Then the following rules describe the formal semantics of π-calculus:

$$\textsc{Congruence}\ \frac{P \equiv Q \qquad Q \to Q' \qquad Q' \equiv R}{P \to R} \qquad\qquad \textsc{Message}\ \frac{}{\overline{x}\langle y\rangle.P | x(z).Q | R \to P | Q[y/z] | R}$$

Because of structural congruence, these two rules are all that is needed to define the semantics of π-calculus. By Congruence, P can reduce to R if P is equivalent to some Q which can reduce to some Q', and Q' is equivalent to R. That is, reduction is ambivalent to structural congruence in either its "from" or "to" state. Message describes the only actual reduction step in our semantics: if there is a process to send a message, and a process to receive a message on the same channel, then we may take a step in both, by removing the send from the sending process, removing the receipt from the receiving process, and substituting the variable in the receiving process with the value sent by the sending process.

Message itself is deterministic. The non-determinism in π-calculus is introduced by Congruence. Every program P has infinitely many structurally congruent equivalents. Some number of those structurally congruent programs are able to take steps with Message. Each of those is equivalently correct, and none has priority; all are valid reduction steps.

Consider our previous example:

$$!x(a).0 | \overline{x}\langle b\rangle.0 | \overline{x}\langle c\rangle.0 | \overline{x}\langle d\rangle.0$$

We may now define one possible sequence formally, with structural congruence and reduction:

	$!x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle c\rangle.0 \mid \overline{x}\langle d\rangle.0$	
$\equiv$	$x(a).0 \mid !x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle c\rangle.0 \mid \overline{x}\langle d\rangle.0$	(C_Replication)
$\equiv$	$\overline{x}\langle c\rangle.0 \mid x(a).0 \mid !x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle d\rangle.0$	(C_Order)
$\to$	$0 \mid 0 \mid !x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle d\rangle.0$	(Message)
$\equiv$	$!x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle d\rangle.0$	(C_Termination)
$\equiv$	$x(a).0 \mid !x(a).0 \mid \overline{x}\langle b\rangle.0 \mid \overline{x}\langle d\rangle.0$	(C_Replication)
$\equiv$	$\overline{x}\langle b\rangle.0 \mid x(a).0 \mid !x(a).0 \mid \overline{x}\langle d\rangle.0$	(C_Order)
$\to$	$0 \mid 0 \mid !x(a).0 \mid \overline{x}\langle d\rangle.0$	(Message)
$\equiv$	$!x(a).0 \mid \overline{x}\langle d\rangle.0$	(C_Termination)
$\equiv$	$x(a).0 \mid !x(a).0 \mid \overline{x}\langle d\rangle.0$	(C_Replication)
$\equiv$	$\overline{x}\langle d\rangle.0 \mid x(a).0 \mid !x(a).0$	(C_Order)
$\to$	$0 \mid 0 \mid !x(a).0$	(Message)
$\equiv$	$!x(a).0$	(C_Termination)

2.4 The Use of π-Calculus

In concurrent programming, most problems—at least, most that are unique to concurrent programming, and aren't just logic bugs—can be simplified to *happens-before* relationships. That is, with multiple processes able to perform tasks concurrently, you want to guarantee that some task happens before some other task. Concurrent systems are modeled in terms of π-calculus to prove these kinds of happens-before relationships.

For instance, let's say we want to verify that a given program always sends a message on channel x before sending a message on channel y. Here is a program that fails to guarantee such a relationship:

$$\overline{a}\langle x\rangle.\overline{b}\langle y\rangle.0|a(m).\overline{m}\langle h\rangle.0|b(n).\overline{n}\langle h\rangle.0|x(q).0|y(q).0$$

We can demonstrate this by showing a reduction that sends on y before sending on x:

$$\begin{aligned}
&\to\ \overline{b}\langle y\rangle.0|\overline{x}\langle h\rangle.0|b(n).\overline{n}\langle h\rangle.0|x(q).0|y(q).0\\
&\equiv\ \overline{b}\langle y\rangle.0|b(n).\overline{n}\langle h\rangle.0|\overline{x}\langle h\rangle.0|x(q).0|y(q).0\\
&\to\ 0|\overline{y}\langle h\rangle.0|\overline{x}\langle h\rangle.0|x(q).0|y(q).0\\
&\equiv\ \overline{y}\langle h\rangle.0|y(q).0|\overline{x}\langle h\rangle.0|x(q).0\\
&\to\ 0|0|\overline{x}\langle h\rangle.0|x(q).0\\
&\quad\ \text{(Premise violated)}
\end{aligned}$$

Proving that happens-before relationships hold is, of course, far more complicated, since it is impossible to enumerate the infinitely many possible structurally congruent programs. Luckily, C_Replication is the only case that can introduce infinite reducible programs, and the difference between them is uninteresting (only how many times the replicated subprogram has been expanded). So, in many cases, it is possible to enumerate all *interesting* reductions. If the program can reduce forever, then it is instead necessary to use inductive proofs for most interesting properties.

