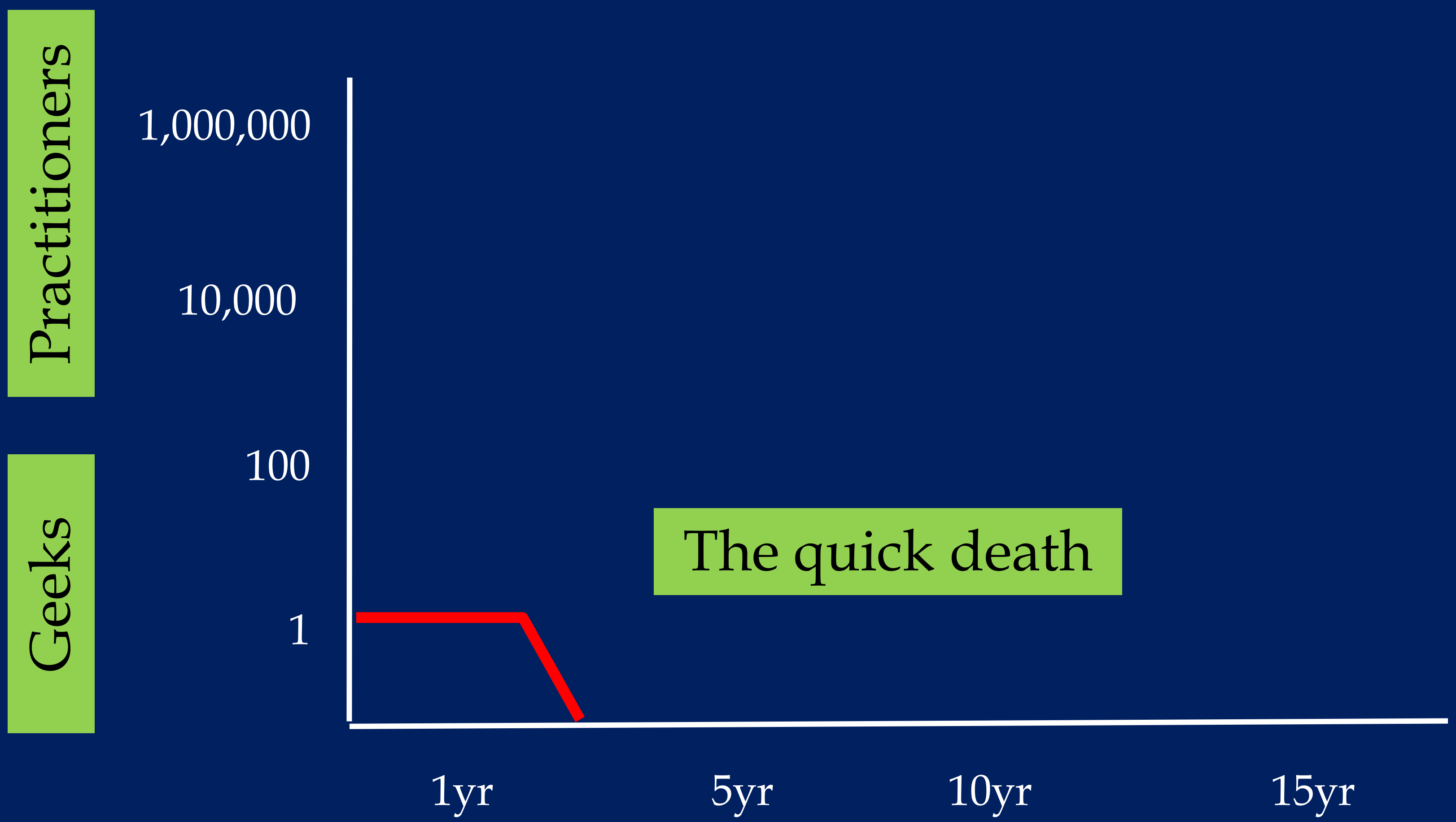
# HASKELL AND TRANSACTIONAL MEMORY

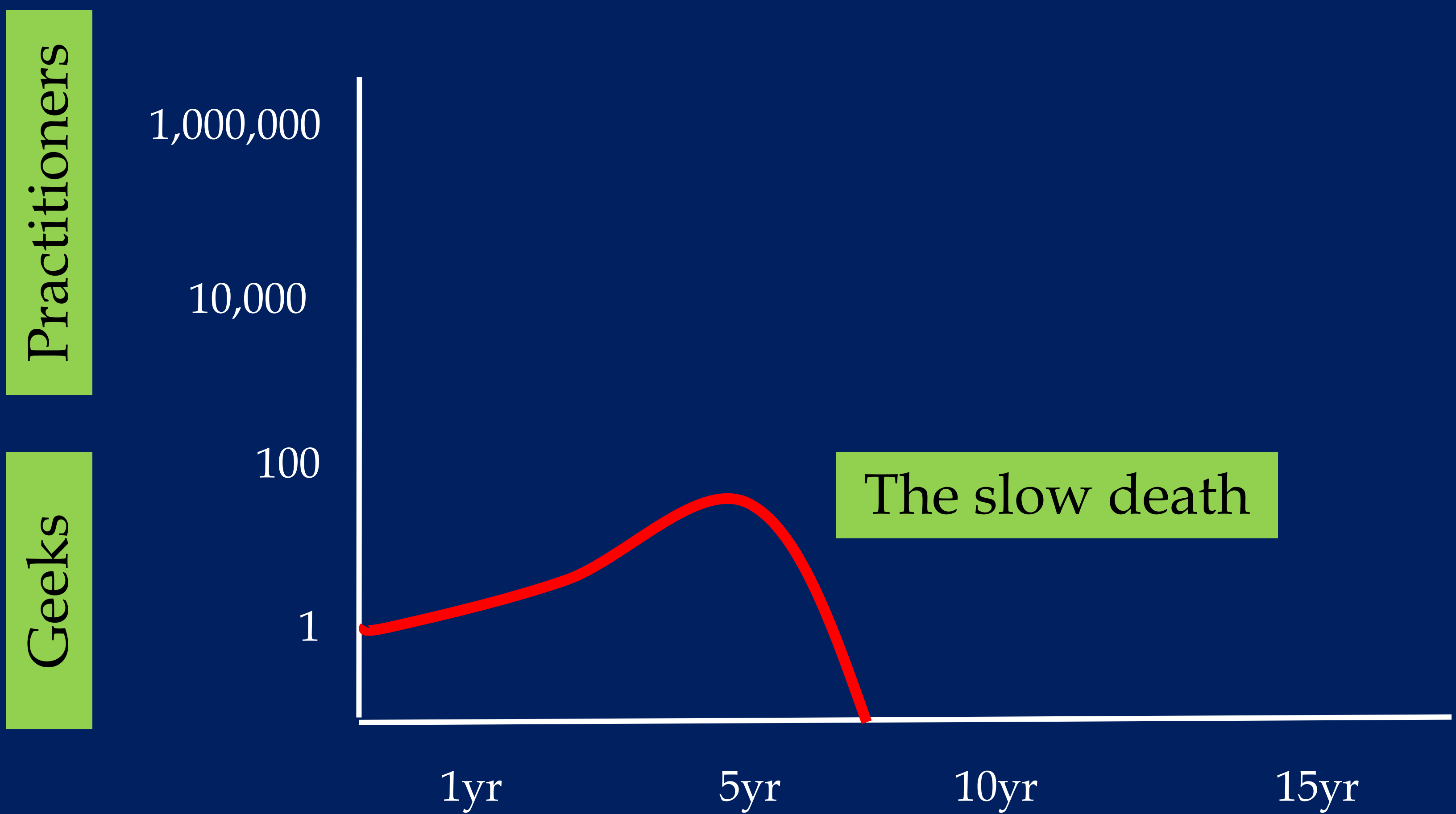
Simon Peyton Jones (Microsoft Research)

