Concurrency

Lecture 24 Thursday, April 24, 2014

1 Concurrency

With increasingly parallel hardware, there is an corresponding increase in the need for concurrent programs that can take advantage of this parallelism. However, writing correct concurrent programs can be difficult. Language abstractions are a promising way to allow programmers to express concurrent computation in a way that makes it easy (or at least, easier) for both the programmer and compiler to reasoning about the computation.

In this lecture, we'll explore how to model concurrent execution, and investigate an *effect system* to ensure that concurrent execution is deterministic despite threads potentially sharing memory.

1.1 A simple concurrent lambda calculus

Let's consider a lambda calculus with a concurrent operator $||$. That is, expression $e_1||e_2$ will concurrently evaluate e_1 and e_2. If expression e_1 evaluates to v_1 and e_2 evaluates to v_2, the result of evaluating $e_1||e_2$ will be the pair (v_1, v_2).

We add first-class references, to make things interesting.

$$\begin{aligned} e &::= x \mid n \mid \lambda x.\, e \mid e_1\; e_2 \mid e_1||e_2 \mid (e_1, e_2) \mid \#1\; e \mid \#2\; e \mid \mathsf{ref}\; e \mid\; !e \mid e_1 := e_2 \mid \ell \\ v &::= n \mid \lambda x.\, e \mid (v_1, v_2) \mid \ell \end{aligned}$$

We define a small step operational semantics for the language as follows. Most of the rules are standard. The rules for the concurrent construct, however, are new.

$$E ::= [\cdot] \mid E\; e \mid v\; E \mid (E, e) \mid (v, E) \mid \#1\; E \mid \#2\; E \mid \mathsf{ref}\; E \mid\; !E \mid E := e \mid v := E$$

$$\frac{\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\langle E[e], \sigma\rangle \longrightarrow \langle E[e'], \sigma'\rangle} \qquad \frac{}{\langle (\lambda x.\, e)\; v, \sigma\rangle \longrightarrow \langle e\{v/x\}, \sigma\rangle} \qquad \frac{}{\langle \mathsf{ref}\; v, \sigma\rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto v]\rangle}\; \ell \notin \mathrm{dom}(\sigma)$$

$$\frac{}{\langle !\ell, \sigma\rangle \longrightarrow \langle v, \sigma\rangle}\; \sigma(\ell) = v \qquad \frac{}{\langle \ell := v, \sigma\rangle \longrightarrow \langle v, \sigma[\ell \mapsto v]\rangle} \qquad \frac{}{\langle \#1\; (v_1, v_2), \sigma\rangle \longrightarrow \langle v_1, \sigma\rangle}$$

$$\frac{}{\langle \#2\; (v_1, v_2), \sigma\rangle \longrightarrow \langle v_2, \sigma\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle e_1||e_2, \sigma\rangle \longrightarrow \langle e_1'||e_2, \sigma'\rangle} \qquad \frac{\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle}{\langle e_1||e_2, \sigma\rangle \longrightarrow \langle e_1||e_2', \sigma'\rangle} \qquad \frac{}{\langle v_1||v_2, \sigma\rangle \longrightarrow \langle (v_1, v_2), \sigma\rangle}$$

Note that this operational semantics is nondeterministic. There are two rules for evaluating subexpressions of the parallel $e_1||e_2$. One rule evaluates one step of the left expression e_1, and the other evaluates one step of the right expression e_2. (We could equivalently have added two more evaluation contexts, $E||e$ and $e||E$.) Indeed, this nondeterminism gets at the heart of concurrent execution.

Consider the following program, which models an account bank balance, with two concurrent deposits.

$$\mathsf{let}\; \mathsf{bal} = \mathsf{ref}\; 0\; \mathsf{in}\; (\mathsf{let}\; y = (\mathsf{bal} := !\mathsf{bal} + 25 || \mathsf{bal} := !\mathsf{bal} + 50)\; \mathsf{in}\; !\mathsf{bal})$$

There are several possible final values that this program could evaluate to: 50, 25, and 75.

In the absence of any synchronization mechanism, communication mechanism, or shared resource, the concurrent evaluation of e_1 and e_2 does not allow e_1 and e_2 to communicate, or interfere with each other at all. That is, it is pretty easy to execute e_1 and e_2 at the same time, since they cannot interfere with each other. This is a good thing. Indeed, if we have a pure expression e in our language (i.e., no use of references) then even though evaluation may be nondeterministic, the final result will always be the same. With side-effects, however, the final result may differ, as shown above.

