Concurrency

CS 152 (Spring 2022)

Harvard University

Tuesday, April 26, 2022

Today, we will learn about

- A simple concurrent λ-calculus
- Weak memory models
- Effect system for determinism
- Message passing

A simple concurrent λ-calculus

concurrent operator ||

The 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).

A simple concurrent λ-calculus (syntax)

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

A simple concurrent λ-calculus

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

A simple concurrent λ-calculus

$$\frac{< e, \sigma > \longrightarrow < e', \sigma' >}{< E[e], \sigma > \longrightarrow < E[e'], \sigma' >}$$

$$\frac{}{< (\lambda x.\, e)\ v, \sigma > \longrightarrow < e\{v/x\}, \sigma >}$$

$$\frac{}{< \mathsf{ref}\ v, \sigma > \longrightarrow < \ell, \sigma[\ell \mapsto v] >}\ \ell \notin \mathsf{dom}(\sigma)$$

$$\frac{}{< !\ell, \sigma > \longrightarrow < v, \sigma >}\ \sigma(\ell) = v$$

$$\frac{}{< \ell := v, \sigma > \longrightarrow < v, \sigma[\ell \mapsto v] >}$$

A simple concurrent λ-calculus

$$\frac{}{< \#1\ (v_1, v_2), \sigma > \longrightarrow < v_1, \sigma >}$$

$$\frac{}{< \#2\ (v_1, v_2), \sigma > \longrightarrow < v_2, \sigma >}$$

$$\frac{< e_1, \sigma > \longrightarrow < e_1', \sigma' >}{< e_1 || e_2, \sigma > \longrightarrow < e_1' || e_2, \sigma' >}$$

$$\frac{< e_2, \sigma > \longrightarrow < e_2', \sigma' >}{< e_1 || e_2, \sigma > \longrightarrow < e_1 || e_2', \sigma' >}$$

$$\frac{}{< v_1 || v_2, \sigma > \longrightarrow < (v_1, v_2), \sigma >}$$

Example: concurrent deposits

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{let bal} = \text{ref } 0 \text{ in} \\ (\text{let } y = (\text{bal} := !\text{bal} + 25 \,||\, \text{bal} := !\text{bal} + 50) \text{ in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad !\text{bal}) \end{array}$$

Sequential consistency

the result of any execution is as if the memory operations of all the threads were executed in some global sequential order, and the memory operations of each individual thread appear in this sequence in the same order they appear in the thread.

i.e. as if there were a single global memory, and only one thread at a time is allowed to access the memory.

Hardware optimization

- write buffers with bypassing capabilities
- each core has its own write buffer
- when it issues a write, the write goes into the buffer and the program can continue
- when the core wants to read a (different) memory location, it can "bypass" the write buffer, i.e. it can go immediately to memory to read the memory location, even if all of the writes it issued have not yet finished
- common hardware optimization used in uniprocessors, since it helps hide the latency of write operations
- but with multiple cores, this can violate sequential consistency

Hardware optimization (Example)

- Two memory locations a and b both containing zero.
- First core executes
  $\mathsf{a} := 1; \mathsf{if}\ !\mathsf{b} = 0\ \mathsf{then}\ e\ \mathsf{else}\ ()$
- Second core executes
  $\mathsf{b} := 1; \mathsf{if}\ !\mathsf{a} = 0\ \mathsf{then}\ e\ \mathsf{else}\ ()$
- Under sequential consistency, at most one of these cores will execute expression e.
- With write buffers, both cores may read 0 and both cores execute expression e!

Effect system for determinism

Memory region

Syntax Changes

$$e ::= \cdots \mid \mathsf{ref}_\alpha\ e \mid \ell_\alpha$$
$$v ::= \cdots \mid \ell_\alpha$$

Computational effect

New Judgement

We write $\Gamma, \Sigma \vdash e : \tau \triangleright R, W$ to mean that 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).

Extended Function Type

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.

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

Static Rules

$$\frac{}{\Gamma, \Sigma \vdash n : \mathbf{int} \triangleright \emptyset, \emptyset}$$

$$\frac{\Gamma(x) = \tau}{\Gamma, \Sigma \vdash x : \tau \triangleright \emptyset, \emptyset}$$

$$\frac{\Gamma[x \mapsto \tau], \Sigma \vdash e : \tau' \triangleright R, W}{\Gamma, \Sigma \vdash \lambda x : \tau.\, e : \tau \xrightarrow{R,W} \tau' \triangleright \emptyset, \emptyset}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau \xrightarrow{R,W} \tau' \triangleright R_1, W_2 \quad \Gamma, \Sigma \vdash e_2 : \tau \triangleright R_2, W_2}{\Gamma, \Sigma \vdash e_1\ e_2 : \tau' \triangleright R_1 \cup R_2 \cup R, W_1 \cup W_2 \cup W}$$

Static Rules

$$\frac{\Gamma, \Sigma \vdash e : \tau \triangleright R, W}{\Gamma, \Sigma \vdash \mathsf{ref}_\alpha\ e : \tau\ \mathbf{ref}_\alpha \triangleright R, W}$$

$$\frac{\Gamma, \Sigma \vdash e : \tau\ \mathbf{ref}_\alpha \triangleright R, W}{\Gamma, \Sigma \vdash\ !e : \tau \triangleright R \cup \{\alpha\}, W}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau\ \mathbf{ref}_\alpha \triangleright R_1, W_2 \quad \Gamma, \Sigma \vdash e_2 : \tau \triangleright R_2, W_2}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau \triangleright R_1 \cup R_2, W_1 \cup W_2 \cup \{\alpha\}}$$

$$\frac{}{\Gamma, \Sigma \vdash \ell_\alpha : \tau\ \mathbf{ref}_\alpha \triangleright \emptyset, \emptyset}\ \Sigma(\ell_\alpha) = \tau\ \mathbf{ref}_\alpha$$

