Concurrency Semantics

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Shared Data

Consider the **Readers and Writers** problem:

Problem

We have a large data structure (i.e. a structure that cannot be updated in one atomic step) that is shared between some number of writers who are updating the data structure and some number of readers who are attempting to retrieve a coherent copy of the data structure.

Desiderata:

- We want *atomicity*, in that each update happens in one go, and updates-in-progress or partial updates are not observable.
- We want *consistency*, in that any reader that starts after an update finishes will see that update.
- We want to minimise *waiting*.

A Crappy Solution

Treat both reads and updates as critical sections — use any old critical section solution (locks, etc.) to sequentialise all reads and writes to the data structure.

Observation

Updates are *atomic* and reads are *consistent* — but reads can't happen concurrently, which leads to unnecessary *contention*.

A Crappy Solution

Treat both reads and updates as critical sections — use any old critical section solution (locks, etc.) to sequentialise all reads and writes to the data structure.

Observation

Updates are *atomic* and reads are *consistent* — but reads can't happen concurrently, which leads to unnecessary *contention*.

A Better Solution

A more elaborate locking mechanism (*condition variables*) could be used to to allow multiple readers to read concurrently, but writers are still executed individually and atomically.

Observation

We have atomicity and consistency, and now multiple reads can execute concurrently. Still, we don't allow updates to execute concurrently with reads, to prevent partial updates from being observed by a reader.

A Better Solution

A more elaborate locking mechanism (*condition variables*) could be used to to allow multiple readers to read concurrently, but writers are still executed individually and atomically.

Observation

We have atomicity and consistency, and now multiple reads can execute concurrently. Still, we don't allow updates to execute concurrently with reads, to prevent partial updates from being observed by a reader.

Reading and Writing

Complication

Now suppose we don't want readers to wait (much) while an update is performed. Instead, we'd rather they get an *older version* of the data structure.

Trick: Rather than update the data structure in place, a writer creates *their own local copy* of the data structure, and then merely updates the (shared) *pointer* to the data structure to point to their copy.

Liam: Draw on the board

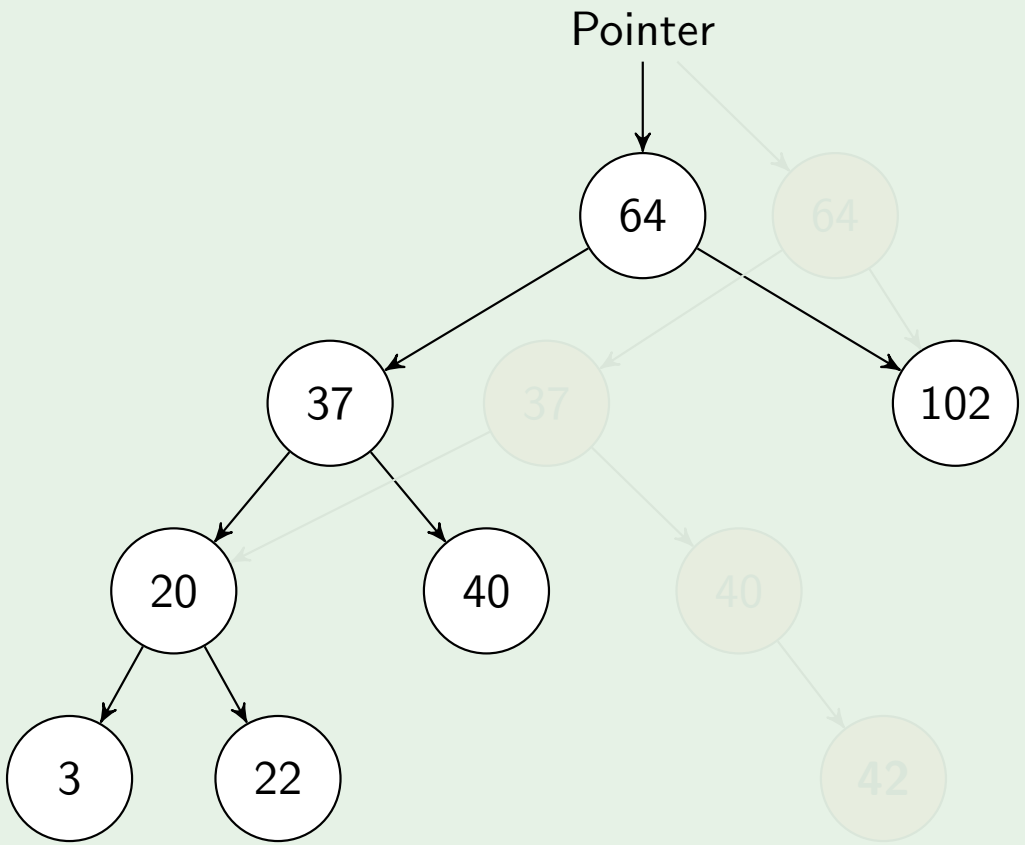
Atomicity The only shared write is now just to one pointer.

