Haskell Concurrency and STM

Johannes Åman Pohjola
UNSW
Term 3 2022
Consider the **Readers and Writers** problem:

**Problem**

We have a **large shared data structure**, that can’t be updated in one atomic step. **Writers** are updating it, and **readers** try to retrieve a coherent copy of it.

We want:

- **Atomicity**: partial updates are not observable.
- **Consistency**: any reader that starts after a finished update will see that update.
- Minimal **waiting**.

**Shared Data**
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.
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.

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 allow multiple concurrent readers. Writers still require exclusive access.
A Better Solution

A more elaborate locking mechanism (condition variables) could be used to allow multiple concurrent readers. Writers still require exclusive access.

Observation

This reduces contention. We can’t let updates execute concurrently with reads; otherwise, partial updates would be observable.
Complication

Suppose we don’t want readers to wait (much) while an update is performed. Instead, we’ll give them an older version of the data.

**Trick:** A writer creates *their own local copy* of the data, and then updates just the *pointer* to the data.

**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 $O(n)$ in the worst case, but we can do better for many tree-like structures.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like structures.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like structures.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like structures.

Example (Binary Search Tree)
Purely Functional Data Structures

Persistent data structures that use of copying (rather than mutation) are called \textit{purely functional} data structures. Operations on them can be expressed as mathematical functions that, given an input structure, return a \textit{new} output structure:

\begin{align*}
\text{insert } v \text{ Leaf} &= \text{Branch } v \text{ Leaf Leaf} \\
\text{insert } v \text{ (Branch } x \text{ l r)} &= \text{if } v \leq x \text{ then} \\
&\quad \text{Branch } x \text{ (insert } v \text{ l) r} \\
&\text{else} \\
&\quad \text{Branch } x \text{ l (insert } v \text{ r)}
\end{align*}
Computing with Functions

We model processes in Haskell using the IO type. We’ll treat IO as an abstract type for now, and give it a formal semantics later if we have time:

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

Note the semantics of evaluation and execution are different things.
Building up IO

Recall monads:

\[
\begin{align*}
return & :: \forall a. \ a \to \ IO \ a \\
(\gg=) & :: \forall a b. \ IO \ a \to (a \to \ IO \ b) \to \ IO \ b \\
getChar & :: \ IO \ Char \\
putChar & :: \ Char \to \ IO \ ()
\end{align*}
\]

Example (Echo)

\[
\begin{align*}
\text{echo} & :: \ IO \ () \\
\text{echo} & = \ getChar \ (\gg=) (\lambda x. \ putChar \ x \ (\gg=) \ \lambda y. \ \text{echo})
\end{align*}
\]

Or, with do notation:

\[
\begin{align*}
\text{echo} & :: \ IO \ () \\
\text{echo} & = \ do \ x \gets \ getChar \ putChar \ x \ \text{echo}
\end{align*}
\]
Adding Concurrency

We can have multiple threads easily enough:

\[
\text{forkIO} :: \text{IO ()} \rightarrow \text{IO ()}
\]

Example (Dueling Printers)

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

But what sort of \textit{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 :: ∀a. a → IO (MVar a)` Create a new MVar
- `takeMVar :: ∀a. MVar a → IO a` Read/remove the value
- `putMVar :: ∀a. MVar a → a → 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:

\[
\text{writeMVar } m \ v = \text{do} \ \text{takeMVar } m ; \ \text{putMVar } m \ v \\
\text{readMVar } m = \text{do} \ v \leftarrow \text{takeMVar } m ; \ \text{putMVar } m \ v ; \ \text{return} \ v
\]

\[
\text{problem :: } DB \rightarrow IO () \\
\text{problem initial = do} \\
\ db \leftarrow \text{newMVar} \ \text{initial} \\
\ \text{wl} \leftarrow \text{newMVar} () \\
\ \text{let} \ \text{reader} = \text{readMVar} \ db \gg= \cdot \cdot \\
\ \text{let} \ \text{writer} = \text{do} \\
\ \ \ \text{takeMVar} \ \text{wl} \\
\ \ \ d \leftarrow \text{readMVar} \ db \\
\ \ \ \text{let} \ \ d' = \text{update} \ d \\
\ \ \ \text{evaluate} \ d' \\
\ \ \ \text{writeMVar} \ db \ d' \\
\ \ \ \text{putMVar} \ \text{wl} ()
\]
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.*
The Problem with Locks

