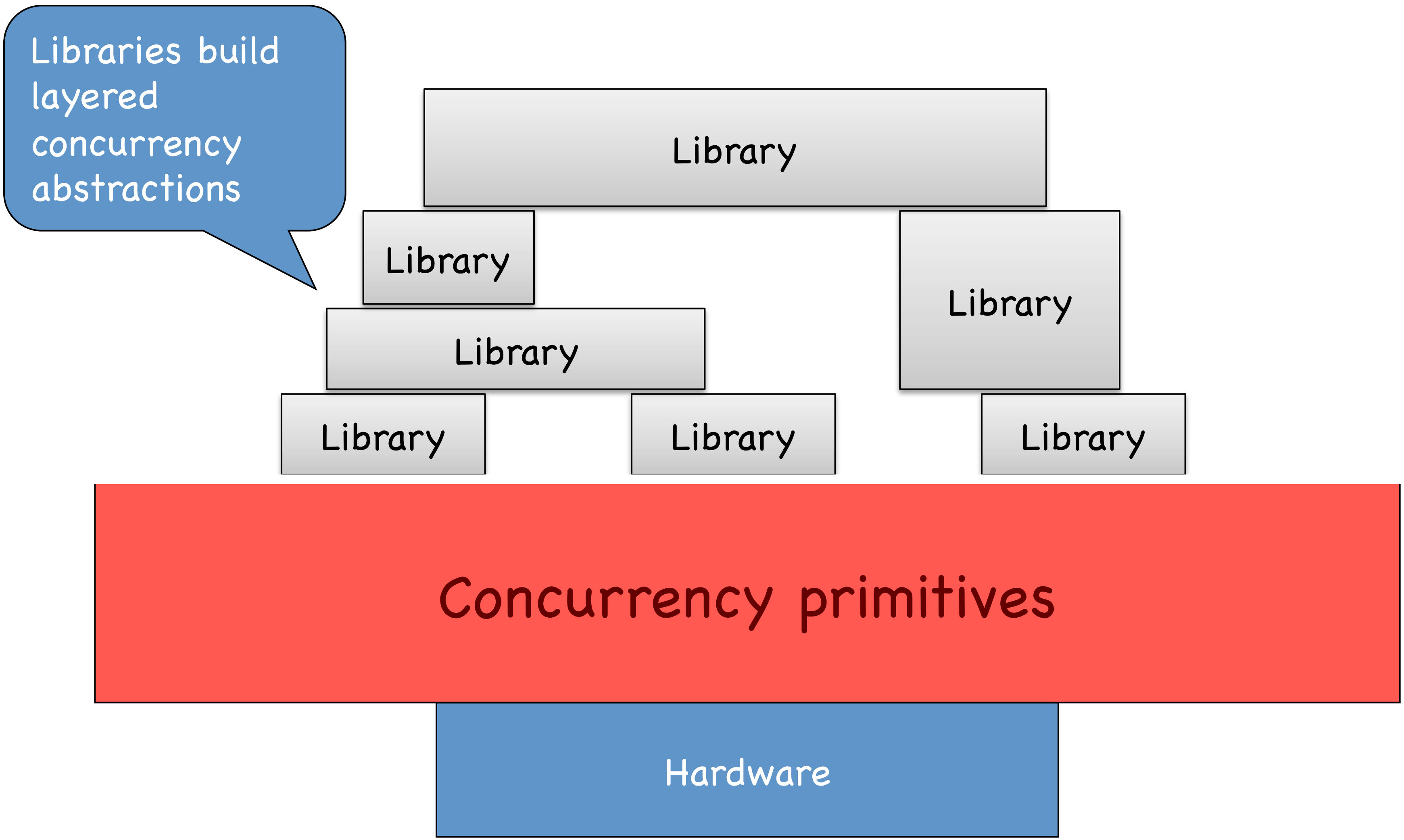
Software Transactional Memory

COS 326
David Walker
Princeton University

Learn You a Haskell!

http://learnyouahaskell.com/

What we want



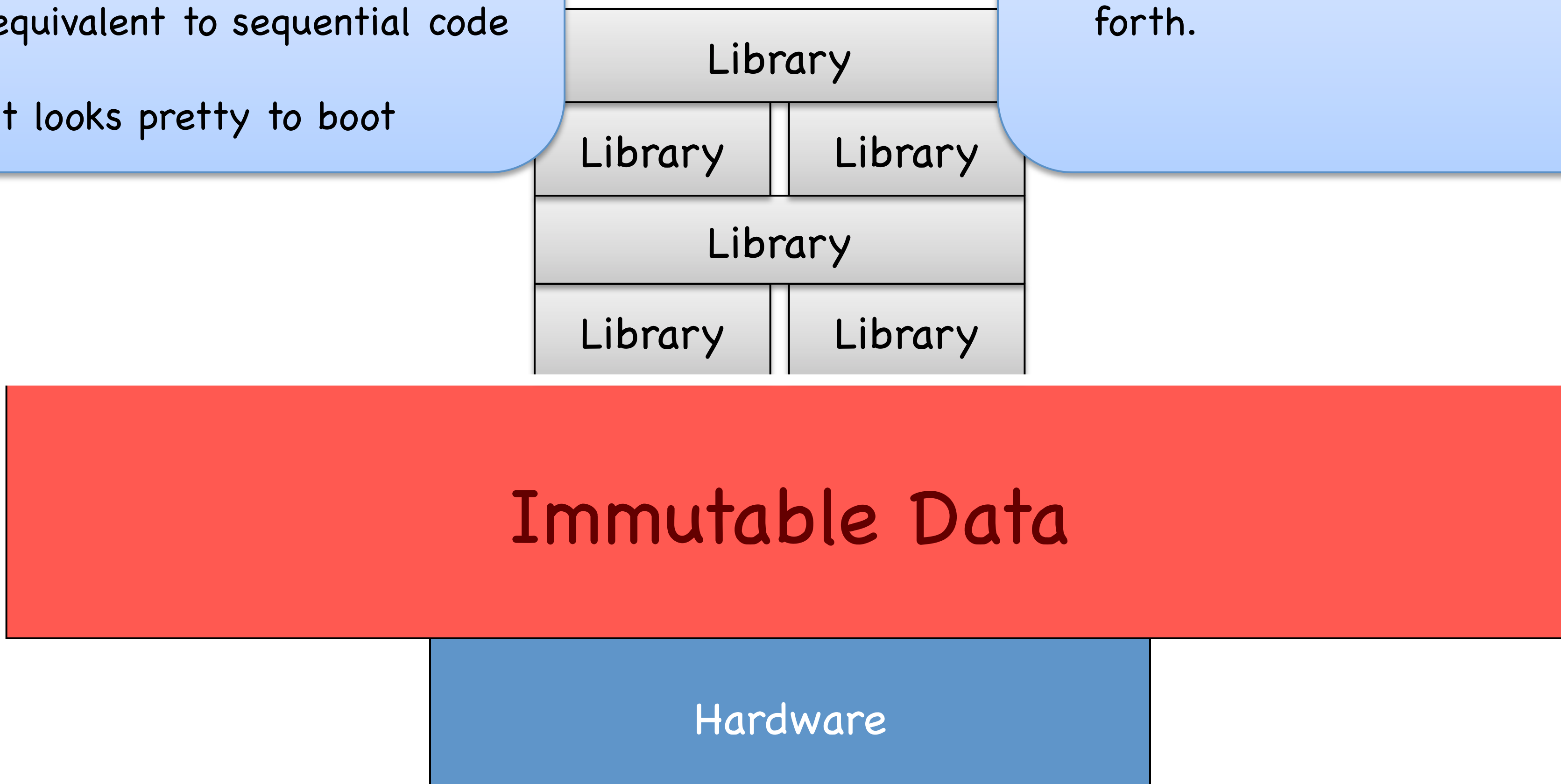
Libraries build layered concurrency abstractions
Library
Library
Library
Library
Library
Library
Library
Concurrency primitives
Hardware

First Idea: Don't Use Mutable Data/Effects

Good:
- no race conditions,
- no deadlock
- interleavings don't matter
- deterministic
- equivalent to sequential code

it looks pretty to boot

Bad:
- Can't interact with the world.
- The world changes.
- Threads can't talk back and forth.



Second Option: Locks

Associate a lock with each mutable object.

Acquire & release surrounding each access

Reason as if a sequence of instructions occurs sequentially.

```
let with_lock l f =
  Mutex.lock l;
  let res =
    try f () with exn ->
     (Mutex.unlock l;
      raise exn)
  in
  Mutex.unlock l;
  res
```

```
let deposit (a:account) (amount:int) : unit =
  with_lock a.lock (fun () ->
    if a.bal + amount < max_balance then
      a.bal <- a.bal + amount))
```

But ....