Consistency Reads that start before the pointer update get the older version, but reads that start after get the latest.

Persistent Data Structures

Copying is $\mathcal{O}(n)$ in the worst case, but we can do better for many tree-like types of data structure.

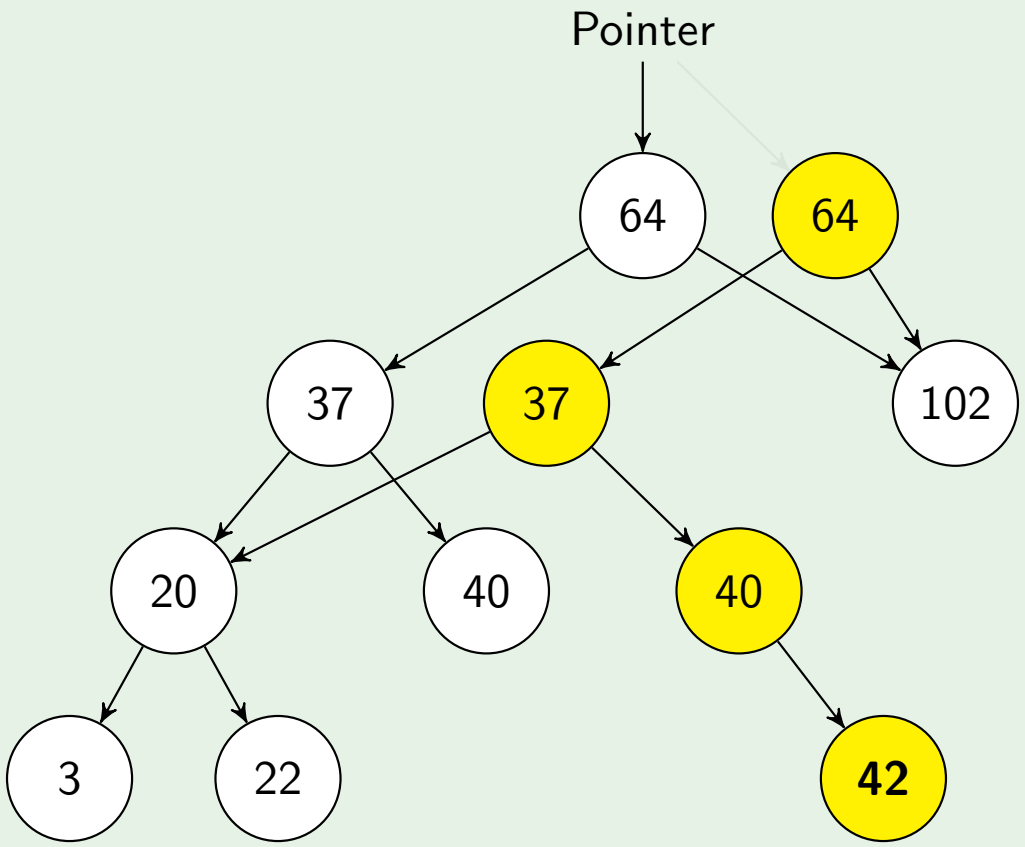
Example (Binary Search Tree)



Persistent Data Structures

Copying is $\mathcal{O}(n)$ in the worst case, but we can do better for many tree-like types of data structure.