Generally, π-calculus is extended with other features to represent the actual computation that each process performs, rather than performing computation through message passing. For instance, the λ-calculus and π-calculus can be overlain directly by allowing processes which contain λ-applications to proceed as in the λ-calculus, while process pairs containing a matching send and receive can proceed as in π-calculus. Such combinations are often used for proving type soundness of concurrent languages.

3 Exemplar: Erlang

In π-calculus, we have found a formal semantics for message-passing concurrency. Although there are many programming languages with support for concurrency, and even many programming languages with support for message-passing concurrency, we're looking for an *examplar* of message-passing concurrency, not just a language with it as a feature. There is a stand-out example which is to message-passing concurrency as Smalltalk is to object orientation: Erlang.

Erlang[3] is a language built on the principle that "everything is a process". It was created in the late 1980s at Ericsson by Joe Armstrong, Robert Virding, and Mike Williams, to manage telecom systems. There were three primary goals in that context:

- that the system scale from single systems (where many processes would run on one computer) to distributed systems (where processes could be distributed across many computers) with little or no rewriting,
- that processes would be sufficiently isolated that faults in one process would not (necessarily) affect the rest of the system, and
- that individual processes could be replaced live in a running system, allowing for smooth upgrades without any downtime.

These goals led Erlang to a quite extreme design, whereby Erlang programs use processes in the same way as Smalltalk programs use objects. Nearly all compound data is bound in processes, and one interacts with processes by sending and receiving messages. Just like in π-calculus, one can create processes and send channels in messages, allowing sophisticated interactions.

In fact, unlike Smalltalk's objects, Erlang does support some primitive data types which are not processes. Integers, floating point numbers, tuples, lists, key-value maps, and Prolog-like atoms are all supported, and ports—Erlang's name for one end of a communication channel—are not themselves processes (how would one ever send

[3]Pronounced roughly like "air-lang" by people who know how to pronounce words, or "ur-lang" by the kind of troglodytes who pronounce "wiki" as "wick-y".

a message if message channels were themselves processes which were controlled by messages?). So, it's not quite true that everything is a process, but everything* is a process. In fact, it's perfectly possible to treat Erlang as a mostly-pure functional language and write totally non-concurrent code. However, we won't focus on that aspect of Erlang. Instead we'll look only at its concurrency features.

Although Erlang has its own unique syntax, by this point, you should be able to guess how most of it works. Like Prolog, variables in Erlang are named with capital letters, and atoms and functions are named with lower-case letters, but its behavior is otherwise more similar to a functional language than a logic language.

3.1 Modules

Erlang divides code into modules. We've avoided discussing modules for most of this course, as they're usually uninteresting for language semantics, but creating a process in Erlang requires modules, so we will briefly discuss them. An Erlang file is a module, and must start with a declaration of the module's name. For instance, a module named `sorter` begins as follows:

```
-module(sorter).
```

Most modules define some public functions and some private functions. Any functions which should be usable from other modules must be *exported*. For instance, if we define a function `merge` taking two arguments, we make that function visible like so:

```
-export([merge/2])
```

Note that functions in this context are named with their arity, in this case `2`, in the same fashion as Prolog, so `merge/2` is a function named `merge` that takes two arguments.

Functions in the same module can be called with only their name:

```
merge([1, 2, 3], [2, 2, 4])
```

Functions in other modules need to be prefixed with the target module:

```
sorter:merge([1, 2, 3], [2, 2, 4])
```

3.2 Processes

A process is created in Erlang with the built-in `spawn` function. `spawn` is called with a module and function name, and the arguments for that function, and the newly created process starts running that function. `spawn` returns a process reference, which can then be used to communicate with the process.