Tokyo Haskell Users Group
April 2010

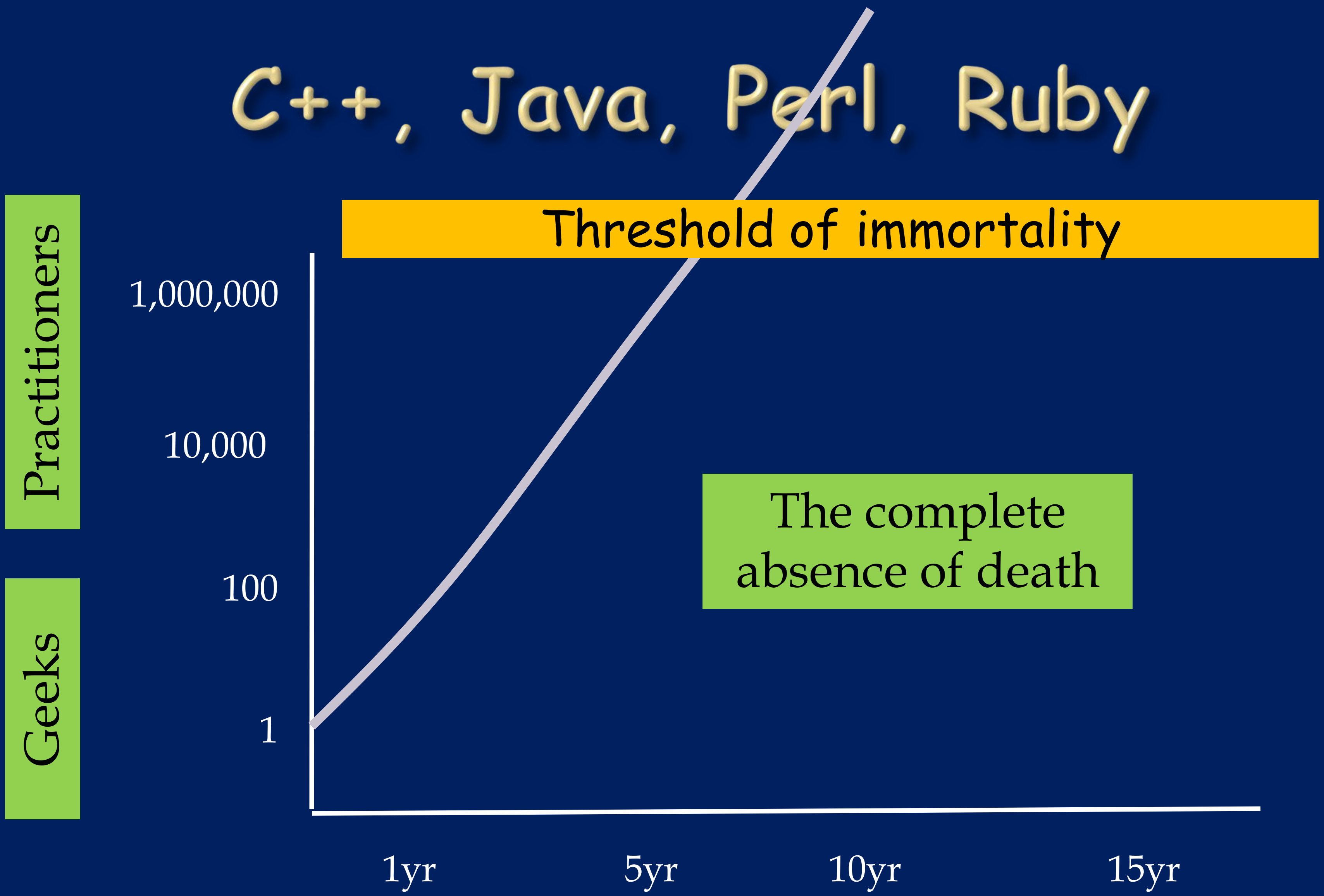
# Most new programming languages



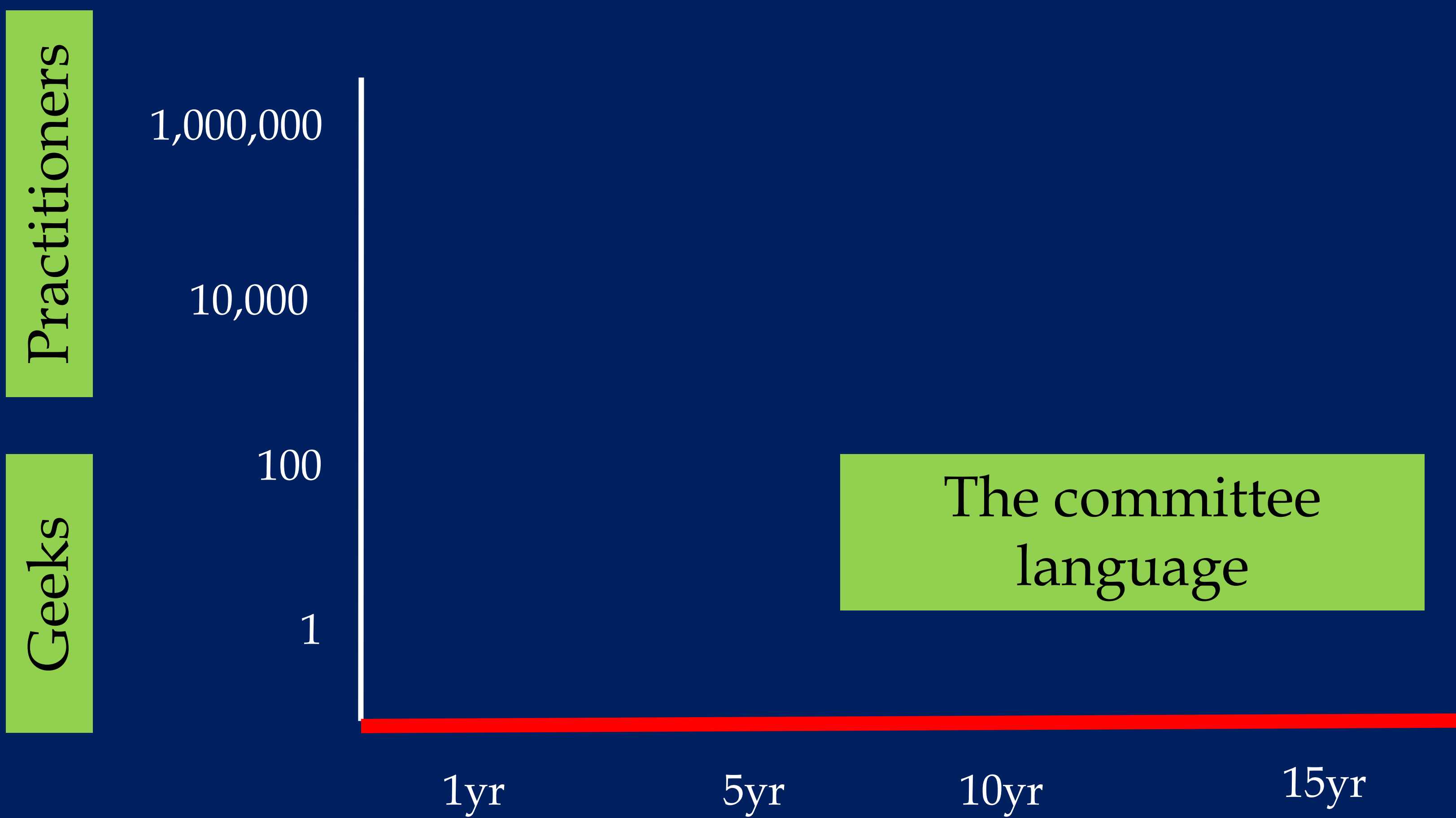
# Successful research languages



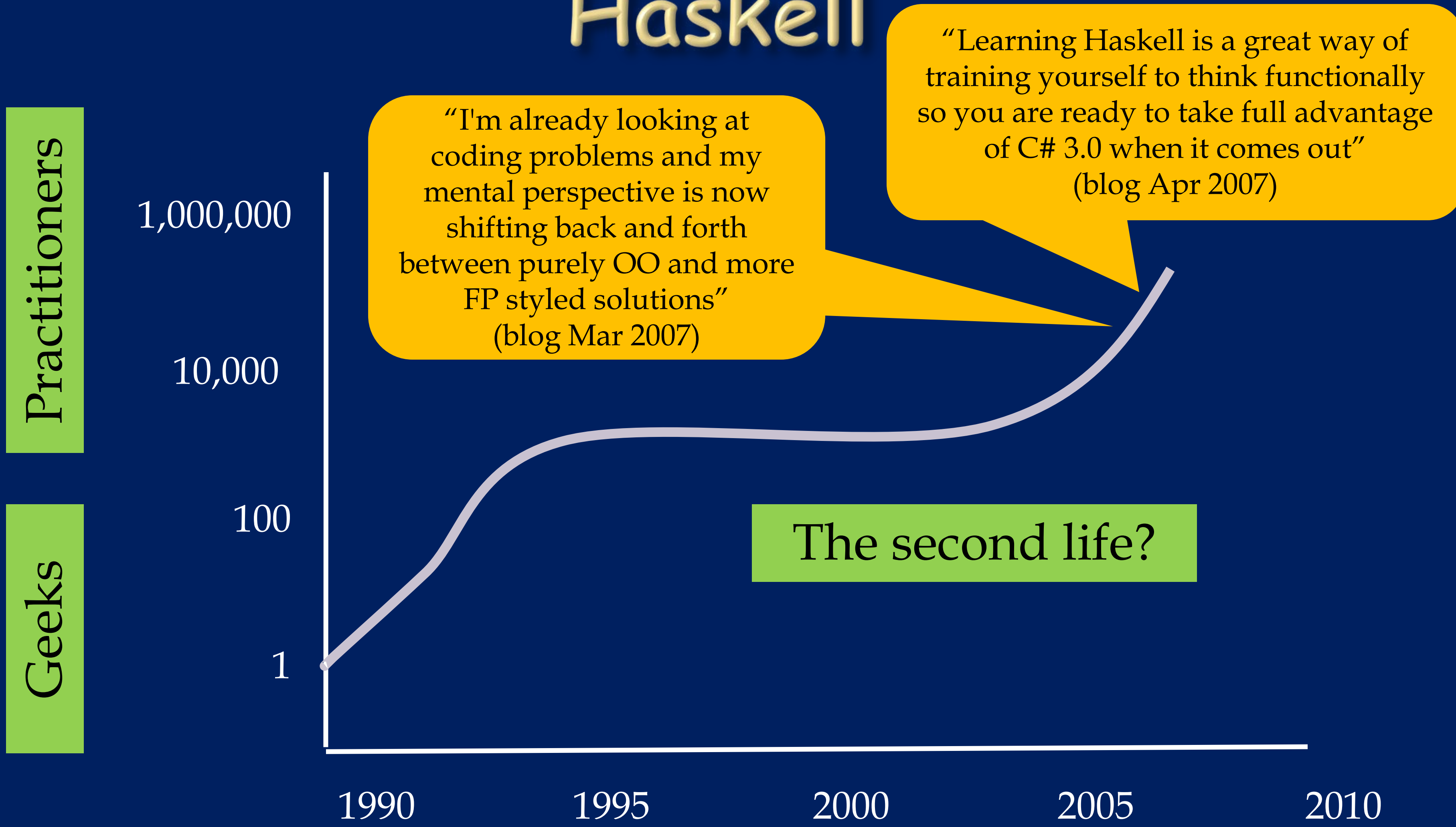
