COMP3161/9164
Concepts of Programming Languages

Haskell Concurrency and STM

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 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 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.

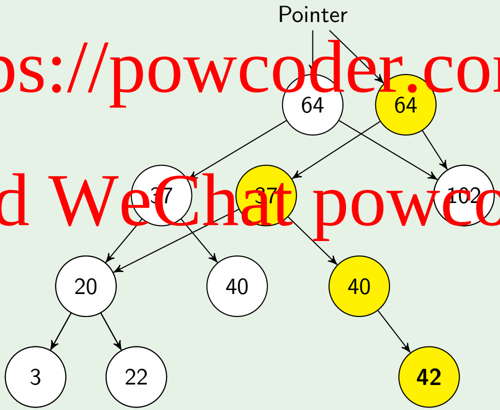
Atomicity The only shared write is now just to one pointer.

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 expressed in the form of mathematical functions that, given an input structure, return a *new* output structure:

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

Computing with Functions

We model real 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:

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

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

Building up IO

Recall monads:

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

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:

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

Adding Concurrency

We can have multiple threads easily enough:

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

Example (Dueling Printers)

$$\begin{array}{l}\textbf{let}\ \mathit{loop}\ c = \textbf{do}\ \mathit{putChar}\ c; \mathit{loop}\ c \\ \textbf{in do}\ \mathit{forkIO}\ (\mathit{loop}\ \text{'a'}); \mathit{loop}\ \text{'z'}\end{array}$$

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

$\mathit{newMVar} :: \forall a.\ a \rightarrow \mathrm{IO}\ (\mathrm{MVar}\ a)$	Create a new MVar
$\mathit{takeMVar} :: \forall a.\ \mathrm{MVar}\ a \rightarrow \mathrm{IO}\ a$	Read/remove the value
$\mathit{putMVar} :: \forall a.\ \mathrm{MVar}\ a \rightarrow a \rightarrow \mathrm{IO}\ ()$	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 = do takeMVar m; putMVar m v
readMVar m = do v ← takeMVar m; putMVar m v; return v

problem :: DB → IO ()
problem initial = do
  db ← newMVar initial
  wl ← newMVar ()
  let reader = readMVar db >>= ···
  let writer = do
    takeMVar wl
    d ← readMVar db
    let d′ = update d
    evaluate d′
    writeMVar db d′
    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.*

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:

$$\textbf{type}\ \mathrm{Balance} = \mathrm{Int}$$
$$\textbf{type}\ \mathrm{Account} = \mathrm{MVar\ Balance}$$

$$\mathit{withdraw} :: \mathrm{Account} \rightarrow \mathrm{Int} \rightarrow \mathrm{IO}\ ()$$
$$\mathit{withdraw}\ a\ m = \mathit{takeMVar}\ a \gg\!= (\mathit{putMVar}\ a\ \circ \mathit{subtract}\ m)$$

$$\mathit{deposit} :: \mathrm{Account} \rightarrow \mathrm{Int} \rightarrow \mathrm{IO}\ ()$$
$$\mathit{deposit}\ a\ m = \mathit{withdraw}\ a\ (-m)$$

Attempt #1

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

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[a].

[a] We're not CBA

Attempt #2


```haskell
transfer f t m = do
  fb ← takeMVar f
  tb ← takeMVar t
  putMVar t (tb + m)
  putMVar f (fb − m)
```


Problem

We can have *deadlock* here, when two people transfer to each other simultaneously 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.

$$\textbf{type}\ \mathrm{Account} = (\mathrm{MVar\ Balance}, \mathrm{AccountNo})$$

$$
\begin{array}{l}
\mathit{transfer}\ (f, \mathit{fa})\ (t, \mathit{ta})\ m = \textbf{do} \\
\quad (\mathit{fb}, \mathit{tb}) \leftarrow \textbf{if}\ \mathit{fa} \leq \mathit{ta} \\
\qquad \textbf{then do} \\
\qquad\quad \mathit{fb} \leftarrow \mathit{takeMVar}\ f \\
\qquad\quad \mathit{tb} \leftarrow \mathit{takeMVar}\ t \\
\qquad\quad \mathit{pure}\ (\mathit{fb}, \mathit{tb}) \\
\qquad \textbf{else do} \\
\qquad\quad \mathit{tb} \leftarrow \mathit{takeMVar}\ t \\
\qquad\quad \mathit{fb} \leftarrow \mathit{takeMVar}\ f \\
\qquad\quad \mathit{pure}\ (\mathit{fb}, \mathit{tb}) \\
\quad \mathit{putMVar}\ t\ (\mathit{tb} + m) \\
\quad \mathit{putMVar}\ f\ (\mathit{fb} - m)
\end{array}
$$

It Gets Complicated

Problem

Now suppose that some accounts can be configured with a "backup" account that is withdrawn from if insufficient funds are available in the normal 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.