Problem

Write a procedure to transfer money from one bank account to another. To keep things simple, both accounts are held in memory: no interaction with databases is required. The procedure must operate correctly in a concurrent program, in which many threads may call transfer simultaneously. No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).
The Problem with Locks

Assume some infrastructure for accounts:

```haskell
type Balance = Int
```
The Problem with Locks

Assume some infrastructure for accounts:

```haskell
type Balance = Int
type Account = MVar Balance
```

withdraw :: Account → Int → IO ()
withdraw a m = takeMVar a >>= putMVar a . subtract m

deposit :: Account → Int → IO ()
deposit a m = withdraw a (−m)
The Problem with Locks

Assume some infrastructure for accounts:

```haskell
 type Balance = Int
 type Account = MVar Balance

 withdraw :: Account → Int → IO ()
 withdraw a m = takeMVar a >>= (putMVar a ∘ subtract m)
```
The Problem with Locks

Assume some infrastructure for accounts:

```haskell
  type Balance = Int
  type Account = MVar Balance

  withdraw :: Account → Int → IO ()
  withdraw a m = takeMVar a >>> (putMVar a ◦ subtract m)

  deposit :: Account → Int → IO ()
  deposit a m = withdraw a (−m)
```

Attempt #1

\[ \text{transfer } f \ t \ m = \text{do } \text{withdraw } f \ m; \text{deposit } t \ m \]
Attempt #1

\[
\text{transfer } f \ t \ m = \textbf{do} \ \text{withdraw } f \ m; \text{deposit } t \ m
\]

Problem

The intermediate states where a transaction has only been partially completed are externally observable.

In a bank, we might want the invariant that at all points during the transfer, the total amount of money in the system remains constant. We should have no money go missing.
Attempt #2

\[
\text{transfer } f \ t \ m = \text{do}
\]
\[
\begin{align*}
fb & \leftarrow \text{takeMVar } f \\
tb & \leftarrow \text{takeMVar } t \\
\text{putMVar } t (tb + m) \\
\text{putMVar } f (fb - m)
\end{align*}
\]
Attempt #2

\[
\text{transfer } f \ t \ m = \textbf{do}
\]
\[
\quad fb \leftarrow \text{takeMVar } f
\]
\[
\quad tb \leftarrow \text{takeMVar } t
\]
\[
\quad \text{putMVar } t (tb + m)
\]
\[
\quad \text{putMVar } f (fb - m)
\]

Problem

We can have \textit{deadlock} here, when two people transfer to each other simultaneously and both transfers proceed in lock-step.

Also, we can’t just compose our existing \textit{withdrawal} and \textit{deposit} operations. That’s sad.
Solution

We should enforce a *global* ordering of locks.

\[
\textbf{type} \quad \text{Account} = (\text{MVar Balance}, \text{AccountNo})
\]

\[
\text{transfer} \, (f, \, fa) \, (t, \, ta) \, m = \textbf{do}
\]
\[(fb, \, tb) \leftarrow \textbf{if} \, fa \leq ta\]
\[
\text{then do}
\]
\[
\begin{align*}
fb & \leftarrow \text{takeMVar} \, f \\
tb & \leftarrow \text{takeMVar} \, t \\
pure \, (fb, \, tb)
\end{align*}
\]
\[
\text{else do}
\]
\[
\begin{align*}
tb & \leftarrow \text{takeMVar} \, t \\
fb & \leftarrow \text{takeMVar} \, f \\
pure \, (fb, \, tb)
\end{align*}
\]

\[
\text{putMVar} \, t \, (tb + m) \\
\text{putMVar} \, f \, (fb - m)
\]
It Gets Complicated

Problem

Now suppose that some accounts can be configured with a “backup” account, that is used if insufficient funds are available in the default account.
It Gets Complicated

Problem

Now suppose that some accounts can be configured with a “backup” account, that is used if insufficient funds are available in the default account.