Managing multiple mutable objects got really hard because you can't easily put two good program components together to make another good program component:

```
let withdraw (a:account) = ...

let deposit (a:account) = ...

let transfer (a:account)(b:account)(amt:float) =
      withdraw a amt;

      deposit b amt
```

other threads can see a bad balance value in between withdrawal and deposit

huge problem: programmers still have to think about all possible interleavings

And ....

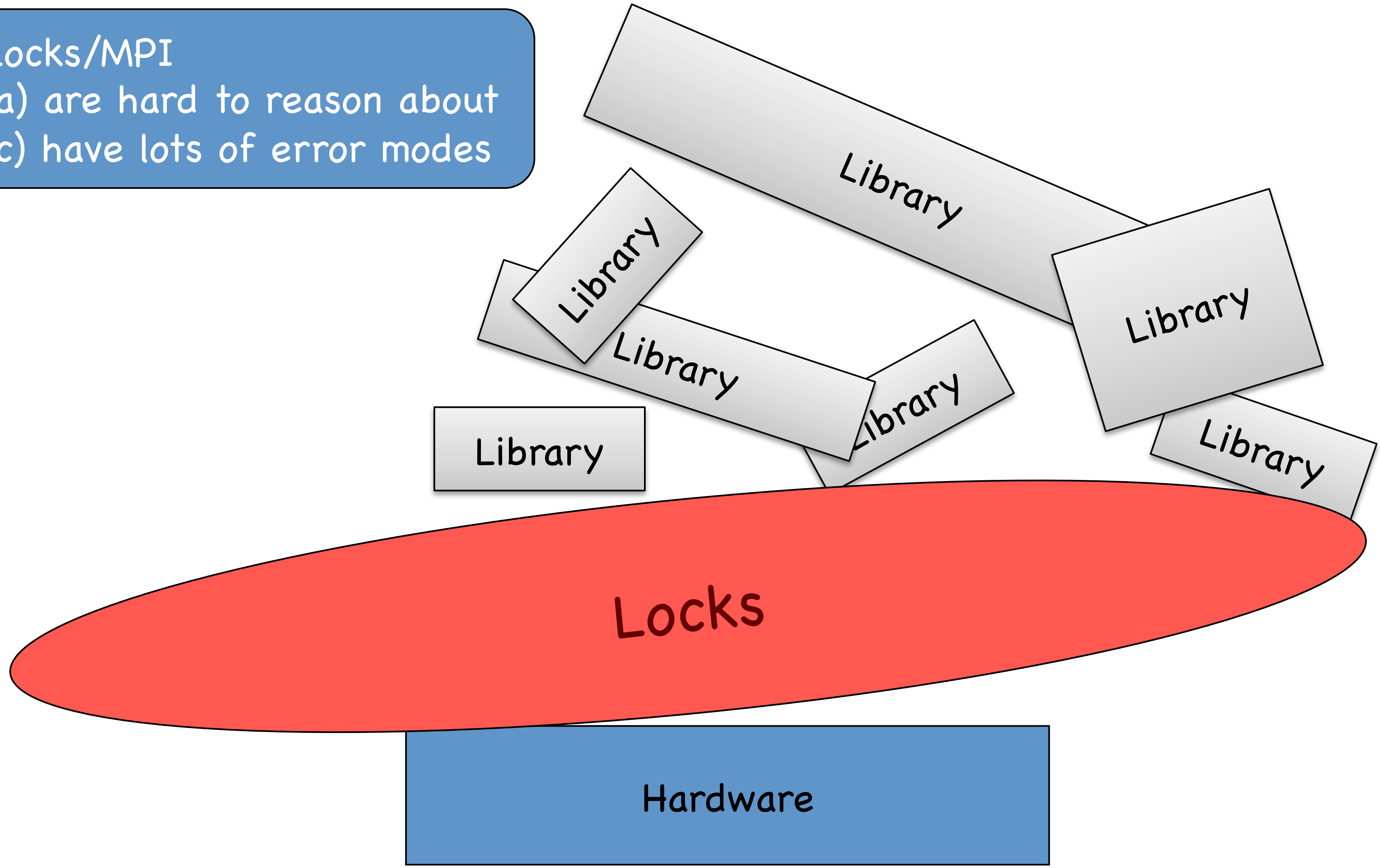
Managing multiple mutable objects got really hard because programs that use locks just don't compose very well:

```
let pop_two (s1:'a stack)
            (s2:'a stack) : ('a * 'a) option =
  with_lock s1.lock (fun _ ->
  with_lock s2.lock (fun _ ->
  match no_lock_pop s1, no_lock_pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> no_lock_push s1 x ; None
     | None, Some y -> no_lock_push s2 y ; None))
```

And we had to worry about forgetting to acquire locks or creating deadlocks ...

Second Idea: Use Locks or MPI

Locks/MPI
(a) are hard to reason about
(c) have lots of error modes



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

The Problem

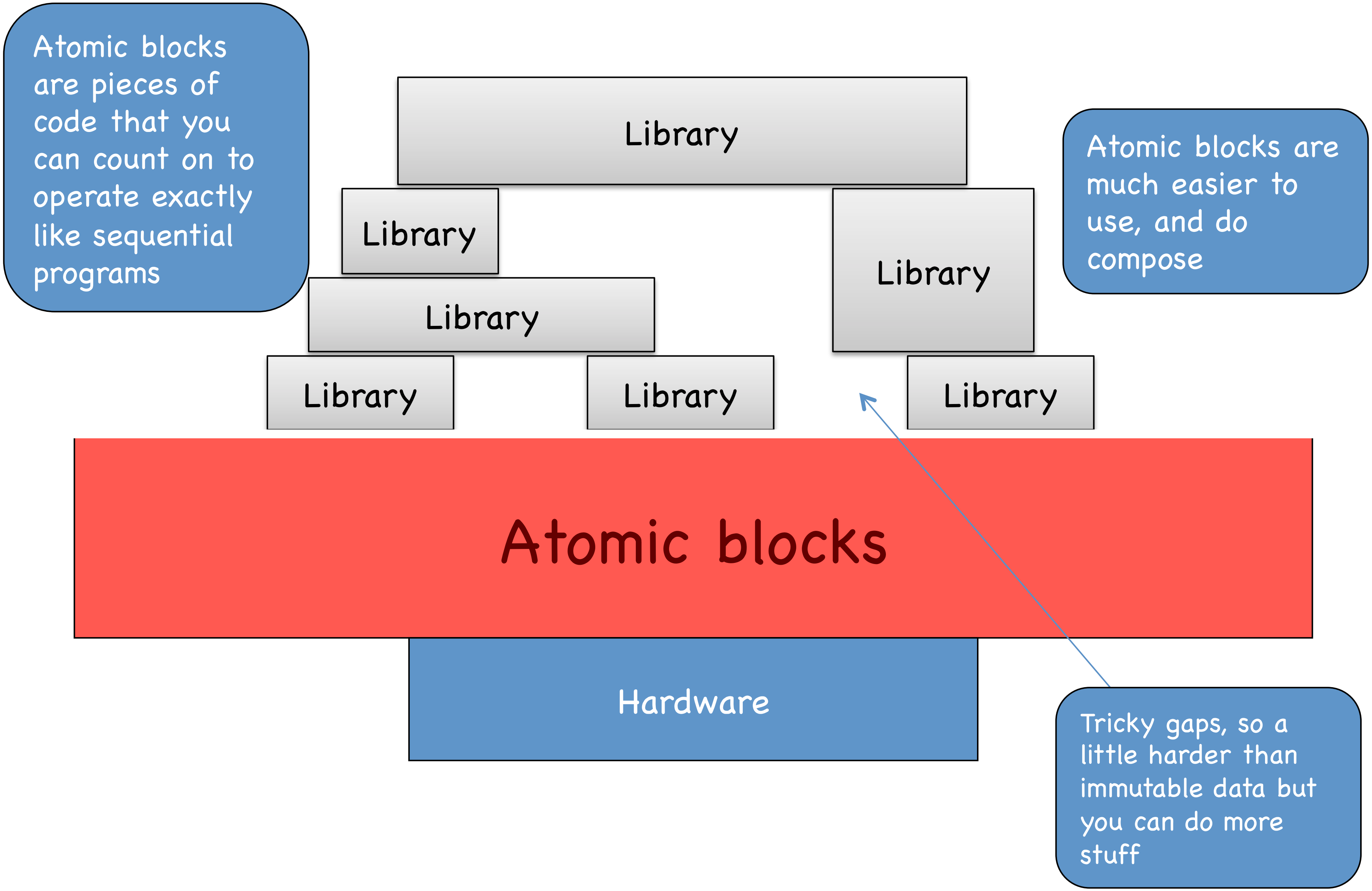
Locks are an indirect way at getting to what we really want:

```
a;
b;
c;
d;
```

execute these instructions in order without interruption by other threads

How might we design a language primitive that encapsulates this notion?