For instance, the following function spawns two processes, passing the process reference of the first to the second. The first process runs the `pong` function in the `pingpong` module with no arguments, and the second runs the `ping` function in the `pingpong` module with the arguments `5` and the reference to the `pong` process:

```
start() ->
    Pong = spawn(pingpong, pong, []),
    spawn(pingpong, ping, [5, Pong]).
```

Generally, functions perform a list of comma-separated actions like this.

Now, let's write the `ping` and `pong` functions. `ping` will send the given number of "ping" messages to the given process, and expect an equal number of "pong" messages in response. `pong` will expect a sequence of "ping" message, and send a "pong" to each. For this to work, we need to know Erlang's syntax for sending and receiving messages.

Messages are sent in Erlang with the `!` operator, as `Target ! Message`. The message can be any Erlang value, but in practice, it is either an atom or a tuple in which the first element is an atom, where the atom specifies the kind of message, and any arguments that the message fills the rest of the tuple. This convention is simply so that when a message is received, the receiver knows what kind of message it is; the atom is a tag. In our case, the "pong" process does not have a reference to the "ping" process, but the "ping" process does a have a reference to the "pong" process, so the "ping" message will need to send a reference along in order for the "pong" process to know how to reply. In π-calculus terms, "ping" must send the channel on which "pong" is to reply.

Messages are received in Erlang with a `receive` expression, which resembles a pattern match, in that it matches the shape of the message received. For a message to be successfully sent, the target process must be running a `receive` with a matching pattern, in the same way that for a π-calculus program to make progress, a sending process must have a matching receiving process. Because `receive` matches particular shapes of messages, a process can receive messages in any order, but process them in the order it chooses, simply by performing `receive`s in sequence that match only the kinds of messages it wishes to process.

Knowing this, we will write `ping` first:

```
ping(0, _) -> io:format("Ping finished~n", []);

ping(N, Pong) ->
    Pong ! {ping, self()},
    receive
        pong -> io:format("Pong received~n", [])
    end,
    ping(N - 1, Pong).
```

Like in Haskell, functions can be declared in multiple parts with implicit patterns. In this case, the `ping` function simply outputs "Ping finished" to standard out and terminates if the first argument (the number of times to ping) is 0. If `N` is not 0, it sends a message to the `Pong` process, and then awaits a `pong` message back. The sent message is a tuple containing the atom `ping`, to indicate that this is a ping message, and a reference to the current process, obtained with the built-in `self` function. Once a ping has been sent and a pong received, it prints "Pong received", and then recurses with one fewer ping left to send.

Now, let's write `pong`:

```
pong() ->
    receive
        {ping, Ping} ->
            Ping ! pong,
            io:format("Ping received~n", [])
    end,
    pong().
```

Where `ping` starts with a send, `pong` instead starts with a receive. Once `pong` has received a message matching the pattern `{ping, Ping}` (remember, `Ping` is a variable because it starts with a capital letter), it sends a `pong` message back (`pong` is an atom in this case, not the function), and then prints "Ping received". We've intentionally written this with the print after the send, to demonstrate concurrency.

If the `start` function we wrote above is run, one possible output is:

```
Ping received
Pong received
Ping received
Pong received
Ping received
Pong received
Ping received
Pong received
Ping received
Pong received
Ping finished
```

However, because the `pong` process sends its `pong` *before* printing that the `ping` was received, other orders are possible, such as this one:

```
Pong received
Ping received
Ping received
Pong received
Ping received
Pong received
Pong received
Ping received
Pong received
Ping finished
Ping received
```

Exercise 2. What orders are *not* possible?

In this example, the `pong` process never actually terminates: only `ping` knew how many times to `ping`, so `pong` is left waiting endlessly for another `ping` that will never arrive. For cleanliness, we could instead add a `terminate` message like so:

```
ping(0, Pong) ->
    io:format("Ping finished~n", []),
    Pong ! terminate;

ping(N, Pong) ->
    Pong ! {ping, self()},
    receive
        pong -> io:format("Pong received~n", [])
    end,
    ping(N - 1, Pong).

pong() ->
    receive
        terminate ->
            io:format("Pong finished~n", []);

        {ping, Ping} ->
            Ping ! pong,
            io:format("Ping received~n", []),
            pong()
    end.
```

Since `pong` does not recurse if `terminate` is received, it instead simply ends, terminating the process.

4 Processes as References

Erlang does not have mutable variables. But, surprisingly, they can be built with nothing but processes!