# C++, Java, Perl, Ruby



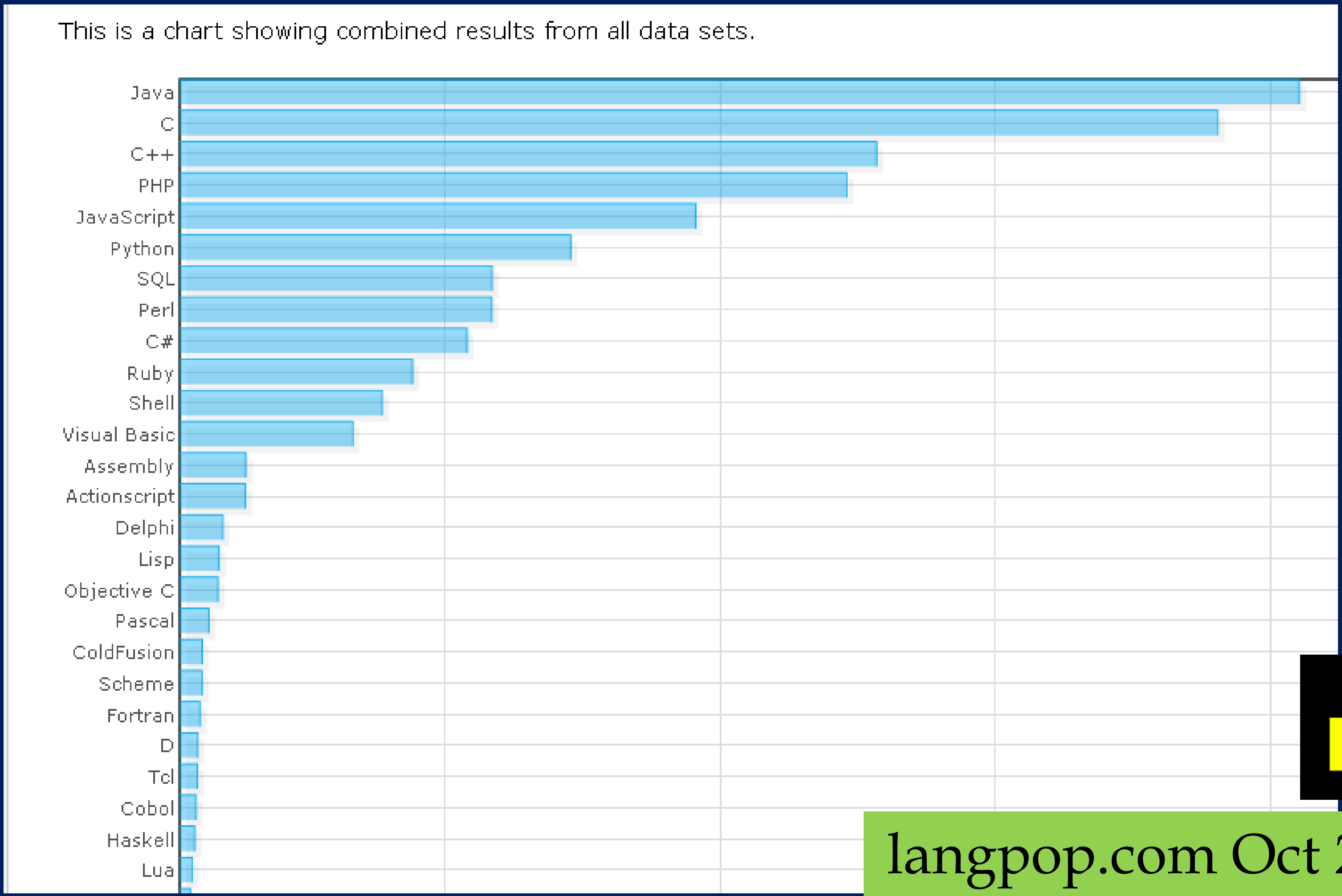
# Committee languages



# Haskell

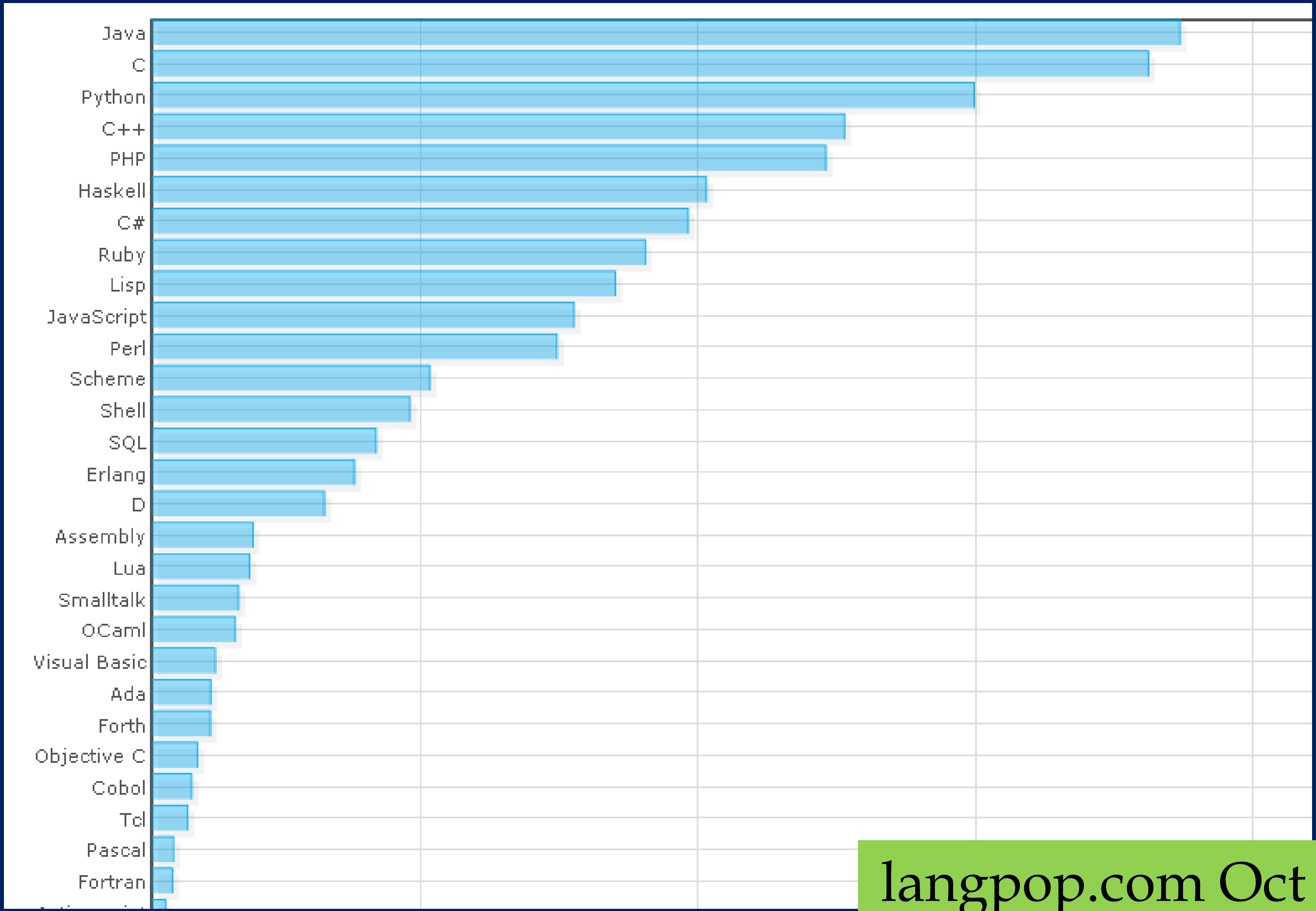

“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out”
