Atomicity and Software Transactional Memory

(thanks to SPJ for many slides)

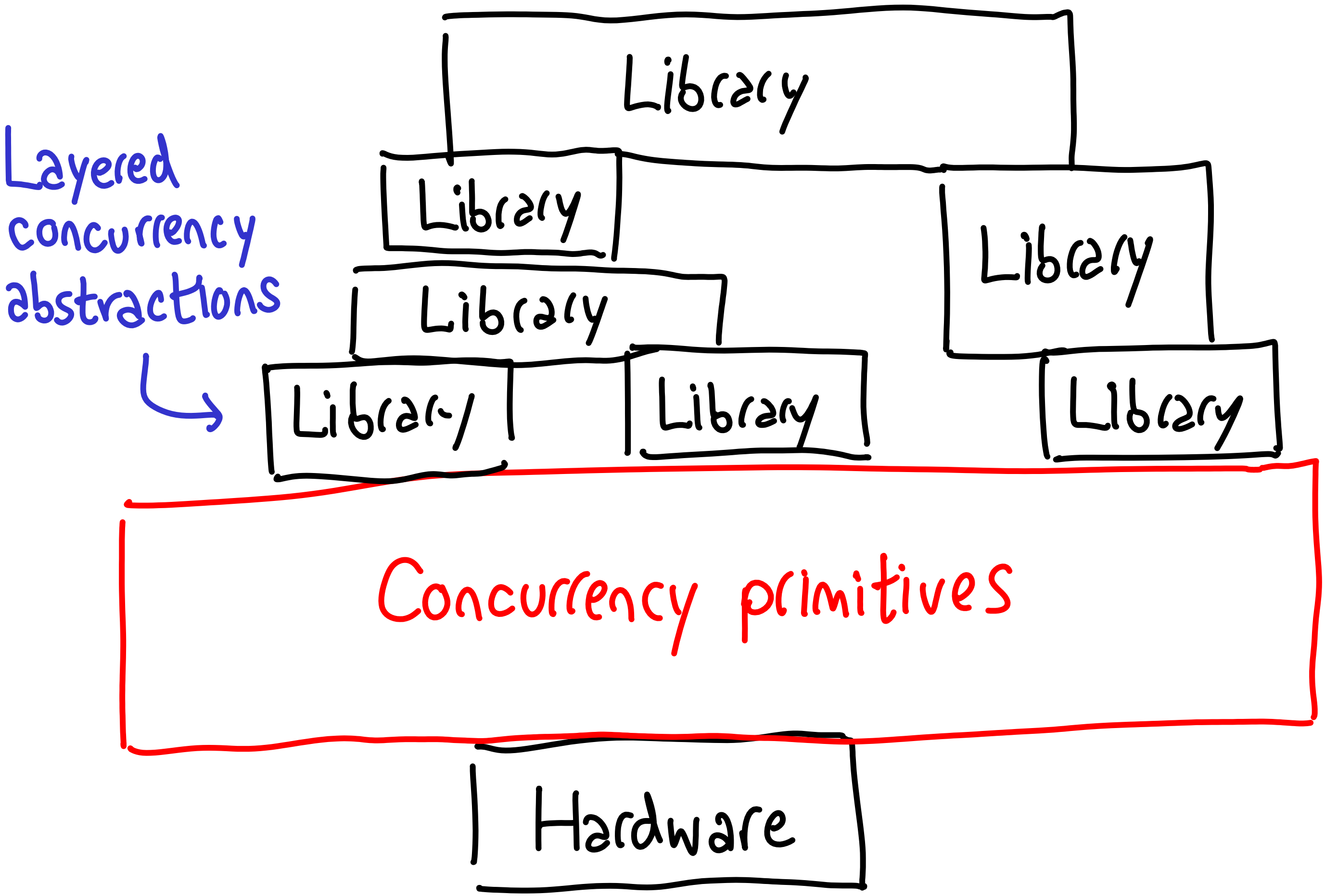
The Problem

locks / condition variables

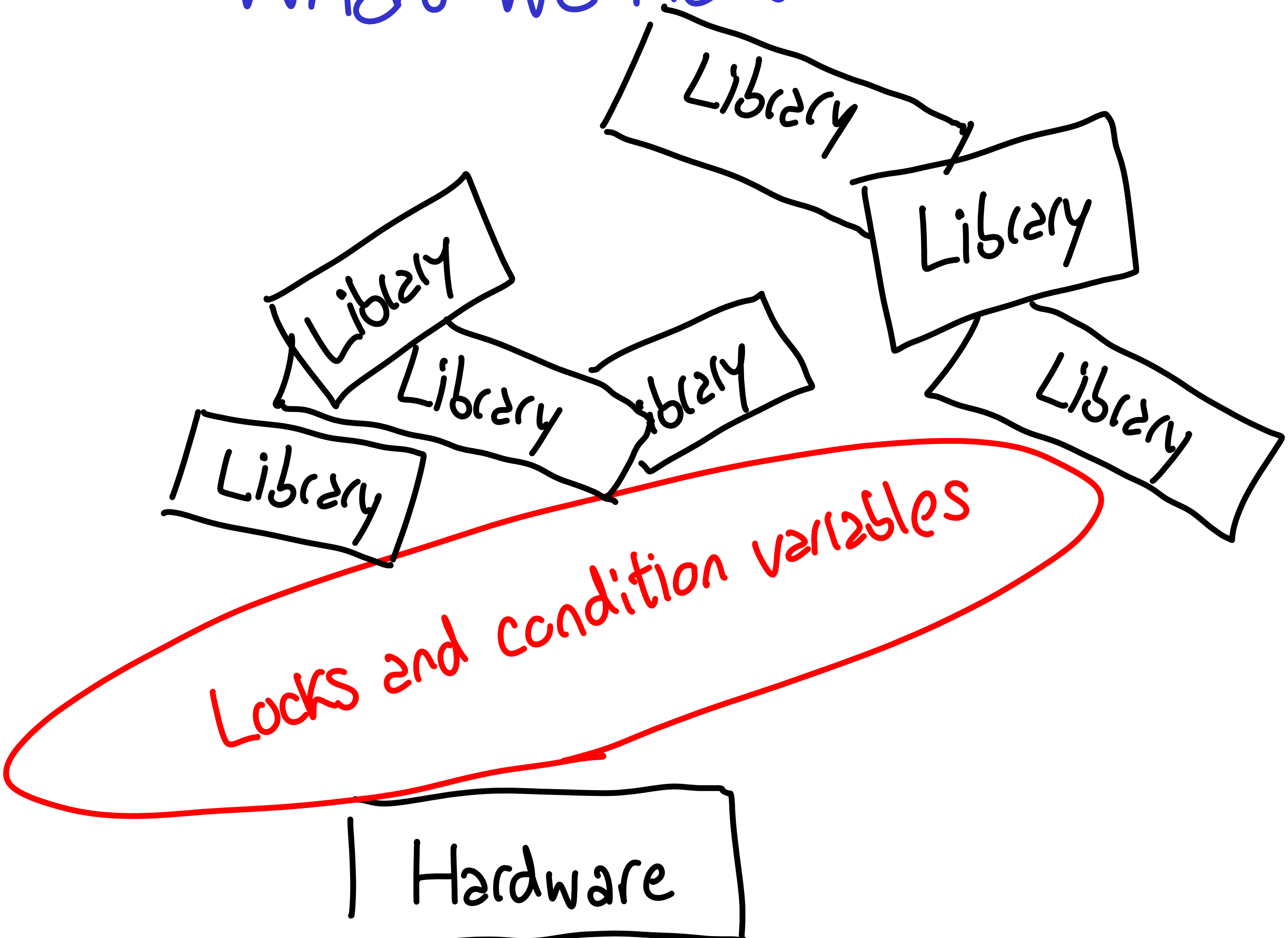
Traditional concurrency does not compose well

deadlock / lost wakeups...

