Communication

Lecture 24 Thursday, April 25, 2013

1 Communication between concurrent programs

Last lecture, we were introduced to the notions of concurrency and nondeterminism. We added these to a familiar lambda calculus by augmenting it with a concurrency operator $e_1||e_2$. Concurrently evaluating expressions were able to communicate by reading and writing shared memory.

Today, we will look at one of the other major mechanisms by which concurrent programming languages allow for such communication: *message passing*. We have already seen message passing in this course, as a mechanism for implementing objects.

1.1 Process Calculi

Throughout much of this course, we have focused on individual, isolated language features by adding them to a small lambda calculus. For example, we added records to the lambda calculus in order to study subtyping. To add records, we extended the notions of expressions and values with new forms for records. We also had to extend the semantics of the language to describe how to compute with record creation and projection, as well as our typing judgment.

While it would be possible to do the same for message passing, today we will do something different. Rather than adding message passing primitives to an existing calculus, we will see that message passing between concurrent processes is able to stand on its own as a model of computation. While the resulting language will not be directly useful as a programming language (just as the pure lambda calculus is not), we will see that it can be extended with all the other concepts we have studied thus far, much as the lambda calculus may be.

Unlike the lambda calculus, which is essentially a canonical, minimal *functional* language, there is no single preferred language for reasoning about communicating processes. Instead, there are many *process calculi* which each emphasize different aspects or tradeoffs in this style of computation. We will single out the *join calculus*, a process calculus which is in many ways reminiscent of ML.

2 The Join Calculus

2.1 Syntax

We first present the core syntax of the join calculus, in which there are three productions: processes P and Q, join patterns J and J', and definitions D and D'. x, v and so on are variables, or channel names. In both processes and join patters, the notation $\bar{v}$ represents a tuple of names, of arbitrary (including empty) size.

$$\begin{aligned} P, Q &::= x\langle\bar{v}\rangle \mid P|Q \mid \mathsf{def}\ D\ \mathsf{in}\ P \mid 0 \\ J &::= x\langle\bar{v}\rangle \mid J|J' \\ D &::= J \rhd P \mid D \wedge D' \end{aligned}$$

A process, denoted P or Q, intuitively represents a single communicating entity. $x\langle\bar{v}\rangle$ is a process which sends a message on channel x; the message is the tuple $\bar{v}$. $P|Q$ denotes the parallel composition of processes P and Q. The process $\mathsf{def}\ D\ \mathsf{in}\ P$ is our mechanism for defining new names in processes, similarly to how $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ binds the variable x in e_2 in the lambda calculus. Finally, 0 is a null or inert process. Intuitively, 0 is a process which has halted its computation.

Join patterns J are our mechanism for allowing concurrent computations to communicate or interact. The join pattern $x\langle\bar{v}\rangle$ (which looks just like the corresponding message send) represents the receipt of a message $\bar{v}$ on the x channel. The joint patterns $J|J'$ represents the receipt of multiple messages.

Finally, definitions D bind messages received to names in processes. For example, the definition $c\langle x, y\rangle \triangleright P$ means that whenever a message is received on channel c, process P will be run, with the parameters x and y set to the respective components of the message tuple. The definition $c_1\langle\bar{x}\rangle|c_2\langle\bar{y}\rangle \triangleright P$ is more interesting. The pattern $c_1\langle\bar{x}\rangle|c_2\langle\bar{y}\rangle$ represents the receipt of messages on *both* channels c_1 and c_2. Only then can process P continue on.

In multiple definitions, such as $\mathsf{def}\ J_1 \triangleright P_1 \wedge J_2 \triangleright P_2 \wedge \cdots\ \mathsf{in}\ P$, all of the channel names defined by the $J_i \triangleright P_i$ are all mutually recursively bound in P and in each P_i.

2.2 Semantics

To perform or reason about substitution in the lambda calculus, we had to compute the set of bound and free variables in an expression. In the join calculus, things are slightly more complicated: we must keep track of the free variables (fv) of definitions and processes; defined variables (dv) of join patterns and of definitions; and the received variables (rv) of join patterns. These are computed as follows:

$$\begin{aligned}
\mathsf{rv}(x\langle\bar{v}\rangle) &= \{u | u \in \bar{v}\} \\
\mathsf{rv}(J|J') &= \mathsf{rv}(J) \cup \mathsf{rv}(J')
\end{aligned}$$

$$\begin{aligned}
\mathsf{dv}(x\langle\bar{v}\rangle) &= \{x\} \\
\mathsf{dv}(J|J') &= \mathsf{dv}(J) \cup \mathsf{dv}(J')
\end{aligned}$$

$$\begin{aligned}
\mathsf{dv}(J \triangleright P) &= \mathsf{dv}(J) \\
\mathsf{dv}(D \wedge D') &= \mathsf{dv}(D) \cup \mathsf{dv}(D')
\end{aligned}$$

$$\begin{aligned}
\mathsf{fv}(J \triangleright P) &= \mathsf{dv}(J) \cup (\mathsf{fv}(P) - \mathsf{rv}(J)) \\
\mathsf{fv}(D \wedge D') &= \mathsf{fv}(D) \cup \mathsf{fv}(D')
\end{aligned}$$

$$\begin{aligned}
\mathsf{fv}(x\langle\bar{v}\rangle) &= \{x\} \cup \{u | u \in \bar{v}\} \\
\mathsf{fv}(P|Q) &= \mathsf{fv}(P) \cup \mathsf{fv}(Q) \\
\mathsf{fv}(\mathsf{def}\ D\ \mathsf{in}\ P) &= (\mathsf{fv}(D) \cup \mathsf{fv}(P)) - \mathsf{dv}(D)
\end{aligned}$$

As with the concurrent lambda calculus we saw previously, defining a big-step operational or a denotational semantics for this language would be difficult.