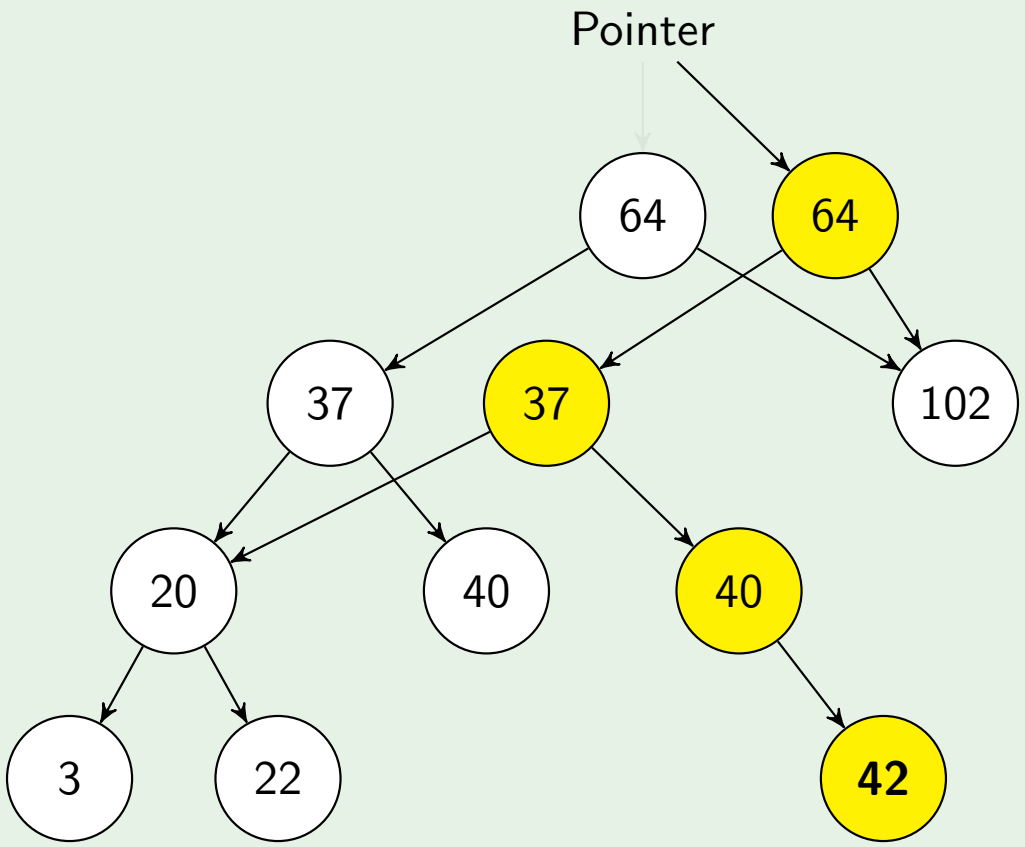
Example (Binary Search Tree)



Persistent Data Structures

Copying is $\mathcal{O}(n)$ in the worst case, but we can do better for many tree-like types of data structure.