Should you take the lock for the backup account?
Problem

Now suppose that some accounts can be configured with a “backup” account, that is used if insufficient funds are available in the default account.

Should you take the lock for the backup account?

To make life even harder: What if we want to block if insufficient funds are available?
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
Conclusion

Lock-based methods have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
- Remember not to take the locks in the wrong order.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
- Remember not to take the locks in the wrong order.
- Remember to deal with locks when an error occurs.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
- Remember not to take the locks in the wrong order.
- Remember to deal with locks when an error occurs.
- Remember to signal condition variables and release locks at the right time.
Conclusion

*Lock-based methods* have their place, but from a software engineering perspective they’re a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
- Remember not to take the locks in the wrong order.
- Remember to deal with locks when an error occurs.
- Remember to signal condition variables and release locks at the right time.

Most importantly, *modular programming* becomes impossible.
The Solution

Represent an account as a simple shared variable containing the balance.

\[ \text{transfer } f \ t \ m = \text{atomically } \$ \ \text{do} \]
\[ \text{withdraw } f \ m \]
\[ \text{deposit } t \ m \]

Where \textit{atomically } P \ guarantees:

- **Atomicity** The effects of the action \( P \) become visible all at once.
- **Isolation** The effects of action \( P \) is not affected by any other threads.

**Problem**

How can we implement \textit{atomically}?
The Global Lock

We can adopt the solution of certain reptilian programming languages.

**Problem**

Atomicity is guaranteed, but what about *isolation*? Also, performance is predictably garbage.
Ensuring Isolation

Rather than use regular shared variables, use special *transactional variables*. 

\[
\begin{align*}
\text{newTVar} & :\quad a \rightarrow \text{STM} (\text{TVar } a) \\
\text{readTVar} & :\quad \text{TVar } a \rightarrow \text{STM } a \\
\text{writeTVar} & :\quad \text{TVar } a \rightarrow a \rightarrow \text{STM } () \\
\text{atomically} & :\quad \text{STM } a \rightarrow \text{IO } a
\end{align*}
\]

The type constructor STM is also an instance of the *Monad* type class, and thus supports the same basic operations as IO.

\[
\begin{align*}
\text{pure} & :\quad a \rightarrow \text{STM} (\text{TVar } a) \\
(\gg=) & :\quad \text{STM } a \rightarrow (a \rightarrow \text{STM } b) \rightarrow \text{STM } b
\end{align*}
\]
Implementing Accounts

type Account = TVar Int

withdraw :: Account → Int → STM ()
withdraw a m = do
  balance ← readTVar m
  writeTVar a (balance − m)

deposit a m = withdraw a (−m)

Observe: withdraw (resp. deposit) can only be called inside an atomically
Implementing Accounts

type Account = TVar Int

withdraw :: Account → Int → STM ()
withdraw a m = do
  balance ← readTVar m
  writeTVar a (balance − m)

deposit a m = withdraw a (−m)

Observe: withdraw (resp. deposit) can only be called inside an atomically ⇒ We have isolation.
Implementing Accounts

type Account = TVar Int

withdraw :: Account → Int → STM ()
withdraw a m = do
    balance ← readTVar m
    writeTVar a (balance - m)

deposit a m = withdraw a (-m)

Observe: withdraw (resp. deposit) can only be called inside an atomically ⇒ We have isolation.

But, we’d still like to run more than one transaction at once — one global lock isn’t good enough.
Optimistic Execution

Each transaction (atomically block) is executed *optimistically*. This means they do not need to check that they are allowed to execute the transaction first (unlike, say, locks, which prefer a *pessimistic* model).
Optimistic Execution

Each transaction (atomically block) is executed *optimistically*. This means they do not need to check that they are allowed to execute the transaction first (unlike, say, locks, which prefer a *pessimistic* model).

