SOFTWARE TRANSACTIONAL MEMORY
(WITH A DETOUR THROUGH HASKELL & MONADS)

COS 326

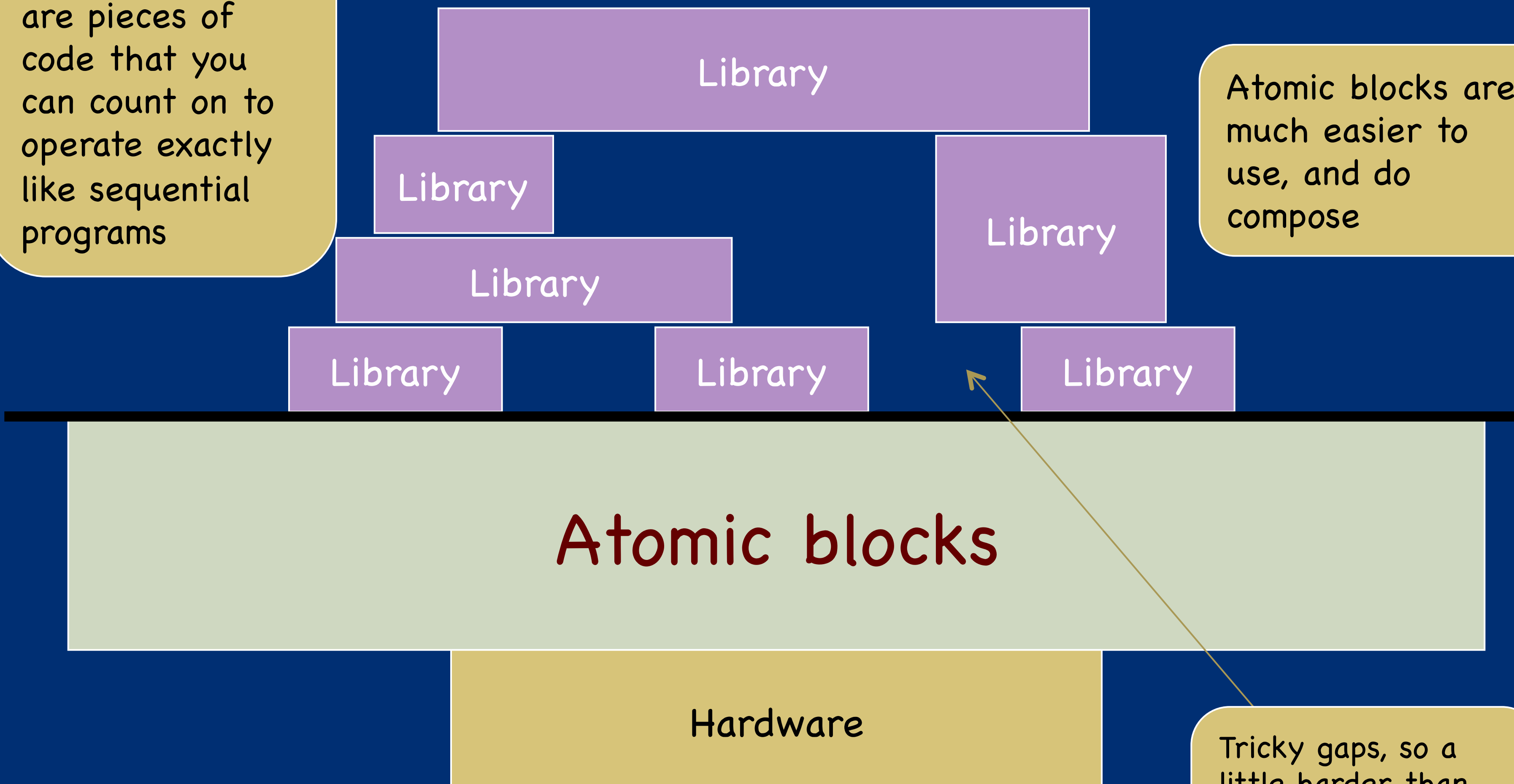
David Walker

Thanks to Kathleen Fisher and recursively to
Simon Peyton Jones for much of the content of these slides.

Optional Reading:
"Beautiful Concurrency",
"The Transactional Memory / Garbage Collection Analogy"
"A Tutorial on Parallel and Concurrent Programming in Haskell"

Second Idea: Replace locks with atomic blocks

Atomic blocks are pieces of code that you can count on to operate exactly like sequential programs



Atomic blocks are much easier to use, and do compose

Tricky gaps, so a little harder than immutable data but you can do more stuff action 1:

read x
write x
read x
write x

action 2:

read x
write x
read x
write x

Software Transactions:
A means to cut down program non-determinism

action 1:

read x
write x
read x
write x

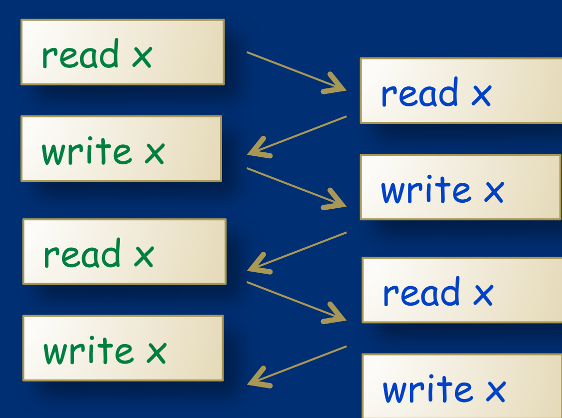
action 2:

read x
write x
read x
write x

with transactions:

read x write x read x write x read x write x read x write x	or	read x write x read x write x read x write x read x write x

without atomic transactions:



STM in Haskell

Concurrent Threads in Haskell

- The fork function spawns a thread.
- It takes an action as its argument.

```
fork :: IO a -> IO ThreadId
```

```
main = do
          id <- fork action1
          action2
          ...
```

action 1 and
action 2 in
parallel

Atomic Blocks in Haskell

- Idea: add a function atomic that guarantees atomic execution of a suspended (effectful) computation

```
main = do
          id <- fork (atomic action1)
          atomic action2
          ...
```

action 1 and
action 2
atomic
and parallel

```
main = do
        id <- fork (atomic action1)
        atomic action2
        ...
```

with transactions:

action 1:

read x
write x
read x
write x

action 2:

read x
write x
read x
write x

read x
write x
read x
write x

read x
write x
read x
write x

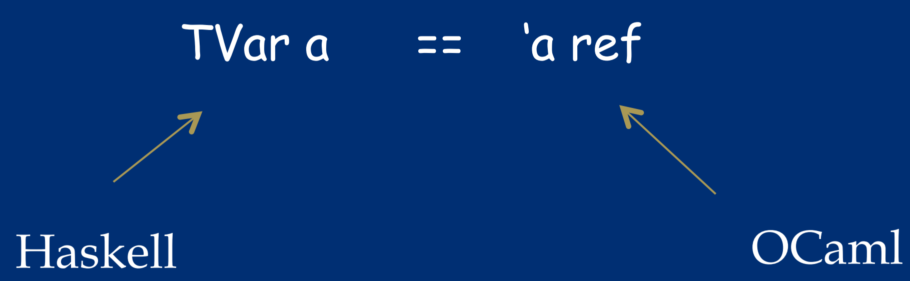
or

read x
write x
read x
write x

read x
write x
read x
write x

Atomic Details

- Introduce a type for imperative transaction variables (TVar) and a new Monad (STM) to track transactions.
  - STM a == a computation producing a value with type a that does transactional memory book keeping on the side
  - Haskell type system ensures TVars can only be modified in transactions.




```
atomic    :: STM a -> IO a
new       :: a -> STM (TVar a)
read      :: TVar a -> STM a
write     :: TVar a -> a -> STM ()
```