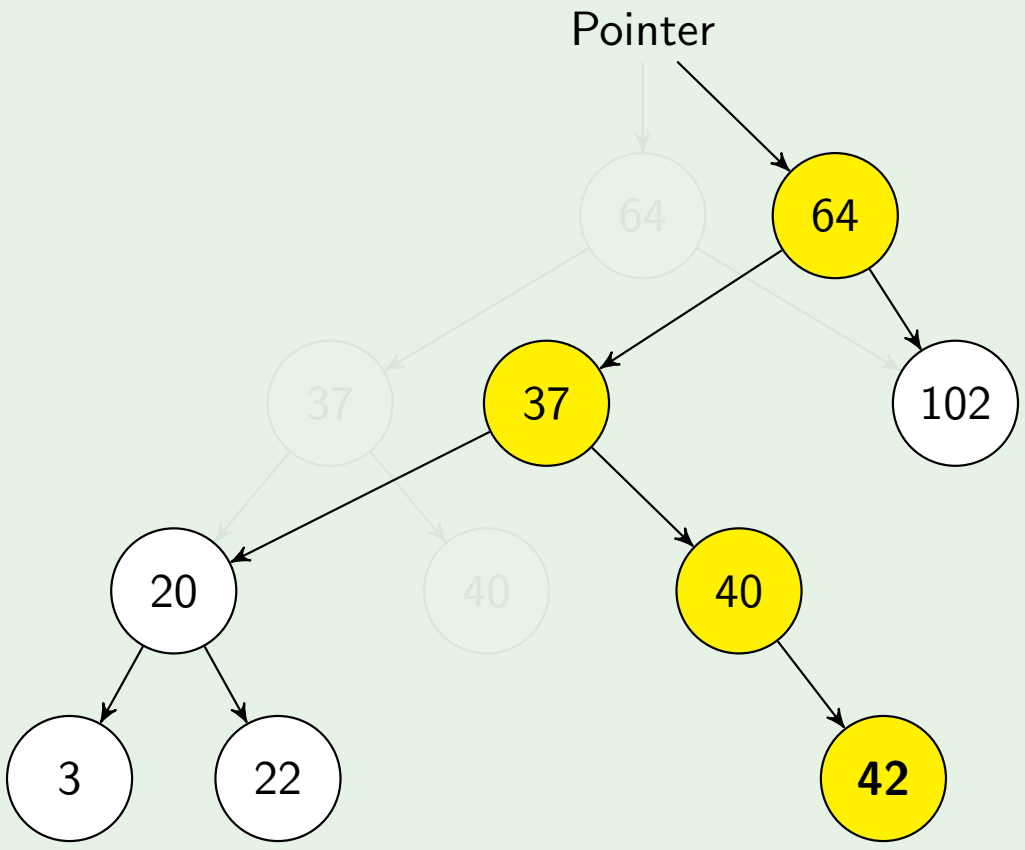
Example (Binary Search Tree)



Persistent Data Structures

Copying is $\mathcal{O}(n)$ in the worst case, but we can do better for many tree-like types of data structure.

Example (Binary Search Tree)



Purely Functional Data Structures

Persistent data structures that exclusively make use of copying over mutation are called *purely functional* data structures. They are so called because operations on them are best expressed in the form of mathematical functions that, given an input structure, return a *new* output structure:

$$
\begin{aligned}
&\mathit{insert}\ v\ \mathrm{Leaf} &&= \mathrm{Branch}\ v\ \mathrm{Leaf}\ \mathrm{Leaf} \\
&\mathit{insert}\ v\ (\mathrm{Branch}\ x\ l\ r) &&= \textbf{if}\ v \leq x\ \textbf{then} \\
&&&\quad \mathrm{Branch}\ x\ (\mathit{insert}\ v\ l)\ r \\
&&&\textbf{else} \\
&&&\quad \mathrm{Branch}\ x\ l\ (\mathit{insert}\ v\ r)
\end{aligned}
$$

Computing with Functions

We model real processes in Haskell using the IO type. We have previously seen a definition of IO that resembles the state monad:

$$\textbf{type}\ \mathrm{IO}\ a = (\text{🌍} \to (\text{🌍}, a))$$

But this is not a good model where concurrency is concerned. Why?

Instead, we'll just treat IO as an abstract type for now, and give it a formal semantics later:

$$\mathrm{IO}\ \tau = \text{A (possibly effectful) process that, when executed, produces a result of type } \tau$$

Note the semantics of *evaluation* and *execution* are different things.