There is an operational semantics for the join calculus, featuring a single reduction rule. Much of the work in presenting it is fairly tedious, as it involves a lot of machinery for keeping track of channel names.

Instead, we will look at a very different style of operational semantics: the *reflexive chemical abstract machine* (RCHAM). This semantics is based on an analogy with chemical reactions. There is nondeterminacy is the receipt of messages in the join calculus; this corresponds to the random motion of molecules in a chemical solution. In this lecture we will not worry too much about the precise correspondence.

We write $\mathcal{R} \vdash \mathcal{M}$, where $\mathcal{R}$ is a multiset of definitions and $\mathcal{M}$ is a multiset of processes, to represent a current "solution". Our rules are then between pairs of such solutions.

We will have two sorts of reductions: reversible ones, denoted by back-and-forth arrows $\rightleftharpoons$ (with one direction at a time denoted by the hooked arrows $\rightharpoonup$ and $\rightharpoondown$); and irreversible ones, denoted by the standard $\rightarrow$ arrow. In all of the following rules, only those elements of the multisets that participate in the rule are shown; we elide showing all other elements for clarity. That is, the first pair of rules is really $\mathcal{R} \vdash \mathcal{M}, 0 \rightleftharpoons$

$\mathcal{R} \vdash \mathcal{M}$.

$$\begin{array}{rcll} \vdash 0 & \rightleftharpoons & \vdash & (\textsc{str-null}) \\ \vdash P|Q & \rightleftharpoons & \vdash P, Q & (\textsc{str-join}) \\ D \wedge D' \vdash & \rightleftharpoons & D, D' \vdash & (\textsc{str-and}) \\ \vdash \mathsf{def}\ D\ \mathsf{in}\ P & \rightleftharpoons & D\sigma_{\mathsf{dv}} \vdash P\sigma_{\mathsf{dv}} & (\textsc{str-def}) \\ J \rhd P \vdash J\sigma_{\mathsf{rv}} & \to & J \rhd P \vdash P\sigma_{\mathsf{rv}} & (\textsc{react}) \end{array}$$

In these rules, σ_{dv} and σ_{rv} represent arbitrary substitutions obeying some side conditions. In STR-DEF, σ_{dv} substitutes fresh names for all of the channel names in $\mathsf{dv}(D)$. In REACT, σ_{rv} substitutes the received names into the process P.

Let's look at the intuitive meanings of each of theses rules. The STR-NULL rules mean that we can freely create or destroy null processes in the solution. The STR-JOIN rules allow us to pull apart (or push back together) concurrent processes. The STR-AND rules do the same for definitions. All the rules so far allow us to break apart processes and definitions into individual "atoms", rearrange them, and put them back together into new configurations.

The other rules deal with namings. The STR-DEF rules allow processes to place new definitions into the solution, or to capture existing ones in the opposite direction. The σ_{dv} substitution and its side condition only allows this to take place when no unintended variable capture (or unbinding, in the other direction) will occur.

Finally, the one-way REACT rule describes the receipt of messages. Intuitively, if there are a number of processes $J\sigma_{\mathsf{rv}}$ sending messages on some set of channels, and there is a definition which describes a process to spawn upon receipt of those messages, then the REACT rule may fire, consuming the message sends and substituting them into the new process. Here, the substitution σ_{rv} describes which channel names to substitute for which formal parameters in the process.

2.3 Encodings and Extensions

As promised, the join calculus is an expressive enough model of computation to embed the simple lambda calculus in. We do so via a CPS translation. The following encoding is for a call-by-value lambda calculus:

$$\begin{aligned} [\![x]\!]_v &= v\langle x \rangle \\ [\![\lambda x.\, T]\!]_v &= \mathsf{def}\ \kappa\langle x, w \rangle \rhd [\![T]\!]_w\ \mathsf{in}\ v\langle \kappa \rangle \\ [\![T\ U]\!]_v &= \mathsf{def}\ t\langle \kappa \rangle | u\langle w \rangle \rhd \kappa\langle w, v \rangle\ \mathsf{in}\ [\![T]\!]_t | [\![U]\!]_u \end{aligned}$$

Intuitively, $[\![T]\!]_v$ sends the value of T along channel v. A value here is a process which will be sent an argument value x and a continuation w along channel κ.

Interestingly, we allow the translation to evaluate a function and its argument *in parallel*, and join the results back together.

Given the ability to encode the lambda calculus in the join calculus, we can indirectly encode or extend the join calculus with all of the features we have amended the lambda calculus with: numerals and arithmetic, state, records, etc. The join calculus can then be seen as the core of a more realistic concurrent programming language, just as the lambda calculus is the core of functional programming languages.

2.4 Applications

The join calculus has been implemented for many real programming languages. As its syntax suggests, there are implementations for ML, and in particular there is an extension of OCaml, cleverly named JoCaml. There are also libraries implementing join patterns for Java, C++, and C♯ among others. Other process calculi also see use as the basis for concurrency in languages such as Go and Erlang.

The join calculus (and other process calculi) have also been used to model concurrent systems besides computer programs, such as the complex biochemical processes underlying the behavior of cells.