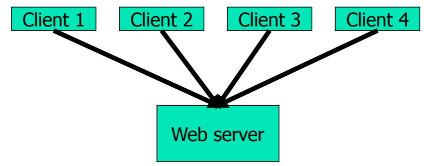
Concurrency

The need for concurrency

- Want one thread ("virtual server") per client
- Threads largely independent, but share some common resources (e.g. file system)



Concurrency vs parallelism

Parallel functional programming:
- Aim = performance through multiple processors (e.g. `e1+e2` in parallel)
- No semantic changes; deterministic results

Concurrent functional programming
- Aim = concurrent, I/O-performing threads
- Makes perfect sense on a uniprocessor
- Non-deterministic interleaving of I/O is inevitable

Concurrent web service

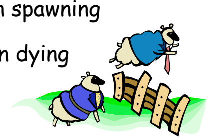
```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
    = forever (do { conn <- accept socket ;
                    forkIO (serviceConn config conn) })
```

```
forkIO :: IO a -> IO ThreadId
```

- forkIO spawns an independent, I/O-performing, thread
- No parameters passed; free variables work fine

Communication and sharing

- What if two threads want to communicate? Or share data?
- Example: keep a global count of how many client threads are running
  - o Increment count when spawning
  - o Decrement count when dying



Communication and sharing

```
data MVar a
newEmptyMVar  :: IO (MVar a)
putMVar       :: MVar a -> a -> IO ()
takeMVar      :: MVar a -> IO a
```

- A value of type `(MVar t)` is a **location** that is either
  - o **empty**, or
  - o **holds a value** of type t 27

Communication and sharing

```
takeMVar    :: MVar a -> IO a
putMVar     :: MVar a -> a -> IO ()
```

	Empty	Full
takeMVar	Block	Return contents, leave MVar empty
putMVar	Fill MVar	Block

70

Using MVars

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = do { count <- newEmptyMVar ;
         putMVar count 0 ;
         forever (do { conn <- accept socket ;
                       forkIO (do { inc count ;
                                    serviceConn config
conn ;
                                    dec count}) }

inc,dec :: MVar Int -> IO ()
inc count = do { v <- takeMVar count; putMVar count (v+1) }
dec count = do { v <- takeMVar count; putMVar count (v-1) }
```

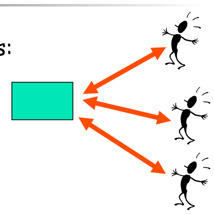
MVar is empty at this point, hence no race hazard

71

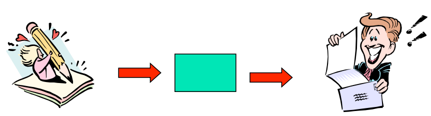
MVars as channels

An MVar directly implements:

- a shared data structure



- a one-place channel



72

Semantics

Fortunately, most of the infrastructure is there already!

Step 1: elaborate the program state

P, Q, R ::=	$\ldots$	
|	$\{M\}_t$	A thread called t
|	$\langle M \rangle_m$	An MVar called m containing M
|	$\langle\rangle_m$	An empty MVar called m

e.g. $\{$`putChar 'c'`$\}_{t1}$ | $\{$`putChar 'd'`$\}_{t2}$

73

Semantics

Step 2: a rule for for forkIO

$$\frac{u \notin fn\,(M, \mathrm{E})}{\{\mathrm{E}[\texttt{forkIO}\ M]\}_t \ \rightarrow\ \nu u.(\{\mathrm{E}[\texttt{return}\ u]\}_t \mid \{M\}_u)} \ (FORK)$$

- Restrict the new thread name (νu)
- Return the thread name to the caller ($\texttt{return}\ u$)
- The new thread ($\{M\}_u$)

74

Semantics

Step 3: rules for new, take, put

$$\{\mathrm{E}[\texttt{takeMVar}\ m]\}_t \mid \langle M \rangle_m \ \rightarrow\ \{\mathrm{E}[\texttt{return}\ M]\}_t \mid \langle\rangle_m$$
$$\{\mathrm{E}[\texttt{putMVar}\ M]\}_t \mid \langle\rangle_m \ \rightarrow\ \{\mathrm{E}[\texttt{return ()}]\}_t \mid \langle M \rangle_m$$

- Same as readIORef, writeIORef, except that MVar is filled/emptied
- Blocking is implicit
- Non-determinism is implicit

75

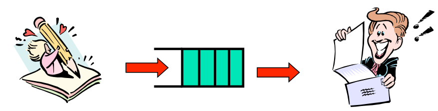
Building abstractions

- MVars are primitive
- Want to build abstractions on top of them
- Example: a buffered channel

76

A buffered channel

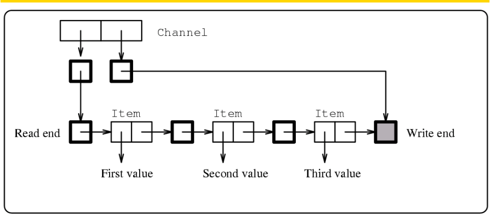
```
type Chan a
newChan   :: IO (Chan a)
putChan   :: Chan a -> a -> IO ()
getChan   :: Chan a -> IO a
```



77

A buffered channel

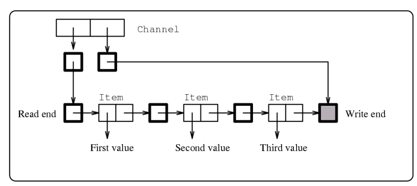
```
type Chan a   = (MVar (Stream a),MVar (Stream a))
type Stream a = MVar (Item a)
data Item a   = MkItem a (Stream a)
```



78

A buffered channel