Atomic Example

```
-- inc adds 1 to the mutable reference r
inc :: TVar Int -> STM ()

inc r = do
           v <- read r
           write r (v+1)


main  = do
           r <- atomic (new 0)
           fork (atomic (inc r))
           atomic (inc r);
```

Atomic Example

```
-- inc adds 1 to the mutable reference r
inc :: TVar Int -> STM ()

inc r = do
           v <- read r
           write r (v+1)


main  = do
           r <- atomic (new 0)
           fork (atomic (inc r))
           atomic (inc r);
```

Haskell is lazy so these computations are suspended and executed within the atomic block

STM in Haskell

```
atomic    :: STM a -> IO a
new       :: a -> STM (TVar a)
read      :: TVar a -> STM a
write     :: TVar a -> a -> STM()
```

The STM monad includes a specific set of operations:

- Can't use TVars outside atomic block
- Can't do IO inside atomic block:

```
atomic (if x<y then launchMissiles)
```

- atomic is a function, not a syntactic construct
  - called *atomically* in the actual implementation
- ...and, best of all...

STM Computations Compose (unlike locks)

```
inc r = do
           v <- read r
           write r (v+1)

inc2 r = do
           inc r
           inc r

foo = atomic (inc2 r)
```

The type guarantees that an STM computation is always executed atomically.

- Glue many STM computations together inside a "do" block
- Then wrap with atomic to produce an IO action.

Composition is THE way to build big programs that work

Exceptions

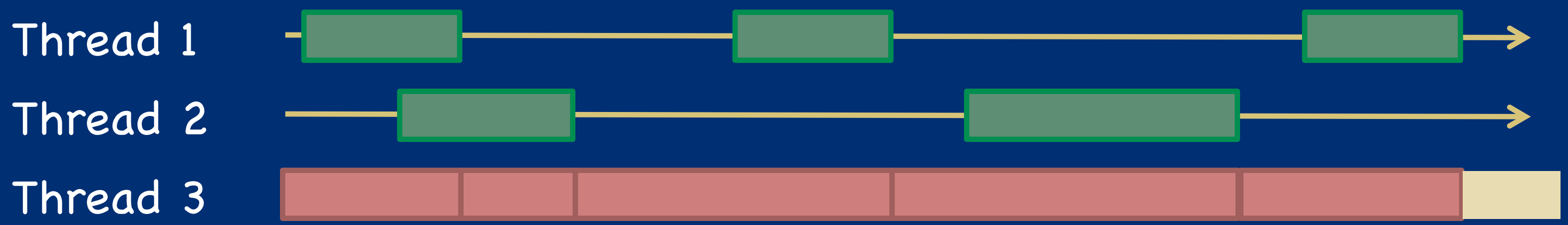
- The STM monad supports exceptions:

```
throw :: Exception -> STM a
catch :: STM a ->(Exception -> STM a) -> STM a
```

- In the call (atomic s), if s throws an exception, *the transaction is aborted with no effect* and the exception is propagated to the enclosing code.
- *No need to restore invariants, or release locks!*

Starvation

- Worry: Could the system "thrash" by continually colliding and re-executing?
- No: A transaction can be forced to re-execute only if another succeeds in committing. That gives a strong *progress guarantee*.
- But: A particular thread could starve:



Three more 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
      writeTVar acc (bal-n)
```

```
retry :: STM ()
```

- retry means "abort the current transaction and re-execute it from the beginning".
- Implementation avoids early retry using reads in the transaction log (i.e. acc) to wait on all read variables.
  - ie: retry only happens when one of the variables read on the path to the retry changes

Compositional Blocking

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

- Retrying thread is woken up automatically when acc is written, so there is no danger of forgotten notifies.
- No danger of forgetting to test conditions again when woken up because the transaction runs from the beginning.
- *Correct-by-construction design!*

What makes Retry Compositional?

- **retry** can appear anywhere inside an atomic block, including nested deep within a call. For example,

```
atomic (do { withdraw a1 3;
             withdraw a2 7 })
```

waits for:

- a1 balance > 3
- *and* a2 balance > 7
- *without any change to withdraw function.*

Idea 2: Choice

- Suppose we want to transfer 3 dollars from either account a1 or a2 into account b.

Try this

...and if it retries, try this

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

then afterward, do this

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

Choice is composable, too!

```
transfer ::
    TVar Int ->
    TVar Int ->
    TVar Int ->
    STM ()

transfer a1 a2 b =
  do
    withdraw a1 3 `orElse` withdraw a2 3
    deposit b 3
```

```
atomic (
  transfer a1 a2 b
    `orElse` transfer a3 a4 b
)
```

- The function transfer calls orElse, but calls to transfer can still be composed with orElse.

Composing Transactions

- A transaction is a value of type STM a.
- Transactions are first-class values.
- Build a big transaction by composing little transactions: in sequence, using orElse and retry, inside procedures....
- Finally seal up the transaction with

  atomic :: STM a -> IO a

Equational Reasoning

STM supports nice equations for reasoning:

```
a `orElse` (b `orElse` c) == (a `orElse` b) `orElse` s
retry `orElse` s == s

s `orElse` retry == s
```

(These equations make STM an instance of a structure known as a MonadPlus -- a Monad with some extra operations and properties.)

Idea 3: Invariants

The route to sanity is to *establish invariants* that are *assumed on entry*, and *guaranteed on exit*, by *every atomic block*.

- just like in a module with *representation invariants*
- this gives you *local reasoning about your code*

- We want to check these guarantees. But we don't want to test every invariant after every atomic block.
- Hmm.... Only test when something read by the invariant has changed.... rather like retry.

Invariants: One New Primitive

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