(blog Apr 2007)
“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions”
(blog Mar 2007)
Practitioners
Geeks
1,000,000
10,000
100
1
The second life?
1990
1995
2000
2005
2010

# Language popularity
# how much language X is used



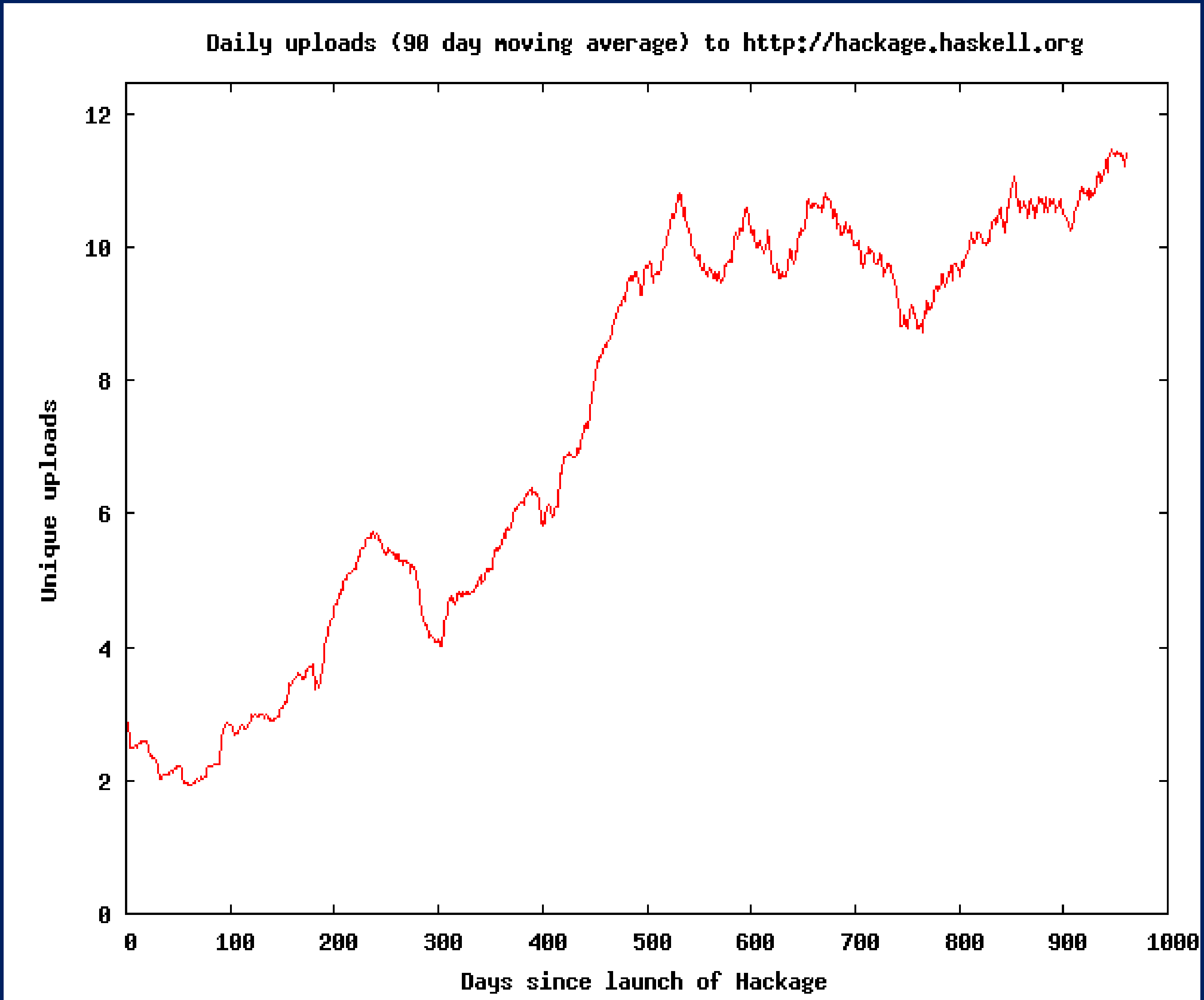

langpop.com Oct 2008

# Language popularity
# how much language X is talked about




langpop.com Oct 2008

# Hackage




1976 packages

533 developers

256 new packages Jan-Mar 2010

11.5 uploads/day

4k downloads/day

# Parallelism is a big opportunity for Haskell

- The language is naturally parallel (the opposite of Java)
- Everyone is worried about how to program parallel machines

