Haskell Concurrency a

Christine Rizkallah
CSE, UNSW
Term 3 2020

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
structure and som
of the data structu

Desiderata:

- We want *atomicity*, in that each update happens in on
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, et

Observation

Updates are *atomic* and reads are *consistent* — b tly, which leads to unnecessary *contention*.

A Better Solution

A more elaborate locking mechanism (*condition variables*) could be used to to allow multiple readers t
atomically.

Observation

We have atomicity and consistency, and now multiple reads c
Still, we don't allow updates to execute concurrently with read
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 rath

Trick: Rather tha *their own local copy* of the data structure, and then merely updates the (shar to the data structure to point to their copy.

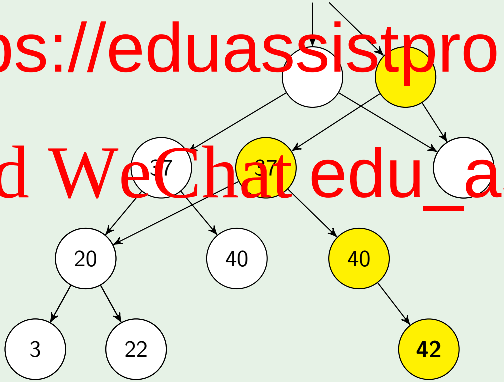
Atomicity The only shared write is now just to one pointer.

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)



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 expr

structure, return

$$\textit{insert}\ v\ (\mathrm{Branch}\ x\ l\ r) \quad = \quad \textbf{if}$$

els

$$\mathrm{Branch}\ x\ l\ (\textit{insert}\ v\ r)$$

Computing with Functions

We model real processes in Haskell using the IO type. We'll treat IO as an abstract type for now, and gi

$$\mathrm{IO}\ \tau \ = \ \text{result of type } \tau$$

Note the semantics of *evaluation* and *execution*

Building up IO

Recall monads:

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

Example (Ech

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

Or, with **do** notation:

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

Adding Concurrency

We can have multiple threads easily enough

Example (Duel

$$\textbf{let } loop\ c = \textbf{do } putC$$
$$\textbf{in do } forkIO\ (loop\ \text{'a'}$$

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 ta

an empty box to upd

MVar Functio

$newMVar :: \forall a.\ a \rightarrow \mathrm{IO}\ (\mathrm{MVar}\ a)$	Create a new MVar
$takeMVar :: \forall a.\ \mathrm{MVar}\ a \rightarrow \mathrm{IO}\ a$	Read/r
$putMVar :: \forall a.\ \mathrm{MVar}\ a \rightarrow a \rightarrow \mathrm{IO}\ ()$	Update/inse

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:

$$\begin{array}{l} \mathit{writeMVar}\ m\ v = \mathbf{do}\ \mathit{takeMVar}\ m;\ \mathit{putMVar}\ m\ v \\ \mathit{readMVar}\ m = \mathbf{do}\ v \leftarrow \mathit{takeMVar}\ m;\ \mathit{putMVar}\ m\ v;\ \mathit{return}\ v \end{array}$$

$$\begin{array}{l} \mathbf{let}\ \mathit{reader} = \mathit{readMVar}\ \mathit{db} \gg\!= \cdots \\ \mathbf{let}\ \mathit{writer} = \mathbf{do} \\ \quad \mathit{takeMVar}\ \mathit{wl} \\ \quad d \leftarrow \mathit{readMVar}\ \mathit{db} \\ \quad \mathbf{let}\ d' = \mathit{update}\ d \\ \quad \mathit{evaluate}\ d' \\ \quad \mathit{writeMVar}\ \mathit{db}\ d' \\ \quad \mathit{putMVar}\ \mathit{wl}\ () \end{array}$$

Fairness

Each MVar has an a
fairness propert

> *No thread ca*
> *that MVar indefinitely.*

The Problem with Locks

Problem

Write a procedur

simple, both acco
The procedure must operate correctly in a concurrent program, in which many threads may call transfer simultaneously. No thread should be able to o
the money has left one account, but not arrived in the other (or vice v

The Problem with Locks

Assume some infrastructure for accounts:

t

t

w

$withdraw\ a\ m = takeMVar\ a \gg= (\qquad)$

$deposit :: \mathrm{Account} \to \mathrm{Int} \to \mathrm{IO}\ ()$

$deposit\ a\ m = withdraw\ a\ (-m)$

Attempt #1

$$\mathit{transfer}\ f\ t\ m = \mathbf{do}\ \mathit{withdraw}\ f\ m;\ \mathit{deposit}\ t\ m$$

Problem

The intermediat
externally observable.

