Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Concurrency

Lecture 26 Tuesday, April 27, 2021

1 Semantics of 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.

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

There are several possible final values that this program could evaluate to: 25, 50, 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 Weak memory models

In the calculus above, we have used a very simple model of shared memory, i.e., of imperatively updatable memory shared by multiple threads. In particular, our model assumes *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.

That is, our model behaves as if there were a single global memory, and only one thread at a time is allowed to access the memory. While this may be a reasonable model for how uniprocessor machines execute, it is not a good description for how multi-processor machines (such as CPUs with multiple cores) execute. Indeed, requiring sequential consistency prevents many compiler and hardware optimizations, because it enforces a strict order among memory operations.

Let's consider an example hardware optimization: write buffers with bypassing capabilities. Here, each core has its own write buffer, and 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. This is a common hardware optimization used in uniprocessors, since it helps hide the latency of write operations. But with multiple cores, this can violate sequential consistency! Suppose that a and b are two memory locations, both containing zero, and we have one core that executes the expression $\mathsf{a} := 1; \mathsf{if}\ !\mathsf{b} = 0\ \mathsf{then}\ e\ \mathsf{else}\ ()$ and the other 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. However, with write buffers, both cores may read 0 and both cores execute expression e!

In practice, machines do not provide sequential consistency. Instead they offer weaker guarantees. It is important for there to be a clear specification of what guarantees are actually provided for shared memory. These "memory models" allow programmers (including compiler writers) to reason about the correctness of their code, and affect the performance of the system, because it restricts the kinds of optimizations that hardware architects can provide. An ongoing area of research is in using programming language techniques to formally specify these weak memory models, to enable formal and precise reasoning about the correctness of programs and programming language abstractions. Although the specifics of these weak memory models are beyond the scope of this course,[1] we will examine programming language abstractions and techniques that enable programmers to ignore the details of the weak memory model, and deal with higher-level abstractions. (Of course, the language implementors must then understand the appropriate weak memory model in order to implement the abstraction correctly.)

1.3 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

[1]For a tutorial about weak memory models, look at the paper "Shared memory consistency models: a tutorial" by S.V. Adve, S.V. and K. Gharachorloo, *IEEE Computer*, vol. 29, no. 12, pp. 66–76, IEEE Computer Society, 1996. For a formalization of weak memory models, see "Noninterference under Weak Memory Models" by H. Mantel, M. Perner and J. Sauer, in *Proceedings of the 27th IEEE Computer Security Foundations Symposium*, pages 80–94, IEEE Computer Society, 2014.

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 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.

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

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

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

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

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

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

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.

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

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.

2 Abstractions for Concurrency: Message passing

We just looked at a shared memory model of concurrency: different threads could concurrently access the same memory locations. A shared memory model is commonly used in many programming languages. However, shared memory can make it difficult to reason about the interaction between threads.

Now we consider a different model of concurrency: message passing. In this model, 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.

The following grammar describes our language.

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

The language is a lambda calculus with integers, fixpoints ($\mu f.\ e$), and primitives for creating threads, and creating and using channels. Primitive spawn e creates a new thread, and expression e will execute in the new thread. A new channel can be created with primitive newchan$_\tau$. The type annotation τ will be used to enforce that the new channel is used to send and receive values of type τ.

Expression send e_1 to e_2 computes e_1 to a value v, computes e_2 to a channel c, and sends v over channel c. The send "blocks" until some other thread executes a recv on the same channel, and then evaluates to the unit value $()$. That is, execution of the send e_1 to e_2 expression doesn't complete until some thread receives the value. Expression recv from e evaluates e to a channel c and then blocks until it receives a value on channel c. The expression evaluates to the value received.

The type system for this language expresses some aspects of the intended behavior of these new primitives.

$$
\frac{\Gamma \vdash e\!:\!\tau}{\Gamma \vdash \mathsf{spawn}\ e\!:\!\textbf{unit}}
\qquad
\frac{}{\Gamma \vdash \mathsf{newchan}_\tau\!:\!\tau\ \textbf{chan}}
\qquad
\frac{\Gamma \vdash e_1\!:\!\tau \quad \Gamma \vdash e_2\!:\!\tau\ \textbf{chan}}{\Gamma \vdash \mathsf{send}\ e_1\ \mathsf{to}\ e_2\!:\!\textbf{unit}}
\qquad
\frac{\Gamma \vdash e\!:\!\tau\ \textbf{chan}}{\Gamma \vdash \mathsf{recv\ from}\ e\!:\!\tau}
$$

2.1 Operational semantics

A configuration is now a list of expressions, one expression for each thread. That is, configuration $\langle e_1, \ldots, e_n\rangle$ represents the concurrent execution of n threads.

