

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:

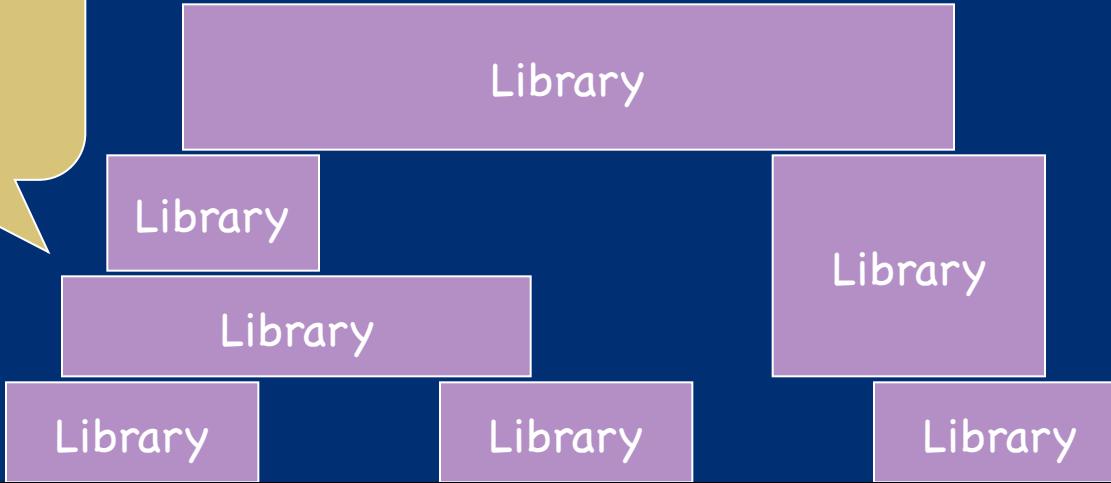
“Beautiful Concurrency”,

“The Transactional Memory / Garbage Collection Analogy”

“A Tutorial on Parallel and Concurrent Programming in Haskell”

What we want

Libraries build
layered
concurrency
abstractions

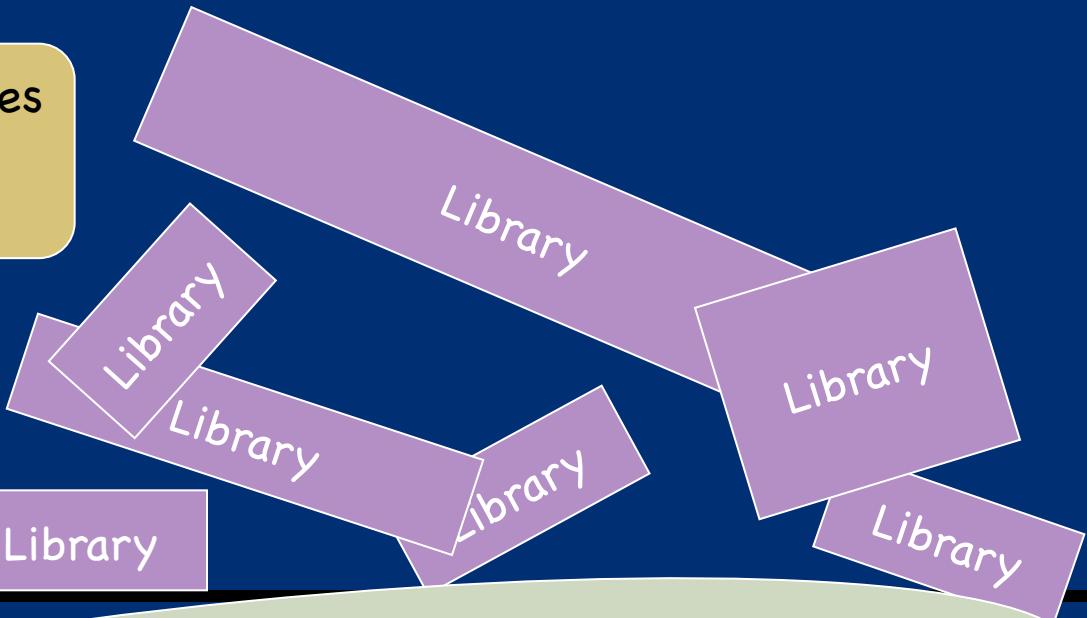


Concurrency primitives

Hardware

What we have using conventional techniques

Locks and condition variables
(a) are hard to use and
(b) do not compose



Locks and condition variables

Hardware

“Building complex parallel programs is like building a sky scraper out of bananas.” -- Simon Peyton Jones

Recall with Locks & Mutable Data, Imperative Parallel Programming is Hard: **1 + 1 ain't always 2!**

```
int x = 0;
```

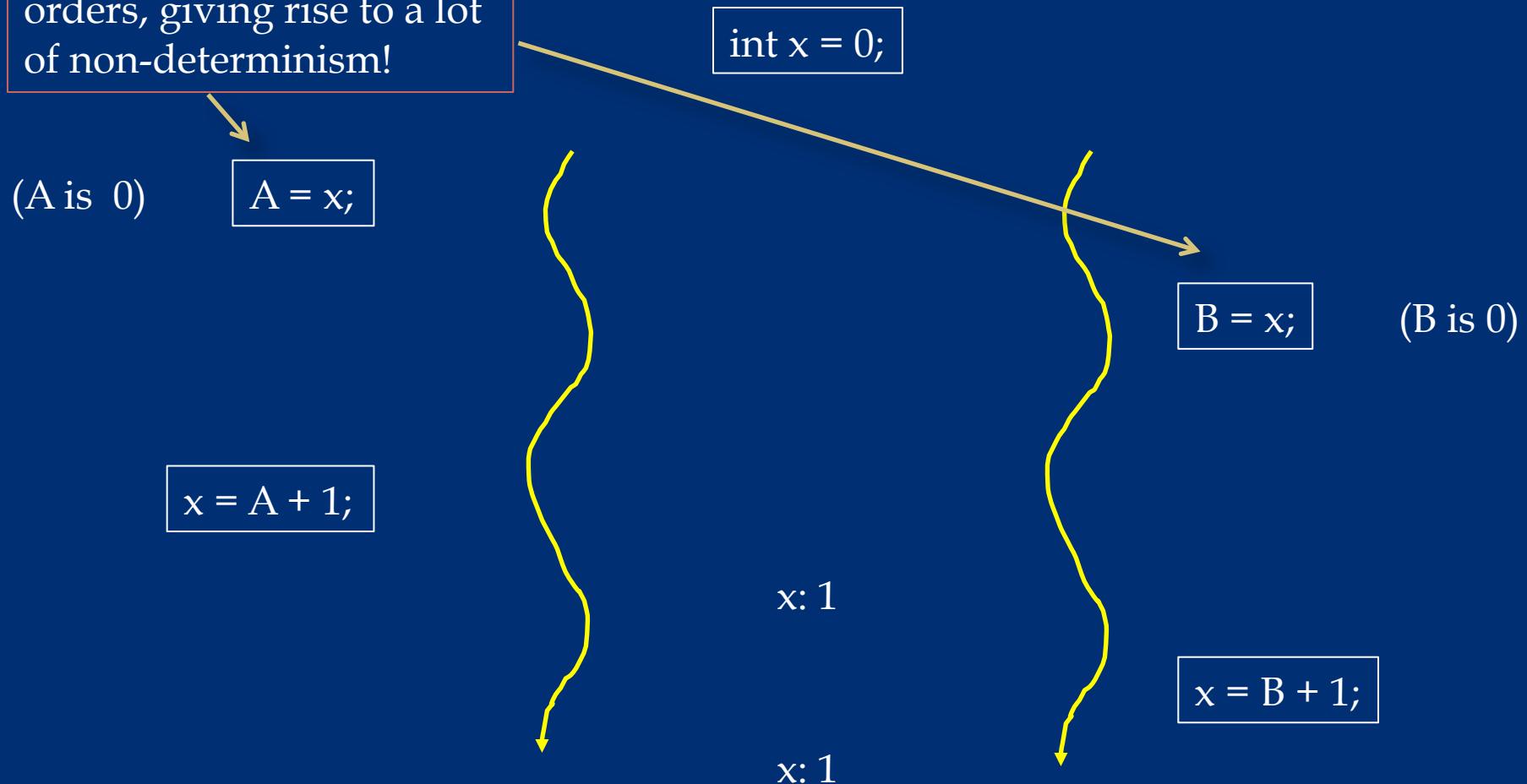
```
A = x;  
x = A + 1
```



```
B = x;  
x = B + 1
```

Recall with Locks & Mutable Data, Imperative Parallel Programming is Hard: **1 + 1 ain't always 2!**

Execution of operations
in threads interleaved
in many different possible
orders, giving rise to a lot
of non-determinism!