**Implementation Strategy**

Each transaction has an associated *log*, which contains:

- The values written to any *TVars* with *writeTVar*.
- The values read from any *TVars* with *readTVar*, consulting earlier log entries first.

First the log is *validated*, and, if validation succeeds, changes are *committed*. Validation and commit are *one atomic step*.

What can we do if validation fails?
Optimistic Execution

Each transaction (atomically block) is executed \textit{optimistically}. This means they do not need to check that they are allowed to execute the transaction first (unlike, say, locks, which prefer a \textit{pessimistic} model).

\begin{itemize}
  \item The values written to any \texttt{TVar}s with \texttt{writeTVar}.
  \item The values read from any \texttt{TVar}s with \texttt{readTVar}, consulting earlier log entries first.
\end{itemize}

First the log is \textit{validated}, and, if validation succeeds, changes are \textit{committed}. Validation and commit are \textit{one atomic step}.

What can we do if validation fails? \textit{We re-run the transaction!}
Re-running transactions

```haskell
atomically $ do
    x ← readTVar xv
    y ← readTVar yv
    if x > y then detonateTNT else pure ()
```

To avoid harmful side-effects, the transaction must be repeatable. We can’t change the world until commit time.

A real implementation is smart enough not to retry with exactly the same schedule.
Blocking and *retry*

**Problem**

We want to *block* if insufficient funds are available.

We can use the helpful action *retry :: STM a.*
Blocking and *retry*

**Problem**
We want to *block* if insufficient funds are available.

We can use the helpful action *retry :: STM a*.

```haskell
withdraw' :: Account → Int → STM ()
withdraw' a m = do
  balance ← readTVar a
  if m > 0 && m > balance then
    retry
  else
    writeTVar a (balance − m)
```
Choice and \textit{orElse}

**Problem**

We want to transfer from a backup account if the first account has insufficient funds, and \textit{block} if neither account has insufficient funds.

We can use the helpful action

\[
\text{orElse} :: \text{STM } a \rightarrow \text{STM } a \rightarrow \text{STM } a
\]
Choice and *orElse*

Problem

We want to transfer from a backup account if the first account has insufficient funds, and *block* if neither account has insufficient funds.

We can use the helpful action

\[ \text{orElse} :: \text{STM} \ a \rightarrow \text{STM} \ a \rightarrow \text{STM} \ a \]

\[ \text{wdBackup} :: \text{Account} \rightarrow \text{Account} \rightarrow \text{Int} \rightarrow \text{STM} \ () \]

\[ \text{wdBackup} \ a_1 \ a_2 \ m = \text{orElse} \ (\text{withdraw}' \ a_1 \ m) \ (\text{withdraw}' \ a_2 \ m) \]
Evaluating STM

STM is *modular*. We can compose transactions out of smaller transactions. We can hide concurrency behind library boundaries without worrying about deadlock or global invariants.

Lock-free data structures and transactional memory based solutions work well if contention is low and under those circumstances scale better to higher process numbers than lock-based ones.

Most importantly, the resulting code is often simpler and more robust. Profit!
Progress

One transaction can force another to abort only when it commits.

At any time, at least one currently running transaction can successfully commit.

Traditional deadlock scenarios are impossible, as is cyclic restarting where two transactions constantly cancel each other.

Starvation is possible (when?), however uncommon in practice. So, we technically don’t have eventual entry.
Database Guarantees

**Atomicity** ✓ Each transaction should be ‘all or nothing’.

**Consistency** ✓ Each transaction in the future sees the effects of transactions in the past.

**Isolation** ✓ The transaction’s effect on the state cannot be affected by other transactions.

**Durability** The transaction’s effect on the state survives power outages and crashes.

STM gives you 75% of a database system. The Haskell package *acid-state* builds on STM to give you all four.
That’s it

We have now covered all the content in COMP3161/COMP9164. Thanks for sticking with the course.