Computing with Functions

We model real processes in Haskell using the IO type. We have previously seen a definition of IO that resembles the state monad:

$$\textbf{type}\ \mathrm{IO}\ a = (\text{🌍} \to (\text{🌍}, a))$$

But this is not a good model where concurrency is concerned.
Why?
Instead, we'll just treat IO as an abstract type for now, and give it a formal semantics later:

$$\mathrm{IO}\ \tau\ = \begin{array}{l}\text{A (possibly effectful) process that, when executed,}\\ \text{produces a result of type } \tau\end{array}$$

Note the semantics of *evaluation* and *execution* are different things.

Building up IO

Recall monads:

$$\begin{aligned}
&\mathit{return} :: \forall a.\ a \to \mathrm{IO}\ a \\
&(\gg\!=) :: \forall a\ b.\ \mathrm{IO}\ a \to (a \to \mathrm{IO}\ b) \to \mathrm{IO}\ b \\
&\mathit{getChar} :: \mathrm{IO}\ \mathrm{Char} \\
&\mathit{putChar} :: \mathrm{Char} \to \mathrm{IO}\ ()
\end{aligned}$$

Example (Echo)

$$\begin{aligned}
&\mathit{echo} :: \mathrm{IO}\ () \\
&\mathit{echo} = \mathit{getChar} \gg\!= (\lambda x.\ \mathit{putChar}\ x \gg\!= \lambda y.\ \mathit{echo})
\end{aligned}$$

Or, with **do** notation:

$$\begin{aligned}
&\mathit{echo} :: \mathrm{IO}\ () \\
&\mathit{echo} = \mathbf{do}\ \ x \leftarrow \mathit{getChar} \\
&\qquad\qquad\ \ \mathit{putChar}\ x \\
&\qquad\qquad\ \ \mathit{echo}
\end{aligned}$$

Adding Concurrency

We can have multiple threads easily enough:

$$\mathit{forkIO} :: \mathrm{IO}\ () \ \rightarrow \mathrm{IO}\ ()$$

Example (Dueling Printers)

let *loop c* = **do** *putChar c*; *loop c*
in do *forkIO* (*loop* 'a'); *loop* 'z'

But what sort of *synchronisation primitives* are available?

MVars

The MVar is the simplest synchronisation primitive in Haskell. It can be thought of as a shared box which holds at most one value.

Processes must take the value out of a full box to read it, and must put a value into an empty box to update it.

MVar Functions

$newMVar :: \forall a.\ a \to \mathrm{IO}\ (\mathrm{MVar}\ a)$	Create a new MVar
$takeMVar :: \forall a.\ \mathrm{MVar}\ a \to \mathrm{IO}\ a$	Read/remove the value
$putMVar :: \forall a.\ \mathrm{MVar}\ a \to a \to \mathrm{IO}\ ()$	Update/insert a value

Taking from an empty MVar or putting into a full one results in blocking.
An MVar can be thought of as channel containing at most one value.

Readers and Writers

We can treat MVars as shared variables with some definitions:

$$writeMVar\ m\ v = \mathbf{do}\ takeMVar\ m; putMVar\ m\ v$$
$$readMVar\ m = \mathbf{do}\ v \leftarrow takeMVar\ m; putMVar\ m\ v; return\ v$$

$$problem :: DB \to \mathrm{IO}\ ()$$
$$problem\ initial = \mathbf{do}$$
$$\quad db \leftarrow newMVar\ initial$$
$$\quad wl \leftarrow newMVar\ ()$$
$$\quad \mathbf{let}\ reader = readMVar\ db \gg= \cdots$$
$$\quad \mathbf{let}\ writer = \mathbf{do}$$
$$\quad\quad takeMVar\ wl$$
$$\quad\quad d \leftarrow readMVar\ db$$
$$\quad\quad \mathbf{let}\ d' = update\ d$$
$$\quad\quad evaluate\ d'$$
$$\quad\quad writeMVar\ db\ d'$$
$$\quad\quad putMVar\ wl\ ()$$