Recall: Imperative Parallel Programming is Hard: Locks and Critical Sections

critical
section

```
int x = 0;
```

```
acquire(L);  
A = x;  
x = A + 1;  
release(L);
```

x: 1

```
acquire(L);  
B = x;  
x = B + a;  
release(L);
```

x: 2



Recall: Why Imperative Parallel Programming is Hard: **Synchronized Methods**

acquires and releases the lock
associated with the object (1 lock per object)

Java Synchronized Methods:

```
class Adder{  
    int x;  
  
    synchronized void add() {  
        x = x+1;  
    }  
}
```

```
Adder a = new Adder(0);
```

```
a.add()
```

x: 1

```
a.add()
```

x: 2

What's wrong with locks?

Correct use of locks can solve concurrency problems, but locks are amazingly difficult to use correctly

- **Races**: forgotten locks (or synchronization commands) lead to inconsistent views
- **Deadlock**: locks acquired in “wrong” order
- **Lost wakeups**: forget to notify condition variables
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers. Yikes!
- These are serious problems. But even worse...

Locks are Non-Compositional

Consider a (correct) Java bank **Account** class:

```
class Account{
    float balance;

    synchronized void deposit(float amt) {
        balance += amt;
    }

    synchronized void withdraw(float amt) {
        if (balance < amt)
            throw new OutOfMoneyError();
        balance -= amt;
    }
}
```

Now suppose we want to add the ability to transfer funds from one account to another.

Locks are Non-Compositional

Simply calling `withdraw` and `deposit` to implement `transfer` causes a race condition:

```
class Account{  
    float balance;  
  
    ...  
    void badTransfer(Acct other, float amt) {  
  
        other.withdraw(amt); ←  
  
        this.deposit(amt); ←  
    }  
}
```

but clients can see a bad total balance value in between withdraw and deposit

Main point: you still have to think about all possible interleavings ==> *THIS IS TOO HARD!*

Locks are Non-Compositional

- Synchronizing transfer can cause deadlock:

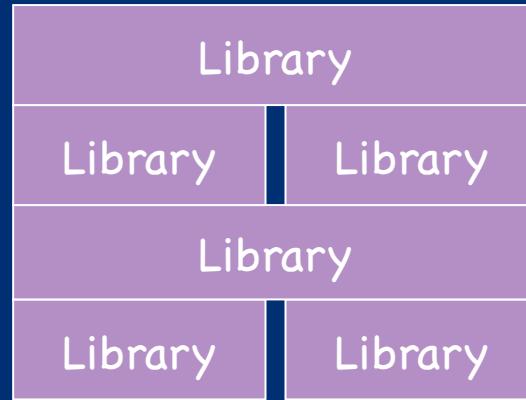
```
class Account{  
    float balance;  
    synchronized void deposit(float amt) {  
        balance += amt;  
    }  
    synchronized void withdraw(float amt) {  
        if(balance < amt)  
            throw new OutOfMoneyError();  
        balance -= amt;  
    }  
    synchronized void badTrans(Acct other, float amt) {  
        // can deadlock with parallel reverse-transfer  
        this.deposit(amt);  
        other.withdraw(amt);  
    }  
}
```

First Idea: Don't Use Mutable Data/ Effects

Good:

if you can build it:
no race conditions,
no deadlock,
interleavings don't matter,
it's deterministic,
it's equivalent to sequential code

it looks pretty to boot



Problem: You can't interact with the world.

The world changes (mutates). We can't stop it.

Threads can't talk back and forth.

Immutable Data

Hardware

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

Atomic blocks

Hardware

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

The Punchline for STM

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate (COS 226)
Efficient parallel code with locks and condition variables	Publishable result at international conference ¹
Parallel code with STM	<i>Undergraduate</i>

¹

Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.

in parallel:

(atomic action1)

(atomic action2)



action 1:

read x
write x
read x
write x

action 2:

read x
write x
read x
write x

just a function call in Haskell

in parallel:

(atomic action1)

(atomic action2)

just a function call in Haskell

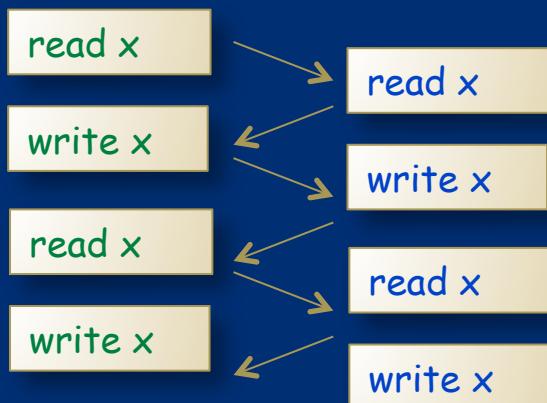
action 1:

action 2:

read x
write x
read x
write x

read x
write x
read x
write x

without atomic transactions:



(some interleaving -- the programmer must worry about which one)

in parallel:

(atomic action1)

(atomic action2)

just a function call in Haskell

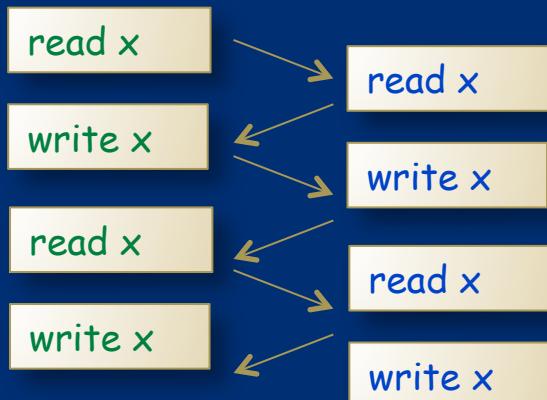
action 1:

action 2:

read x
write x
read x
write x

read x
write x
read x
write x

without atomic transactions:



with transactions:

read x
write x
read x
write x

or

(programmer gets to cut down non-determinism
as much as he/she wants)

(some interleaving -- the programmer must worry about which one)

in parallel:

(atomic action1)

(atomic action2)

just a function call in Haskell

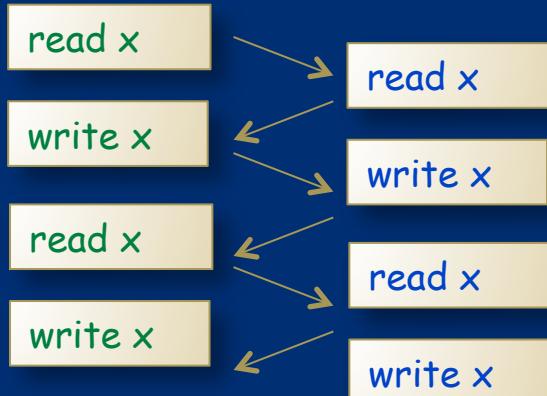
action 1:

action 2:

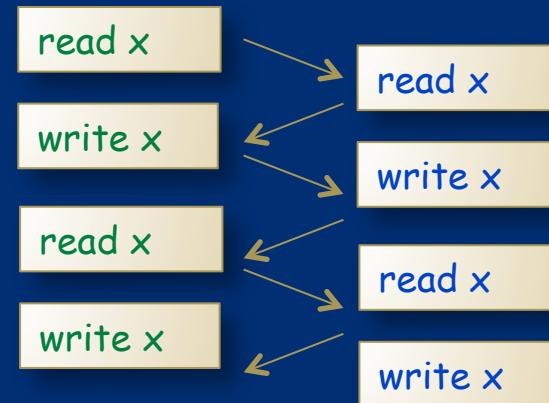
read x
write x
read x
write x

read x
write x
read x
write x

without atomic transactions:



but the actual implementation
with transactions:



(plus validation to ensure it *appears* as though
either action1 or action2 went first in its
entirety)

(some interleaving -- the programmer must worry about which one)

STM = Atomic Memory Transactions

Like database
transactions

```
atomic { ... sequential code ... }
```

- To a first approximation, just write the sequential code, and wrap **atomic** around it
- All-or-nothing semantics: **Atomic** commit
- Atomic block executes in **Isolation**
 - with automatic retry if another conflicting atomic block interferes
- Cannot deadlock (there are no locks!)
 - guarantees about progress on retry
- Atomicity makes error recovery easy (e.g. throw exception inside sequential code)

How do you implement it?

read y;
read z;
write 10 x;
write 42 z;
...

```
atomic { ... <code> ... }
```

One possibility:

- Execute the code *optimistically* without taking any locks.
- Log each read and write to a thread-local transaction log.
 - note: *book keeping*
- Writes go to the log only, not to memory.
- At the end, the transaction validates the log:
 - Are the values I read the same now as when I read them?
 - If valid, *atomically commits changes* to memory.
 - If not valid, *re-run from the beginning*, discarding changes.

Why STM in Haskell (or OCaml)?

- Logging memory effects is *expensive*.
 - Haskell already partitions the world:
 - immutable values (zillions and zillions)
 - mutable locations (some or none)
 - *Only the mutable must be logged!*
 - **Already paid the bill:** Simply reading or writing a mutable location in Haskell is *expensive* so transactions don't add much more overhead
 - (you can read that as a bad thing about Haskell)
 - **Monad infrastructure:** Ideal for the book keeping needed to build logs
- Functional
programmers brutally
trained from birth to
use memory effects
sparingly.

A Detour: Haskell and Monads

Back to Basics: What's an interface?

An interface declares some new abstract types and some operations over values with those abstract types. For example:

```
module type CONTAINER = sig
  type 'a t           (* the type of the container *)
  val empty : 'a t
  val insert : 'a -> 'a t -> 'a t
  val remove : 'a t -> 'a option * 'a t
  val fold : ('a -> 'b -> 'b) -> 'b -> 'a t -> 'b
end
```

There are lots of different implementations of such containers: queues, stacks, sets, randomized sets, ...

Interfaces can come with some *equations* one expects every implementation to satisfy. eg:

`fold f base empty == base`

The equations specify some, but not all of the behavior of the module (eg: stacks and queues remove elements in different orders)

Monads

A *monad* is just a particular *interface*. Two views:

- interface for a **very generic container**, with operations designed to support **composition** of computations over the contents of containers
- interface for an **abstract computation** that does some “book keeping” on the side. Book keeping is code for “has an effect”. Once again, the support for composition is key.
- since functional programmers know that functions are data, the two views actually coincide

Many different kinds of monads:

- monads for handling/accumulating errors (last week)
- monads for processing collections en masse
- monads for logging strings that should be printed
- monads for coordinating concurrent threads (Jane St. Talk)
- monads for back-tracking search
- monads for *transactional memory*

Because a monad is just a particular interface (with many useful implementations), *you can implement monads in any language*

- But, Haskell is famous for them because it has a special built-in syntax that makes monads particularly easy and elegant to use
- F#, Scala have adopted similar syntactic ideas
- Monads also play a very special role in the overall design of the Haskell language

What is the monad interface?

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

+ some equations specifying
how return and bind are
required to interact

Consider first the “container interpretation”:

- ' $'a M$ is a container for values with type ' a
- $\text{return } x$ puts x in the container
- $\text{bind } c \ f$ takes the values in c out of the container and applies f to them, forming a new container holding the results
 - $\text{bind } c \ f$ is often written as: $c \gg= f$

The Options as a Container

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

```
module OptionMonad = struct
```

```
  type 'a M = 'a option
```

```
  let return x = Some x
```

```
  let (=>) c f =  
    match c with  
      None -> None  
    | Some v -> f v
```

```
end
```

put value in
a container

take value v out
of a container c
and then apply f,
producing a new container

The Options as a Container

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

using the option container:

```
type file_name = string  
  
val read_file : file_name -> string M  
  
let concat f1 f2 =  
  readfile f1      =>= (fun contents1 ->  
  readfile f2      =>= (fun contents2 ->  
  return (contents1 ^ contents2)  
  ::
```

```
module OptionMonad = struct  
  
  type 'a M = 'a option  
  
  let return x = Some x  
  
  let (=>) c f =  
    match c with  
      None -> None  
    | Some v -> f v  
  
end
```

put value in a container

take value v out of a container c and then apply f, producing a new container

The Option Monad as Possibly Erroneous Computation

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

using the error monad:

```
type file_name = string  
  
val read_file : file_name -> string M  
  
let concat f1 f2 =  
  readfile f1      =>= (fun contents1 ->  
  readfile f2      =>= (fun contents2 ->  
  return (contents1 ^ contents2)  
  ::
```

```
module ErrorMonad = struct
```

```
  type 'a M = 'a option
```

```
  let return x = Some x
```

```
  let (=>) c f =  
    match c with  
      None -> None  
    | Some v -> f v
```

```
end
```

check to see if
error has occurred,
if so return None,
else continue

setting up
book keeping
for error
processing

Lists as Containers

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

using the list monad:

```
random_sample : unit -> int M  
monte_carlo : int -> int -> int -> result  
  
let experiments : result M =  
  random_sample() => (fun s1 ->  
    random_sample() => (fun s2 ->  
      random_sample() => (fun s3 ->  
        return (monte_carlo s1 s2 s3)  
      ))
```

```
module ListMonad = struct
```

```
  type 'a M = 'a list
```

```
  let return x = [x]
```

```
  let (=>) c f =  
    List.flatten (List.map f c)
```

```
end
```

apply f to all elements
of the list c, creating a
list of lists and then
flatten results in to
single list

put element
in to list
container

The List Monad as Nondeterministic Computation

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

using the non-determinism monad:

```
random_sample : unit -> int M  
monte_carlo : int -> int -> int -> result  
  
let experiments : result M =  
  random_sample() => (fun s1 ->  
    random_sample() => (fun s2 ->  
      random_sample() => (fun s3 ->  
        return (monte_carlo s1 s2 s3)  
      ))  
  :::
```

```
module ListMonad = struct
```

```
  type 'a M = 'a list
```

```
  let return x = [x]
```

```
  let (=>) c f =  
    List.flatten (List.map f c)
```

```
end
```

one result;
no non-determinism
compose many
possible results (c)
with a non-deterministic
continuation f

A Container with a String on the Side (aka: A logging/printing monad)

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

using the logging monad:

```
record : ('a -> 'b) -> 'a -> string -> 'b M
let record f x s = (f x, s)
let do x =
  record read x "read it" =>= (fun v ->
    record write v "wrote it" =>= (fun _ ->
      record write v "wrote it again" =>= (fun _ ->
        return v
      ::))
```

```
module LoggingMonad = struct
  type 'a M = 'a * string
  let return x = (x, "")
  let (=>) c f =
    let (v, s) = c in
    let (v', s') = f v in
    (v', s ^ s')
end
```

concatenate the log of c with the log produced by running f

nothing logged yet

Monad Laws

Just like one expects any CONTAINER to behave in a particular way,
one has expectations of MONADs.