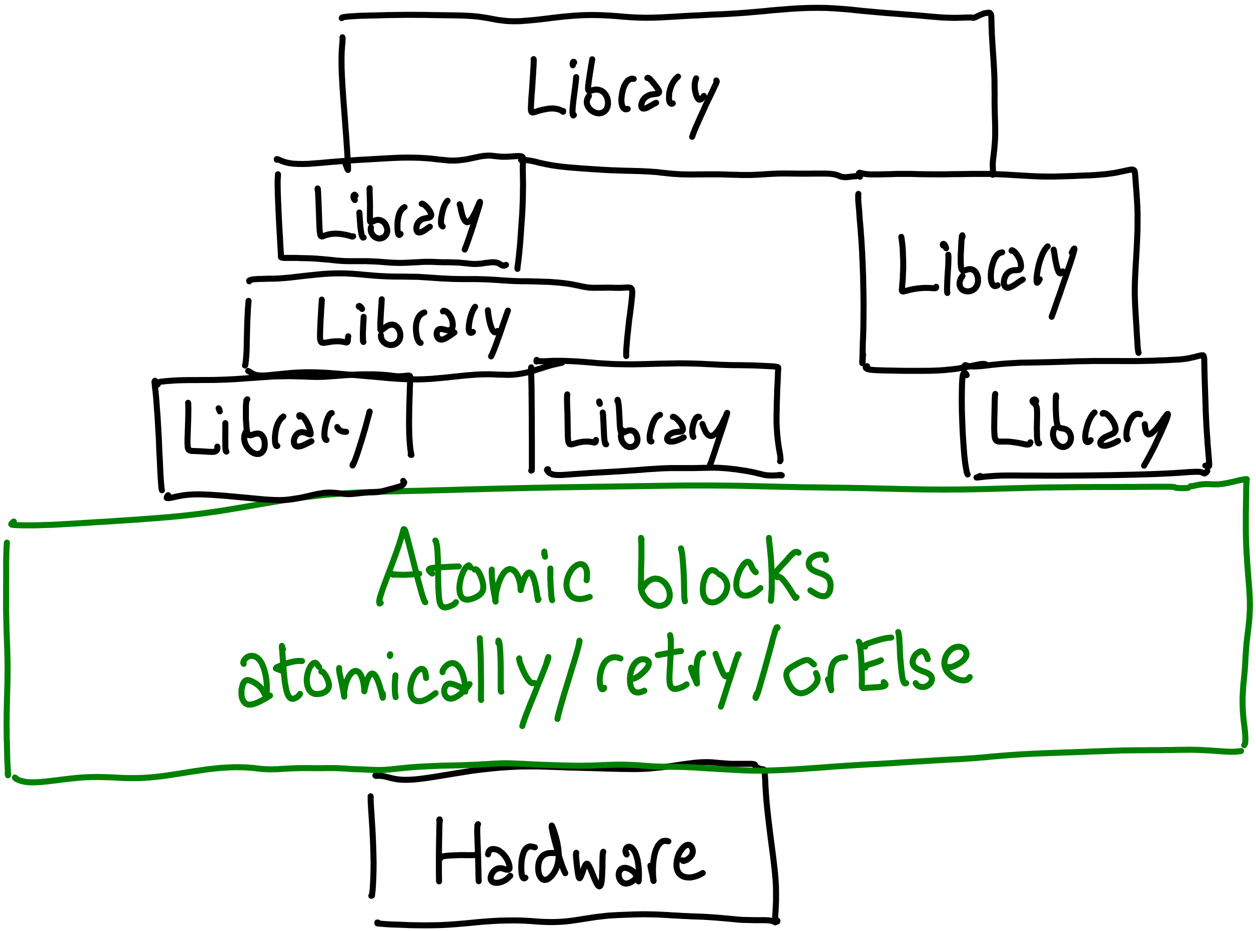
What we want


Library
Library
Library
Library
Layered concurrency abstractions
Library
Library
Library
Concurrency primitives
Hardware

What we have


Library
Library
Library
Library
Library
Library
Library
Locks and condition variables
Hardware

Idea: Replace locks with atomic blocks



What's wrong with locks?

- Races — forgot to lock
- Deadlock — used wrong order
- Lost wakeups — forgot to notify
- Error recovery — forgot to cleanup on exception

Locks are non-compositional

```
class Account {
    int balance;
    synchronized void deposit(int amt) {
        balance += amt; }
    synchronized void withdraw(int amt) {
        if(balance < amt)
            throw new OutOfMoney();
        balance -= amt; }
}
```

add fund transfer?

Locks are non-compositional

```
class Account {
    int balance;
    synchronized void deposit(int amt) {
        balance += amt; }
    synchronized void withdraw(int amt) {
        if(balance < amt)
            throw new OutOfMoney();
        balance -= amt; }

    void transfer_wrong1(Account o, int amt) {
        o.withdraw(amt);
        this.deposit(amt);      race condition
    }
}
```

Locks are non-compositional

```
class Account {
    int balance;
    synchronized void deposit(int amt) {
        balance += amt; }
    synchronized void withdraw(int amt) {
        if(balance < amt)
            throw new OutOfMoney();
        balance -= amt; }
    synchronized void transfer_wrong2
                        (Account o, int amt) {
        o.withdraw(amt);
        this.deposit(amt);
    }
}
```

deadlock

Limitations of race-freedom

```
class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
        }
        synchronized (this) {
            i = t+1;
        }
    }
}
```

Race free!
Doesn't do what you want!

Race freedom does not eliminate concurrency errors.
(SC is still not easy!)

Limitations of race-freedom

```
class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t+1;
        }
    }
    int read() { return i; }
}
```

Has a race!

Does what you want.

Race freedom is not necessary to eliminate errors!