When representing mutable data in functional languages, we needed a way to put that data aside, separate from the program, in the heap (Σ). But, concurrent processes are already "aside" and separate from one another, so all we actually need is a way for a process to store a piece of data like a single mapping in the heap, similarly to Haskell's monads.

To achieve this, we will make a module which exports three functions: `ref/1`, `get/1`, and `put/2`. The `ref` function will generate a reference, like OCaml's `ref`. The `get` function will retrieve the value stored in a reference, like OCaml's `!`. The `put` function will store a value in a reference, like OCaml's `:=`. The actual value used for the reference will be a process reference, to a process carefully designed to work this way.

The complete solution follows:

```
-module(refs).
-export([ref/1, refproc/1, get/1, put/2]).

ref(V) ->
    spawn(refs, refproc, [V]).

refproc(V) ->
    receive
        {get, Return} ->
            Return ! {refval, V},
            refproc(V);

        {put, Return, NV} ->
            Return ! {refval, NV},
            refproc(NV)
    end.

get(Ref) ->
    Ref ! {get, self()},
    receive
        {refval, V} -> V
    end.

```

```
put(Ref, V) ->
    Ref ! {put, self(), V},
    receive
        {refval, _} -> Ref
    end.
```

The `ref` function spawns a new process, returning the process reference. The new process represents the reference, and runs the function `refproc`, with the initial value as its arguments. The `get` function sends a `get` message to the given process (which must be a reference process for this to work), and expects a `refval` message in response with the value stored in the reference. The `put` function sends a `put` message to the given process, containing the value to put in the reference, and also waits for a `refval` message. `put` doesn't actually care about the value returned by `refval`; it's only used to make sure that the message has been received and acted on before the current process continues. This isn't *strictly* necessary, but we don't want to get stuck in the quagmire of happens-before relationships here!

The `refproc` function contains all of the interesting behavior, as it is the function used by the actual reference process. `refproc` must be exported because of how `spawn` works—there are ways to get around "polluting" the exported names in this way, but they're not important for our purposes. `refproc`'s behavior is quite similar regardless of whether it receives a `get` message or a `put` message: it returns a value and then recursively calls `refproc` again. The difference is in which value. The value stored in the reference is in the (immutable) `V` variable. With `get`, it returns that value, and then recurses with the same value. With `put`, it instead expects a new value, `NV`, and returns and recurses with `NV` instead of `V`. In this way, although no variables are mutable, the reference itself is, since if it receives a `put` message, then it will respond to future `get` messages with the new value, until another `put` is received.

In shared-memory concurrency, two tasks may access the same mutable memory, and each may mutate it. With these references, we have in fact implemented shared-memory concurrency on top of message-passing concurrency: if two processes each have a reference to the reference process, then either may mutate its value, and both can see the other's mutations. The fact that it is then extremely difficult to guarantee that the processes mutate things in the correct order is the usual argument for using message-passing concurrency instead of shared-memory concurrency in the first place, but this form of references demonstrates that neither is more powerful than the other.

5 Processes as Objects

If you're accustomed to object-oriented programming, you've probably noticed that references from the reference module above act a lot like objects with a `get` and `put` method. Indeed, we can extend this metaphor to implement objects with only processes with immutable variables. For instance, this module implements a reverse polish notation calculator object very similar to the one we wrote for the Smalltalk segment of Module 1:

```
-module(rpncalc).
-export([newrpn/0, rpn/1, push/2, binary/2, add/1, sub/1, mul/1, divide/1]).

newrpn() ->
    spawn(rpncalc, rpn, [[]]).

rpn(Stack) ->
    receive
        {push, Return, V} ->
            Return ! {rpnval, V},
            rpn([V | Stack]);

        {op, Return, F} ->
            rpnop(Stack, F, Return)
    end.

rpnop([R, L | Rest], F, Return) ->
    V = F(L, R),
    Return ! {rpnval, V},
    rpn([V | Rest]).

push(RPN, V) ->
    RPN ! {push, self(), V},
```

```
24      receive
25          {rpnval, _} -> V
26      end.
27
28  binary(RPN, F) ->
29      RPN ! {op, self(), F},
30      receive
31          {rpnval, V} -> V
32      end.
33
34  add(RPN) ->
35      binary(RPN, fun(L, R) -> L + R end).
36
37  sub(RPN) ->
38      binary(RPN, fun(L, R) -> L - R end).
39
40  mul(RPN) ->
41      binary(RPN, fun(L, R) -> L * R end).
42
43  divide(RPN) ->
44      binary(RPN, fun(L, R) -> L / R end).
```

An RPN's sole field, the stack, is represented by the `Stack` variable of the `rpn` function, which is the function run by an RPN process. The `push` and `binary` functions send an RPN process messages corresponding to one of its two supported "methods": `push` and `op`. The `op` message carries a function, representing the binary operation to perform, so like in the Smalltalk version, specific operations can be implemented in terms of it.