1.2 Effect system for determinism

Let's consider a type system that ensures that when we execute a concurrent program, the result is always deterministic. To do so, we will introduce two new concepts: *memory regions* and *effects*.

A *memory region* is a set of memory locations. For our purposes, every location ℓ will belong to exactly one region, and we will annotated locations with the region to which they belong. For example, we will write ℓ_α to indicate that location ℓ belongs to region α. We will assume that the programmer provides us with region annotations at allocation sites. We are going to use regions to help us track which locations a program may read and write, in order to ensure determinism during evaluation. However, regions can be used to help manage memory effectively (for example, deallocating an entire region at a time, instead of individual locations), and it is often possible to infer regions of memory locations.

The modified grammar of the language, with region annotations, is as follows.

$$\begin{aligned} e &::= \cdots \mid \mathsf{ref}_\alpha\, e \mid \ell_\alpha \\ v &::= \cdots \mid \ell_\alpha \end{aligned}$$

A *computational effect* is an observable event that occurs during computation. The canonical example is side-effects, such as reading or writing memory during execution or performing input or output (i.e., interaction with the external environment). However, depending on what we regard as "observable", it may also include the termination behavior of a program, or whether a computation will produce an exception or run-time error.

Whereas a type τ describes the final value produced by a computation e, the effects of e describe observable events during the execution of e. An *effect system* describes or summarizes effects that may occur during computation. (What do you think effects mean under the Curry-Howard isomorphism?) One use of monads is to cleanly separate effectful computation from pure (i.e., non-effectful) computation.

For our language, we are interested in memory effects: that is, what memory locations a computation may read or write during execution. We define a type and effect system to track these effects.

We write $\Gamma, \Sigma \vdash e : \tau \rhd R, W$ to mean that expression under variable context Γ and store typing Σ, expression e has type τ, and that during evaluation of e, any location read will belong to a region in set R (the read effects of e), and any locations written will belong to a region in set W (the write effects of e).

We extend function types with read and write effects. A function type is now of the form $\tau_1 \xrightarrow{R,W} \tau_2$. A function of this type takes as an argument a value of type τ_1, and produces a value of type τ_2; R and W describe, respectively, the read and write effects that may occur during execution of the function.

$$\tau ::= \textbf{int} \mid \tau_1 \xrightarrow{R,W} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau\ \textbf{ref}_\alpha$$

$$\frac{}{\Gamma, \Sigma \vdash n : \textbf{int} \rhd \emptyset, \emptyset} \qquad \frac{\Gamma(x) = \tau}{\Gamma, \Sigma \vdash x : \tau \rhd \emptyset, \emptyset} \qquad \frac{\Gamma[x \mapsto \tau], \Sigma \vdash e : \tau' \rhd R, W}{\Gamma, \Sigma \vdash \lambda x : \tau.\, e : \tau \xrightarrow{R,W} \tau' \rhd \emptyset, \emptyset}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau \xrightarrow{R,W} \tau' \rhd R_1, W_2 \quad \Gamma, \Sigma \vdash e_2 : \tau \rhd R_2, W_2}{\Gamma, \Sigma \vdash e_1\ e_2 : \tau' \rhd R_1 \cup R_2 \cup R, W_1 \cup W_2 \cup W} \qquad \frac{\Gamma, \Sigma \vdash e : \tau \rhd R, W}{\Gamma, \Sigma \vdash \mathsf{ref}_\alpha\, e : \tau\ \textbf{ref}_\alpha \rhd R, W}$$

$$\frac{\Gamma, \Sigma \vdash e : \tau\ \textbf{ref}_\alpha \rhd R, W}{\Gamma, \Sigma \vdash !e : \tau \rhd R \cup \{\alpha\}, W} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \tau\ \textbf{ref}_\alpha \rhd R_1, W_2 \quad \Gamma, \Sigma \vdash e_2 : \tau \rhd R_2, W_2}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau \rhd R_1 \cup R_2, W_1 \cup W_2 \cup \{\alpha\}}$$

$$\frac{}{\Gamma, \Sigma \vdash \ell_\alpha : \tau\ \textsf{\textbf{ref}}_\alpha \rhd \varnothing, \varnothing}\ \Sigma(\ell_\alpha) = \tau\ \textsf{\textbf{ref}}_\alpha \qquad \frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \rhd R_1, W_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2 \rhd R_2, W_2}{\Gamma, \Sigma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \rhd R_1 \cup R_2, W_1 \cup W_2}$$

$$\frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2 \rhd R, W}{\Gamma, \Sigma \vdash \#1\ e : \tau_1 \rhd R, W} \qquad \frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2 \rhd R, W}{\Gamma, \Sigma \vdash \#2\ e : \tau_2 \rhd R, W}$$

The rule for dereferencing a location adds the appropriate region to the read effect. The rule for updating locations adds the appropriate region to the write effect. The other rules just propagate read and write effects as needed.