Locks are absurdly hard to get right

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate

Locks are absurdly hard to get right

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks	Publishable result at PODC[1]

Locks are absurdly hard to get right

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks	Publishable result at PODC[1]
Atomic blocks	Undergraduate

[1] Simple, fast, and practical non-blocking concurrent queue algorithms

Atomic: easier-to-use, harder to implement

```
void deposit(int x) {
synchronized(this) {
    int tmp = balance;
    tmp += x;
    balance = tmp;
}}
```

semantics:
lock acquire/release

```
void deposit(int x) {
atomically {
    int tmp = balance;
    tmp += x;
    balance = tmp;
}}
```

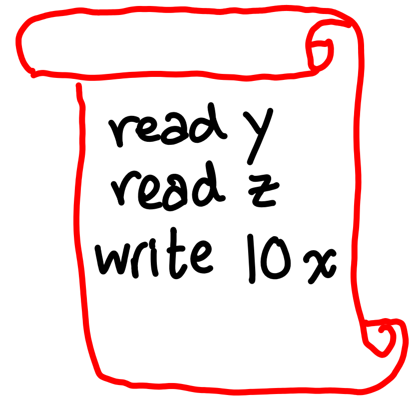
semantics: (behave as if)
no interleaved execution

Atomic memory transactions

atomically{ ... sequential code ... }

- All or nothing! (error recovery)
- Isolated!
- No deadlocks! (no locks!)

How does it work?



atomically{ … <code> … }

One possibility:

1. Execute <code> w/o any locks
2. Log each read/write in a thread-local transaction log
3. Writes go to log only, not memory

At the end, validate the log

- If valid, atomically commit changes to memory
- If invalid, re-run transaction from beginning

Software Transactional Memory

- Original paper/patent Tom Knight
- Many research/experimental implementations
  C/C++, C#, Java, OCaml, Python, Scala
- Haskell STM!
  Functional core simplifies STM concepts
  (Transaction access of non-transactional memory?)

Some questions

- Java has locking & Condition variables

Some questions

- Java has ~~locking~~ & Condition variables

atomically

Some questions

- Java has ~~locking~~ & Condition variables ?

  atomically

  blocking?

Some questions

?

- Java has ~~locking~~ & Condition variables
  atomically blocking?

- Is programming with atomic really this easy?
  efficiency?

STM in Haskell

Why Haskell?

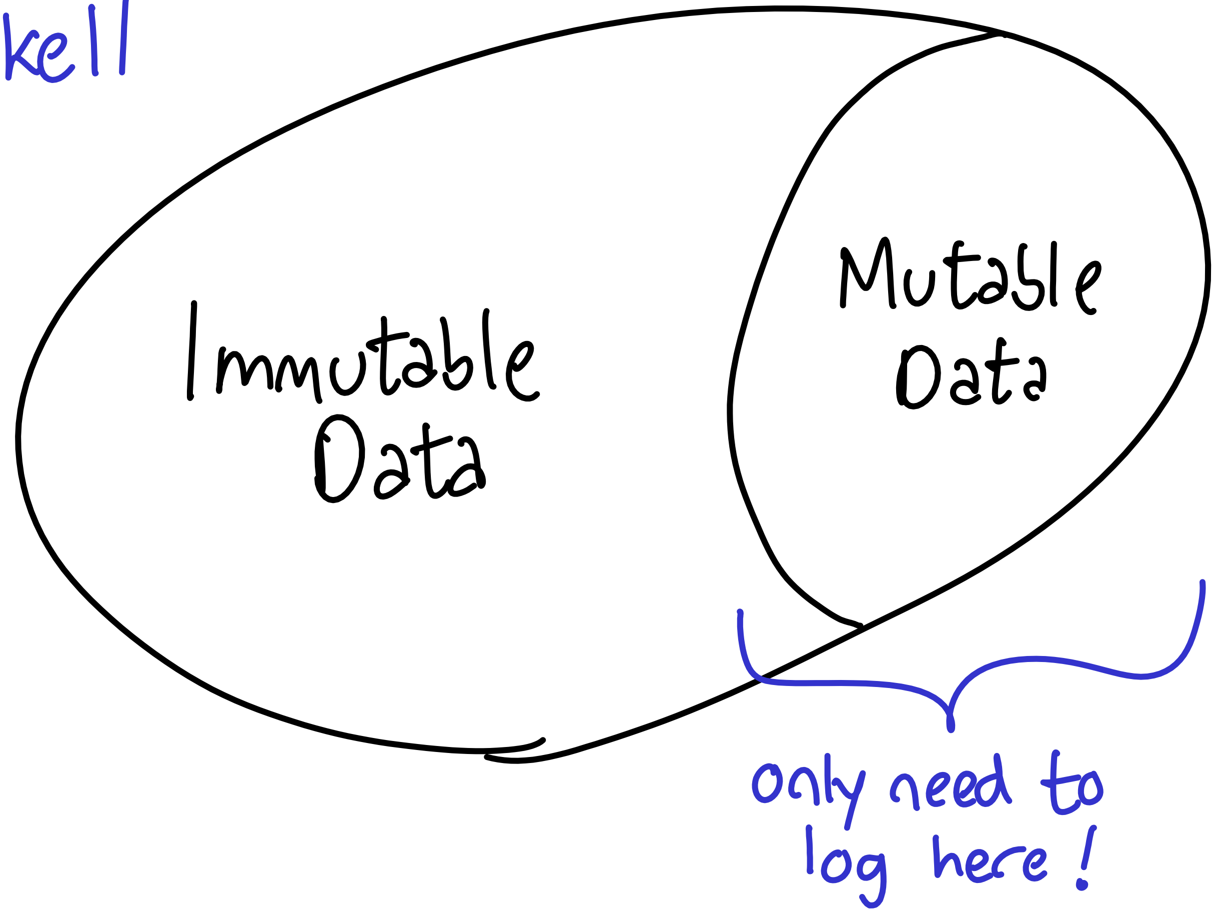
Other languages:

Mutable data

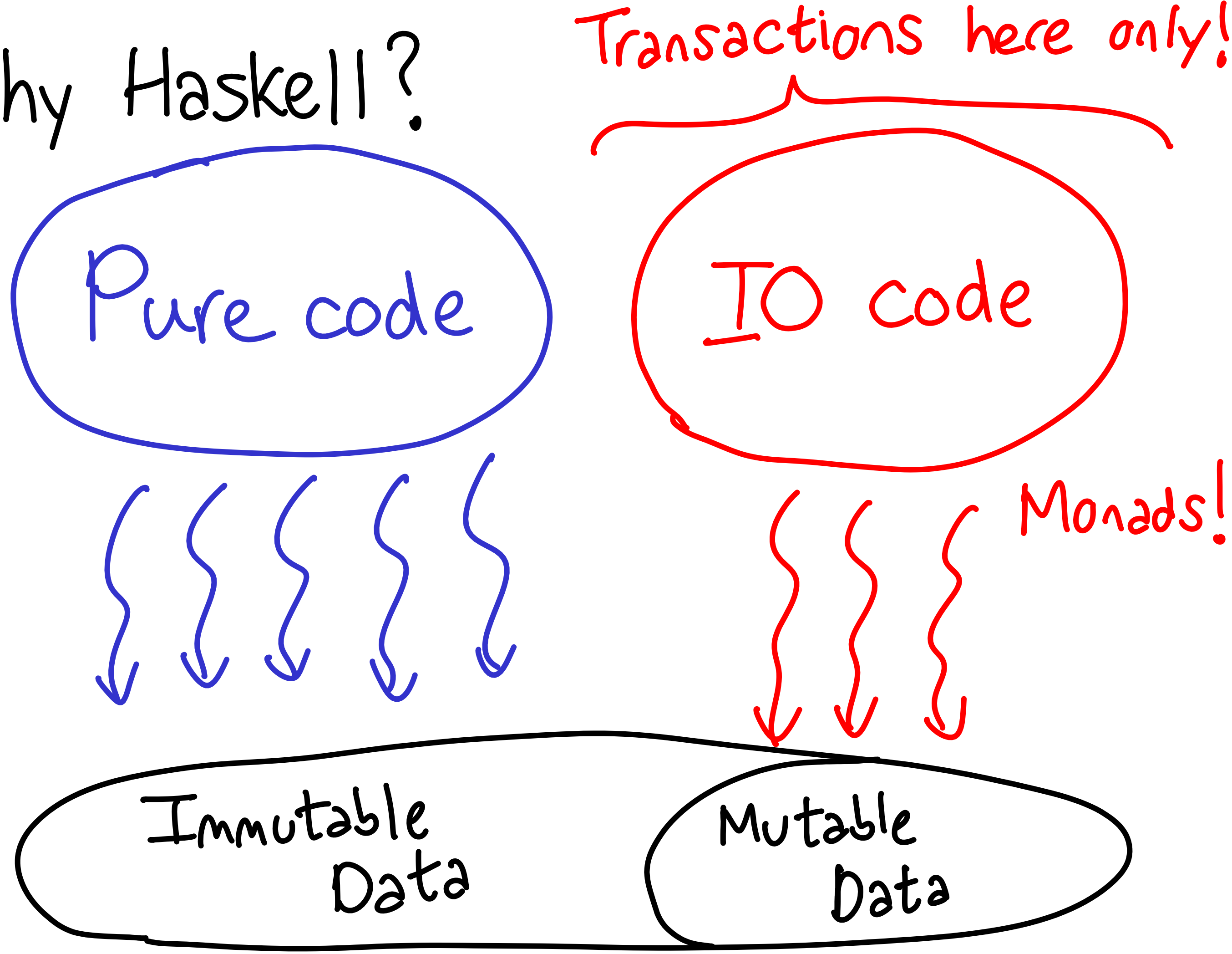
Logging = Expensive

Why Haskell?

Haskell



Why Haskell?


Transactions here only!
Pure code
IO code
Monads!
Immutable Data
Mutable Data

Why Haskell?


Guest Check
453272
TAX
Thank You - Call Again
GUEST RECEIPT
453272

Recap: Effects in the type system

$$
\begin{aligned}
\texttt{main = do \{ putStr } & \overbrace{\texttt{(reverse "yes")}}^{::\ \texttt{String}\ \text{no effects}}\texttt{;} \\
& \underbrace{\texttt{putStr "no" \}}}_{::\ \texttt{IO ()}\ \text{effects}}
\end{aligned}
$$

Recap: Mutable State

reads & writes
100% explicit

```
main = do { r <- newIORef 0;
            incR r;
            s <- readIORef r;
            print s }

incR :: IORef Int -> IO ()
incR r = do { v <- readIORef r;
              writeIORef r (v+1) }
```

r+1 disallowed!

Recap: Concurrency in Haskell

```
forkIO :: IO () -> IO ThreadId
```

```
main = do { r <- newIORef 0;
            forkIO (incR r);
            s <- readIORef r;
            print s }
```

race

(could fix with IORef → MVar)

STM in Haskell

almost!

Idea: `atomically :: IO a → IO a`

```
main = do { r ← newIORef 0;
            forkIO (atomically (incR r));
            s ← readIORef r;
            print s }
```

Problem: shouldn't allow <u>non-transactional</u> access to r...

STM in Haskell

a monad!

Better:

```
atomically :: STM a -> IO a
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM
```

TVar can only be modified in a transaction!

can't modify TVar outside of transaction

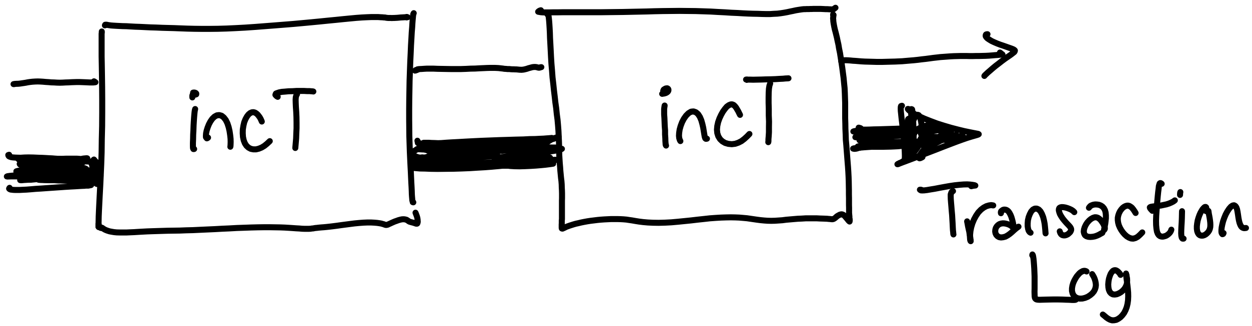
```
main = do { r <- atomically (newTVar 0);
            forkIO (atomically (incT r));
            s <- atomically (readTVar r);
            print s }
```

atomically is just a function

```
incT :: TVar Int -> STM ()
incT r = do { v <- readTVar r;
              writeTVar r (v+1) }
```

can't do IO inside transaction

Recap: Monads



$$(\gg\!=) :: STM\ a \rightarrow (a \rightarrow STM\ b) \rightarrow STM\ b$$

STM composes

Recap: Exceptions

$$\text{throwSTM} :: \text{Exception } e \Rightarrow e \to \text{STM } a$$

$$\begin{aligned}\text{catchSTM} :: \text{Exception } e \Rightarrow\ & \text{STM } a \\ & \to (e \to \text{STM } a) \\ & \to \text{STM } a\end{aligned}$$

If s throws exception in atomically s, abort transaction! No cleanup needed

Three new ideas:

retry   orElse   always

Idea 1: Compositional Blocking

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n =
    do { bal <- readTVar acc;
         if bal < n then retry
           else writeTVar acc (bal-n) }
```

retry :: STM ()

Abort transaction and try again from beginning. (Impl!)

Idea 1: Compositional Blocking

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n =
    do { bal <- readTVar acc;
         if bal < n then retry
          else writeTVar acc (bal-n) }
```

- No condition variables
- Retrying thread woken up automatically
- No danger of forgetting to re-test conditions

Why is retry compositional?