```
newAccount :: STM (TVar Int)

newAccount =
  do { r <- new 0;
       always (accountInv r);
       return v }


accountInv r = do { x <- read r;
                    return (x >= 0)};
```

An arbitrary boolean valued STM computation

Any transaction that modifies the account will check the invariant (no forgotten checks). If the check fails, the transaction restarts. A persistent assert!!

What always does

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

- The function **always** adds a new invariant to a global pool of invariants.
- Conceptually, every invariant is checked as every transaction commits.
- But the implementation checks only invariants that read TVars that have been written by the transaction
- ...and garbage collects invariants that are checking dead Tvars.

What does it all mean?

- Everything so far is intuitive and arm-wavey.
- But what happens if it's raining, and you are inside an orElse and you throw an exception that contains a value that mentions...?
- We need a precise specification!

One exists

IO transitions $\quad P;\Theta \xrightarrow{a} Q;\Theta'$

$$
\begin{array}{lcll}
\mathbb{P}[\texttt{putChar}\ c];\ \Theta & \xrightarrow{!c} & \mathbb{P}[\texttt{return}\ ()];\ \Theta & (PUTC)\\
\mathbb{P}[\texttt{getChar}];\ \Theta & \xrightarrow{?c} & \mathbb{P}[\texttt{return}\ c];\ \Theta & (GETC)\\
\mathbb{P}[\texttt{forkIO}\ M];\ \Phi,\Delta & \rightarrow & (\mathbb{P}[\texttt{return}\ t] \mid M_t);\ \Phi,\Delta\cup\{t\} \quad t \notin \Delta & (FORK)
\end{array}
$$

$$
\frac{M \rightarrow N}{\mathbb{P}[M];\ \Theta \rightarrow \mathbb{P}[N];\ \Theta}\ (ADMIN)
$$

$$
\frac{M;\ \Theta \stackrel{*}{\Rightarrow} \texttt{return}\ N;\ \Theta'}{\mathbb{P}[\texttt{atomically}\ M];\ \Theta \rightarrow \mathbb{P}[\texttt{return}\ N];\ \Theta'}\ (ARET)
\qquad
\frac{M;\ \Phi,\Delta \stackrel{*}{\Rightarrow} \texttt{throw}\ N;\ \Phi,\Delta'}{\mathbb{P}[\texttt{atomically}\ M];\ \Phi,\Delta \rightarrow \mathbb{P}[\texttt{throw}\ N];\ \Phi,\Delta'}\ (ATHROW)
$$

Administrative transitions $\quad M \rightarrow N$

$$
\begin{array}{rcll}
M & \rightarrow & V \quad \text{if } \mathcal{E}[\![M]\!] = V \text{ and } M \not\equiv V & (EVAL)\\
\texttt{return}\ N \texttt{>>=} M & \rightarrow & M\ N & (BIND)\\
\texttt{throw}\ N \texttt{>>=} M & \rightarrow & \texttt{throw}\ N & (THROW)\\
\texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (CATCH1)\\
\texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (CATCH2)
\end{array}
$$

STM transitions $\quad M;\Theta \Rightarrow N;\Theta'$

$$
\begin{array}{rcll}
\mathbb{E}[\texttt{readTVar}\ r];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ \Phi(r)];\ \Phi,\Delta \quad \text{if } r \in dom(\Phi) & (READ)\\
\mathbb{E}[\texttt{writeTVar}\ r\ N];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ ()];\ \Phi[r \mapsto M],\Delta \quad \text{if } r \in dom(\Phi) & (WRITE)\\