Fairness

Each MVar has an attached FIFO queue, so GHC Haskell can ensure the following fairness property:

> *No thread can be blocked indefinitely on an MVar unless another thread holds that MVar indefinitely.*

Evaluation Semantics

The semantics of Haskell's evaluation are interesting but not particularly relevant for us. We will assume that it happens quietly without a fuss:

$$\begin{array}{lrcl} \beta\textbf{-equivalence} & (\lambda x.\ M[x])\ N & \equiv_\beta & M[N] \\ \alpha\textbf{-equivalence} & \lambda x.\ M[x] & \equiv_\alpha & \lambda y.\ M[y] \\ \eta\textbf{-equivalence} & \lambda x.\ M\ x & \equiv_\eta & M \end{array}$$

Let our ambient congruence relation $\equiv$ be $\equiv_{\alpha\beta\eta}$ enriched with the following extra equations, justified by the *monad laws*:

$$\begin{array}{lcl} \mathit{return}\ N \gg\!= M & \equiv & M\ N \\ (X \gg\!= Y) \gg\!= Z & \equiv & X \gg\!= (\lambda x.\ Y\ x \gg\!= Z) \\ X & \equiv & X \gg\!= \mathit{return} \end{array}$$

Processes

This means that a Haskell expression of type $\mathrm{IO}\ \tau$ for will boil down to either *return* x where x is a value of type τ; or $a \gg\!= M$ where a is some *primitive* IO *action* (*forkIO* p, *readMVar* v, etc.) and M is some function producing another $\mathrm{IO}\ \tau$. This is the *head normal form* for IO expressions.

Definition

Define a language of *processes* P, which contains all (head-normal) expressions of type $\mathrm{IO}\ ()$.

We want to define the semantics of the *execution* of these processes. Let's use *operational semantics*:

$$(\mapsto) \subseteq P \times P$$

Semantics for forkIO

To model *forkIO*, we need to model the parallel execution of multiple processes in our process language. We shall add a *parallel composition* operator to the language of processes:

$$\begin{array}{rcl} P, Q & ::= & a \gg\!= M \\ & \mid & \mathit{return}\ () \\ & \mid & P \parallel Q \\ & \mid & \ldots \end{array}$$

And the following ambient congruence equations:

$$\begin{array}{rcl} P \parallel Q & \equiv & Q \parallel P \\ P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R \end{array}$$

Semantics for forkIO

To model *forkIO*, we need to model the parallel execution of multiple processes in our process language. We shall add a *parallel composition* operator to the language of processes:

$$\begin{array}{rcl} P, Q & ::= & a \gg\!= M \\ & | & \mathit{return}\ () \\ & | & P \parallel Q \\ & | & \ldots \end{array}$$

And the following ambient congruence equations:

$$\begin{array}{rcl} P \parallel Q & \equiv & Q \parallel P \\ P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R \end{array}$$

Semantics for forkIO

If we have multiple processes active, pick one of them non-deterministically to move:

$$\frac{P \mapsto P'}{P \parallel Q \mapsto P' \parallel Q}$$

The *forkIO* operation introduces a new process:

$$(\mathit{forkIO}\ P \gg\!= M) \quad \mapsto \quad P \parallel (\mathit{return}\ () \gg\!= M)$$

Semantics for forkIO

If we have multiple processes active, pick one of them non-deterministically to move:

$$\frac{P \mapsto P'}{P \parallel Q \mapsto P' \parallel Q}$$

The *forkIO* operation introduces a new process:

$$(\mathit{forkIO}\ P \gg\!= M) \quad \mapsto \quad P \parallel (\mathit{return}\ () \gg\!= M)$$

Semantics for MVars

MVars are modelled as a special type of *process*, identified by a *unique name*. Values of MVar type merely contain the name of the process, so that *putMVar* and friends know where to look.