Retry can occur anywhere in an atomic block

atomically (withdraw a1 3 »
withdraw a2 7)

waits on **both** conditions

Non-composition alternative:

declare all conditions upfront

Idea 2: Choice

Suppose we want to transfer three dollars from either account a1 or a2 to account b

try this

...and if it retries

```
atomic ((withdraw a1 3
          `orElse`
        withdraw a2 3) >>
        deposit b 3 )
```

... then do this.

```
orElse :: STM a -> STM a -> STM a
```

Choice composes too!

```
transfer2 a1 a2 b
  `orElse`
transfer2 a3 a4 b
```

It's associative! (but not commutative)

An algebra!

```
retry `orElse` s ≡ s
s `orElse` retry ≡ s
```

MonadPlus!

Idea 3: Invariants

Goal: Establish invariants which are true on entry & exit from atomic

```
always :: STM Bool -> STM ()

newAccount :: STM (TVar Int)
newAccount = do
    v <- newTVar 0
    always (do cts <- readTVar v
               return (cts >= 0))

    return v
```

arbitrary STM code

Idea 3: Invariants

Goal: Establish invariants which are true on entry & exit from atomic

`always :: STM Bool → STM ()`

- Adds a new invariant to pool of invariants
- Conceptually: checked after every txn

(Actually, only check for <u>modified</u> TVars; garbage collect based on dead TVars) spec?

See Composable Memory Transactions for details!

GHC ships w/ a complete STM impl

import Control.Concurrent.STM

Microbenchmarks:

~50-80ns TVar read/put

~20ns MVar take/put

(Mutex cost)

Worse: readTVar O(n) in number of TVars in txn

STM in mainstream languages?

```
class Account {
      int balance;
      atomic void deposit(int amt) {
            balance += amt;}
      atomic void withdraw(int amt) {
            if(balance < amt)
                  throw new OutOfMoney();
            balance -= amt;}
      atomic void transfer(Account o, int amt) {
            o.withdraw(amt);
            this.deposit(amt);
      }
}
```

STM in mainstream languages?

Trouble: type system doesn't control effects

Weak atomicity

Non-transactional code may see inconsistent states on transactional code.

Strong atomicity

Non-transactional code guaranteed to see a consistent state (performance hit!)

<u>Enforcing Isolation and Ordering in STM</u>

Performance?

- Naive STM is hopelessly inefficient (think 6x slowdown or more)
- HTM: hardware transactional memory: hardware support for "lock ellision"
- Hybrid STM: use HTM support to implement STM efficiently

research direction

Atomicity not a silver bullet

How big should atomic blocks be?

Races vs. Starvation

Thread 1

```
atomic {x = 1;}
atomic {if (y==0) retry;}
```

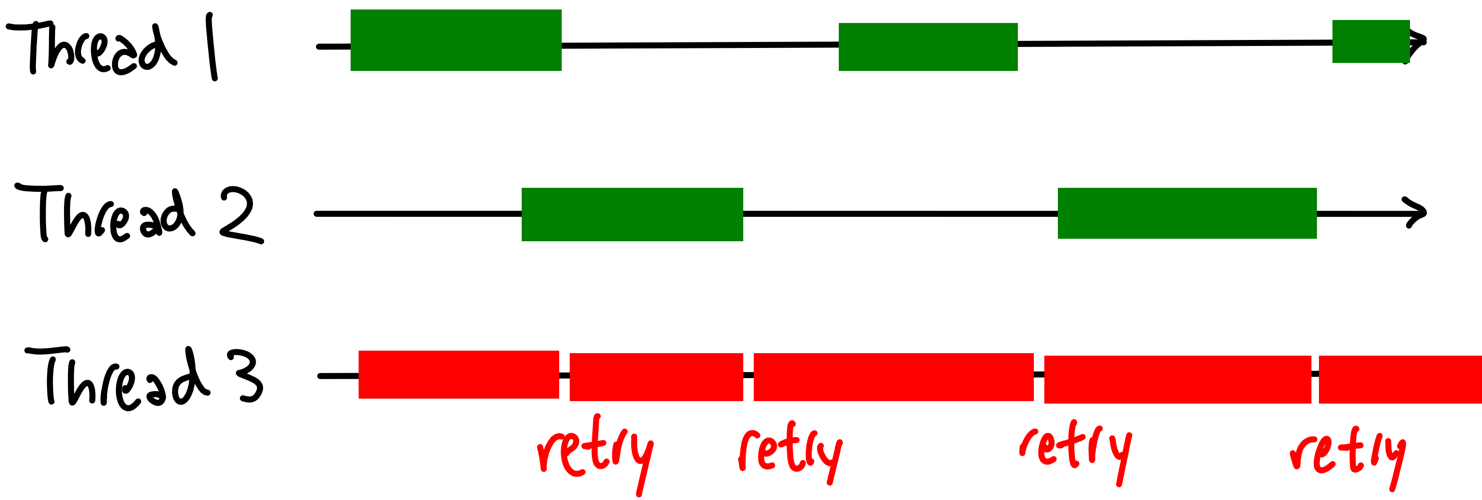
incorrect to make this a single atomic block