Left identity: “return does nothing observable”

$$(1) \text{ return } v \gg= f == f v$$

Right identity: “return still doesn’t do anything observable”

$$(2) m \gg= \text{return} == m$$

Associativity: “composing m with f first and then doing g is the same as
doing m with the composition of f and g ”

$$(3) (m \gg= f) \gg= g == m \gg= (\text{fun } x \rightarrow f x \gg= g)$$

Breaking the Law

Just like one expects any CONTAINER to behave in a particular way,
one has expectations of MONADs.

Left identity: “return does nothing observable”

(1) $\text{return } v \gg= f \quad == \quad f \ v$

```
module LoggingMonad = struct
  type 'a M = 'a * string
  let return x = (x, "start")
  let (gg=) c f =
    let (v, s) = c in
    let (v', s') = f v in
    (v', s ^ s')
end
```

```
return 3 >>= fun x -> return x
== (3, "start") >>= fun x -> return x
== (3, "start" ^ "start")
== (3, "startstart")
```

```
(fun x -> return x) 3
== return 3
== (3, "start")
```

Breaking the Law

What are the consequences of breaking the law?

Well, if you told your friend you've implemented a monad and they can use it in your code, they will expect that they can rewrite their code using equations like this one:

```
return x >>= f == f x
```

If you tell your friend you've implemented the monad interface but none of the monad laws hold your friend will probably say: Ok, tell me what your functions do then and please stop using the word monad because it is confusing. It is like you are claiming to have implemented the QUEUE interface but insert and remove are First-In, First-Out like a stack.

In Haskell or Fsharp or Scala, breaking the monad laws may have more severe consequences, because the compiler actually uses those laws to do some transformations of your code.

Monads in Haskell

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

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

Another Haskell Detail:

Haskell function types are pure -- totally effect-free

foo : int -> int



Haskell's type system *forces** purity on functions with type $a \rightarrow b$

- 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

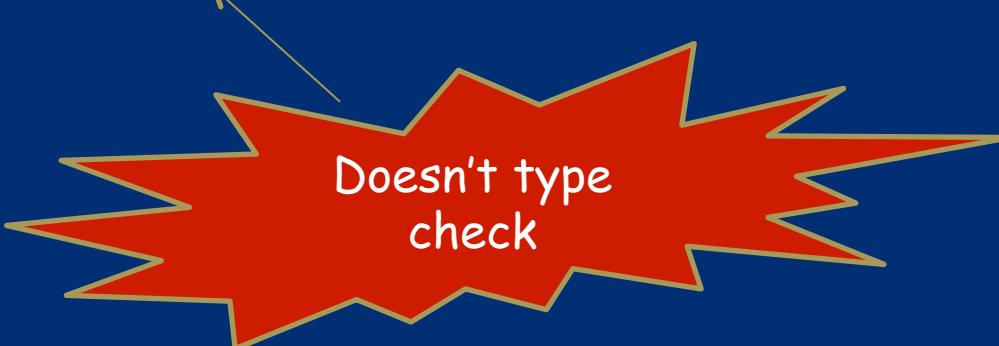
Another Example

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 = 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, ...

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>

Back to 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  
    ...
```

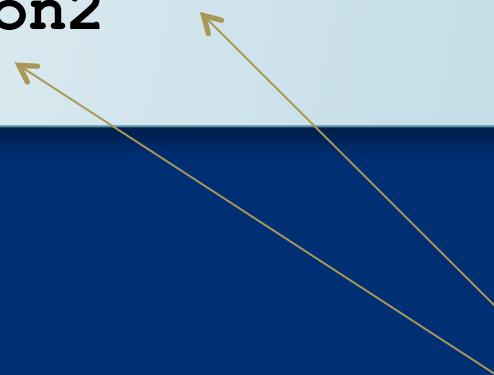


action 1 and
action 2 in
parallel

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
    ...

```

action 1:

```

read x
write x
read x
write x

```

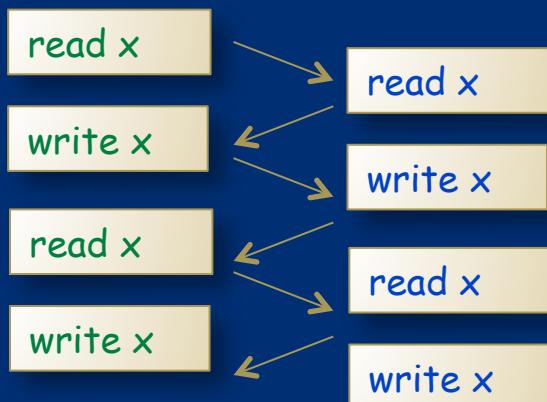
action 2:

```

read x
write x
read x
write x

```

without transactions:



with transactions:

```

read x
write x
read x
write x

```

```

read x
write x
read x
write x

```

```

read x
write x
read x
write x

```

```

read x
write x
read x
write x

```

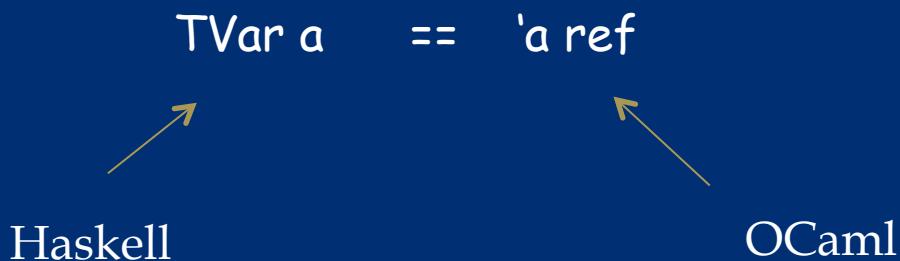
or

(programmer gets to cut down non-determinism
as much as he/she wants)

(some interleaving -- the programmer must worry about which one)

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.



<code>atomic</code>	<code>:: STM a -> IO a</code>
<code>new</code>	<code>:: a -> STM (TVar a)</code>
<code>read</code>	<code>:: TVar a -> STM a</code>
<code>write</code>	<code>:: TVar a -> a -> STM ()</code>

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

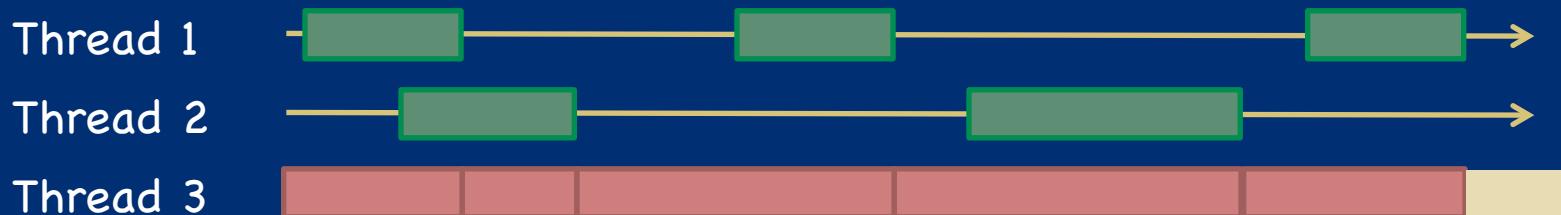
- The STM monad supports exceptions:

```
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#(TVar#(Int), Int) 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:

$a \text{ `orElse`} (b \text{ `orElse`} c) == (a \text{ `orElse`} b) \text{ `orElse`} s$

$\text{retry } \text{ `orElse`} s == s$

$s \text{ `orElse`} \text{ retry } == s$

(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 }
```