$$
\begin{array}{rcl}
P, Q & ::= & a \gg\!= M \\
& \mid & \mathit{return}\ () \\
& \mid & P \parallel Q \\
& \mid & \langle\rangle_n \quad \mid \quad \langle v \rangle_n \\
& \mid & \cdots
\end{array}
$$

$$\langle\rangle_n \parallel (\mathit{putMVar}\ n\ v \gg\!= M) \quad \mapsto \quad \langle v \rangle_n \parallel (\mathit{return}\ () \gg\!= M)$$

$$\langle v \rangle_n \parallel (\mathit{takeMVar}\ n \gg\!= M) \quad \mapsto \quad \langle\rangle_n \parallel (\mathit{return}\ v \gg\!= M)$$

Semantics for MVars

MVars are modelled as a special type of *process*, identified by a *unique name*. Values of MVar type merely contain the name of the process, so that *putMVar* and friends know where to look.

$$\begin{array}{rcl} P, Q & ::= & a \gg\!= M \\ & | & return\ () \\ & | & P \parallel Q \\ & | & \langle\rangle_n \quad | \quad \langle v \rangle_n \\ & | & \cdots \end{array}$$

$$\langle\rangle_n \parallel (putMVar\ n\ v \gg\!= M) \quad \mapsto \quad \langle v \rangle_n \parallel (return\ () \gg\!= M)$$

$$\langle v \rangle_n \parallel (takeMVar\ n \gg\!= M) \quad \mapsto \quad \langle\rangle_n \parallel (return\ v \gg\!= M)$$

Semantics for MVars

MVars are modelled as a special type of *process*, identified by a *unique name*. Values of MVar type merely contain the name of the process, so that *putMVar* and friends know where to look.

$$
\begin{array}{rcl}
P, Q & ::= & a \gg\!= M \\
& | & return\ () \\
& | & P \parallel Q \\
& | & \langle\rangle_n \quad | \quad \langle v \rangle_n \\
& | & \cdots
\end{array}
$$

$$\langle\rangle_n \parallel (putMVar\ n\ v \gg\!= M) \quad \mapsto \quad \langle v \rangle_n \parallel (return\ () \gg\!= M)$$

$$\langle v \rangle_n \parallel (takeMVar\ n \gg\!= M) \quad \mapsto \quad \langle\rangle_n \parallel (return\ v \gg\!= M)$$

Semantics for newMVar

We might think that *newMVar* should have semantics like this:

$$\frac{}{(\mathit{newMVar}\ v \gg\!= M) \ \mapsto\ \langle v \rangle_n \parallel (\mathit{return}\ n \gg\!= M)}(n \text{ fresh})$$

But this approach has a number of problems:

- The name n is now globally-scoped, without an explicit binder to introduce it.
- It doesn't accurately model the *lifetime* of the MVar, which should be garbage-collected once all processes that can access it finish.
- It makes MVars *global* objects, so our semantics aren't very *abstract*. We would like local communication to be local in our model.

Restriction Operator

We introduce a *restriction operator* ν to our language of processes:

$$\begin{array}{rcl} P, Q & ::= & a \gg\!= M \\ & \mid & \mathit{return}\ () \\ & \mid & P \parallel Q \\ & \mid & \langle\rangle_n \quad \mid \quad \langle v \rangle_n \\ & \mid & (\nu\ n)\ P \end{array}$$

Writing $(\nu\ n)\ P$ says that the MVar name n is *only* available in process P. Mentioning n outside P is not well-formed. We need the following additional congruence equations:

$$\begin{array}{rcll} (\nu\ n)\ (\nu\ m)\ P & \equiv & (\nu\ m)\ (\nu\ n)\ P & \\ (\nu\ n)(P \parallel Q) & \equiv & P \parallel (\nu\ n)\ Q & (\text{if } n \notin P) \end{array}$$

Restriction Operator

We introduce a *restriction operator* ν to our language of processes:

$$\begin{array}{rcl} P, Q & ::= & a \gg\!= M \\ & \mid & \mathit{return}\ () \\ & \mid & P \parallel Q \\ & \mid & \langle\rangle_n \quad \mid \quad \langle v \rangle_n \\ & \mid & (\nu\ n)\ P \end{array}$$