We use two judgements to define the operational semantics, one to indicate how a configuration (i.e., a list of threads) takes a step, and one to indicate how an individual thread takes a step. We use judgment $\langle e_1, \ldots, e_n\rangle \Longrightarrow \langle e'_1, \ldots, e'_m\rangle$ to indicate that configuration $\langle e_1, \ldots, e_n\rangle$ can take a small step to $\langle e'_1, \ldots, e'_m\rangle$ and we write $e \longrightarrow e'$ to indicate that thread e can take a small step to e'. For clarity, we present the entire operational semantics for the language here.

We first present the inference rules for the thread judgment $e \longrightarrow e'$.

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

$$
\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}
\qquad
\frac{}{(\lambda x\!:\!\tau.\, e)\ v \longrightarrow e\{v/x\}}
\qquad
\frac{}{n_1 + n_2 \longrightarrow m}\ m = n_1 + n_2
\qquad
\frac{}{\mu x\!:\!\tau.\ e \longrightarrow e\{(\mu x\!:\!\tau.\ e)/x\}}
$$

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

We now present the rules for judgment $\langle e_1, \ldots, e_n\rangle \Longrightarrow \langle e'_1, \ldots, e'_m\rangle$. As a notational convenience, we write $\langle e_1, \ldots, e_n\rangle_{i \mapsto e'}$ as shorthand for the configuration $\langle e_1, \ldots, e_{i-1}, e', e_{i+1}, \ldots, e_n\rangle$, i.e., where the ith thread is replaced with expression e'.

$$
\frac{e_i \longrightarrow e'_i}{\langle e_1, \ldots, e_n\rangle \Longrightarrow \langle e_1, \ldots, e_n\rangle_{i \mapsto e'_i}}
\qquad
\frac{e_i = E[\mathsf{spawn}\ e]}{\langle e_1, \ldots, e_n\rangle \Longrightarrow \langle e_1, \ldots, e_n, e\rangle_{i \mapsto E[()]}}
$$

$$
\frac{e_i = E_i[\mathsf{send}\ v\ \mathsf{to}\ c] \quad e_j = E_j[\mathsf{recv\ from}\ c]}{\langle e_1, \ldots, e_n\rangle \Longrightarrow \langle e_1, \ldots, e_n\rangle_{i \mapsto E_i[()],\, j \mapsto E_j[v]}}
$$

The first rule for configurations states allows a thread of the configuration to take a step. This rule is nondeterministic: any thread that can take a step is allowed to. That is, we are not modeling any scheduler that restricts the order in which threads may execute.

The next rule describes spawning a new thread ($\mathsf{spawn}\ e$), which is added to the end of the list of threads. The last rule handles a matching send and receive on the same channel. The thread evaluating the receive term evaluates to the value sent by the other thread. This semantics enforces blocking, in that the thread sending cannot complete the send operation until some other thread executes a receive, and vice versa.

2.2 Example

For convenience, we write $e_1; e_2$ as shorthand for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ where $x \notin FV(e_2)$. (This notation is only useful in a language with side-effects. Why?)

The follow program creates a new channel, spawns a thread that sends an integer value on the channel, and the original thread receives the value and adds 7 to it.

$$\begin{array}{l}\mathsf{let}\ c = \mathsf{newchan}_{\mathsf{int}}\ \mathsf{in}\\ \mathsf{spawn}\ (\mathsf{send}\ 35\ \mathsf{to}\ c);\\ \mathsf{recv\ from}\ c + 7\end{array}$$

The final configuration for this program will be $\langle 42, ()\rangle$. Make sure you understand how this configuration is derived from an initial configuration that contains a single thread that executes the program above.

What about the following program? What does it do?

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

Note that even though there is no shared memory, the language is still non-deterministic. Consider the following program, where 2 threads are executing a receive instruction, and one thread is executing a send, all on the same channel. What are the possible final configurations?

$$\begin{array}{l}\mathsf{let}\ c = \mathsf{newchan}_{\mathsf{int}}\ \mathsf{in}\\ \mathsf{spawn}\ (3 + \mathsf{recv\ from}\ c);\\ \mathsf{spawn}\ (5 + \mathsf{recv\ from}\ c);\\ \mathsf{send}\ 15\ \mathsf{to}\ c\end{array}$$