In a bank, we might want the invariant that at all points during the tr
amount of money in the system remains constant. We should ha
missing[a].

---

[a] We're not CBA

Attempt #2

$$
\begin{array}{l}
\mathit{transfer}\ f\ t\ m = \mathbf{do} \\
\quad \mathit{fb} \leftarrow \mathit{takeMVar}\ f
\end{array}
$$

Problem

We can have *deadlock* here, when two people transfer t
and both transfers proceed in lock-step.

Also, not being able to compose our existing *withdrawal* and *deposit* operations is unfortuitous from a software design perspective.

Solution

We should enforce a *global* ordering of locks.


```
type Account = (MVar Balance, AccountNo)

transfer (f, fa) (t, ta) m = do
```



```
        tb ← takeMVar t
        pure (fb, tb)
```



```
      else do
        tb ← takeMVar t
        fb ← takeMVar f
        pure (fb, tb)
    putMVar t (tb + m)
    putMVar f (fb − m)
```

It Gets Complicated

Problem

Now suppose that
withdrawn from if i

Should you take the lock for the backup account?

To make life even harder: What if we want to
available?

Conclusion

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

- Remembe
- Remembe
- Remembe
- 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 loc

Most importantly, *modular programming* becomes impossible.

The Solution

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

$transfer\ f\ t\ m = atomically\ \$\ \mathbf{do}$

Where *atomic*

Atomicity The effects of the action P becom

Isolation The effects of action P is not affected

Problem

How can we implement *atomically*?

The Global Lock

We can adopt the so

Problem

Atomicity is guar

Also, performance is predictably garbage.

Ensuring Isolation

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

$$\mathit{createTVar} \quad :: \quad a \rightarrow \mathrm{STM}\ (\mathrm{TVar}\ a)$$

The type constructor STM is also an instance of the hus supports the same basic operations as IO.

$$\begin{aligned} \mathit{pure} \quad &:: \quad a \rightarrow \mathrm{STM}\ (\mathrm{TVar}\ a) \\ (\gg\!=) \quad &:: \quad \mathrm{STM}\ a \rightarrow (a \rightarrow \mathrm{STM}\ b) \rightarrow \mathrm{STM}\ b \end{aligned}$$

Implementing Accounts

$\textbf{type}\ \mathrm{Account} = \mathrm{TVar\ Int}$

deposit a m = *withdraw*

Observe: *withdraw* (resp. *deposit*) can only be called i ly ⇒ 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).

Implementati

Each transactio

- The values written to any TVars with *write*
- The values read from any TVars with *read* ies first.

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

What can we do if validation fails? We re-run the transaction!

Re-running transactions

To avoid serious international side-effects, the transaction *able*. We can't change the world until *commit* time.

A real implementation is smart enough not to retry with exactly t

Blocking and *retry*

Problem

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

We can use the helpf

$$
\begin{array}{l}
\mathit{withdraw}\ \ a\ m = \mathbf{do} \\
\quad \mathit{balance} \leftarrow \mathit{readTVa} \\
\quad \mathbf{if}\ m \geq 0\ \&\&\ m > b \\
\qquad \mathit{retry} \\
\quad \mathbf{else} \\
\qquad \mathit{writeTVar}\ \ a\ (\mathit{balance} - m)
\end{array}
$$

Choice and *orElse*

Problem

We want to transfer from a backup account if the first account has insufficient funds, and *block* if neit

We can use the help f

$$orElse :: \mathrm{STM}\ a \rightarrow \mathrm{ST}$$

$$wdBackup :: \mathrm{Account} \rightarrow \mathrm{Account} \rightarrow$$
$$wdBackup\ a_1\ a_2\ m = orElse\ (withdraw'\ a_1\ m)\ (withdraw'\ a_2\ m)$$

Evaluating STM

STM is *modular*. We can compose transactions out of smaller transactions. We can hide concurrenc al invariants.

Lock-free data str contention is low and under those circumstances scale better t numbers than lock-based ones.