An arbitrary boolean
valued STM computation

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

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		$P; \Theta \xrightarrow{\cdot} Q; \Theta'$
$\mathbb{P}[\text{putChar } c]; \Theta$	$\xrightarrow{!c}$	$\mathbb{P}[\text{return } ()]; \Theta$ (PUTC)
$\mathbb{P}[\text{getChar}]; \Theta$	$\xrightarrow{?c}$	$\mathbb{P}[\text{return } c]; \Theta$ (GETC)
$\mathbb{P}[\text{forkIO } M]; \Phi, \Delta$	\rightarrow	$(\mathbb{P}[\text{return } t] \mid M_t); \Phi, \Delta \cup \{t\} \quad t \notin \Delta$ (FORK)
$\frac{M \rightarrow N}{\mathbb{P}[M]; \Theta \rightarrow \mathbb{P}[N]; \Theta}$		(ADMIN)
$M; \Theta \xrightarrow{\cdot} \text{return } N; \Theta'$		$\mathbb{P}[\text{atomically } M]; \Theta \rightarrow \mathbb{P}[\text{return } N]; \Theta'$ (ARET)
		$M; \Phi, \Delta \xrightarrow{\cdot} \text{throw } N; \Phi, \Delta'$
		$\mathbb{P}[\text{atomically } M]; \Phi, \Delta \rightarrow \mathbb{P}[\text{throw } N]; \Phi, \Delta'$ (ATHROW)
Administrative transitions		$M \rightarrow N$
$M \rightarrow V$		if $E[M] = V$ and $M \neq V$ (EVAL)
$\text{return } N >= M \rightarrow MN$		(BIND)
$\text{throw } N >= M \rightarrow \text{throw } N$		(THROW)
$\text{catch } (\text{throw } M) N \rightarrow NM$		(CATCH1)
$\text{catch } (\text{return } M) N \rightarrow \text{return } M$		(CATCH2)
STM transitions		$M; \Theta \Rightarrow N; \Theta'$
$E[\text{readTVar } r]; \Phi, \Delta \Rightarrow E[\text{return } \Phi(r)]; \Phi, \Delta$		if $r \in \text{dom}(\Phi)$ (READ)
$E[\text{writeTVar } r N]; \Phi, \Delta \Rightarrow E[\text{return } ()]; \Phi[r \mapsto M], \Delta$		if $r \in \text{dom}(\Phi)$ (WRITE)
$E[\text{newTVar } M]; \Phi, \Delta \Rightarrow E[\text{return } r]; \Phi[r \mapsto M], \Delta \cup \{r\}$		if $r \notin \Delta$ (NEW)
$\frac{M \rightarrow N}{E[M]; \Theta \rightarrow E[N]; \Theta}$		(ADMIN)
$E[M_1]; \Theta \xrightarrow{\cdot} E[\text{return } N]; \Theta'$		$E[M_1]; \Theta \xrightarrow{\cdot} E[\text{throw } N]; \Theta'$
$E[M_1 \text{ `orElse' } M_2]; \Theta \Rightarrow E[\text{return } N]; \Theta'$	(OR1)	$E[M_1]; \Theta \xrightarrow{\cdot} E[\text{throw } N]; \Theta'$
		$E[M_1 \text{ `orElse' } M_2]; \Theta \Rightarrow E[\text{throw } N]; \Theta'$ (OR2)
$E[M_1]; \Theta \xrightarrow{\cdot} E[\text{retry}]; \Theta'$		$E[M_1 \text{ `orElse' } M_2]; \Theta \Rightarrow E[M_2]; \Theta$ (OR3)

See “[Composable Memory Transactions](#)” for details.

Take COS 510 to understand what it means!

Haskell Implementation

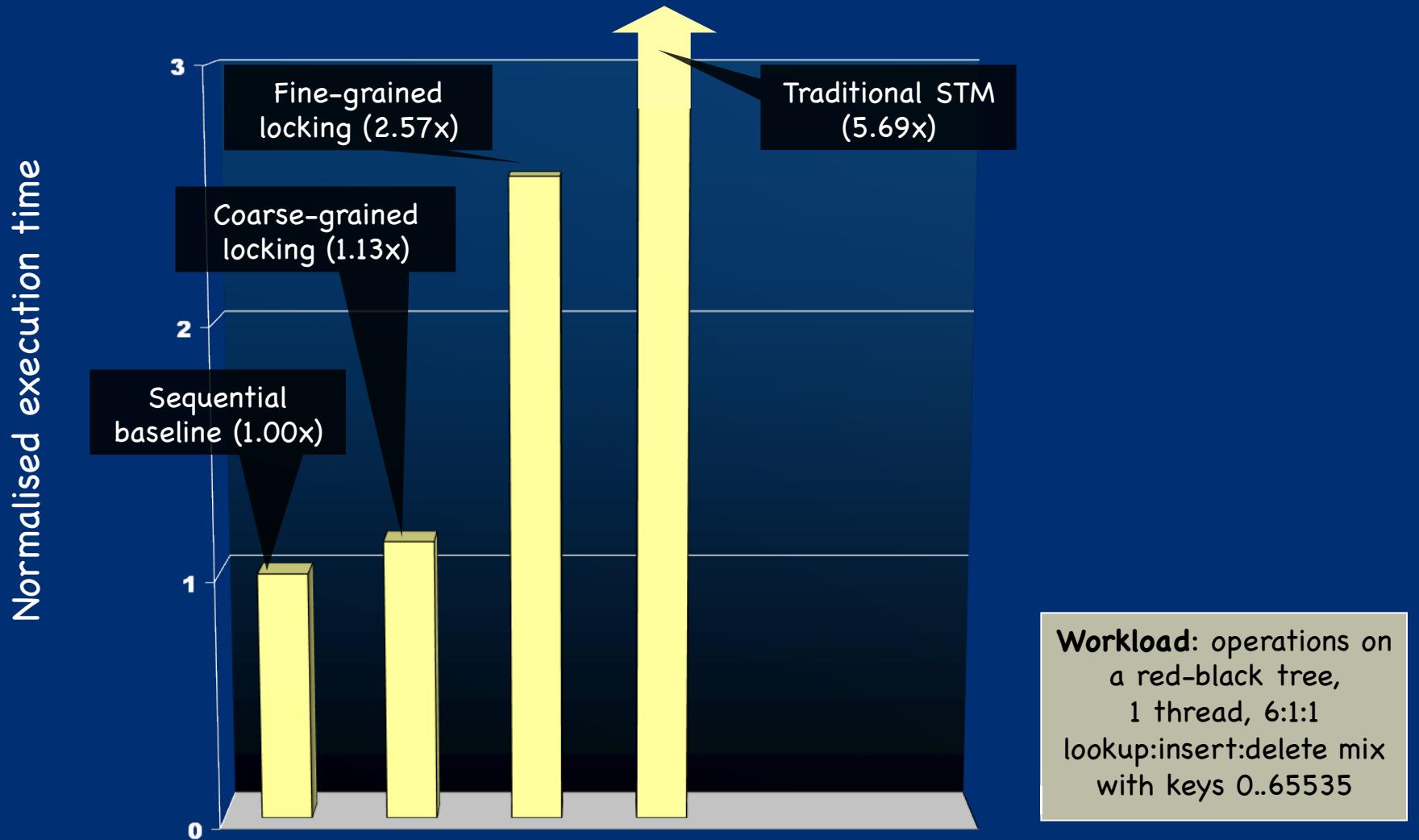
Haskell Implementation

- A complete, multiprocessor implementation of STM exists as of GHC 6.
- **Experience to date:** even for the most mutation-intensive program, the Haskell STM implementation is as fast as the previous MVar implementation.
 - The MVar version paid heavy costs for (usually unused) exception handlers.
- Need more experience using STM in practice, though!
- You can play with it.