The rule for the concurrent operator (below) is the most interesting. A concurrent command $e_1 || e_2$ is well-typed only if the write effect of e_1 does not intersect with the read or write effects of e_2, and vice versa. That is, there is no region such that e_1 writes to that region, and e_2 reads or writes to the same region. This prevents *data races*, i.e., two threads that are concurrently accessing the same location, and one of those accesses is a write.

$$\frac{\begin{array}{c}\Gamma, \Sigma \vdash e_1 : \tau_1 \rhd R_1, W_1 \quad W_1 \cap (R_2 \cup W_2) = \varnothing \\ \Gamma, \Sigma \vdash e_2 : \tau_2 \rhd R_2, W_2 \quad W_2 \cap (R_1 \cup W_1) = \varnothing\end{array}}{\Gamma, \Sigma \vdash e_1 || e_2 : \tau_1 \times \tau_2 \rhd R_1 \cup R_2, W_1 \cup W_2}$$

What is type soundness for this type system? Intuitively, it extends our previous notion of type safety (i.e., not getting stuck), with the notion that R and W correctly characterize the reads and writes that a program may perform. We express this idea with the following theorem. (Note that we assume that evaluation contexts include $E || e$ and $e || E$.)

Theorem 1 (Type soundness). *If $\vdash e : \tau \rhd R, W$ then for all stores σ and σ',*

- *if, for some evaluation context E, we have $\langle e, \sigma \rangle \longrightarrow^* \langle E[!\ell_\alpha], \sigma' \rangle$, then $\alpha \in R$.*
- *if, for some evaluation context E, we have $\langle e, \sigma \rangle \longrightarrow^* \langle E[\ell_\alpha := v], \sigma' \rangle$, then $\alpha \in W$.*
- *if $\langle e, \sigma \rangle \longrightarrow^* \langle e', \sigma \rangle$ then either e' is a value or there exists e'' and σ'' such that $\langle e', \sigma' \rangle \longrightarrow \langle e'', \sigma'' \rangle$.*

The theorem says that if expression e is well typed, and, during its evaluation, it dereferences a location belonging to region α, then the type judgment had α in the read effect of e. It also says that if evaluation updates a location ℓ_α, then α is in the write effect of e. (We could also have tracked the *allocation effect* of e, i.e., in which region e allocates new locations, but we don't need to for our purposes.)

The following theorem says that a well-typed program is deterministic. If there are two executions, then both executions produce the same value.

Theorem 2 (Determinism). *If $\Gamma, \Sigma \vdash e : \tau \rhd R, W$ and $\langle e, \sigma \rangle \longrightarrow^* \langle v_1, \sigma_1 \rangle$ and $e \longrightarrow^* \langle v_2, \sigma_2 \rangle$ then $v_1 = v_2$.*

The proof of this theorem relies on the following key lemma, which says that if a well-typed concurrent expression $e_1 || e_2$ can first take a step with e_2, and then take a step with e_1, then we can first step e_1 and then e_2, and end up at the same state.

Lemma 1. *If for some Σ, τ, R and W we have $\varnothing, \Sigma \vdash e_1 || e_2 : \tau \rhd R, W$, then for all σ such that $\Gamma, \Sigma \vdash \sigma$ if*

$$\langle e_1 || e_2, \sigma \rangle \longrightarrow \langle e_1 || e_2', \sigma' \rangle \longrightarrow \langle e_1' || e_2', \sigma'' \rangle,$$

then there exists σ''' such that

$$\langle e_1 || e_2, \sigma \rangle \longrightarrow \langle e_1' || e_2, \sigma''' \rangle \longrightarrow \langle e_1' || e_2', \sigma'' \rangle.$$

Intuitively, the proof works by showing that given any two executions of a program, they are both equivalent to a third execution in which we always fully evaluate the left side of a concurrent operator first, before starting to evaluate the right side of a concurrent operator. By transitivity, the two executions must be equal, and produce equal values.