- **Syntax Foundations**
  Concrete/Abstract Syntax, Ambiguity, HOAS, Binding, Variables, Substitution, $\lambda$-calculus

- **Semantics Foundations**
  Static Semantics, Dynamic Semantics (Small-Step/Big-Step), Abstract Machines, Environments, Stacks, Safety, Liveness, Type Safety (Progress and Preservation)

- **Features**
  - Algebraic Data Types, Recursive Types
  - Exceptions
  - Polymorphism, Type Inference, Unification
  - Overloading, Subtyping, Abstract Data Types
  - Concurrency, Critical Sections, STM
MyExperience

Please fill out the survey. It helps tremendously.

https://myexperience.unsw.edu.au
Further Learning

- **UNSW courses:**
  - COMP3141 — Software System Design and Implementation
  - COMP6721 — (In-)formal Methods
  - COMP3131 — Compilers
  - COMP4141 — Theory of Computation
  - COMP3151 — Foundations of Concurrency
  - COMP4161 — Advanced Topics in Verification
  - COMP3153 — Algorithmic Verification

- **Online Learning**
  - Oregon Programming Languages Summer School Lectures ([https://www.cs.uoregon.edu/research/summerschool/archives.html](https://www.cs.uoregon.edu/research/summerschool/archives.html)) Videos are available from here! Also some on YouTube.
**What’s next?**

The exam is on **Monday, 5th of December 2022** at 9am.

- I have posted a sample exam with revision questions.
- The final exam will run similar to the sample exam.
- It runs for 2 hours and 10 minutes.
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:

- **β-equivalence**: \((\lambda x. M[x]) \equiv_\beta M[N]\)
- **α-equivalence**: \(\lambda x. M[x] \equiv_\alpha \lambda y. M[y]\)
- **η-equivalence**: \(\lambda x. M x \equiv_\eta M\)

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

\[
\begin{align*}
\text{return } N \gg M &\equiv M N \\
(X \gg Y) \gg Z &\equiv X \gg (\lambda x. Y x \gg Z) \\
X &\equiv X \gg \text{return}
\end{align*}
\]
Processes

This means that a Haskell expression of type \( \text{IO} \ \tau \) for will boil down to either \( \text{return} \ x \) where \( x \) is a value of type \( \tau \); or \( a \gg M \) where \( a \) is some primitive IO action (\( \text{forkIO} \ p \), \( \text{readMVar} \ v \), etc.) and \( M \) is some function producing another 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 \( \text{IO} () \).

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

\[
(\leftrightarrow) \subseteq P \times P
\]
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:

\[
P, Q ::= a \gg M \\
\mid \text{return} () \\
\mid P \parallel Q \\
\mid \ldots
\]
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:

\[
P, Q ::= a \gg M \\
| return () \\
| P \parallel Q \\
| ... 
\]

And the following ambient congruence equations:

\[
P \parallel Q \equiv Q \parallel P \\
P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R
\]
Semantics for forkIO

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

\[
P \mapsto P'  \\
\frac{P \mapsto P'}{P \parallel Q \mapsto P' \parallel Q}
\]
Semantics for forkIO

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

\[ P \leftrightarrow P' \]

\[ P \parallel Q \leftrightarrow P' \parallel Q \]

The \textit{forkIO} operation introduces a new process:

\[ (\text{forkIO } P \gg M) \leftrightarrow P \parallel (\text{return } () \gg M) \]
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.

\[
P, Q ::= a \gg M
\]
\[
\quad \mid \quad \text{return } ()
\]
\[
\quad \mid \quad \text{return } ()
\]
\[
\quad \mid \quad P \parallel Q
\]
\[
\quad \mid \quad \langle \rangle_n \mid \langle v \rangle_n
\]
\[
\quad \mid \quad \ldots
\]
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.