Third Idea: Atomic Blocks



Atomic Blocks Cut Down Nondeterminism

action 1:

```
read x
write x
read x
write x
```

action 2:

```
read x
write x
read x
write x
```

Atomic Blocks Cut Down Nondeterminism

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

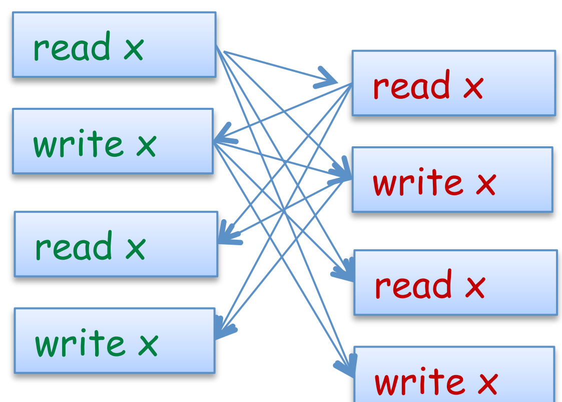
read x
write x
read x
write x

or

read x
write x
read x
write x

read x
write x
read x
write x

without atomic transactions:
vastly more possible interleavings



STM IN HASKELL

Concurrent Threads in Haskell

```
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

Atomic Blocks in Haskell

```
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

Recall Monads

Key idea:

- Monadic typing *constrains the use of effectful operations*

- int -> int:
  - cannot access a reference
  - cannot doo IO so you know that that function is pure
- int -> IO int:
  - returns a suspended computation that does access references
  - IO int computation can be composed with other computations

We will do the same thing to implement transactions. New kind of reference that can only be accessed inside an atomic block

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 with type STM a
  - just like Haskell refs can only be used inside computations with type IO a

TVar a == 'a ref

Haskell

OCaml

```
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

```
inc :: TVar Int -> STM ()
inc r = do
           v <- read r
           write r (v+1)

inc :: TVar Int -> STM ()
inc2 r = do
           inc r
           inc r

inc :: TVar Int -> STM ()
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

```
let with_lock l f =
  Mutex.lock l;
  let res =
    try f ()
    with exn ->
     (Mutex.unlock l;
      raise exn)
  in
  Mutex.unlock l;
  res
```

when exceptions get thrown, we are often in the midst of some complex action.

here, we must unlock the lock to get back to the initial state

more generally, we might have mutated many pieces of state and must "undo" our changes

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!

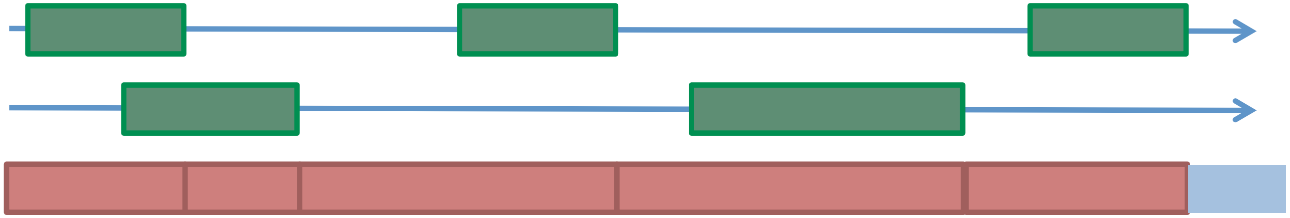
(you still need to deal with the exception ...)

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*.

- much 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

$$\boxed{\text{IO transitions} \qquad P;\Theta \xrightarrow{a} Q;\Theta'}$$

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

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

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

$$\boxed{\text{Administrative transitions} \qquad M \rightarrow N}$$

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

$$\boxed{\text{STM transitions} \qquad M;\Theta \Rightarrow N;\Theta'}$$

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

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

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

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

See "Composable Memory Transactions" for details.

Take COS 510 to understand what it means!