```
putChan :: Chan a -> a -> IO ()
putChan (read,write) val
   = do { new_hole <- newEmptyMVar ;
          old_hole <- takeMVar write ;
          putMVar write new_hole
          putMVar old_hole (MkItem val new_hole) }
```



79

Summary

- forkIO + MVars are very simple.
- MVars are a low-level primitive, but surprisingly often Just The Right Thing
- Some excellent references:
  - Concurrent programming in ML (Reppy, CUP)
  - Concurrent programming in Erlang (Armstrong, Prentice Hall, 2nd edition)

80

Exceptions

81

Why do we need exceptions?

Robust programs deal gracefully with "unexpected" conditions. E.g.

- Disk write fails because disk is full
- Client goes away, so server should time out and log an error
- Client requests seldom-used service; bug in server code gives pattern-match failure or divide by zero

Server should not crash if these things happen!

82

Approach 1: virtue

"A robust program never goes wrong"
(e.g. test for disk full before writing)

BUT:

- Can't test for all errors (e.g. timeouts)

Need a way to recover from ANY error

83

Approach 2: exceptions

Provide a way to say "execute this code, but if anything (at all) goes wrong, abandon it and do this instead".

This might be called

"**Exceptions for disaster recovery**"

- Exception handler typically covers a large chunk of code
- Recovery action typically aborts a whole chunk of work

84

Aside: bad uses of exceptions

Exceptions are often (mis-) used in a different way:

"**Exceptions for extra return values**"

e.g. Look up up something in a table, raising "NotFound" if it's not there.

- Exception handler often encloses a single call
- Recovery action typically does not abort anything

N.b.: This is Simon's view, not universally shared (though I tend to agree)

Exceptions in Haskell 98

Haskell 98 supports exceptions in I/O actions:

```
catch     :: IO a -> (IOError -> IO a) -> IO a
userError :: String -> IOError
ioError   :: IOError -> IO a
```