Thread 2

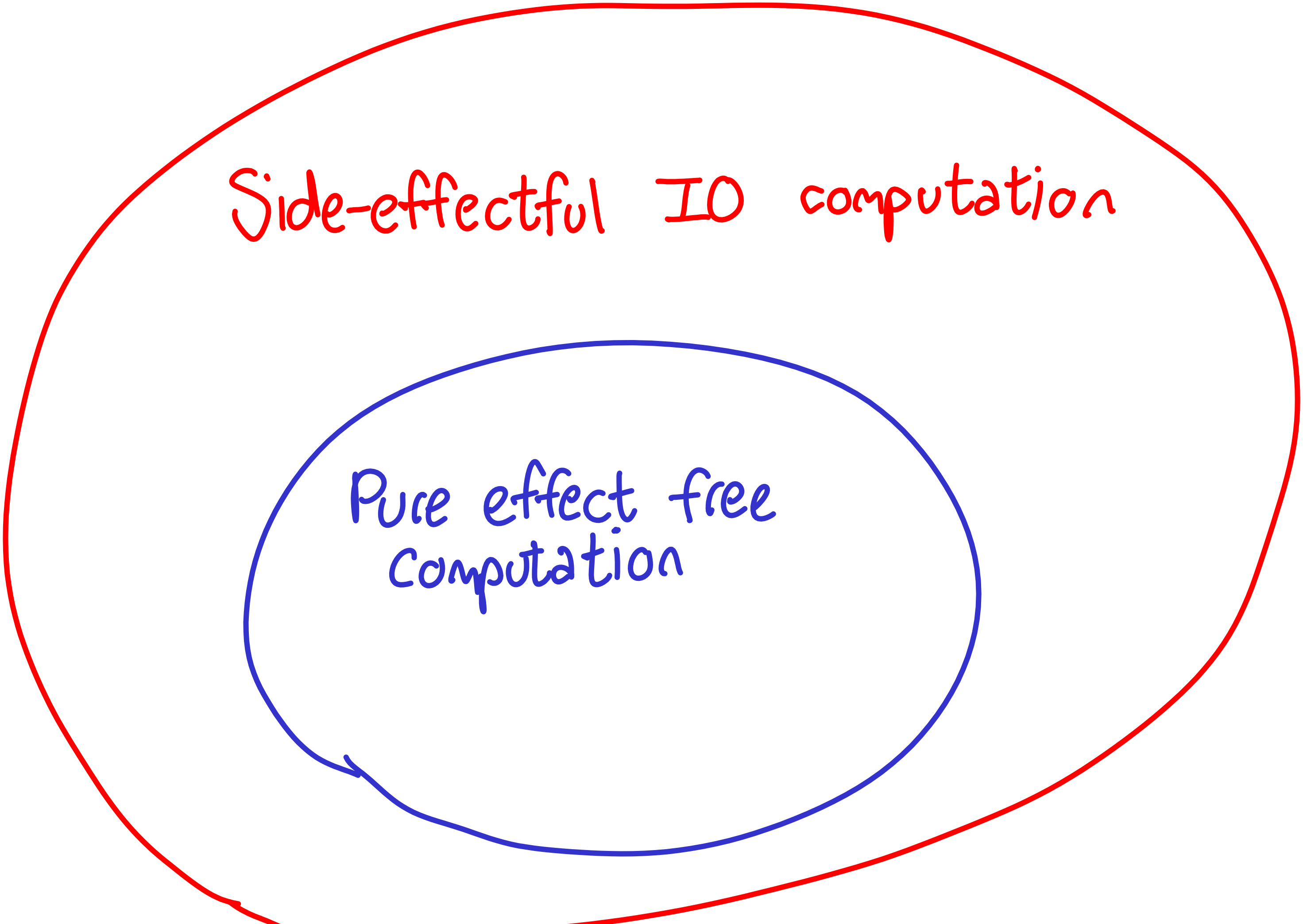
```
atomic {
    if (x==0) retry;
    y=1;
}
```