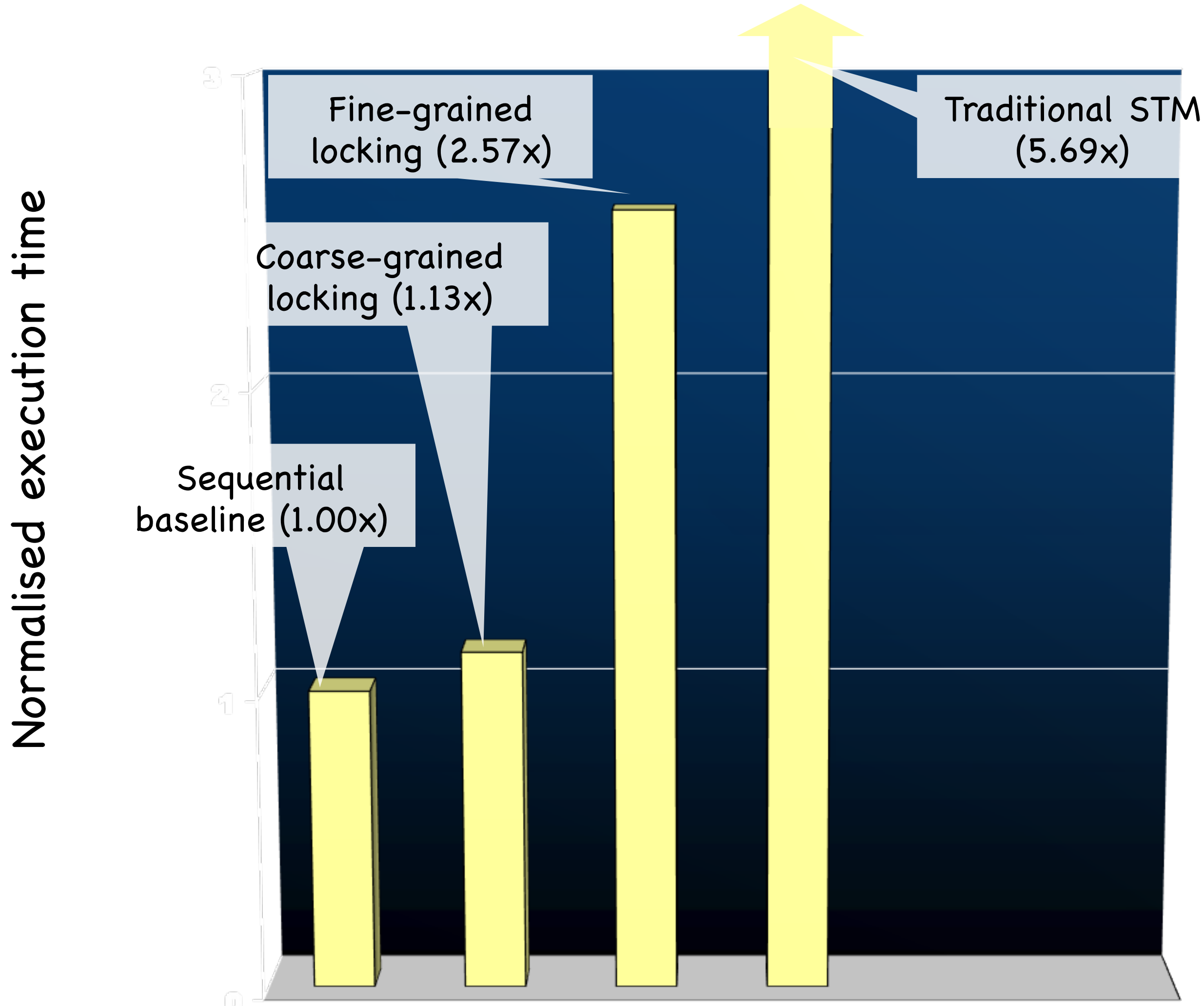
HASKELL IMPLEMENTATION

Implementation

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

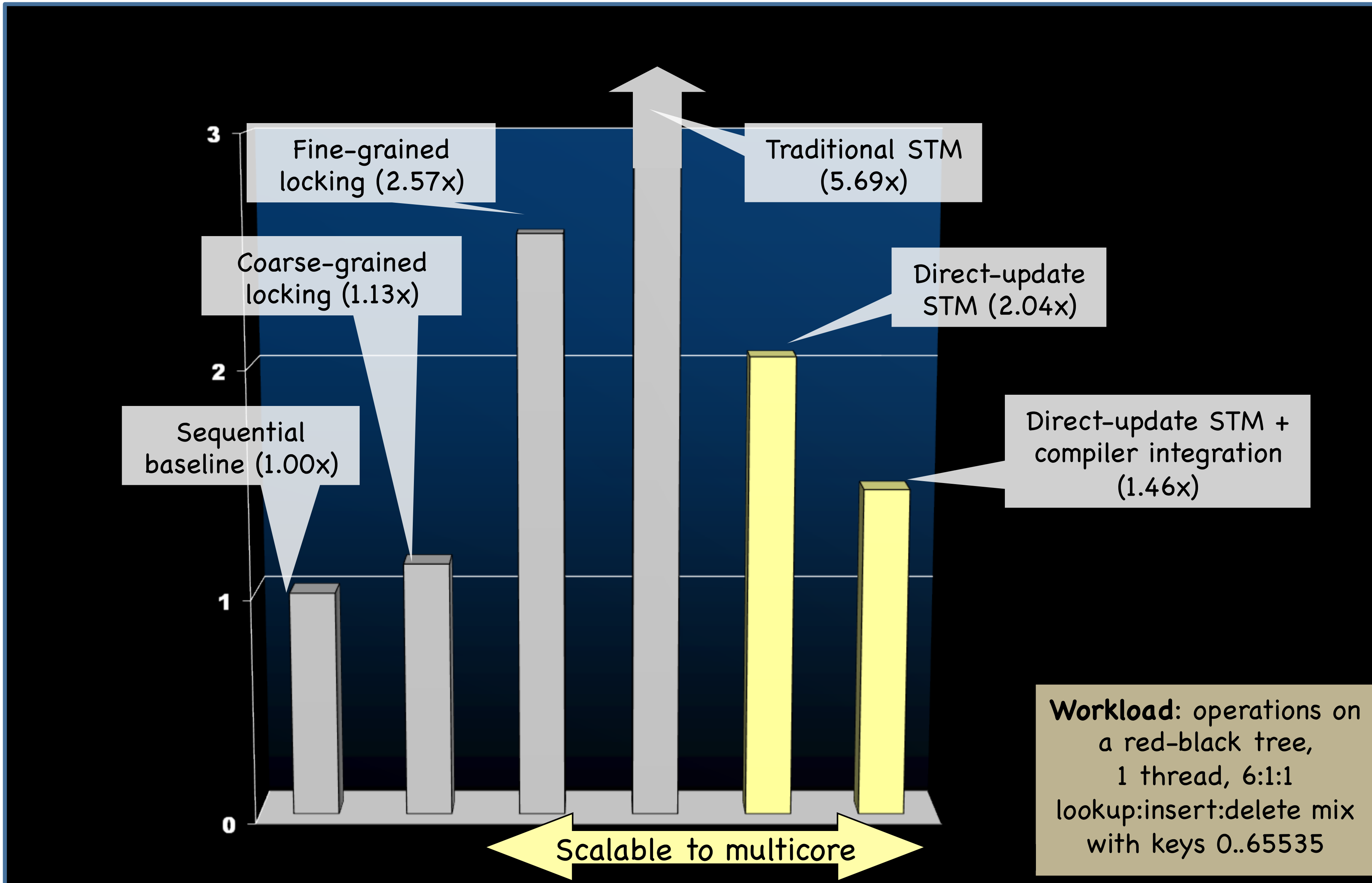
Compiler integration

- Decompose transactional memory operations into primitives
- Expose these primitives to compiler optimization (e.g. to hoist concurrency control operations out of a loop)

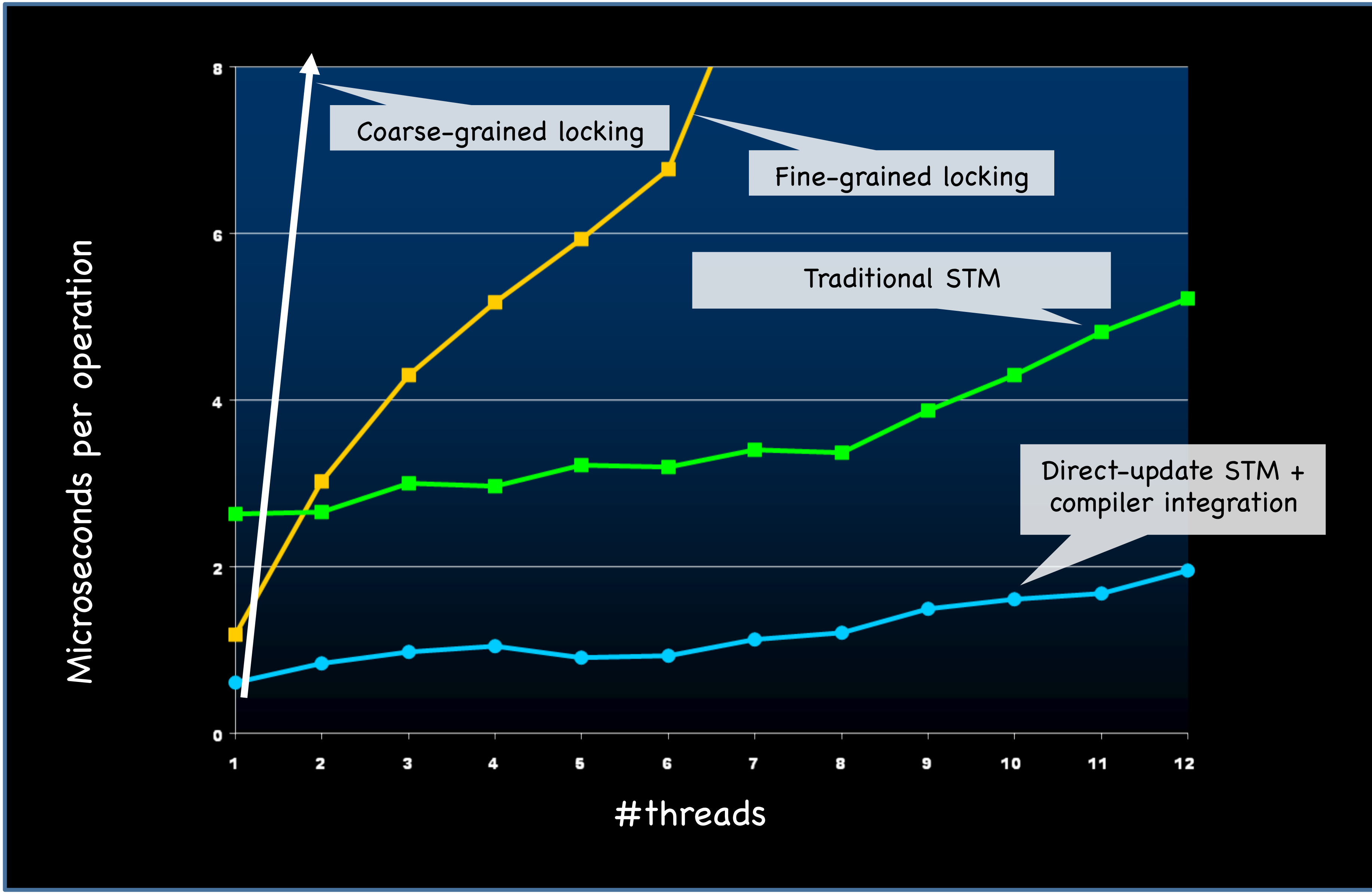
Runtime system integration

- Integrates transactions with the garbage collector to scale to atomic blocks containing 100M memory accesses

Results: Concurrency Control Overhead



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.

- hardware-supported STM

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

- 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"

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 non-terminating to 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

WHAT'S NEXT?

In this course

- An introduction to functional programming
  - immutable data
  - functions as data
  - cool abstractions:
    - futures, lazy computations, streams, parallel collections, atomic blocks, continuations, modules, functors, monads
- In most cases, I tried to explain how an abstraction helped a programmer reason about his or her program
- Because, in the end, that is the most important aspect of a good programming language: it makes it easier to reason about your programs

COS 510 (Spring 2015)

An introduction to *mechanical* reasoning about *programs* and *programming languages*.

It's a grad course but undergrads are encouraged to take it!
Colleen (room 510) will fill out all the signatures on the form concerning undergraduates taking a graduate course!

Programming Language Theory

OCaml's type system makes predictions:

e : int -> int

"expression e, if it terminates, will evaluate to a function value fun x -> ..."

Can we prove that OCaml's predictions always come true?

That's actually a pretty difficult thing to show in general!
There are infinitely many well-typed programs!
And the may execute for arbitrary many steps!