\[
P, Q ::= \begin{align*}
a & \implies M \\
return () \\
P & || Q \\
\langle \rangle_n & || \langle v \rangle_n \\
\ldots
\end{align*}
\]

\[
\langle \rangle_n || (putMVar n v \implies M) \rightarrow \langle v \rangle_n || (return () \implies M)
\]
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.

\[
\begin{align*}
P, Q & ::= a \gg M \\
      & | \quad \text{return} () \\
      & | \quad P \parallel Q \\
      & | \quad \langle \rangle_n \quad | \quad \langle v \rangle_n \\
      & | \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\langle \rangle_n \parallel (\text{putMVar} n \, v \gg M) & \mapsto \langle v \rangle_n \parallel (\text{return} () \gg M) \\
\langle v \rangle_n \parallel (\text{takeMVar} n \gg M) & \mapsto \langle \rangle_n \parallel (\text{return} v \gg M)
\end{align*}
\]
Semantics for newMVar

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

\[
(newMVar \; v \gg M) \mapsto \langle v \rangle_n \parallel (return \; n \gg M) \quad (n \text{ fresh})
\]

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* \( \nu \) to our language of processes:

\[
P, Q ::= a \gg M \\
| \text{return (}) \\
| P \parallel Q \\
| \langle \rangle_n \\
| \langle v \rangle_n \\
| (\nu \ n) \ P
\]
Restriction Operator

We introduce a *restriction operator* \( \nu \) to our language of processes:

\[
P, Q ::= a \gg M \\
| \quad \text{return ()} \\
| \quad P \parallel Q \\
| \quad \langle \rangle_n \\
| \quad \langle \nu \rangle_n \\
| \quad (\nu n) P
\]

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:

\[
(\nu n) (\nu m) P \equiv (\nu m) (\nu n) P \\
(\nu n)(P \parallel Q) \equiv P \parallel (\nu n) Q \quad \text{(if } n \notin P\text{)}
\]
Better Semantics for newMVar

The rule for \textit{newMVar} is much the same as before, but now we explicitly restrict the MVar to $M$.

$$(\textit{newMVar } v \gg M) \quad \leftrightarrow \quad (\nu n)(\langle v \rangle_n \parallel (\text{return } n \gg M))^{(n \text{ fresh})}$$
Better Semantics for newMVar

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

$$\frac{(newMVar \ v \gg M) \rightarrow (\nu \ n)((\nu) n \parallel (return \ n \gg M))}{(n \ fresh)}$$

We can always execute under a restriction:

$$P \leftrightarrow P'$$

$$\frac{(\nu \ n) \ P \leftrightarrow (\nu \ n) \ P'}{(\nu \ n) \ P \leftrightarrow (\nu \ n) \ P'}$$
Better Semantics for newMVar

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

$$(\text{newMVar } v \gg M) \mapsto (\nu n)(\langle v \rangle_n \parallel (\text{return } n \gg M))^{(n \text{ fresh})}$$

We can always execute under a restriction:

$$P \mapsto P'$$

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

**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:

\[
(\nu \ n) \langle \rangle_n \mapsto return () \\
(\nu \ n) \langle v \rangle_n \mapsto return ()
\]
Garbage Collection

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

\[
(\nu \ n) \langle \rangle \n \leftrightarrow \ return \ ()
\]

\[
(\nu \ n) \langle v \rangle \n \leftrightarrow \ return \ ()
\]

Extra processes that have outlived their usefulness disappear:

\[
return \ () \parallel P \quad \leftrightarrow \quad P
\]
Process Algebra

Our language $P$ is called a *process algebra*, a common means of describing semantics for concurrent programs.
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

Simon Marlow (Editor)
Haskell 2010 Language Report
https://www.haskell.org/onlinereport/haskell2010/
Bibliography

Simon Peyton Jones
Beautiful Concurrency

Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy
Composable Memory Transactions
PPoP ’05
Association for Computer Machinery

David Himmelstrup
Acid-State Library
https://github.com/acid-state/acid-state

Ryan Yates and Michael L. Scott
A Hybrid TM for Haskell
TRANSACT ’14
Association for Computer Machinery