- 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.

$$
\begin{aligned}
&\mathit{transfer}\ f\ t\ m = \mathit{atomically}\ \$\ \textbf{do}\\
&\quad \mathit{withdraw}\ f\ m\\
&\quad \mathit{deposit}\ t\ m
\end{aligned}
$$

Where *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 *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*.

$$
\begin{array}{lcl}
\mathit{createTVar} & :: & a \rightarrow \mathrm{STM}\ (\mathrm{TVar}\ a) \\
\mathit{readTVar} & :: & \mathrm{TVar}\ a \rightarrow \mathrm{STM}\ a \\
\mathit{writeTVar} & :: & \mathrm{TVar}\ a \rightarrow a \rightarrow \mathrm{STM}\ () \\
 & & \\
\mathit{atomically} & :: & \mathrm{STM}\ a \rightarrow \mathrm{IO}\ a
\end{array}
$$

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

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

Implementing Accounts

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

$$\begin{array}{l} \textit{withdraw} :: \mathrm{Account} \to \mathrm{Int} \to \mathrm{STM}\ () \\ \textit{withdraw}\ a\ m = \textbf{do} \\ \quad \textit{balance} \leftarrow \textit{readTVar}\ m \\ \quad \textit{writeTVar}\ a\ (\textit{balance} - m) \end{array}$$

$$\textit{deposit}\ a\ m = \textit{withdraw}\ a\ (-m)$$

Observe: *withdraw* (resp. *deposit*) can only be called inside an *atomically* $\Rightarrow$ 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).

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? We re-run the transaction!

Re-running transactions


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


To avoid serious international 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* $:: \mathrm{STM}\ a$.

$$
\begin{array}{l}
\mathit{withdraw}' :: \mathrm{Account} \rightarrow \mathrm{Int} \rightarrow \mathrm{STM}\ ()\\
\mathit{withdraw}'\ a\ m = \mathbf{do}\\
\quad \mathit{balance} \leftarrow \mathit{readTVar}\ a\\
\quad \mathbf{if}\ m > 0\ \&\&\ m > \mathit{balance}\ \mathbf{then}\\
\quad\quad \mathit{retry}\\
\quad \mathbf{else}\\
\quad\quad \mathit{writeTVar}\ a\ (\mathit{balance} - m)
\end{array}
$$

Choice and *orElse*

Problem

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

We can use the helpful action

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

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

Evaluating STM

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

Performance

A concurrent channel using STM was implemented and compared to an MVar version. The STM version performs within 10% of the MVar version, and uses half of the heap space $\longrightarrow$ **Profit**![1]

The implementation is a bit simpler as well. Let's do it if we have time! Just a linked list, really.

[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 ✓ 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.

Hardware Transactional Memory

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

- `XBEGIN` Begin a hardware transaction
- `XEND` End a hardware transaction
- `XTEST` Test if currently executing a transaction.
- `XABORT` Abort the transaction and jump to the abort handler.

The "log" we described earlier is stored in *L1 cache*. Speculative writes are limited to the amount of cache we have. If a speculative read overflows the cache, it may sometimes generate a *spurious conflicts* and cause the transaction to abort.

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/Abstract Syntax, Ambiguity, HOAS, Binding, Variables, Substitution, λ-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

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
  - COMP6752 — Modelling Concurrent Systems
  - 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) 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 **Tuesday, 8th of December 2020** at 2pm.

- 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:

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

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

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

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:

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

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:

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

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.

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

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

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

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:

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

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:

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

Better Semantics for newMVar

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

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

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:

$$
\begin{array}{lcl}
(\nu\ n)\ \langle\rangle_n & \mapsto & \mathit{return}\ () \\
(\nu\ n)\ \langle v \rangle_n & \mapsto & \mathit{return}\ ()
\end{array}
$$

Extra processes that have outlived their usefulness disappear:

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

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.

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/

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, Simon Marlow, Simon Peyton Jones, Maurice Herlihy
Composable Memory Transactions
PPoP '05
Association for Computer Machinery
www.microsoft.com/en-us/research/wp-content/uploads/2005/01/2005-ppopp-composable.pdf

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