Also, recall the midterm:

```
let rec iterate (f:int list -> int list) (x:int list) : int list =
  match x with
    [] -> []
  | hd::tl -> iterate f (f tl)

let rec iterate2 (x
  match x with
    [] -> []
  | hd::tl -> ite
```

Theorem:

for all f:int list -> int

for all x:int list.

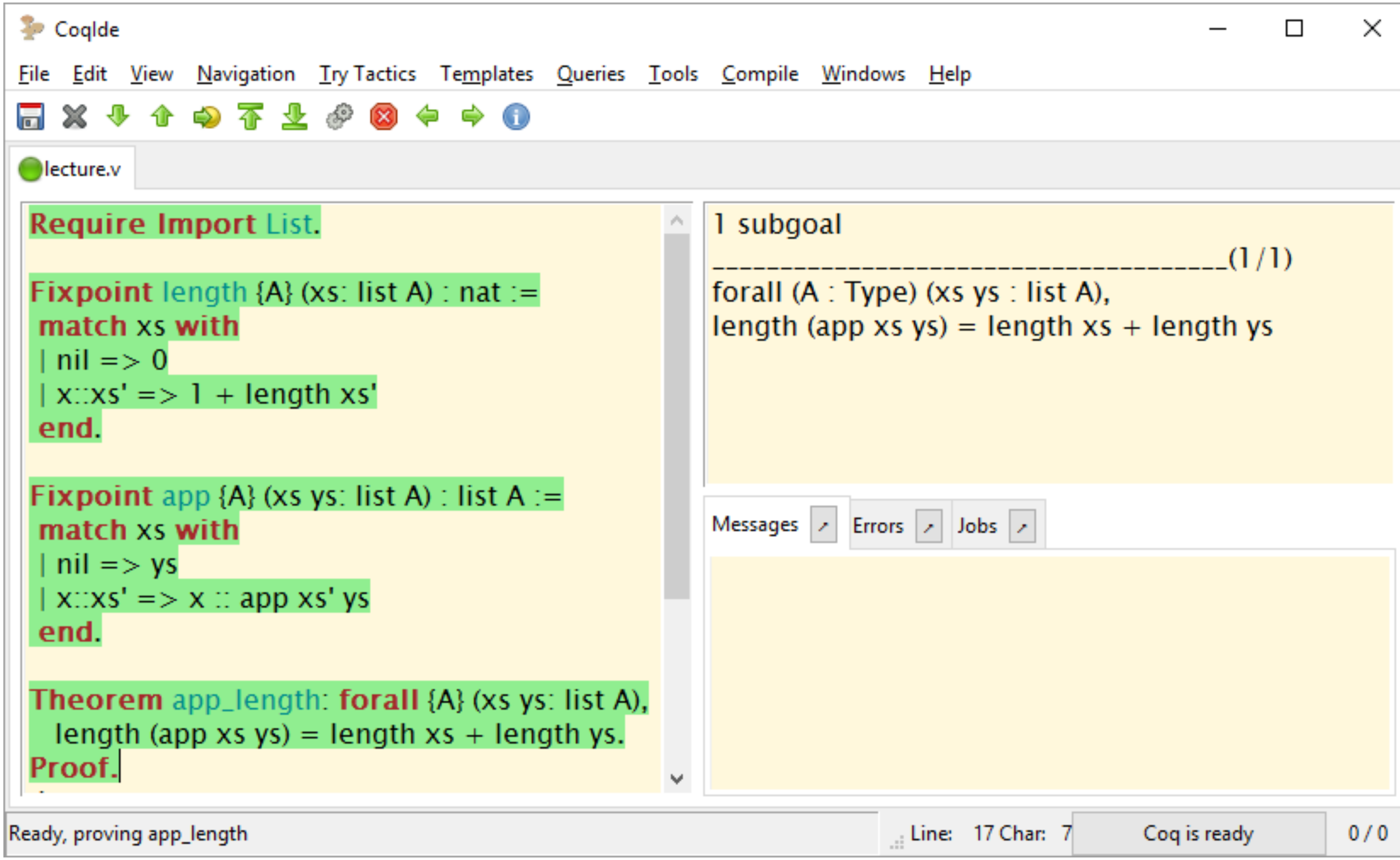
iterate f x == iterate2 x f

Can we devise a programming language that can check the correctness of our proofs?

oops!

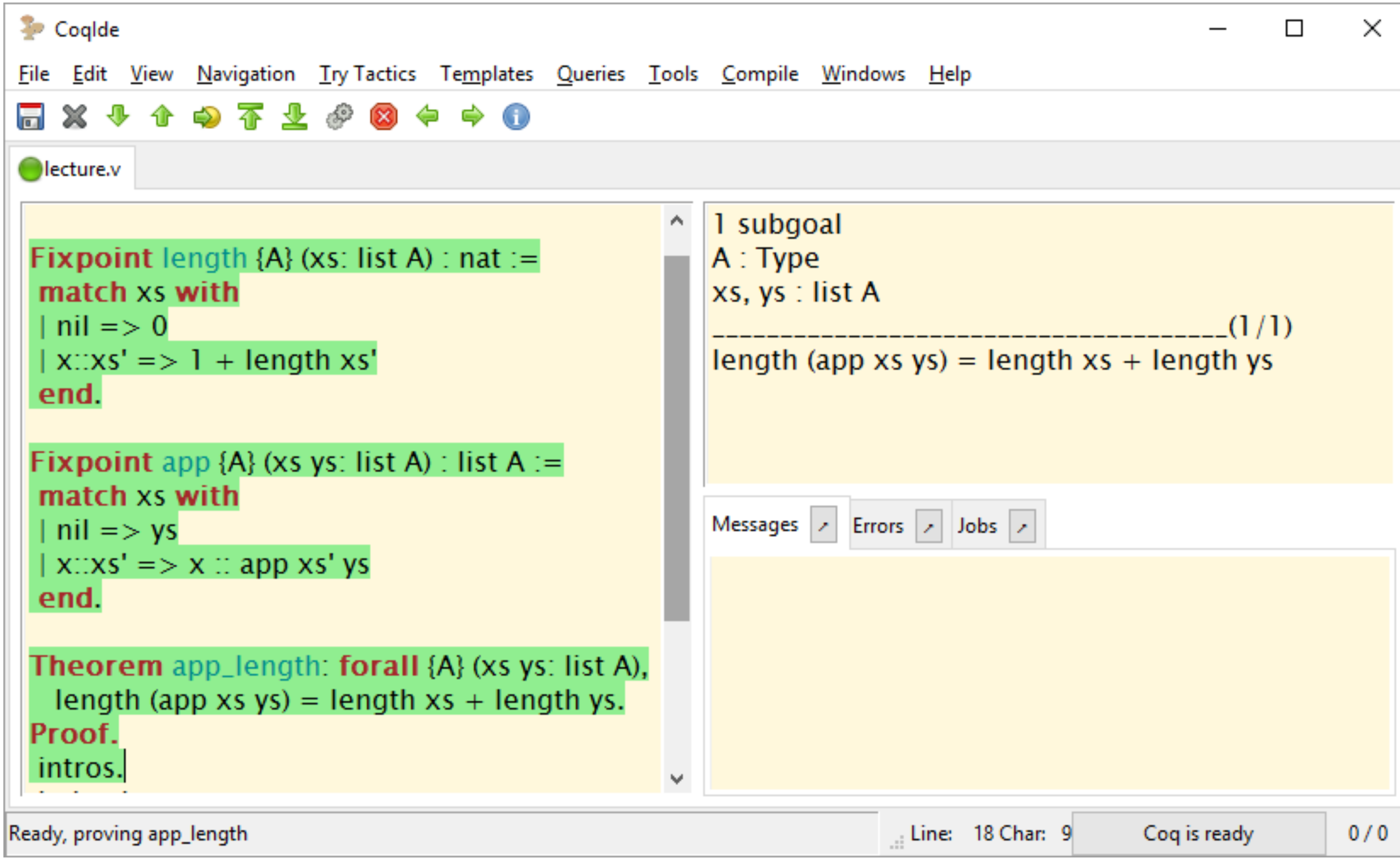
```
   :
      ate f (hd::tl)            (LHS)
== match hd::tl with ...        (eval)
== iterate f (f tl)             (eval)
== iterate2 (f tl) f            (by IH)
== match hd::tl with ...        (rev. eval)
== iterate2 (hd::tl) f          (rev. eval)
```

Yuppers!