\mathbb{E}[\texttt{newTVar}\ M];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ r];\ \Phi[r \mapsto M],\Delta\cup\{r\} \quad \text{if } r \notin \Delta & (NEW)
\end{array}
$$

$$
\frac{M \rightarrow N}{\mathbb{E}[M];\ \Theta \rightarrow \mathbb{E}[N];\ \Theta}\ (AADMIN)
$$

$$
\frac{\mathbb{E}[M_1];\ \Theta \stackrel{*}{\Rightarrow} \mathbb{E}[\texttt{return}\ N];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[\texttt{return}\ N];\ \Theta'}\ (OR1)
\qquad
\frac{\mathbb{E}[M_1];\ \Theta \stackrel{*}{\Rightarrow} \mathbb{E}[\texttt{throw}\ N];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[\texttt{throw}\ N];\ \Theta'}\ (OR2)
$$

$$
\frac{\mathbb{E}[M_1];\ \Theta \stackrel{*}{\Rightarrow} \mathbb{E}[\texttt{retry}];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[M_2];\ \Theta}\ (OR3)
$$

See "[Composable Memory Transactions]" for details.

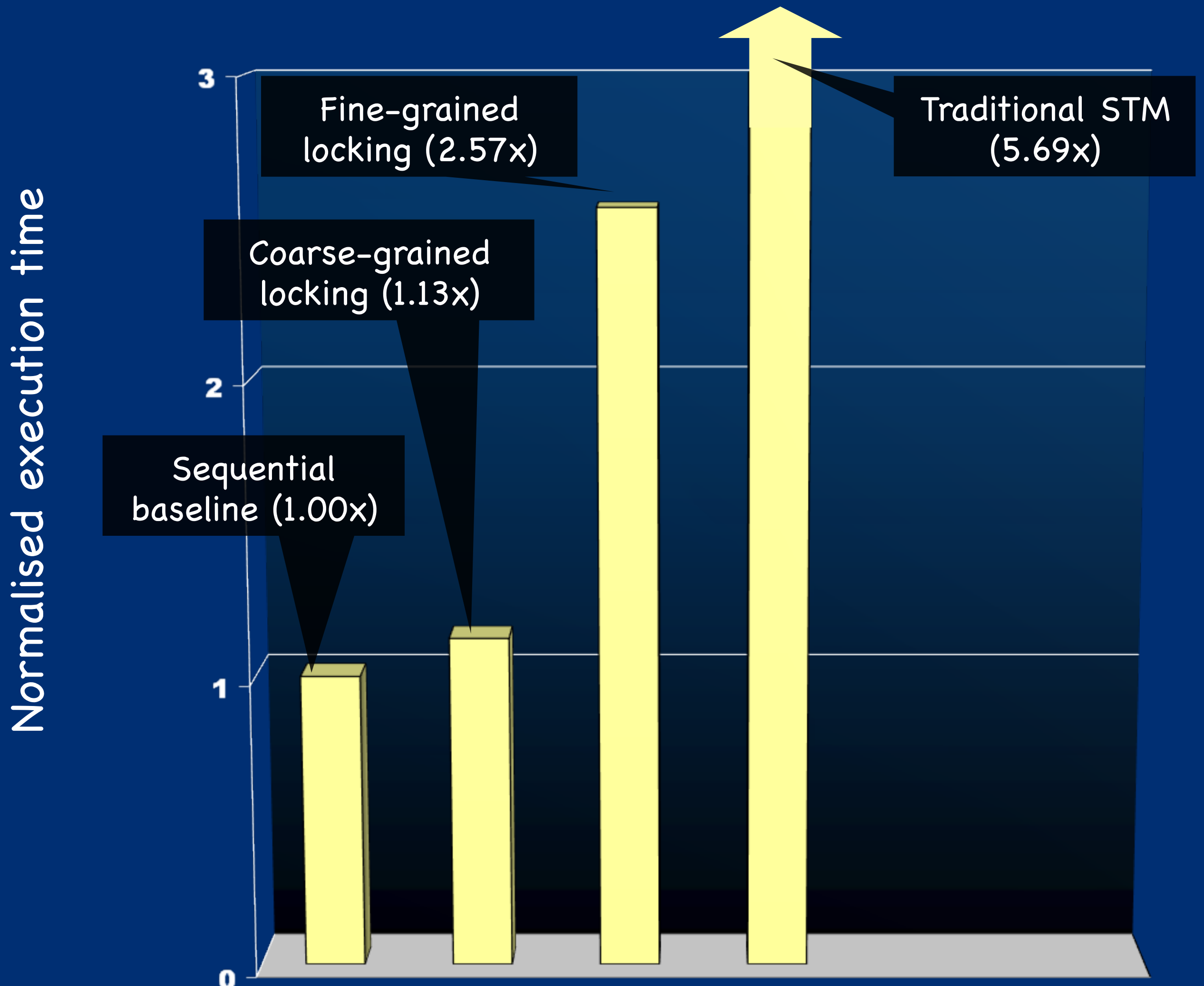
Take COS 510 to understand what it means!

Haskell Implementation

Performance

- At first, atomic blocks look insanely expensive. A naive implementation (c.f. databases):
  - Every load and store instruction logs information into a thread-local log.
  - A store instruction writes the log only.
  - A load instruction consults the log first.
  - Validate the log at the end of the block.
    - If succeeds, atomically commit to shared memory.
    - If fails, restart the transaction.

State of the Art Circa 2003



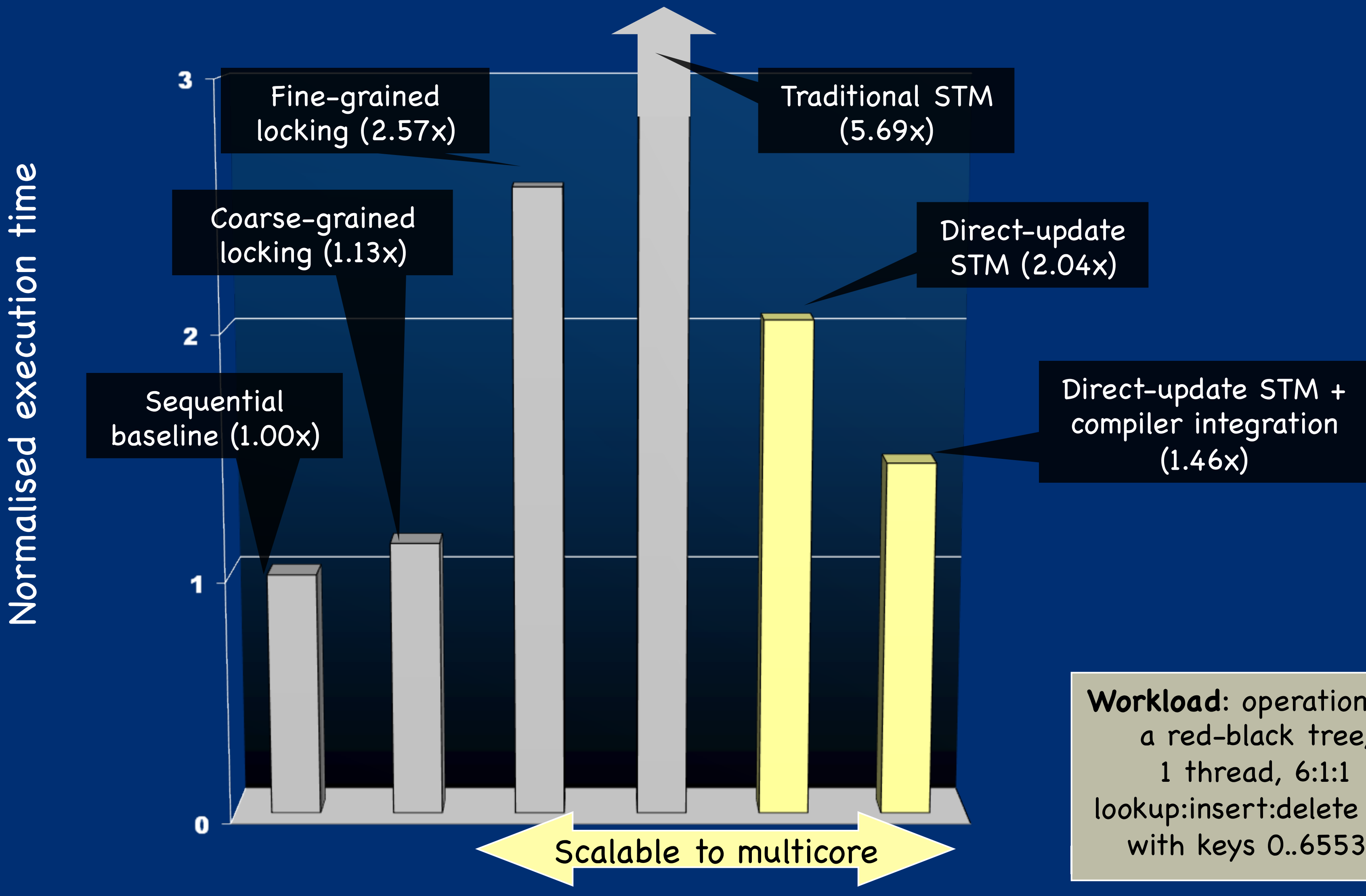
Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

See "Optimizing Memory Transactions" for more information.

New Implementation Techniques

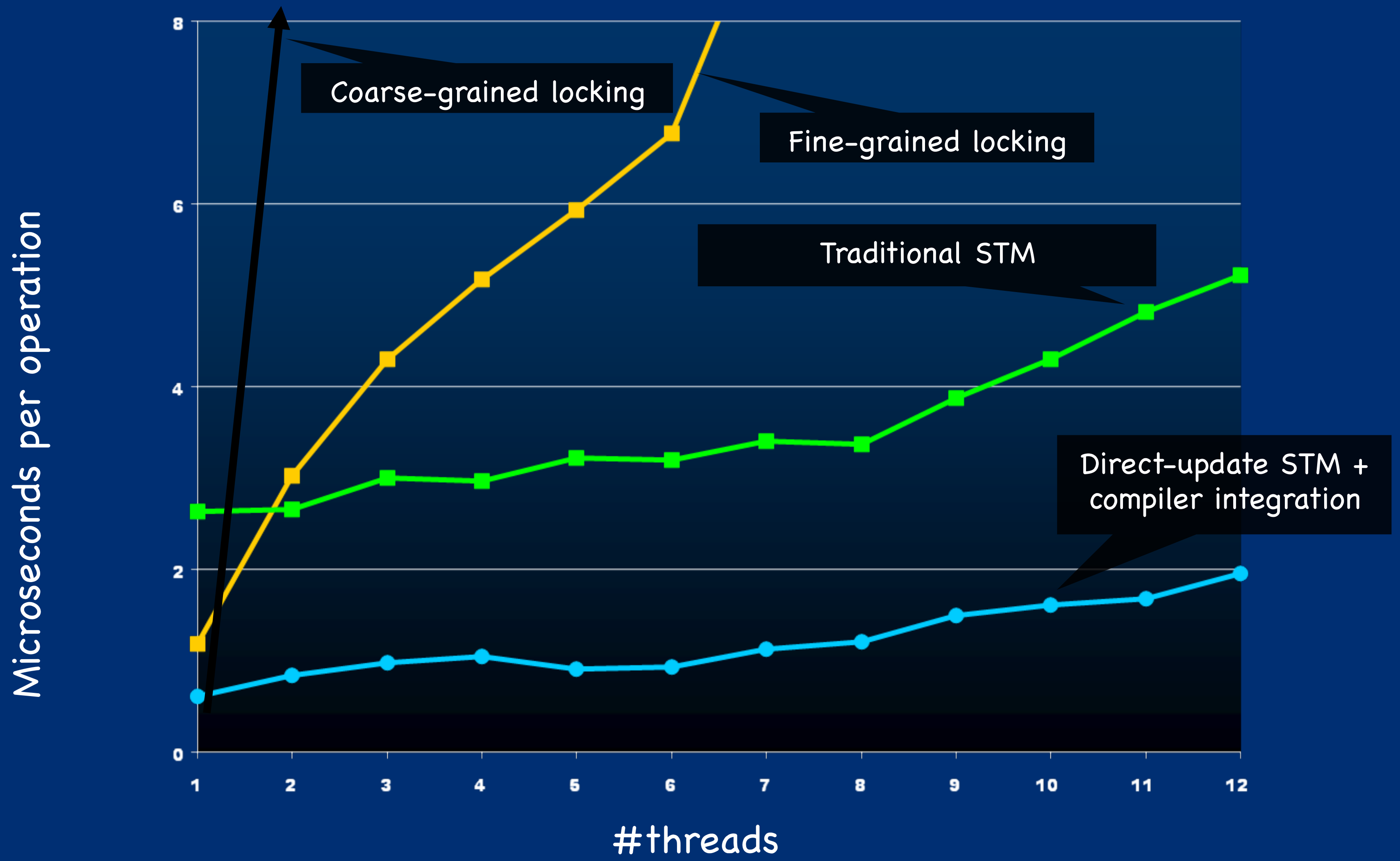
- Direct-update STM
  - Allows transactions to make updates in place in the heap
  - Avoids reads needing to search the log to see earlier writes that the transaction has made
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts
- Compiler integration
  - Decompose transactional memory operations into primitives
  - Expose these primitives to compiler optimization (e.g. to hoist concurrency control operations out of a loop)
- Runtime system integration
  - Integrates transactions with the garbage collector to scale to atomic blocks containing 100M memory accesses

Results: Concurrency Control Overhead



Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

Results: Scalability
(for some benchmark; your experience may vary)



Performance, Summary

- Naïve STM implementation is hopelessly inefficient.
- There is a lot of research going on in the compiler and architecture communities to optimize STM.
- This work typically assumes transactions are smallish and have low contention. If these assumptions are wrong, performance can degrade drastically.
- We need more experience with "real" workloads and various optimizations before we will be able to say for sure that we can implement STM sufficiently efficiently to be useful.

STM Wrapup

STM in Mainstream Languages

- There are similar proposals for adding STM to Java and other mainstream languages.