```
catch (do { h <- openFile "foo";
            processFile h })
      (\e -> putStr "Oh dear")
```

Protected code

Exception handler

Dynamic scope: exceptions raised in processFile are also caught

86

Semantics

Step 1: add a new evaluation context

E ::= [.] | E >>= M | **catch E M**

Says: "evaluate inside the first argument of `catch`"

87

Semantics

Step 2: add propagation rule for `ioError`

$$\{E[\texttt{ioError}\ e \ \texttt{>>=}\ M]\}_t \rightarrow \{E[\texttt{ioError}\ e]\}_t$$

An exception before the (>>=)...

...discards the part after the (>>=)

Standard stack-unwinding implementation is possible

88

Semantics

Step 3: add rules for `catch`

$$\{E[\texttt{catch}\ (\texttt{ioError}\ e)\ M]\}_t \rightarrow \{E[M\ e]\}_t$$
$$\{E[\texttt{catch}\ (\texttt{return}\ N)\ M]\}_t \rightarrow \{E[\texttt{return}\ N]\}_t$$

What to do if an exception ***is*** raised

What to do if an exception is ***not*** raised

89

Synchronous vs asynchronous

- A synchronous exception is raised as a direct, causal result of executing a particular piece of code
  - Divide by zero
  - Disk full
- An asynchronous exception comes from "outside" and can arrive at any moment
  - Timeout
  - Stack overflow

90

Haskell 98 isn't enough

Pure Haskell 98 deals only with synchronous exceptions in the IO monad

Two big shortcomings

- Does not handle things that go wrong in purely-functional code
- Does not deal with asynchronous exceptions

91

Exceptions in pure code

92

Embed exceptions in values

Idea: embed exceptions in values

```
throw :: Exception -> a

divide :: Int -> Int -> Int
divide x y = if y==0 then throw DivZero
             else x/y
```

result type unchanged

A value is
- either an "ordinary" value
- or an "exception" value, carrying an exception

(Just like NaNs in IEEE floating point.)

In a lazy language an exception value might hide inside an un-evaluated data structure, but that's OK.

93

Catching exceptions

New primitive for catching exceptions: BAD BAD!

```
getException :: a -> ExVal a

data ExVal a = OK a
             | Bad Exception
```

Example

```
f x = case getException (goop x) of
         OK result -> result
         Bad exn   -> recovery_goop x
```

94

A well-known problem

What exception is raised by "+"?

```
(throw ex1) + (throw ex2)
```

Usual answer: fix evaluation order

BAD ENOUGH for call-by-value languages

- loss of code-motion transformations
- need for effect analyses

TOTAL CATASTROPHE for Haskell

- evaluation order is deliberately unspecified
- key optimisations depend on changing evaluation order

95

A cunning idea

Return *both* exceptions!

A value is
- either a "normal value"
- or an "exceptional value" containing a **set** of exceptions

Operationally, an exceptional value is
- represented by a single representative
- implemented by the usual stack-unwinding stuff

c.f. infinite lists:
semantically infinite, operationally finite

96

Semantics without exceptions

Denotations of Haskell types, ⟦τ⟧

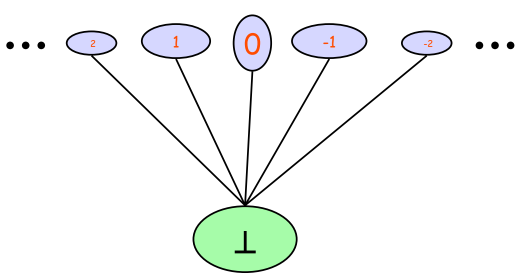
⟦Int⟧ = M **Z**
⟦ t1→t2⟧ = M (⟦t1⟧ → ⟦t2⟧)
⟦ (t1,t2) ⟧ = M (⟦t1⟧ × ⟦t2⟧)

$M\ t = t \cup \{\bot\}$

e.g. ⟦Int->Int⟧ = M (M Int -> M Int)

97

⟦Int⟧ : Ordinary Haskell



98

Semantics with exceptions

Denotations of Haskell types, ⟦τ⟧

⟦Int⟧ = M **Z**
⟦t1→t2⟧ = M (⟦t1⟧ → ⟦t2⟧)
⟦ (t1,t2) ⟧ = M (⟦t1⟧ × ⟦t2⟧)

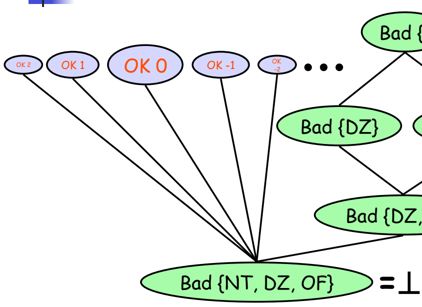
M t = {Ok x | x in t} ∪
{Bad s | s ⊆ **E**} ∪
{⊥}

E = {Overflow, DivZero, …}
Z = the integers

e.g. ⟦Int->Int⟧ = M (M Int -> M Int)

99

⟦Int⟧ : Exceptional Haskell



100

Semantics

$[\![e_1 + e_2]\!]$ = OK (m + n) if OK m = $[\![e_1]\!]$, OK n = $[\![e_2]\!]$

= Bad (S ($[\![e_1]\!]$) ∪ S($[\![e_2]\!]$)) otherwise

where S(Bad s) = s
S(OK n) = {}

Payoff: $[\![e_1 + e_2]\!] = [\![e_2 + e_1]\!]$

101

Whoa! What about getException?

Problem: which exception does getException choose from the set of possibilities?