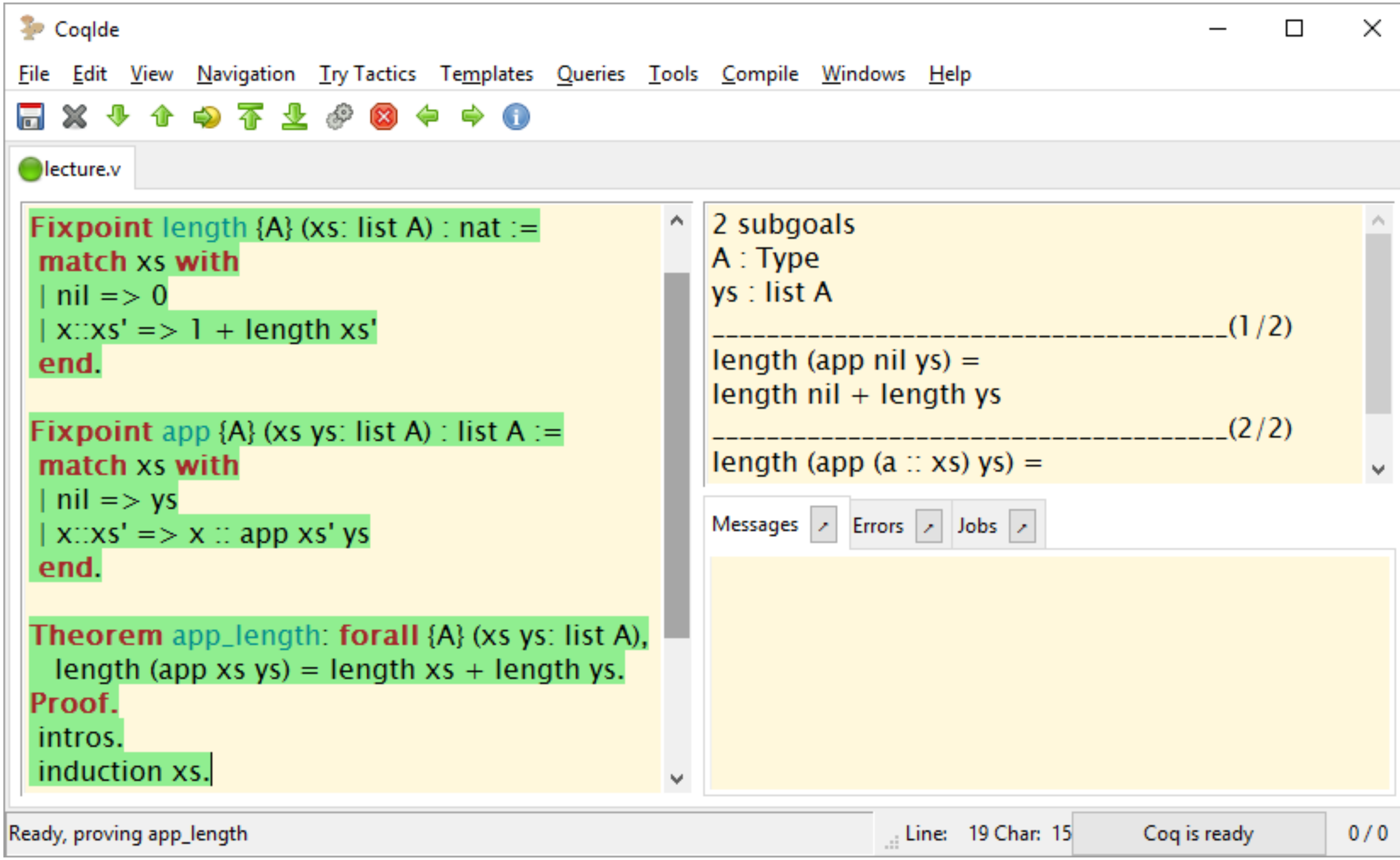


Coq: A theorem proving environment

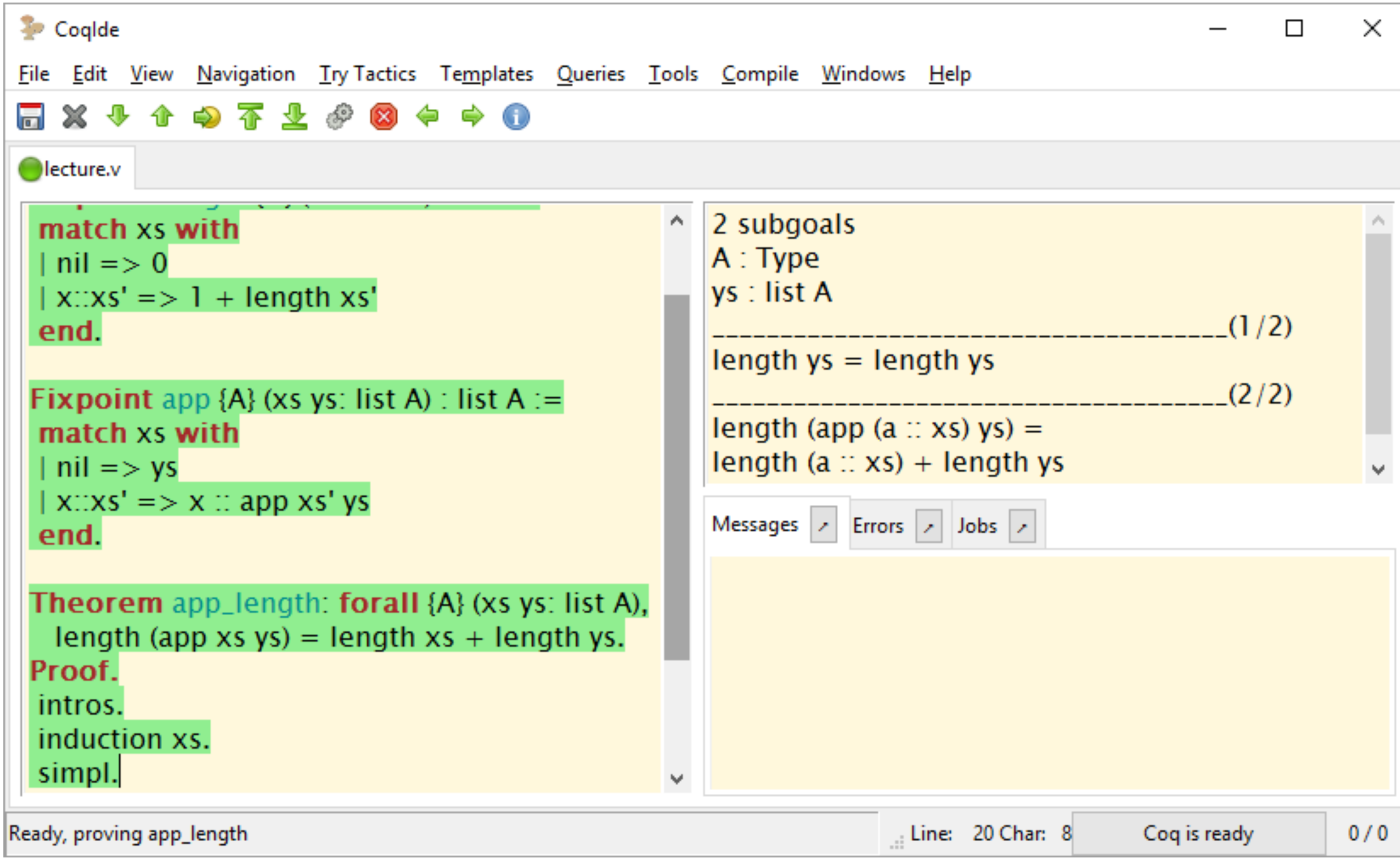
Yuppers!


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
Fixpoint length {A} (xs: list A) : nat :=
match xs with
| nil => 0
| x::xs' => 1 + length xs'
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
1 subgoal
A : Type
xs, ys : list A
______________________________________(1/1)
length (app xs ys) = length xs + length ys
Messages
Errors
Jobs
Ready, proving app_length
Line: 18 Char: 9
Coq is ready
0 / 0

Yuppers!