```
class Account {
  float balance;
  void deposit(float amt) {
    atomic { balance += amt; }
  }
  void withdraw(float amt) {
    atomic {
      if(balance < amt) throw new OutOfMoneyError();
      balance -= amt;  }
  }
  void transfer(Acct other, float amt) {
    atomic {  // Can compose withdraw and deposit.
      other.withdraw(amt);
      this.deposit(amt); }
  }
}
```

Weak vs Strong Atomicity

- Unlike Haskell, type systems in mainstream languages don't control where effects occur.
- What happens if code outside a transaction conflicts with code inside a transaction?
  - Weak Atomicity: Non-transactional code can see inconsistent memory states. Programmer should avoid such situations by placing all accesses to shared state in transaction.
  - Strong Atomicity: Non-transactional code is guaranteed to see a consistent view of shared state. This guarantee may cause a performance hit.

For more information: "Enforcing Isolation and Ordering in STM"

Even in Haskell: Easier, But Not Easy.

The essence of shared-memory concurrency is *deciding where critical sections should begin and end*. This is still a hard problem.

- Too small: application-specific data races (Eg, may see deposit but not withdraw if transfer is not atomic).
- Too large: delay progress because deny other threads access to needed resources.

In Haskell, we can compose STM subprograms but at some point, we must decide to wrap an STM in "atomic"

- When and where to do it can be a hard decision

Programs can still be non-deterministic and hard to debug

Still Not Easy, Example

- Consider the following program:

```
Initially, x = y = 0
```

```
Thread 1
// atomic {                              //A0
     atomic { x = 1; }                   //A1
     atomic { if (y==0) abort; } //A2
//}
```

```
Thread 2
atomic {        //A3
  if (x==0) abort;
  y = 1;
}
```

- Successful completion requires A3 to run after A1 but before A2.
- So deleting a critical section (by uncommenting A0) changes the behavior of the program (from terminating to non-terminating).

STM Conclusions

- Atomic blocks (atomic, retry, orElse) dramatically raise the level of abstraction for concurrent programming.
  - Gives programmer back some control over when and where they have to worry about interleavings
- It is like using a high-level language instead of assembly code. Whole classes of low-level errors are eliminated.
  - Correct-by-construction design
- Not a silver bullet:
  - you can still write buggy programs;
  - concurrent programs are still harder than sequential ones
  - aimed only at shared memory concurrency, not message passing
- There is a performance hit, but it is usually acceptable in Haskell (and things can only get better as the research community focuses on the question.)

Exploring Haskell
In a little more depth

Haskell vs. OCaml

```
module type MONAD = sig
  type 'a M
  return : 'a -> 'a M
  (>>=) : 'a M -> ('a -> 'b M) -> 'b M
end
```

OCaml

```
val read_file : file_name -> string M

let concat f1 f2 =
  readfile f1          >>= (fun contents1 ->
  readfile f2          >>= (fun contents2 ->
  return (contents1 ^ contents2)
;;
```

```
do          readfile f1
then do     readfile f2
then do     contents1 ^
             contents2
```

Another Haskell Detail:

Haskell function types are pure -- totally effect-free

```
foo : int -> int
```

Haskell's type system *forces** purity on functions with type `a -> b`

- no printing
- no mutable data
- no reading from files
- no concurrency
- no benign effects (like memoization)

* except for a function called unsafePerformIO

Another Haskell Detail:

```
foo :: int -> int
```

totally pure function

```
<code> :: IO int
```

suspended (lazy) computation that performs effects when executed

Another Haskell Detail:

```
foo :: int -> int
```

totally pure function

```
<code> :: IO int
```

suspended (lazy) computation that performs effects when executed

```
bar :: int -> IO int
```

totally pure function that returns suspended effectful computation

Another Haskell Detail:

```
foo :: int -> int
```

totally pure function

```
<code> :: IO int
```

suspended (lazy) computation that performs effects when executed

```
bar :: int -> IO int
```

totally pure function that returns suspended effectful computation

use monad operations to compose suspended computations

all effects in Haskell are treated as a kind of book keeping

IO is the catch-all monad

An Example

```
print :: string -> IO ()
```

the "IO monad"
-- contains effectful computations like printing

```
reverse :: string -> string
```

```
reverse "hello" :: string
```

```
print (reverse "hello") :: IO ()
```

the type system always tells you when an effect has happened – effects can't "escape" the I/O monad

Another Example

```
read :: Ref a -> IO a
```

```
(+) :: int -> int -> int
```

```
r :: Ref int
```

```
(read r) + 3  :: int
```

Doesn't type check

Another Example

```
read :: Ref a -> IO a
```

```
(+) :: int -> int -> int
```

```
r :: Ref int
```

```
(read r) >>= \x ->
x + 3  :: IO int
```

Use Bind to keep the computation in the monad!!

Another Example

```
read :: Ref a -> IO a
```

```
(+) :: int -> int -> int
```

```
r :: Ref int
```

```
do
   x <- read r
   return (x + 3)
```

Prettier!!

Mutable State

```
new   :: a -> IO (Ref a)
read  :: Ref a -> IO a
write :: Ref a -> a -> IO ()
```

Haskell uses new, read, and write* functions within the IO Monad to manage mutable state.