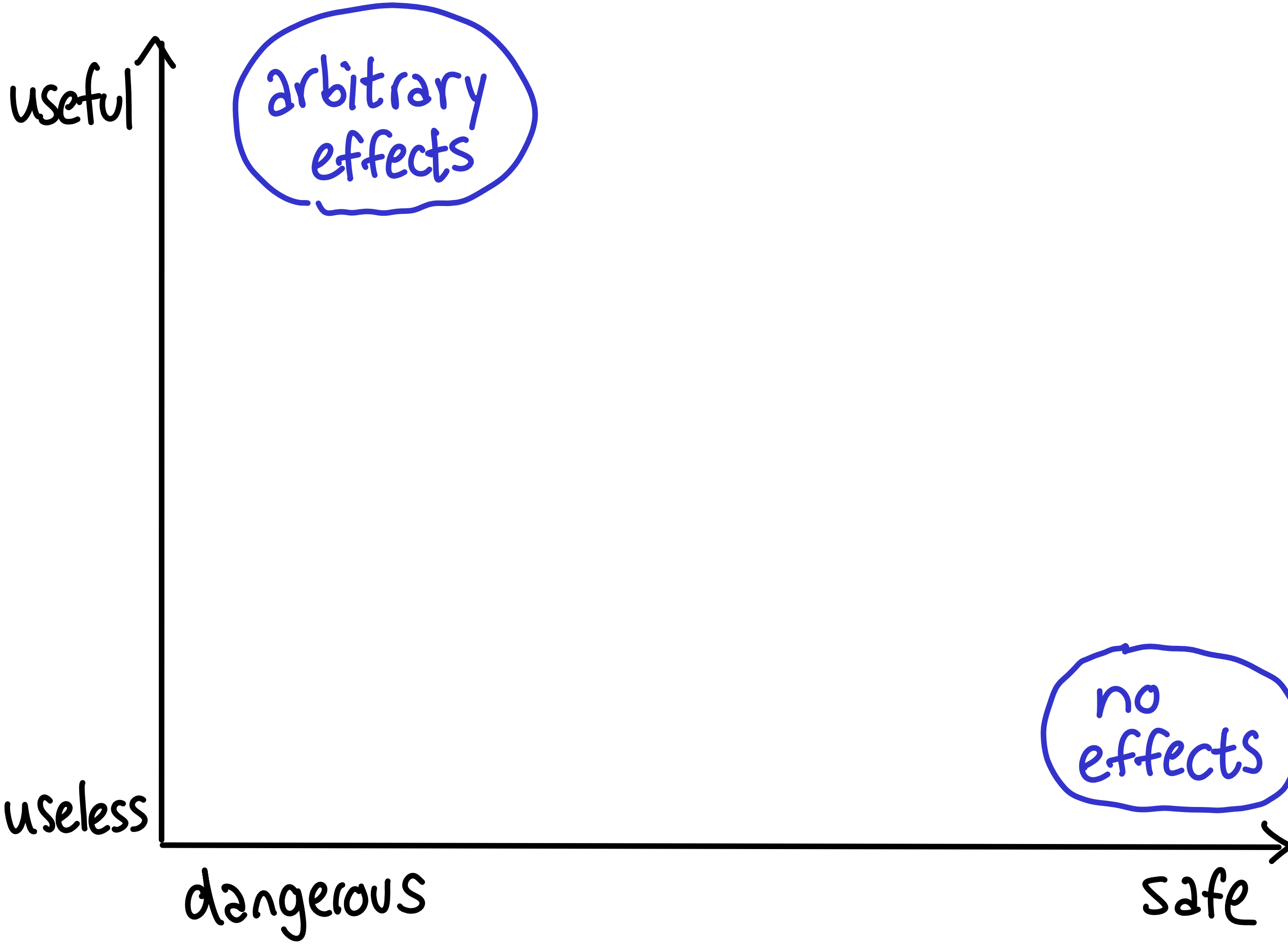
Atomicity not a silver bullet

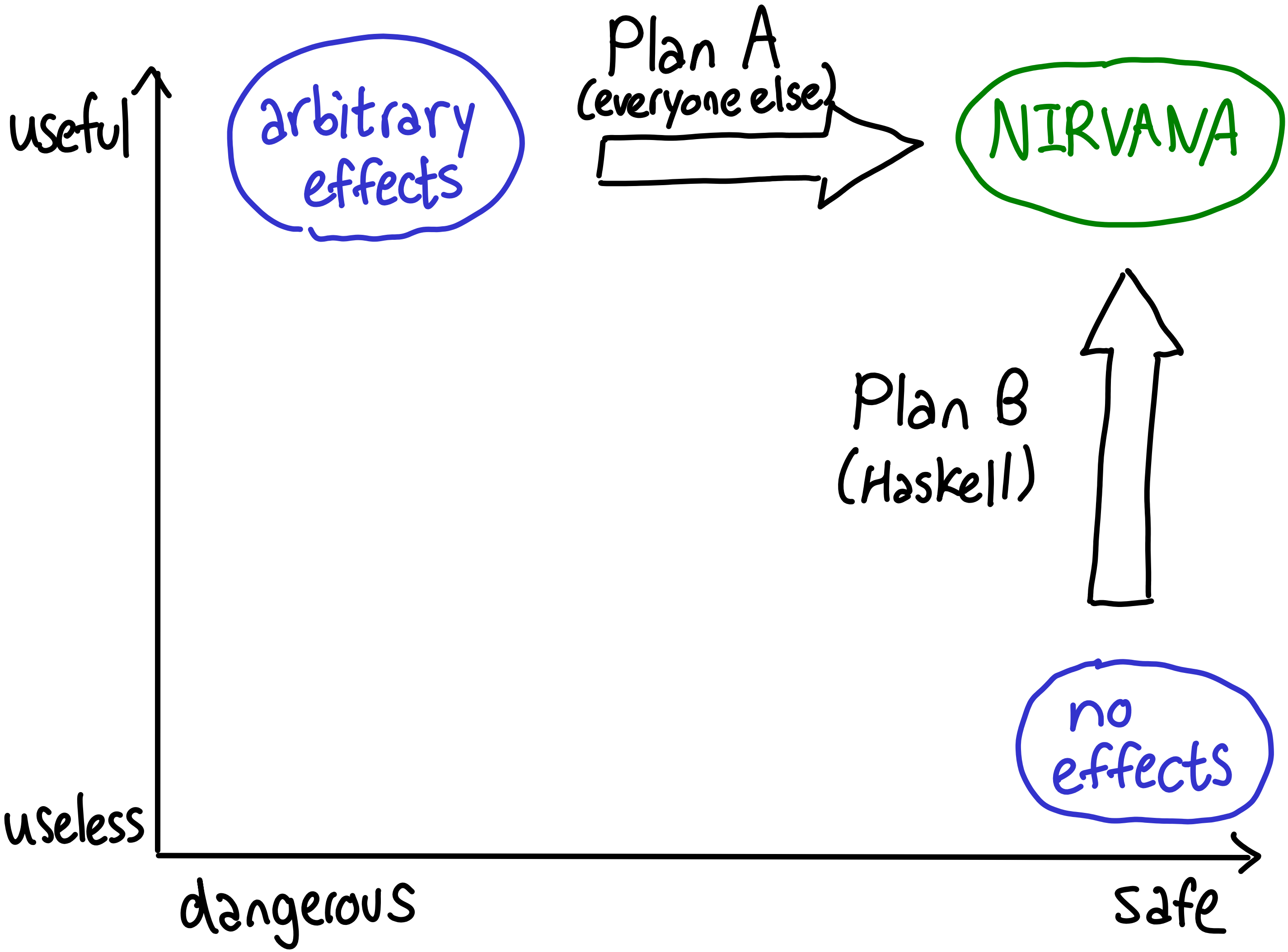
STM can't deadlock, but it's not necessarily fair



Some parting thoughts


Side-effectful IO computation
Pure effect free computation


useful
arbitrary effects
useless
no effects
dangerous
safe

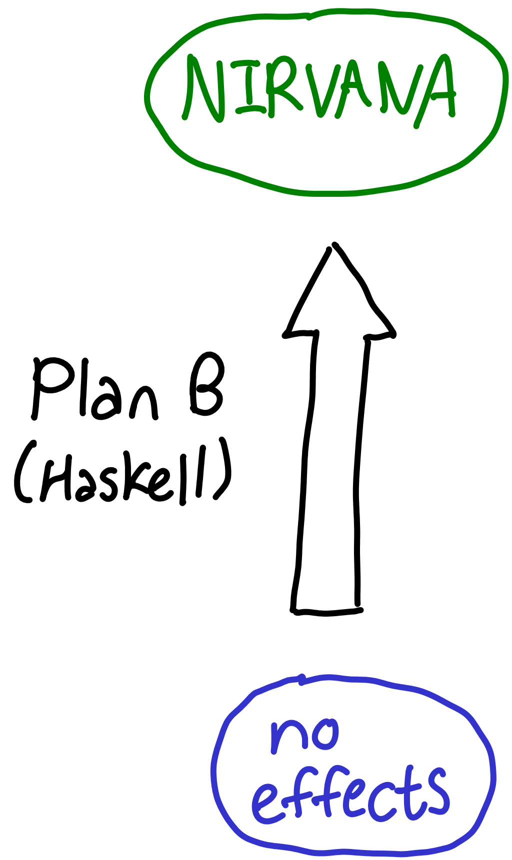

useful
useless
dangerous
safe
arbitrary effects
Plan A
(everyone else)
NIRVANA
Plan B
(Haskell)
no effects