Writing $(\nu\ n)\ P$ says that the MVar name n is *only* available in process P. Mentioning n outside P is not well-formed. We need the following additional congruence equations:

$$\begin{array}{lcll} (\nu\ n)\ (\nu\ m)\ P & \equiv & (\nu\ m)\ (\nu\ n)\ P & \\ (\nu\ n)(P \parallel Q) & \equiv & P \parallel (\nu\ n)\ Q & (\text{if } n \notin P) \end{array}$$

Better Semantics for newMVar

The rule for *newMVar* is much the same as before, but now we explicitly restrict the MVar to M.

$$\frac{}{(\mathit{newMVar}\ v \gg\!= M) \ \mapsto\ (\nu\ n)(\langle v \rangle_n \parallel (\mathit{return}\ n \gg\!= M))}(n \text{ fresh})$$

We can always execute under a restriction:

$$\frac{P \mapsto P'}{(\nu\ n)\ P \mapsto (\nu\ n)\ P'}$$

Question

What happens when you put an MVar inside another MVar?

Better Semantics for newMVar

The rule for *newMVar* is much the same as before, but now we explicitly restrict the MVar to M.

$$\frac{}{(\mathit{newMVar}\ v \gg\!= M) \ \mapsto\ (\nu\ n)(\langle v \rangle_n \parallel (\mathit{return}\ n \gg\!= M))}(n \text{ fresh})$$

We can always execute under a restriction:

$$\frac{P \mapsto P'}{(\nu\ n)\ P \mapsto (\nu\ n)\ P'}$$

Question

What happens when you put an MVar inside another MVar?

Better Semantics for newMVar

The rule for *newMVar* is much the same as before, but now we explicitly restrict the MVar to M.

$$\frac{}{(\mathit{newMVar}\ v \gg\!= M) \ \mapsto\ (\nu\ n)(\langle v \rangle_n \parallel (\mathit{return}\ n \gg\!= M))}(n \text{ fresh})$$

We can always execute under a restriction:

$$\frac{P \mapsto P'}{(\nu\ n)\ P \mapsto (\nu\ n)\ P'}$$

Question

What happens when you put an MVar inside another MVar?

Garbage Collection

If an MVar is no longer used, we just replace it with the do-nothing process:

$$(\nu\ n)\ \langle\rangle_n \quad \mapsto \quad return\ ()$$
$$(\nu\ n)\ \langle v\rangle_n \quad \mapsto \quad return\ ()$$

Extra processes that have outlived their usefulness disappear:

$$return\ ()\ \|\ P \quad \mapsto \quad P$$

Garbage Collection

If an MVar is no longer used, we just replace it with the do-nothing process:

$$(\nu\ n)\ \langle\rangle_n \quad \mapsto \quad return\ ()$$
$$(\nu\ n)\ \langle v\rangle_n \quad \mapsto \quad return\ ()$$

Extra processes that have outlived their usefulness disappear:

$$return\ () \parallel P \quad \mapsto \quad P$$

Process Algebra

Our language P is called a *process algebra*, a common means of describing semantics for concurrent programs.
Process algebras and calculi of various kinds are covered in great detail in COMP6752 with Rob van Glabbeek, who is an expert in this field.
The last topic of this course is Software Transactional Memory, which we will cover in Week 12.

If there's time!

We can talk about more concurrency topics.

Bibliography

Simon Marlow

Parallel and Concurrent Programming in Haskell

O'Reilly, 2013

http://chimera.labs.oreilly.com/books/1230000000929

Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne

Concurrent Haskell

POPL '96

Association for Computer Machinery

http://microsoft.com/en-us/research/wp-content/uploads/1996/01/concurrent-haskell.pdf

Simon Marlow (Editor)

Haskell 2010 Language Report

https://www.haskell.org/onlinereport/haskell2010/