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

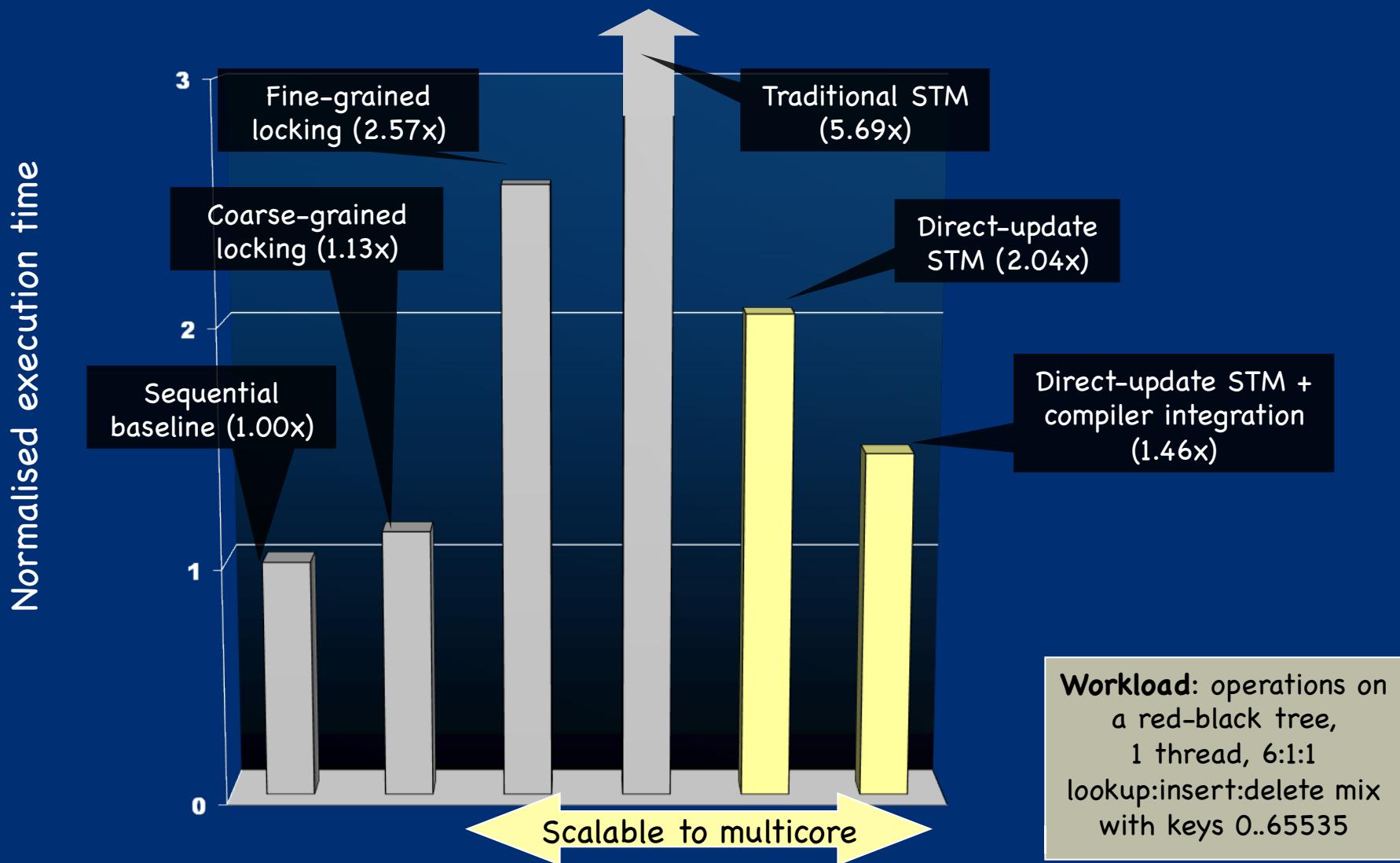


See "[Optimizing Memory Transactions](#)" for more information.