# Haskell has three forms of concurrency

- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit
  - Deterministic
  - Pure: **par** and **seq**
- Data parallel
  - Deterministic
  - Pure: parallel arrays
  - Shared memory initially; distributed memory eventually; possibly even GPUs
- **General attitude**: using **some of** the parallel processors you already have, **relatively easily**

```
main :: IO ()
  = do { ch <- newChan
     ; forkIO (ioManager ch)
     ; forkIO  (worker 1 ch)
     ... etc ... }
```

```
f :: Int -> Int
f x = a `par` b `seq` a + b
    where
       a = f (x-1)
       b = f (x-2)
```

# Haskell has three forms of concurrency

- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit
  - Deterministic
- D
  - ributed memory eventually;

```
main :: IO ()
  = do { ch <- newChan
       ; forkIO (ioManager ch)
       ; forkIO  (worker 1 ch)
       ... etc ... }
```

```
f :: Int -> Int
f x = a `par` b `seq` a + b
    where
       a = f (x-1)
       b = f (x-2)
```

Today's focus

- **General attitude**: using **some of** the parallel processors you already have, **relatively easily**

After 30 years of research, the most widely-used co-ordination mechanism for shared-memory task-level concurrency is....

After 30 years of research, the most widely-used co-ordination mechanism for shared-memory task-level concurrency is....

**Locks and condition variables**

After 30 years of research, the most widely-used co-ordination mechanism for shared-memory task-level concurrency is....

**Locks and condition variables**

**(invented 30 years ago)**

# What's wrong with locks?

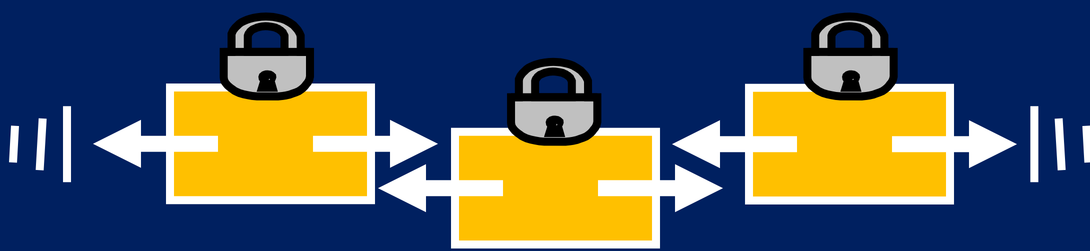
A 10-second review:

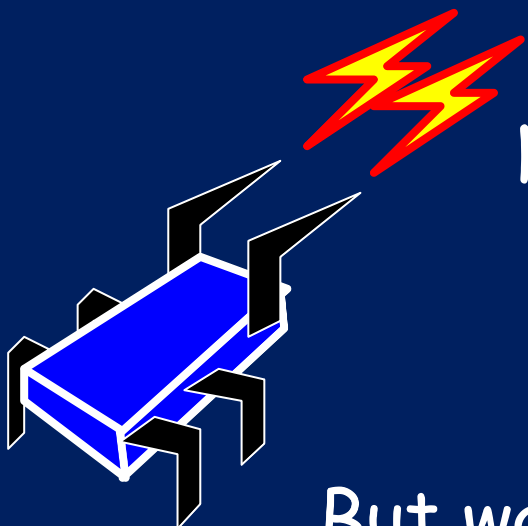
- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in "wrong" order.
- **Lost wakeups:** forgotten notify to condition variable
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...

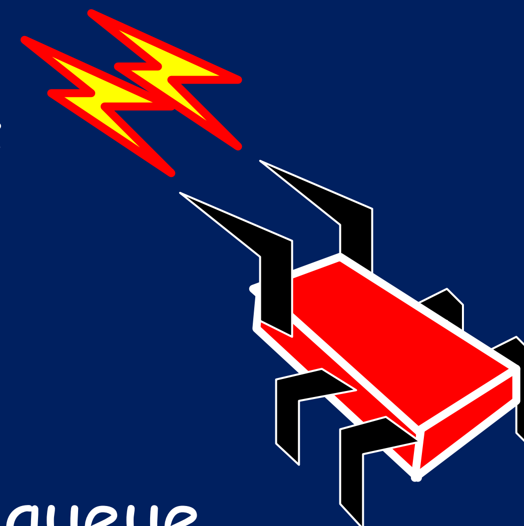
# Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell





No interference if ends "far enough" apart



But watch out when the queue is 0, 1, or 2 elements long!