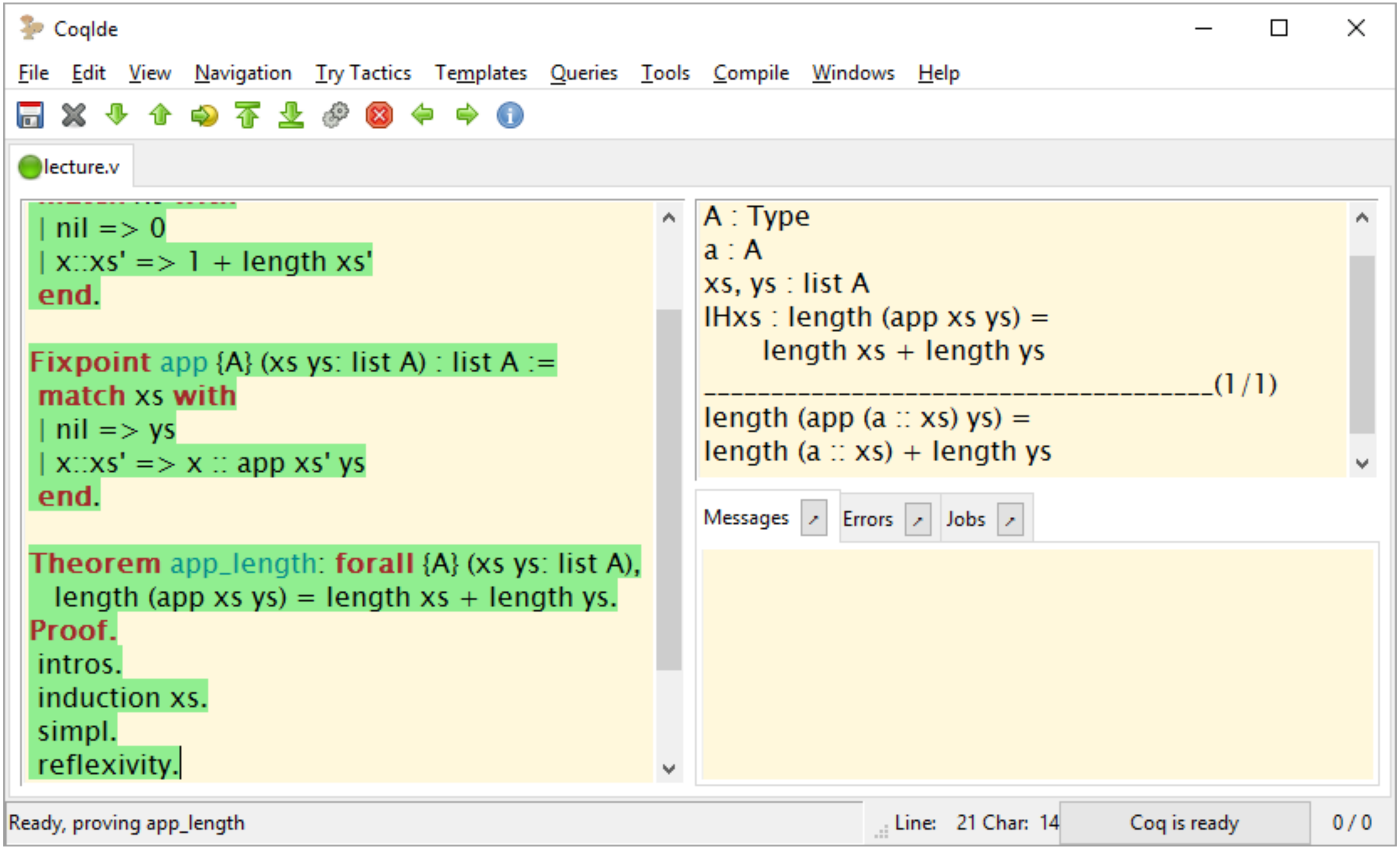

CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
Fixpoint length {A} (xs: list A) : nat :=
match xs with
| nil => 0
| x::xs' => 1 + length xs'
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
2 subgoals
A : Type
ys : list A
______________________________________(1/2)
length (app nil ys) =
length nil + length ys
______________________________________(2/2)
length (app (a :: xs) ys) =
Messages
Errors
Jobs
Ready, proving app_length
Line: 19 Char: 15
Coq is ready
0 / 0

Yuppers!

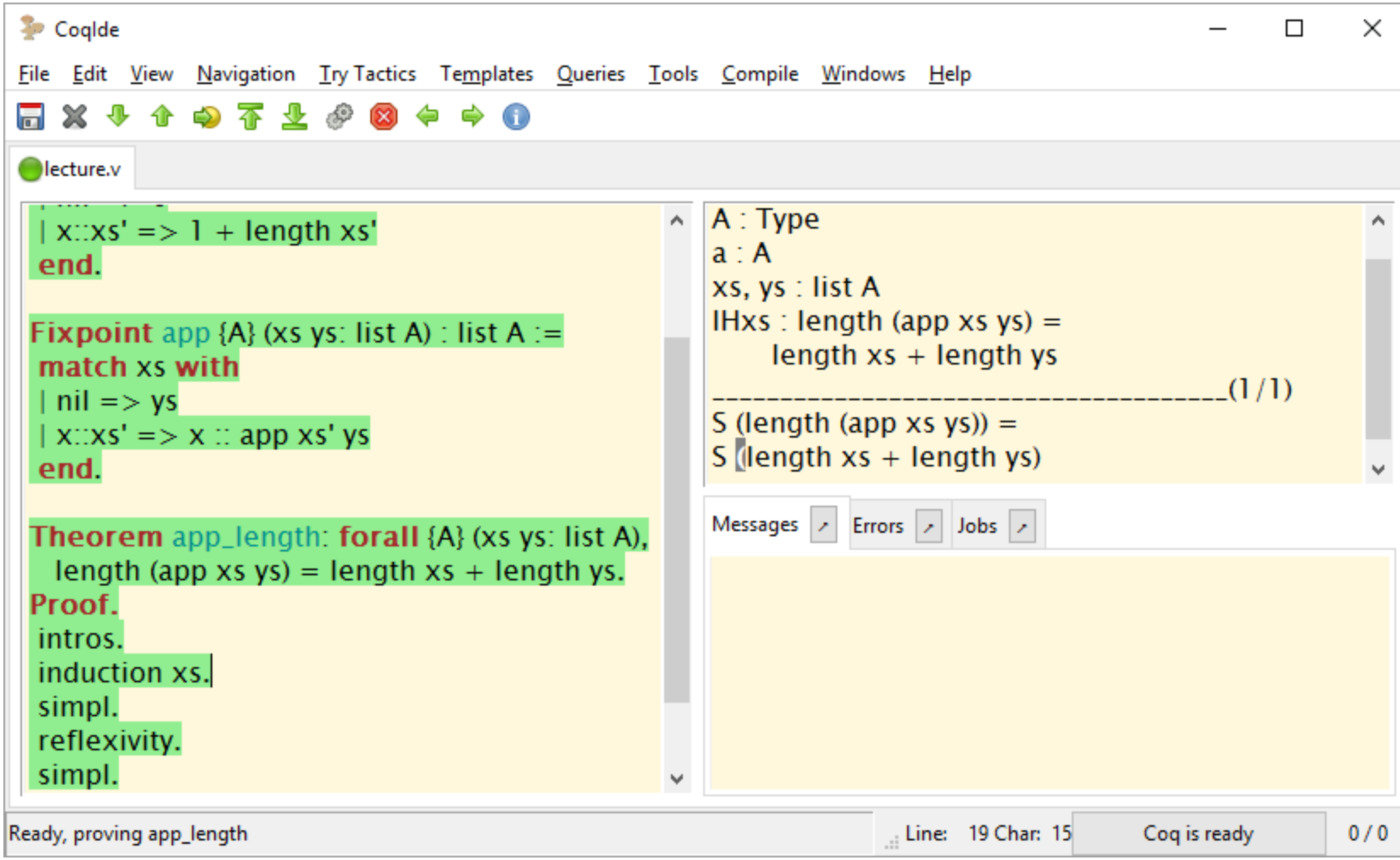


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
match xs with
| nil => 0
| x::xs' => 1 + length xs'
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
simpl.
2 subgoals
A : Type
ys : list A
______________________________________(1/2)
length ys = length ys
______________________________________(2/2)
length (app (a :: xs) ys) =
length (a :: xs) + length ys
Messages
Errors
Jobs
Ready, proving app_length
Line: 20 Char: 8
Coq is ready
0 / 0