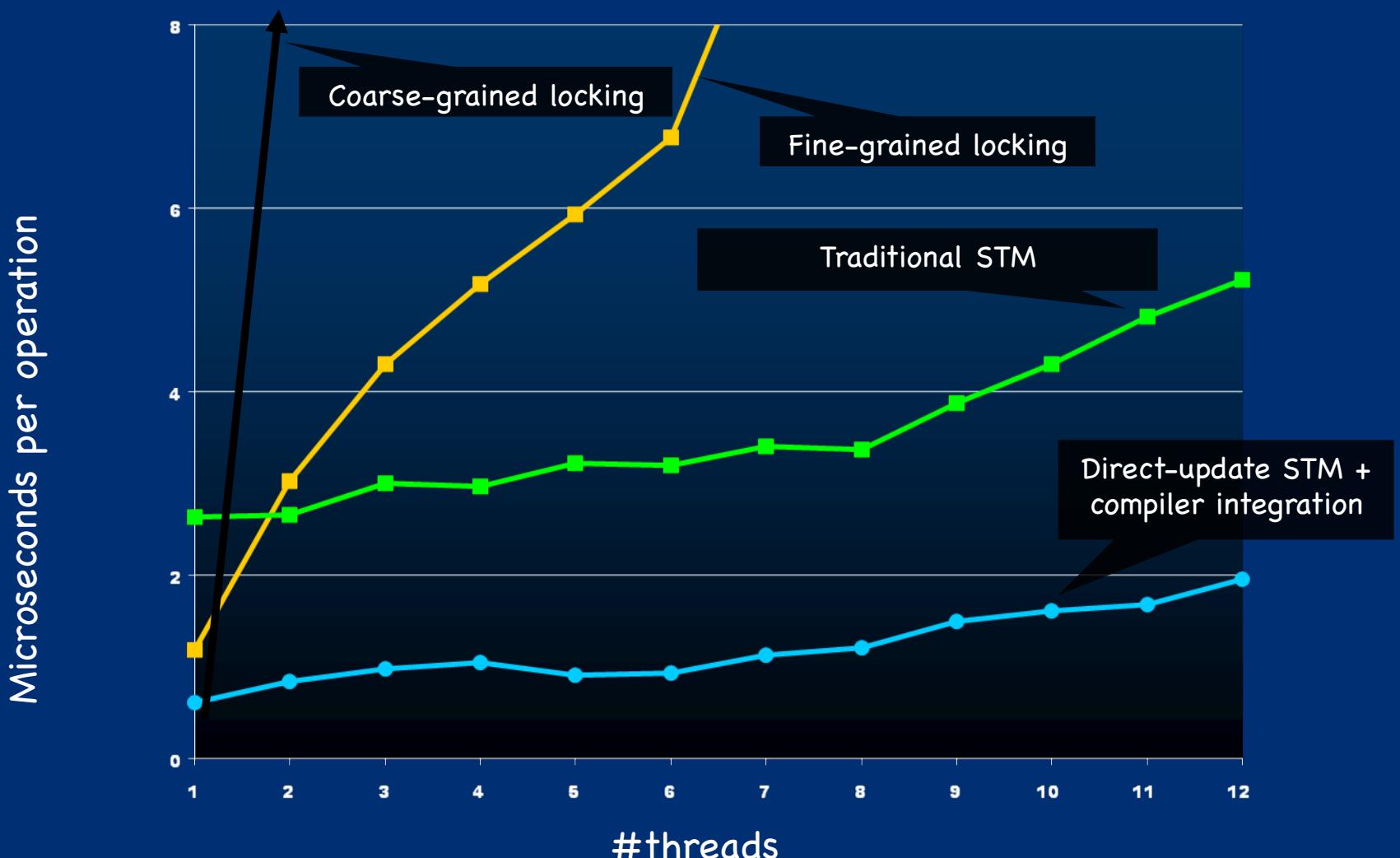
New Implementation Techniques

- **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



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

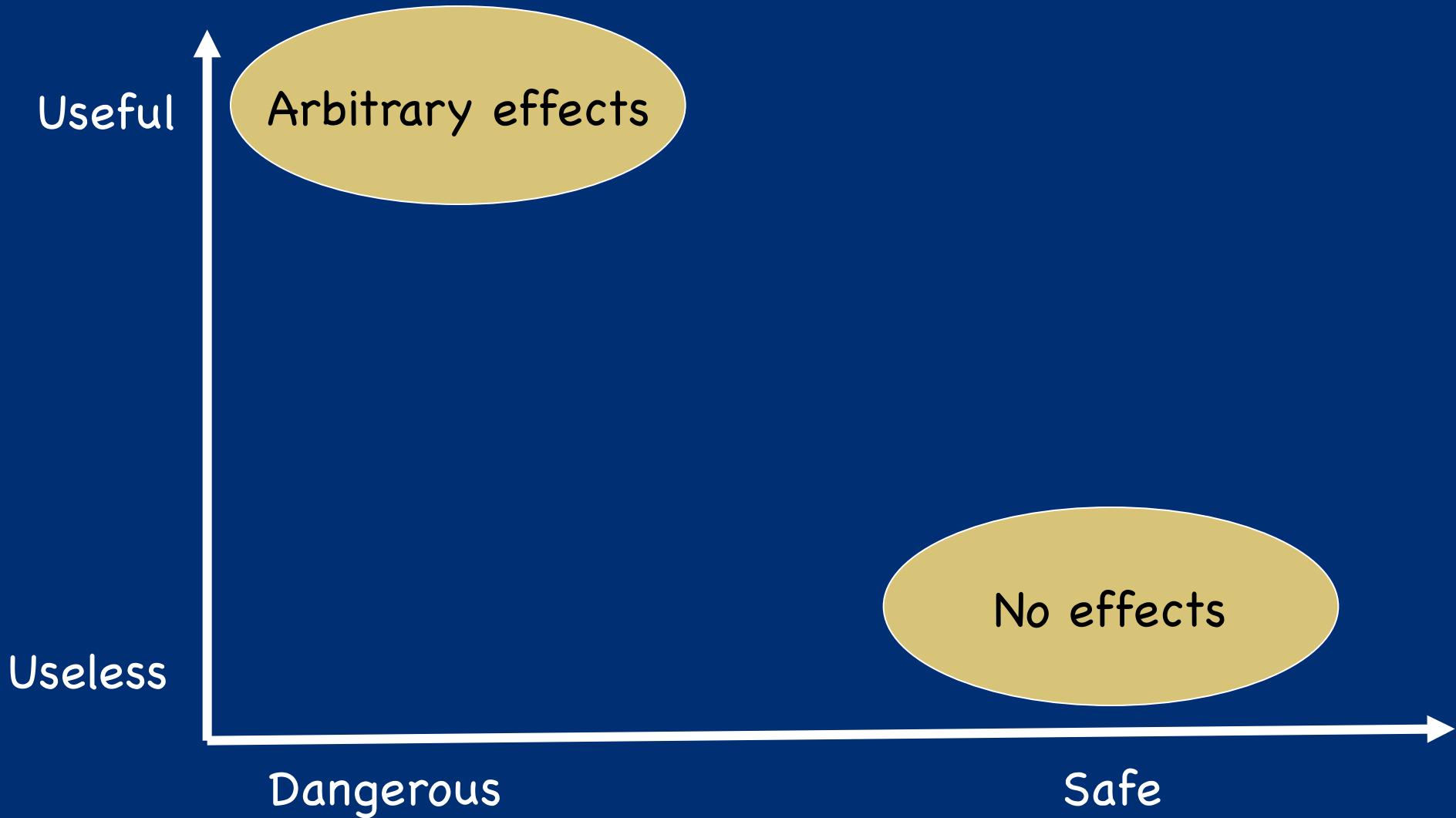
- Atomic blocks (`atomic`, `retry`, `orElse`) dramatically raise the level of abstraction for concurrent programming.
 - Gives programmer back some control over when and where they have to worry about interleavings
- It is like using a high-level language instead of assembly code. Whole classes of low-level 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.)