# Locks are absurdly hard to get right

| Coding style | Difficulty of concurrent queue |
|---|---|
| Sequential code | Undergraduate |

# Locks are absurdly hard to get right

| Coding style | Difficulty of concurrent queue |
|---|---|
| Sequential code | Undergraduate |
| Locks and condition variables | Publishable result at international conference |

# Atomic memory transactions

| Coding style | Difficulty of concurrent queue |
| --- | --- |
| Sequential code | Undergraduate |
| Locks and condition variables | Publishable result at international conference |
| **Atomic blocks** | **Undergraduate** |

# Atomic memory transactions

```
atomic { ... sequential get 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**
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the **get** code)

ACID

# How does it work?

Optimistic concurrency

**atomic** { ... <code> ... }

One possibility:

- Execute <code> without taking any locks
- Each read and write in <code> is logged to a thread-local transaction log
- Writes go to the log only, not to memory
- At the end, the transaction tries to **commit** to memory
- Commit may fail; then transaction is re-run

# Realising STM in Haskell

# Realising STM in Haskell

```
main = do { putStr (reverse "yes")
          ; putStr "no" }
```

- Effects are explicit in the type system
  - (reverse "yes") :: String -- No effects
  - (putStr "no") :: IO () -- Can have effects
- The main program is an effect-ful computation
  - main :: IO ()

# Mutable state

```
newRef :: a -> IO (Ref a)
readRef :: Ref a -> IO a
writeRef :: Ref a -> a -> IO ()
```

```
main = do { r <- newRef 0
          ; incR r
          ; s <- readRef r
          ; print s }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v+1)
            }
```

Reads and writes are 100% explicit!

You can't say (r + 6), because r :: Ref Int

# Concurrency in Haskell

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

- fork spawns a thread
- it takes an action as its argument

```
main = do { r <- newRef 0
          ; fork (incR r)
          ; incR r
          ; ... }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef f; writeRef r (v+1) }
```

A race

# Atomic blocks in Haskell

```
atomic :: IO a -> IO a
```

```
main = do { r <- newRef 0
          ; fork (atomic (incR r))
          ; atomic (incR r)
          ; ... }
```

- atomic is a function, not a syntactic construct
- A worry: what stops you doing incR outside atomic?

# STM in Haskell

- Better idea:

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

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

main = do { r <- atomic (newTVar 0)
          ; fork (atomic (incT r))
          ; atomic (incT r)
          ; ... }
```

# STM in Haskell

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

- Notice that:
- Can't fiddle with TVars outside atomic block [good]
- Can't do IO inside atomic block [sad, but also good]
- No changes to the compiler (whatsoever). Only runtime system and primops.
- ...and, best of all...

# STM computations compose (unlike locks)

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

incT2 :: TVar Int -> STM ()
incT2 r = do { incT r; incT r }

foo :: IO ()
foo = ...atomic (incT2 r)...
```

Composition is THE way we build big programs that work

- An STM computation is always executed atomically (e.g. incT2). The type tells you.
- Simply glue STMs together arbitrarily; then wrap with atomic
- No nested atomic. (What would it mean?)

# Exceptions

- 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 into the IO monad
- No need to restore invariants, or release locks!
- See paper for the question of the exception value itself

# 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;
                    ; writeTVar acc (bal-n) }
```

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

- **retry** means "abort the current transaction and re-execute it from the beginning".
- Implementation avoids the busy wait by using reads in the transaction log (i.e. acc) to wait simultaneously on all read variables

# Compositional blocking

- No condition variables!
- Retrying thread is woken up automatically when acc is written. No lost wake-ups!
- No danger of forgetting to test everything again when woken up; the transaction runs again from the beginning.

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

# Why "compositional"?

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

  e.g. atomic (do { withdraw a1 3
                ; withdraw a2 7 })
- Waits for a1>3 AND a2>7, **without changing withdraw**

- Contrast:

  atomic (a1 > 3 && a2 > 7) { ...stuff... }

  which breaks the abstraction inside "...stuff..."

# Idea 2: Choice

```
atomic (do {
    withdraw a1 3
      `orelse`
    withdraw a2 3
  ;  deposit b 3 })
```

Try this

...and if it retries, try this

...and and then 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)
```

- transfer has an orElse, but calls to transfer can still be composed with orElse

# Composing transactions

- A transaction is a value of type (STM t)
- Transactions are first-class values
- Build a big transaction by composing little transactions: in sequence, using choice, inside procedures....
- Finally seal up the transaction with
  ```
  atomic :: STM a -> IO a
  ```
- No nested atomic! But orElse is like a nested transaction
- No concurrency within a transaction!

# Algebra

Nice equations:

- orElse is associative (but not commutative)
- retry `orElse` s = s
- s `orElse` retry = s

(STM is an instance of MonadPlus)

# Idea 3: invariants

- The route to sanity is by establishing **invariants** that are **assumed on entry**, and **guaranteed on exit**, by every atomic block
- 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 { v <- newTVar 0
                ; always (do  { cts <- readTVar v
                              ; return (cts >= 0) })
                ; return v }
```

An arbitrary boolean-valued STM computation

Any transaction that modifies the account will check the invariant (no forgotten checks)

# What always does

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

- **always** adds a new invariant to a global pool of invariants
- Conceptually, every invariant is checked after every transaction
- 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!

We have one

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

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

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

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

---

**Administrative transitions** $\quad M \rightarrow N$

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

---

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

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

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

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

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

# Conclusions

- Atomic blocks (atomic, retry, orElse) are a real step forward
- It's like using a high-level language instead of assembly code: whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - you can still write buggy programs;
  - concurrent programs are still harder to write than sequential ones;
  - aimed at shared memory
- But the improvement is very substantial