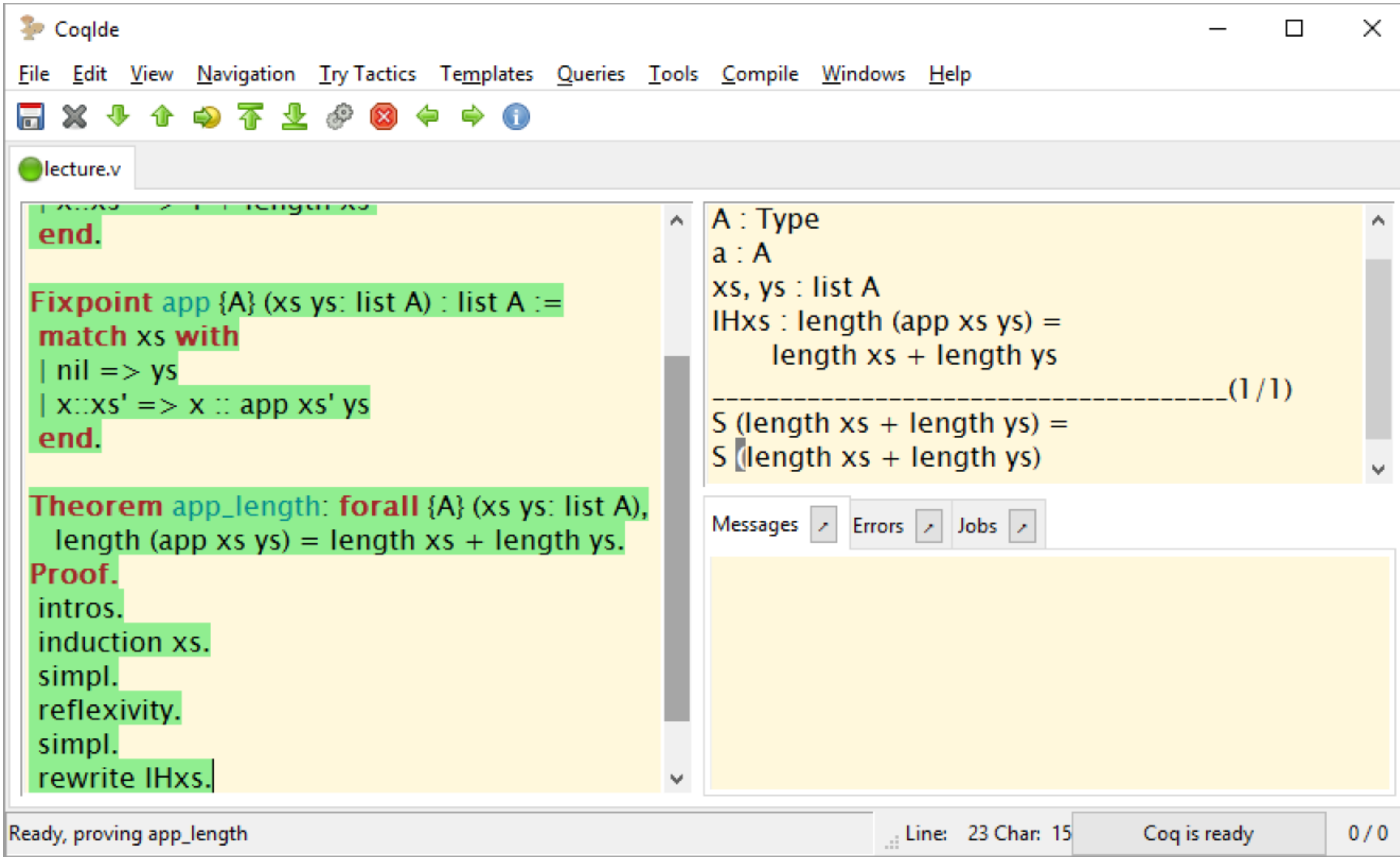
Yuppers!


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
| nil => 0
| x::xs' => 1 + length xs'
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
simpl.
reflexivity.
A : Type
a : A
xs, ys : list A
IHxs : length (app xs ys) =
length xs + length ys
(1 / 1)
length (app (a :: xs) ys) =
length (a :: xs) + length ys
Messages
Errors
Jobs
Ready, proving app_length
Line: 21 Char: 14
Coq is ready
0 / 0

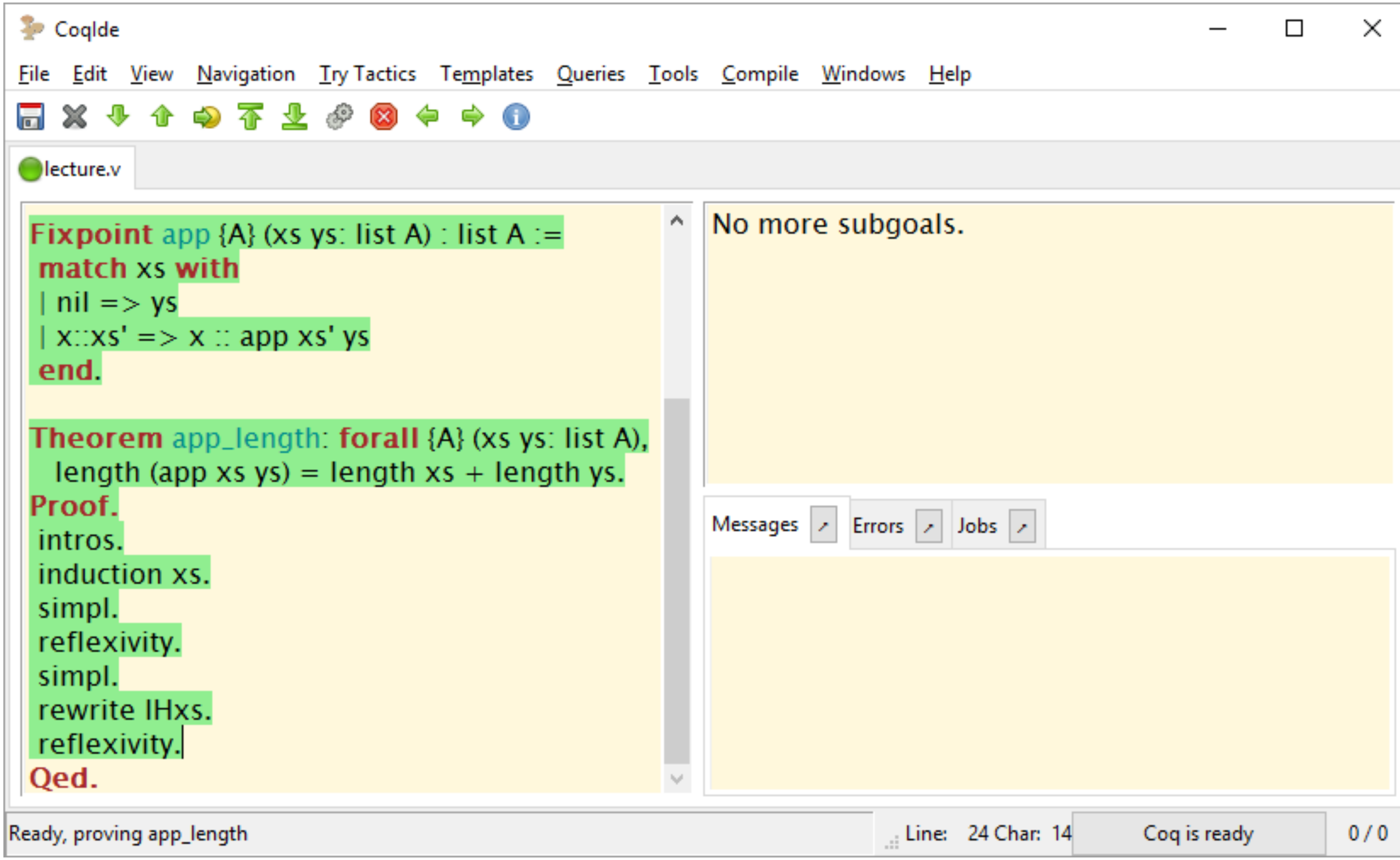
Yuppers!


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
| x::xs' => 1 + length xs'
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
simpl.
reflexivity.
simpl.
A : Type
a : A
xs, ys : list A
IHxs : length (app xs ys) =
length xs + length ys
______________________________________(1 / 1)
S (length (app xs ys)) =
S (length xs + length ys)
Messages
Errors
Jobs
Ready, proving app_length
Line: 19 Char: 15
Coq is ready
0 / 0

Yuppers!


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
end.
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
simpl.
reflexivity.
simpl.
rewrite IHxs.
A : Type
a : A
xs, ys : list A
IHxs : length (app xs ys) =
length xs + length ys
(1 / 1)
S (length xs + length ys) =
S (length xs + length ys)
Messages
Errors
Jobs
Ready, proving app_length
Line: 23 Char: 15
Coq is ready
0 / 0

Yuppers!


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
lecture.v
Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.
Proof.
intros.
induction xs.
simpl.
reflexivity.
simpl.
rewrite IHxs.
reflexivity.
Qed.
No more subgoals.
Messages
Errors
Jobs
Ready, proving app_length
Line: 24 Char: 14
Coq is ready
0 / 0

Summary

If you enjoyed 326, especially the theoretical parts,
there is a lot more fun stuff to learn in 510.

Happy Holidays!