Static Rules

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \triangleright R_1, W_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2 \triangleright R_2, W_2}{\Gamma, \Sigma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \triangleright R_1 \cup R_2, W_1 \cup W_2}$$

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

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

Static Rules

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \triangleright R_1, W_1 \quad W_1 \cap (R_2 \cup W_2) = \emptyset \qquad \Gamma, \Sigma \vdash e_2 : \tau_2 \triangleright R_2, W_2 \quad W_2 \cap (R_1 \cup W_1) = \emptyset}{\Gamma, \Sigma \vdash e_1 \,||\, e_2 : \tau_1 \times \tau_2 \triangleright R_1 \cup R_2, W_1 \cup W_2}$$

Static Rules

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 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.

Type soundness?

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 $e||E$.)

Type Soundness

If $\vdash e : \tau \triangleright R, W$ then for all stores σ and σ',

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

Type soundness

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.)

Determinism

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

Determinism

If $\Gamma, \Sigma \vdash e : \tau \triangleright R, W$ and $< e, \sigma > \longrightarrow^* < v_1, \sigma_1 >$ and $e \longrightarrow^* < v_2, \sigma_2 >$ then $v_1 = v_2$.

Proof of Determinism

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.

Proof of Determinism (Key Lemma)

If for some Σ, τ, R and W we have $\emptyset, \Sigma \vdash e_1||e_2 : \tau \triangleright R, W$, then for all σ such that $\Gamma, \Sigma \vdash \sigma$ if

$$< e_1||e_2, \sigma > \longrightarrow < e_1||e_2', \sigma' > \longrightarrow < e_1'||e_2', \sigma'' >,$$

then there exists σ''' such that

$$< e_1||e_2, \sigma > \longrightarrow < e_1'||e_2, \sigma''' > \longrightarrow < e_1'||e_2', \sigma'' > .$$

Proof of Determinism (Key Lemma)

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.

Abstractions for Concurrency

communication

coordination

atomicity

Message passing

- Shared memory model of concurrency can make it difficult to reason about the interactions between threads.
- Message passing: a different model of concurrency.

Message passing

- Threads communicate by sending and receiving messages over channels.
- Channels are first-class values: they can be created at runtime, and used as values, including being passed as messages over channels.
- Several languages use message passing, including Erlang, Go, Rust, Racket, X10, Smalltalk, F#, Concurrent ML (CML), and others.

Message passing: Syntax

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

Message passing: Type system

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{spawn } e : \textbf{unit}} \qquad \frac{}{\Gamma \vdash \text{newchan}_\tau : \tau \textbf{ chan}}$$

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

Message passing: Operational Semantic

- A configuration is now a list of expressions, one expression for each thread. That is, configuration $< e_1, \ldots, e_n >$ represents the concurrent execution of n threads.
- Use judgement $< e_1, \ldots, e_n > \Longrightarrow < e'_1, \ldots, e'_m >$ to indicate that configuration $< e_1, \ldots, e_n >$ can take a small step to $< e'_1, \ldots, e'_m >$.
- Use judgement $e \longrightarrow e'$ to indicate that thread e can take a small step to e'.

Message passing: $e \longrightarrow e'$

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

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x{:}\tau.\ e)\ v \longrightarrow e\{v/x\}}$$

$$\frac{}{n_1 + n_2 \longrightarrow m}\ m = n_1 + n_2$$

$$\frac{}{\mu x{:}\tau.\ e \longrightarrow e\{(\mu x{:}\tau.\ e)/x\}}$$

$$\frac{}{\text{newchan}_\tau \longrightarrow c}\ c \text{ is fresh}$$

Message passing: Notation

As a notational convenience, we write $< e_1, \ldots, e_n >_{i \mapsto e'}$ as shorthand for the configuration $< e_1, \ldots, e_{i-1}, e', e_{i+1}, \ldots, e_n >$, i.e., where the ith thread is replaced with expression e'.

Message passing:

$$< e_1, \ldots, e_n > \Longrightarrow < e'_1, \ldots, e'_m >$$

$$\frac{e_i \longrightarrow e'_i}{< e_1, \ldots, e_n > \Longrightarrow < e_1, \ldots, e_n >_{i \mapsto e'_i}}$$

$$\frac{e_i = E[\text{spawn } e]}{< e_1, \ldots, e_n > \Longrightarrow < e_1, \ldots, e_n, e >_{i \mapsto E[()]}}$$

$$\frac{e_i = E_i[\text{send } v \text{ to } c] \quad e_j = E_j[\text{recv from } c]}{< e_1, \ldots, e_n > \Longrightarrow < e_1, \ldots, e_n >_{i \mapsto E_i[()], j \mapsto E_j[v]}}$$

Message passing: Notation

For convenience, we write $e_1; e_2$ as shorthand for let $x = e_1$ in e_2 where $x \notin FV(e_2)$. (And let $x = e_1$ in e_2 as itself shorthand for a function application.)

Message passing: Example 1

let $c = \text{newchan}_{\textbf{int}}$ in
spawn (send 35 to c);
recv from $c + 7$

Message passing: Example 2

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

Message passing: Example 3

let $c = \text{newchan}_{\textbf{int}}$ in
spawn $(3 + \text{recv from } c)$;
spawn $(5 + \text{recv from } c)$;
send 15 to c