Default: any effect
Plan: add restrictions

Examples: Regions
Ownership types
Vault, Spec#, Cyclone

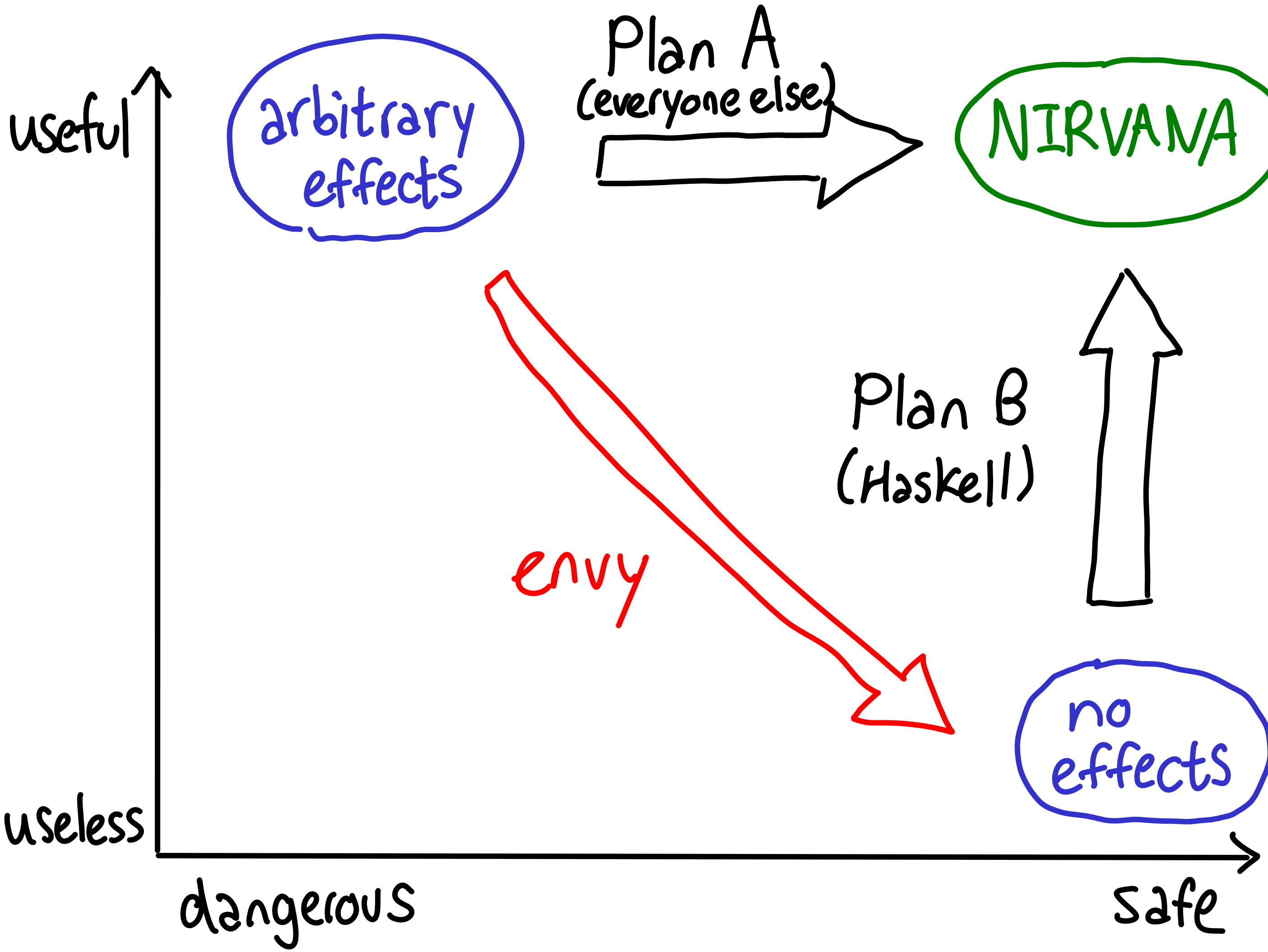
Default: no effects

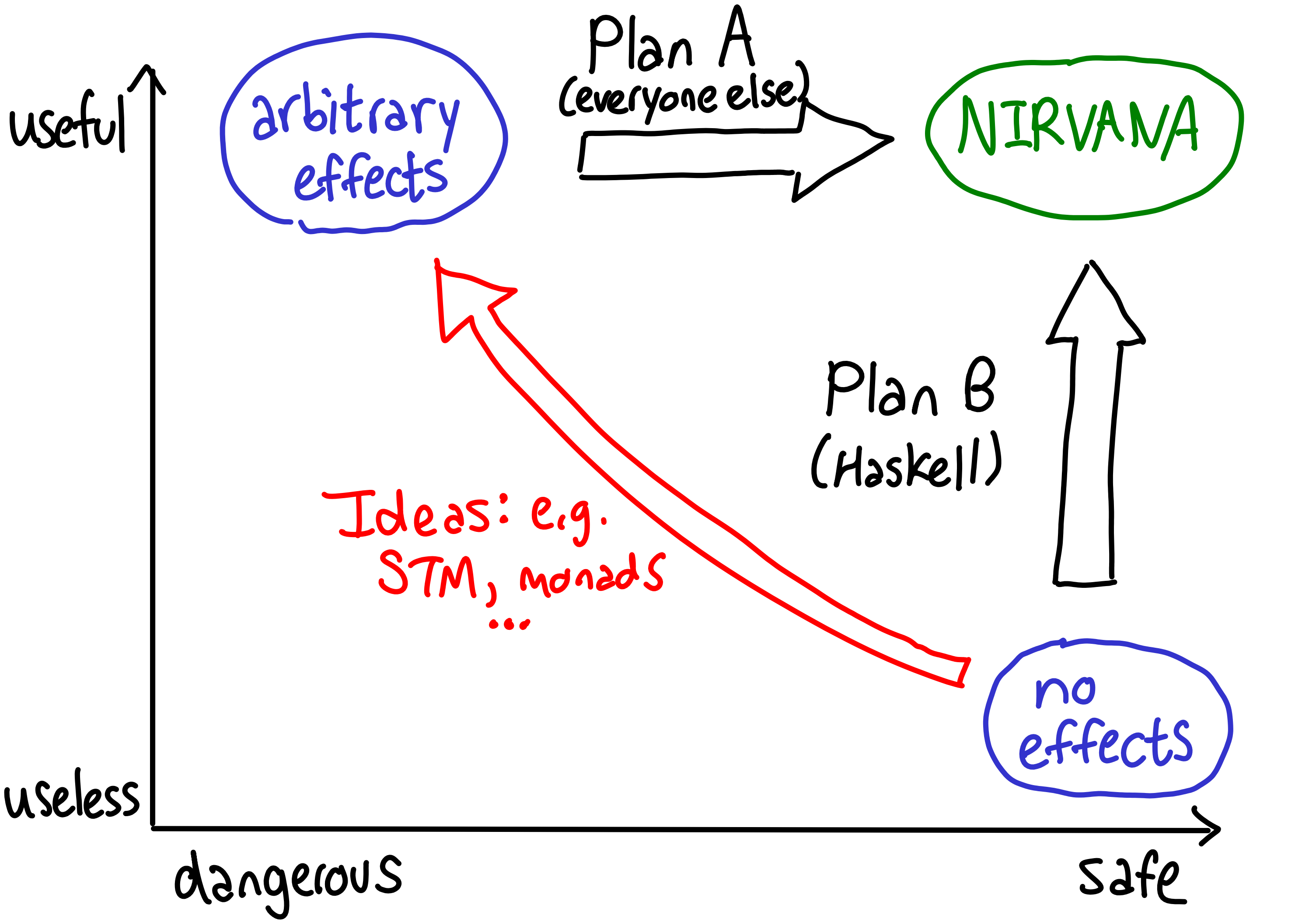
Plan: selectively permit effects



Examples:

- Domain specific languages (SQL, XQuery, MapReduce)
- Functional languages + controlled effects (Haskell)


useful
useless
dangerous
safe
arbitrary effects
Plan A
(everyone else)
NIRVANA
envy
Plan B
(Haskell)
no effects


useful
useless
dangerous
safe
arbitrary effects
Plan A
(everyone else)
NIRVANA
Plan B
(Haskell)
Ideas: e.g. STM, monads ...
no effects

One of Haskell's most significant contributions is to take purity seriously and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

— Simon Peyton Jones

Conclusion

- Atomic blocks raise the level of abstraction for concurrent programming.

  not assembly!

- Not a silver bullet

  bugs! concurrency is hard! fairness

- Performance hit, but it seems acceptible

  HTM! Transactional boosting!