Most importantly, the resulting code is often simpler and mor **rofit!**

Progress

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

At any time, *t.*

Traditional dea

transactions constantly cancel each other.

Starvation is possible (*when?*), however uncommon in pra

don't have eventual entry.

Performance

A concurrent cha MVar version.
The STM version p If of the heap
space ⟶ **Pro**

The implementation is a bit simpler as well. Let's do it if we have time! Jus

[1] Mostly, the MVar implementation performed poorly due to lots of overhead to make it exception-safe.

Database Guarantees

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

Consistency

Isolation p

transactions.

Durability The transaction’s effect on the state survive

STM gives you 75% of a database system. The Haskell package *te* builds on STM to give you all four.

Hardware Transactional Memory

The latest round of Intel processors support *Hardware Transactional Memory* instructions:

XBEGIN Begin a hardware transaction

XEND

XTEST

XABORT

The "log" we described earlier is stored in *L1 cac* ed to the amount of cache we have. If a speculative read overflows the ca sometimes generate a *spurious conflicts* and cause t

For this reason, progress can only be ensured through the *combination* of STM and HTM. Work is currently underway to implement this for Haskell, and prototypes show promising performance improvements.

That's it

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

- **Syntax Foundations**
  Concrete/ tution,
  λ-calculus
- **Semantic**

  Static Semantics, Dynamic Semantics (Small-Step/Big-Step), Abstract Machines, Environments, Stacks, Safety, Liveness, Type Safet
- **Features**

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

MyExperience

`https://myexperience.unsw.edu.au`

Further Learning

- UNSW courses:
  - COMP3141 — Software System Design and Implementation
  - COMP6721 — (In-)formal Methods
  - COMP3131 — Compilers
  - COMP
  - COMP
  - COMP
  - COMP4161 — Advanced Topics in Verification
  - COMP3153 — Algorithmic Verification
- Online Learning
  - Oregon Programming Languages Summer School (https://www.cs.uoregon.edu/research/summerschool/archives.html) Videos are available from here! Also some on YouTube.
  - Bartosz Milewski's Lectures on Category Theory are on YouTube.
- Books — see Liam's Book List!

What's next?

The exam is on

- I have posted
- The final exa
- 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:

Let our ambient congruence relation $\equiv$ be $\equiv_{\alpha\beta\eta}$ tra
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 $\text{IO}\ \tau$ 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 $\text{IO}\ \tau$. This is the *head normal f*

Definition

Define a language ons of type $\text{IO}\ ()$.

We want to define the semantics of the *executio* *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:

$$| \quad \cdots$$

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:

The *forkIO* operation introduces a new process:

$$(\mathit{forkIO}\ P \gg= M) \rightarrow P \parallel$$

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

$$\mid \quad \langle\rangle_n \qquad \langle$$

$$\mid \quad \ldots$$

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

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

Semantics for newMVar

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

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad}{(\qquad\qquad\qquad\qquad)}\quad \text{resh)}$$

But this approach

- The name n is now globally-scoped, without an exp
- It doesn't accurately model the *lifetime* of th garbage-collected once all processes that can access it fi
- 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:

$$P, Q \;::=\; a \gg\!= M$$

Writing $(\nu\ n)\ P$ says that the MVar name n is *o* . Mentioning n outside P is not well-formed. We need the following addi .

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

Better Semantics for newMVar

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

$$\overline{(new \quad \ldots} \;(n \text{ fresh})$$

We can always exe

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

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:

Extra processes that have outlived their usefulness disappear:

$$\mathit{return}\ () \parallel P$$

Process Algebra

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

Process algebras

with Rob van Glab

If there's time!

We can talk about more concurrency topics.

Bibliography

Simon Marlow
Parallel and Concurrent Programming in Haskell
O'Reilly, 2013
http://chi

Simon Peyton
Concurrent H
POPL '96
Association for Computer Machinery
http://microsoft.com/en-us/research/wp-content/uploads/1996/01/c

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

Bibliography

Simon Peyton Jones
Beautiful Concurrency
In 'Beautiful Code', ed. Greg Wilson, O'Reilly, 2007
http://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf

Tim Harris, Si
Composable
PPoP '05
Association for Computer Machinery
www.microsoft.com/en-us/research/wp-content/uploads/2005/01/2005-pp

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