```
main :: IO ()

main = do
           r <- new 0             -- int r := 0
           inc r                  -- r := r+1
           s <- read r            -- s := r;
           print s

inc :: Ref Int -> IO ()
inc r = do
           v <- read r            -- temp = r
           write r (v+1)          -- r = temp+1
```

* actually newRef, readRef, writeRef, …

Haskell vs. OCaml

```
module type MONAD = sig
  type 'a M
  return : 'a -> 'a M
  (>>=) : 'a M -> ('a -> 'b M) -> 'b M
end
```

OCaml

```
val read_file : file_name -> string M

let concat f1 f2 =
  readfile f1            >>= (fun contents1 ->
  readfile f2            >>= (fun contents2 ->
  return (contents1 ^ contents2)
;;
```

```
do           readfile f1
then do      readfile f2
then do      contents1 ^
              contents2
```

the kind of monad is controlled by the type Maybe == option

Haskell

```
concat :: filename -> filename -> Maybe string

concat y z =
  do
     contents1 <- readfile f1
     contents2 <- readfile f2
     return (contents1 ^ contents2)
     .
```

keyword do begins monadic block of code!

syntax is pretty! Compiler automatically translates in to something very similar to the OCaml

In a nutshell

Haskell is already using monads to implement state

It's type system controls where mutation can occur

So now, software transactional memory is just a slightly more sophisticated version of Haskell's existing IO monad.

PS: Scala Monads

Check out James Iry blog:

- http://james-iry.blogspot.com/2007/09/monads-are-elephants-part-1.html + 3 more parts
- he's a hacker and he's using equational reasoning to explain monads!

Main thing to remember:

- bind is called "flatmap" in Scala
- return is called "unit" in Scala
- do notation in Haskell is similar to for notation in Scala

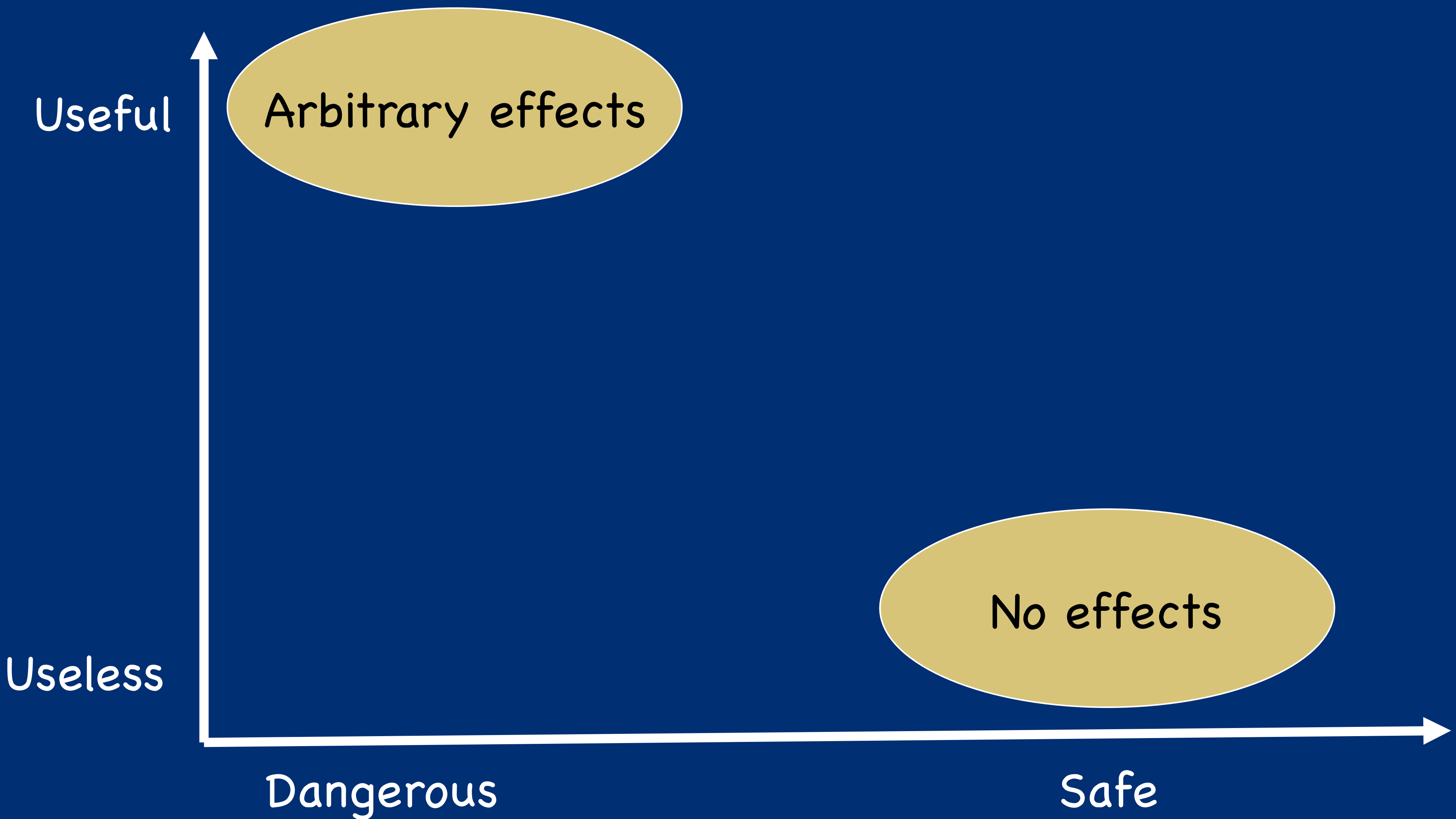
```
for (x <- monad) yield result
== monad >>= (fun x -> return result)
== map (fun x -> result) monad
```

PPS: Check out monads in Python via generators:
http://www.valuedlessons.com/2008/01/monads-in-python-with-nice-syntax.html

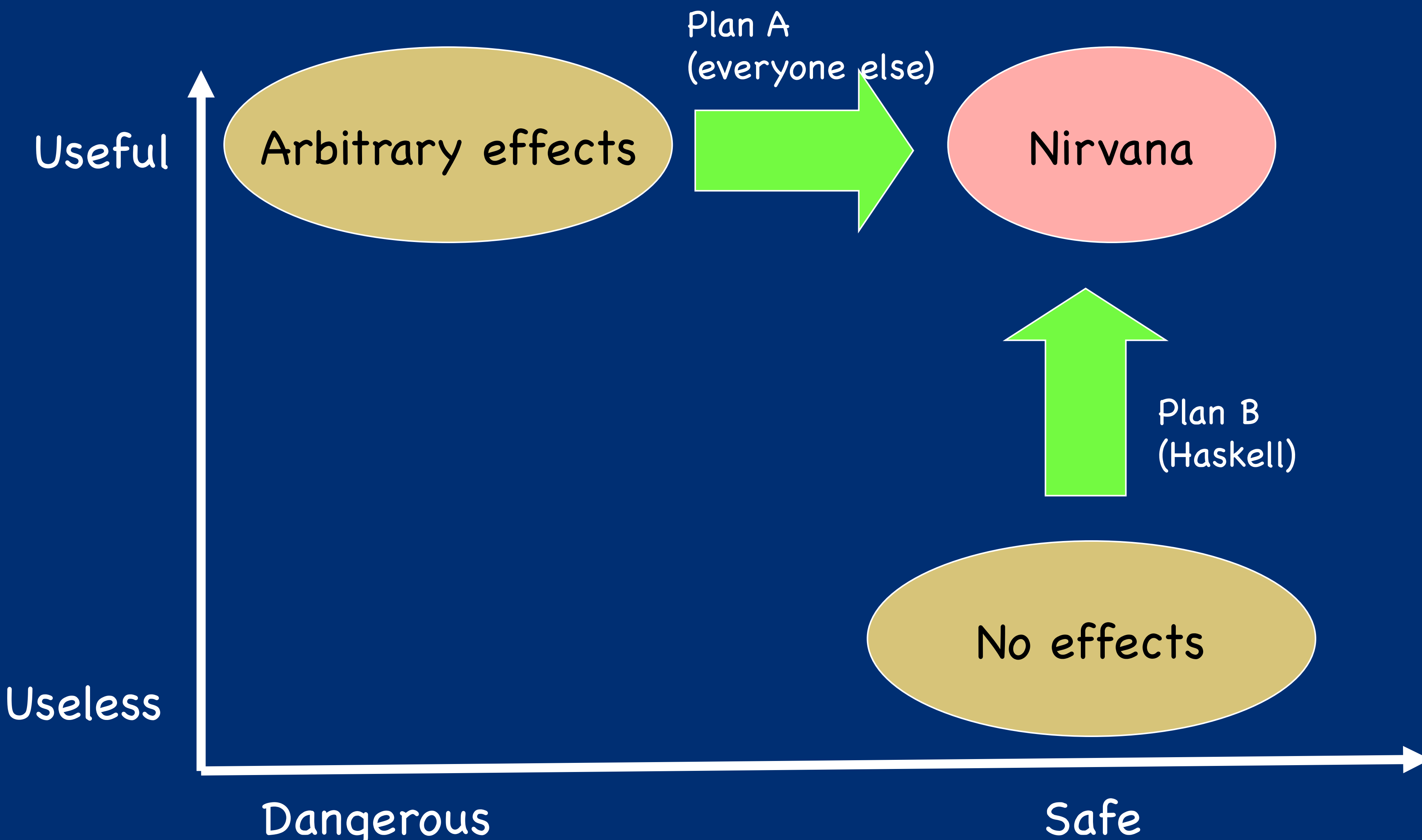
Haskell: A Language with a Monadic Skin

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because IO is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