Again, there are only immutable variables and processes, but the fact that sending and receiving messages establishes a sequence allows us to emulate more sophisticated features. Erlang has several libraries implementing more elegant object orientation, but still using processes as objects. Erlang is designed for programs to have thousands of processes, so it's common to mix these styles as well; for instance, fields can be implemented as references which are in turn implemented as processes as in the previous section.

6 Implementation

Operating systems implement processes and threads: in OS terms, two processes do not share memory, but two threads within the same process do share memory. Erlang uses the term "process" because Erlang's processes do not share memory. However, using operating system processes to implement Erlang processes would result in catastrophically poor performance! Indeed, even using operating system threads to implement Erlang processes would be similarly fraught. The reason is simply that switching between threads or processes in an operating system—that is, context switching—is an expensive process.

Instead, highly-concurrent languages such as Erlang use so-called *green threads*. Basically, the Erlang interpreter must implement its own form of thread switching, and maintain the stack for each thread as a native data structure in the host language. When an Erlang process executes a `receive` and it does not immediately have a matching message available to act upon, Erlang instead sets aside the thread for that process and loads a stack for another process; in effect, it performs the same kind of context switch that an operating system performs, but with much less context to switch. It runs as many operating system threads as there are CPU cores available, but each one can switch between many green threads. It is not uncommon for an Erlang process to have tens of thousands of processes, so keeping green threads light is extremely important.

> **Aside:** The standard term for green threads is the term we use, "green threads", but that term has a curious history. They're not "green" because they're in some way environmentally friendly; they're called "green" because the original implementation of Java had only green threads, and they were called green threads because they were implemented by a team at Sun called The Green Team. These titular green threads, ironically, were very quickly dropped from Java, but their name has been genericized to refer to all user-level threads with similar functionality, even those that predate Java's green threads, such as Erlang's!

To know which processes are available to run, an Erlang process must also be able to pattern match very quickly. Typically, a process that is waiting for a message has its pattern stored, and when another process *sends*

it a message, it performs a pattern match immediately, to determine if the other process can be awoken.

The other major implementation roadblock to message-passing concurrency is the actual message passing. In languages with mutable values, it is necessary to copy the message being passed, so that two processes cannot see the same mutable memory (which would be shared-memory concurrency, not message-passing concurrency). Erlang largely sidesteps this issue by being fundamentally immutable, so that it's harmless to pass around values in any form. Two processes can share a pointer to a value if neither will actually mutate that value.

7 Miscellany

As mentioned previously, we won't mention how to actually solve any interesting problems with concurrency; instead, you've hopefully realized, particularly through the examples of processes as references and objects, that like the λ-calculus before it, π-calculus and Erlang demonstrate that a surprising amount of power is unleashed merely by having concurrency, before we've even reached actual algorithms. Most programming language semantics that aren't designed specifically to address concurrency simply ignore it, because adding non-determinism affects all other aspects of a semantics.

Erlang does actually support some (very limited) mutable data structures, but they may not be sent in a message.

The message-reply style we used in all of our examples was fairly brittle, in that we used a specific atom as the expected reply, but there's nothing to stop another process from sending the same atom. Erlang supports creating unique values, called "references" to be needlessly confusing, with the built-in `create_ref` function. Usually, two processes which communicate would exchange such unique references to make sure that they're receiving messages from the process they thought they were speaking to.

Most Erlang programs are written in a so-called "let it crash" style. That is, instead of trying to anticipate all forms of errors, code is written to simply re-spawn processes that fail unexpectedly. Since processes are mostly independent of each other, large systems can operate even with major bugs. In the Erlang shell, you can re-compile modules, and swap out processes using the old module for processes using the new version, and it is thus often possible to fix bugs in a running system with no downtime. Many Erlang proponents cite this style as the major advantage of Erlang.

8 Fin

In the next (and final) module, we will look at how our mathematical model of programming languages interfaces with the real world, through models of systems programming.

References

[1] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.