Haskell Wrapup

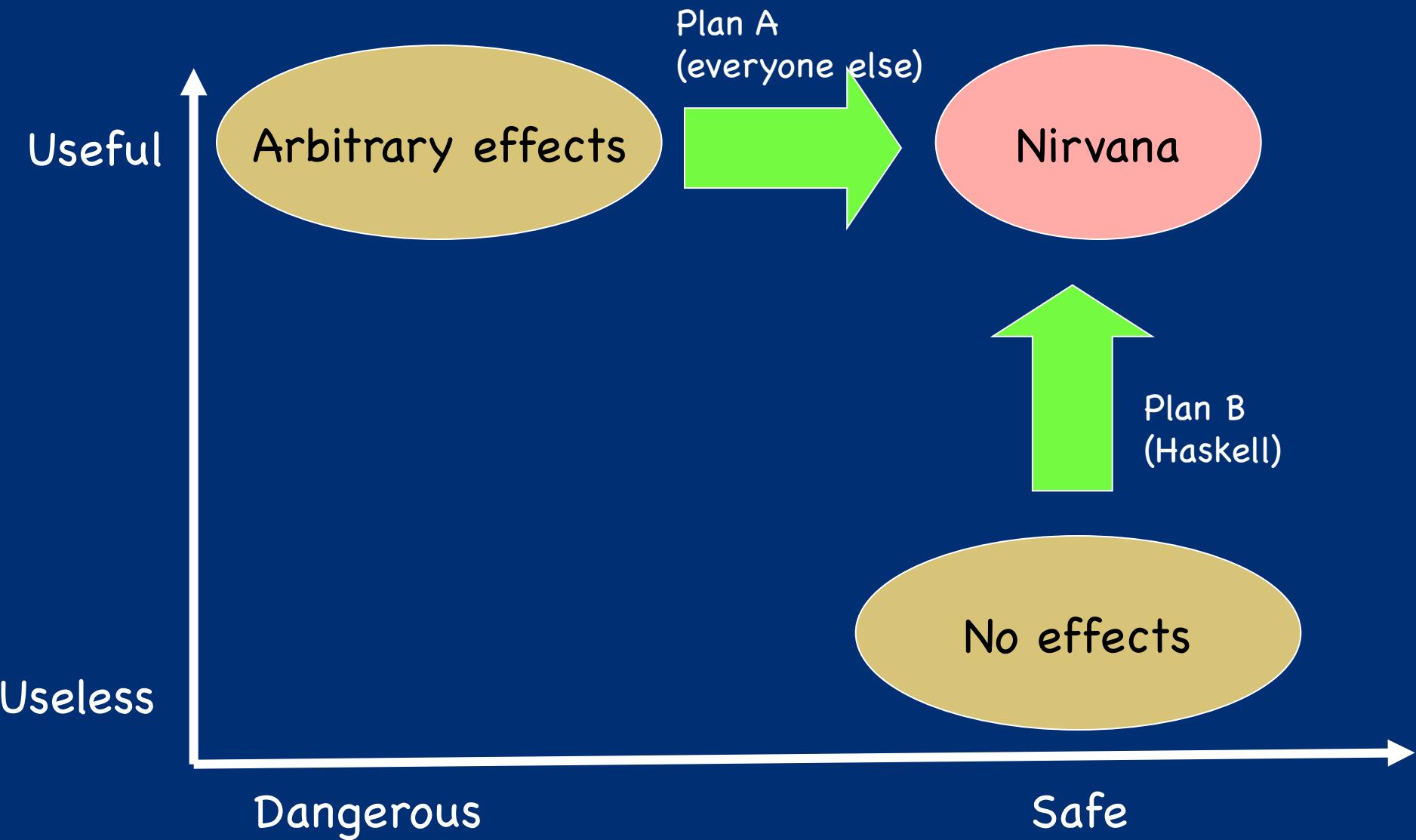
A Monadic Skin

- 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

Arbitrary effects



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

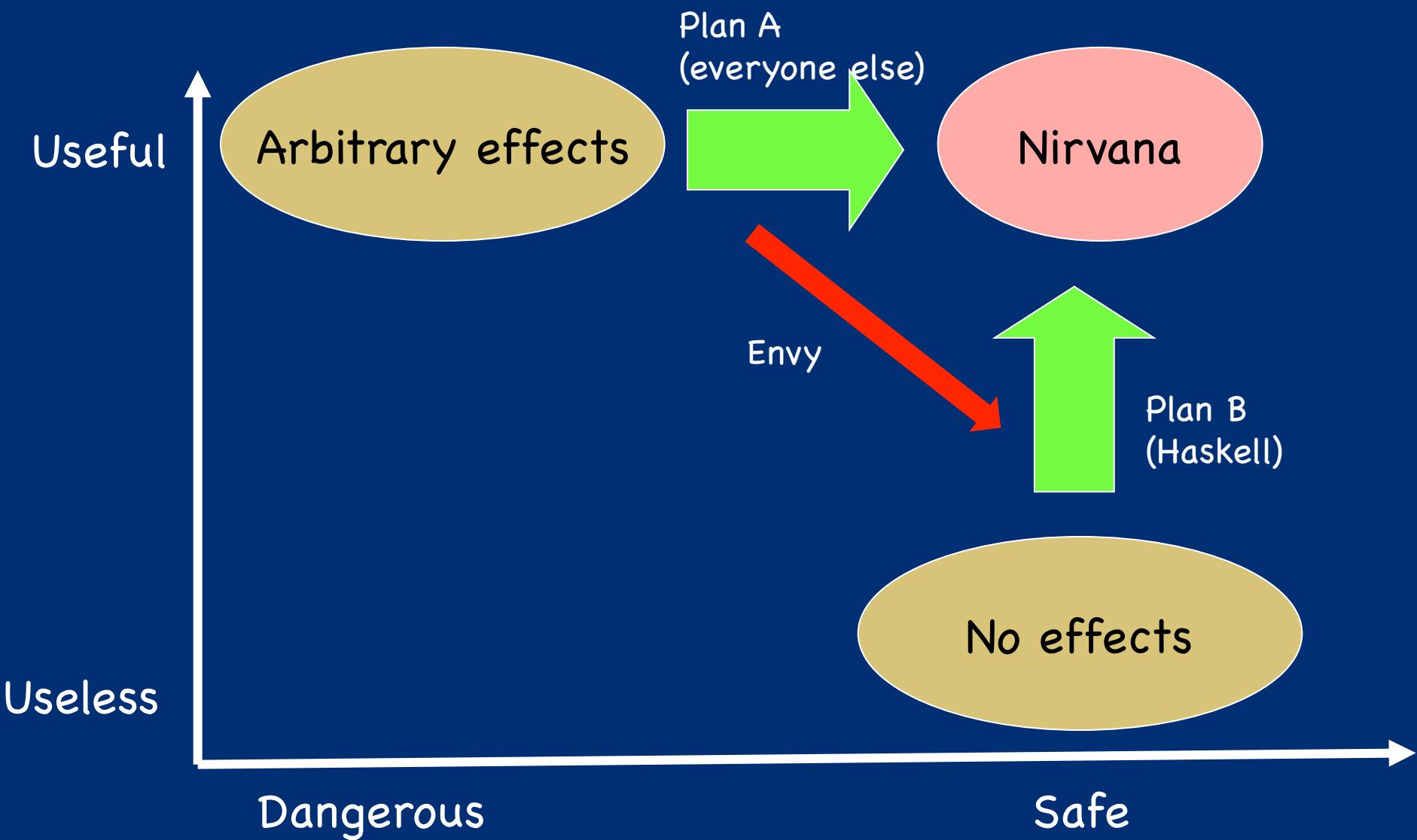
Two main approaches:

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

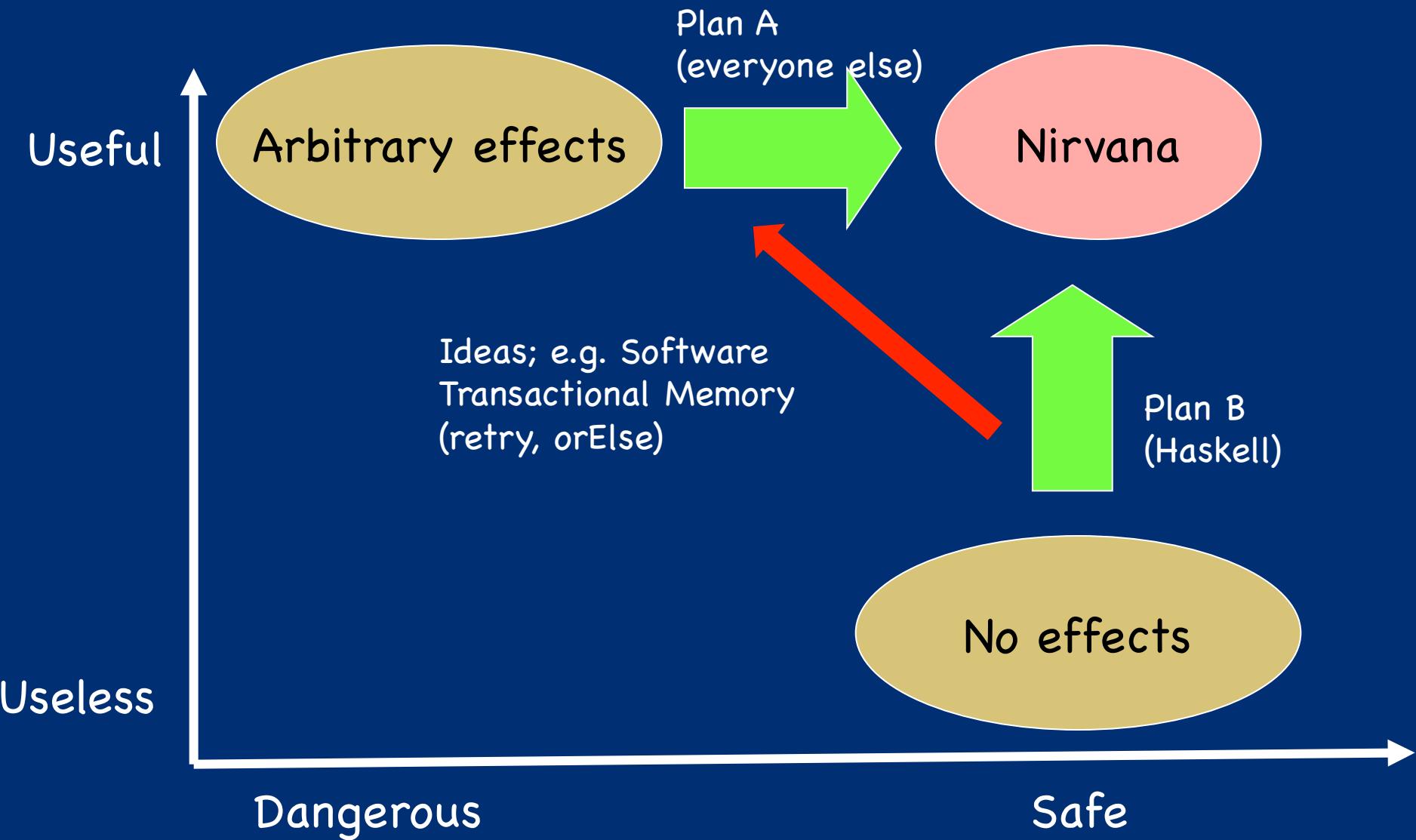


Value oriented
programming

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

End