  - Counter-point: We have shown that it is useful to be able to build pure abstractions using imperative infrastructure (eg: laziness, futures, parallel sequences, memoization). You can't do that in Haskell (without escaping the type system via unsafeIO)
- Interesting perspective: It is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.
- At any rate, a checked pure-impure separation facilitates concurrent programming.

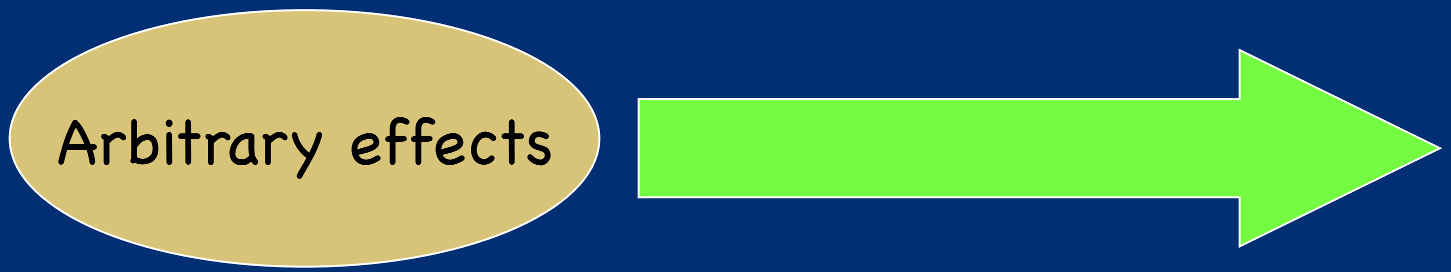
The Central Challenge



The Challenge of Effects



Two Basic Approaches: Plan A



Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone

Default = Any effect
Plan = Add restrictions

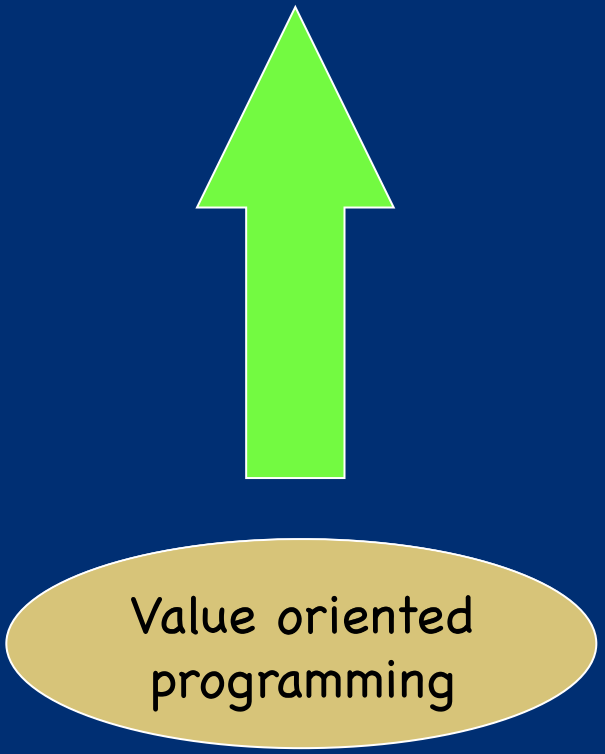
Two Basic Approaches: Plan B

Default = No effects
Plan = Selectively permit effects

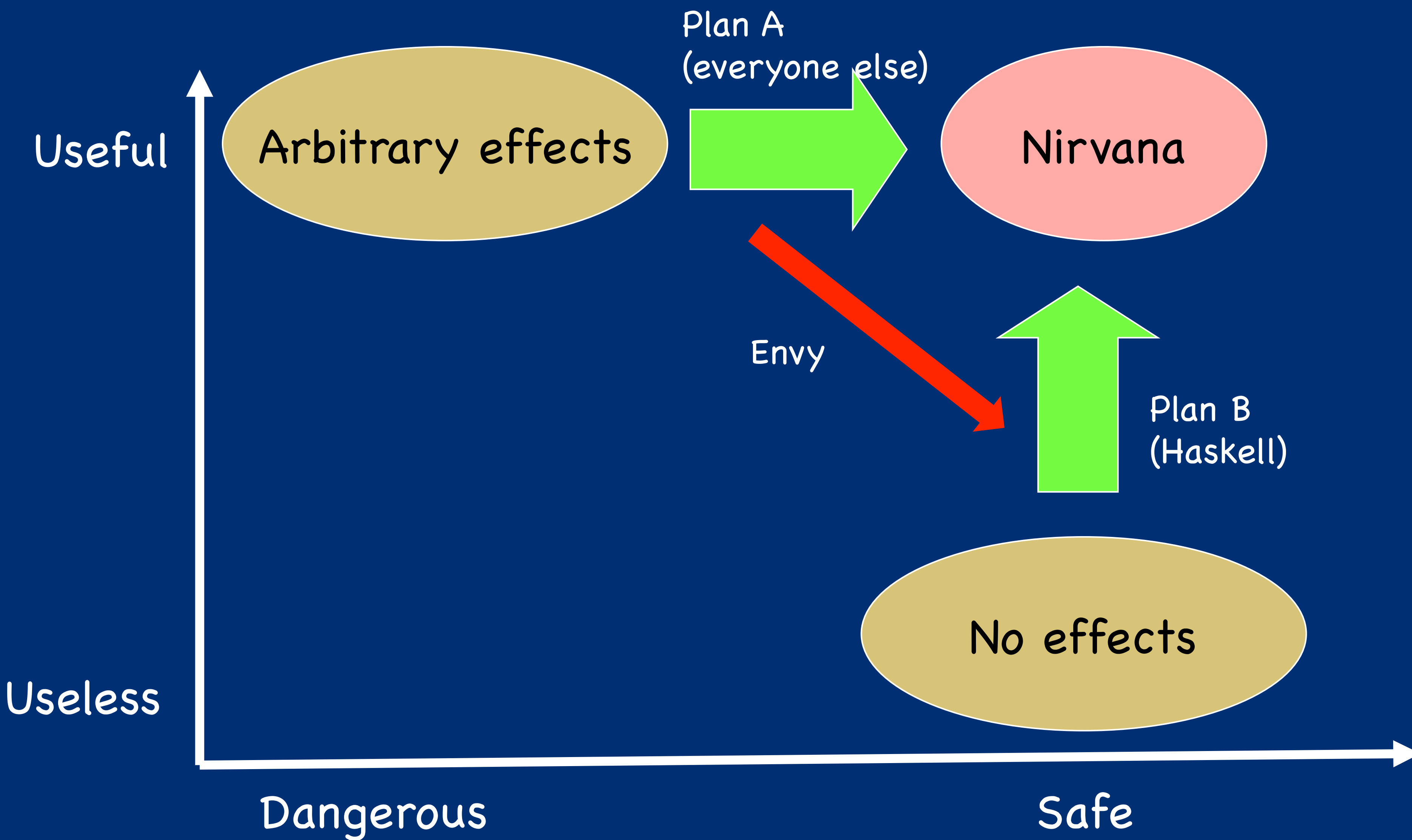
Types play a major role

Two main approaches:

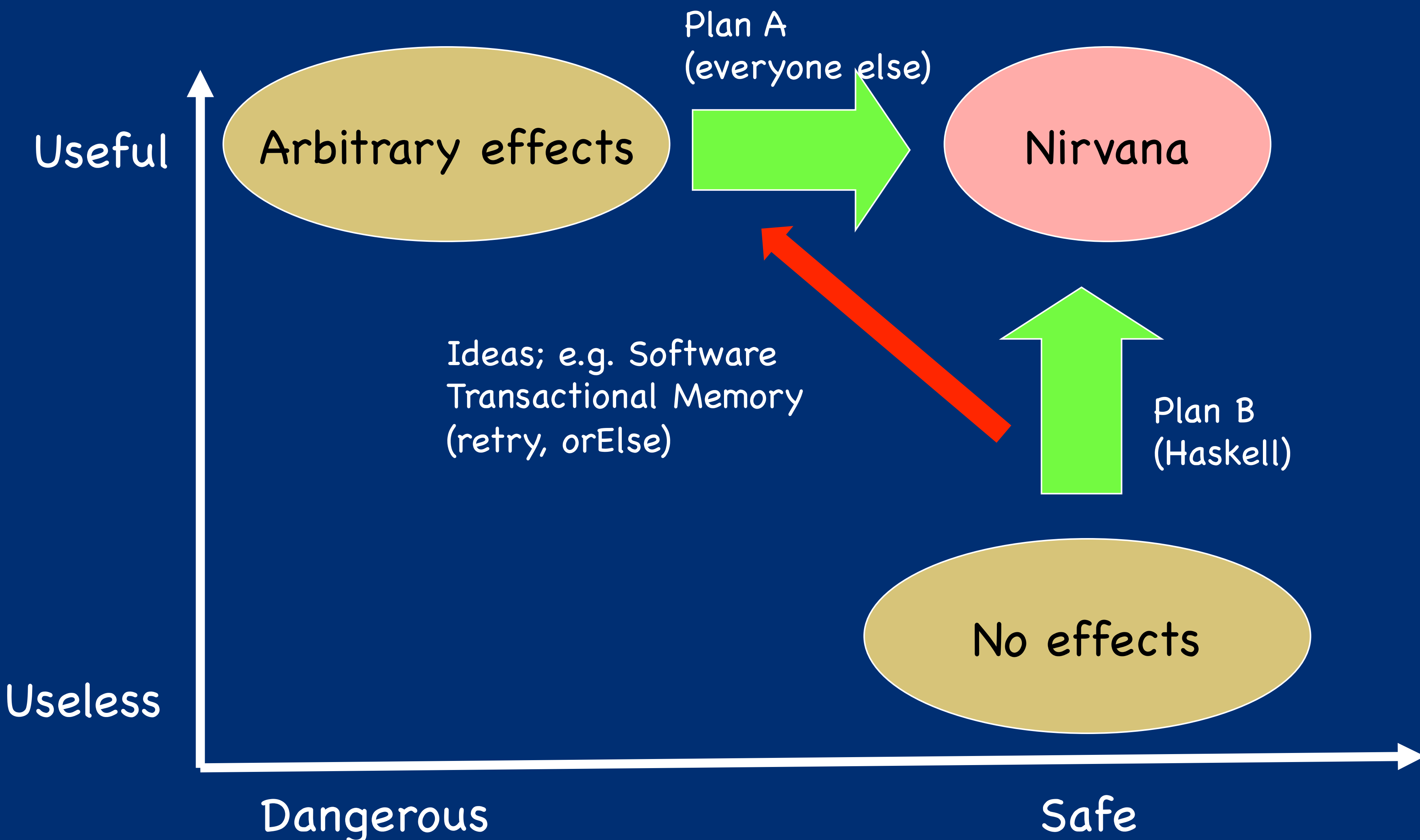
- Domain specific languages (SQL, Xquery, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)



Lots of Cross Over



Lots of Cross Over



An Assessment and a Prediction

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

Take home message: Haskell is cool. Check it out.

End