```
getException :: a -> ExVal a    ?????
data ExVal a = OK a| Bad Exception
```

Solution 1: choose any. But that makes getException non-deterministic. And that loses even β-reduction!

```
let x = getException e in x==x           =    True
(getException e) == (getException e)     ≠    True
```

Verdict: Cure worse than disease.

102

Using the IO monad

Solution 2: put getException in the IO monad:

```
evaluate :: a -> IO a
```

`evaluate` evaluates its argument;

- if it is an ordinary value, it returns it
- if it is an exceptional value, it chooses one of the set of exceptions and raises it as an IO monad exception

103

Using the IO monad

Key idea:

The choice of which exception to raise is made in the IO monad, so it can be non-deterministic (like so much else in the IO monad)

```
evaluate :: a -> IO a
```

104

Using `evaluate`

```
main = do { i <- getInput;
            catch (do { r <- evaluate (goop i);
                        do_good_stuff r })
                  (\ ex -> recover_from ex)
          }
```

You have to be in the IO monad to use `evaluate`

You do **not** have to be in the IO monad to use `ioError`

105

Semantics

Add rules for **evaluate**

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V \quad M \not\equiv V}{\{E[\texttt{evaluate}\ M]\}_t \rightarrow \{E[\texttt{return}\ V]\}_t}$$

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{E[\texttt{evaluate}\ M]\}_t \rightarrow \{E[\texttt{ioError}\ e]\}_t}$$

An ordinary value

An exceptional value

106

Watch out!

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{E[M]\} \rightarrow \{E[V]\}}$$

We've just changed what values look like!

But what if M evaluates to (Bad S)???

107

Watch out!

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V \quad M \not\equiv V}{\{E[M]\}_t \rightarrow \{E[V]\}_t}\ (FUN1)$$

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{E[M]\}_t \rightarrow \{E[\texttt{ioError}\ e]\}_t}\ (FUN2)$$

Ordinary value

Exceptional value

108

Quiz

What does each of these programs do?

```
a1, a2, a3, a4, a5 :: IO ()
a1 = do { x <- evaluate 4; print x }
a2 = do { evaluate (head []); print "no" }
a3 = do { return (head []); print "yes" }
a4 = do { xs <- evaluate [1 `div` 0]; print (length xs) }
a5 = do { xs <- evaluate [1 `div` 0]; print (head xs) }
```

109

Imprecise exceptions

- A decent treatment of exceptions in purely-functional code
- Quite a lot more to say (see PLDI'99 paper)
- No transformations lost!
- Good for disaster recovery, poor for extra return values

110

Asynchronous exceptions

111

Asynchronous exceptions

A flexible form of asynchronous exception:

```
throw   :: Exception -> IO a
throwTo :: ThreadId -> Exception -> IO a
```

112

Timeouts

```
timeout :: Int -> IO a -> IO (Maybe a)
timeout n a
  = do { t <- myThreadId ;

         s <- forkIO (do { sleep n ;
                           throwTo t TimeOut });

         catch (do { r <- a;
                     throwTo s Kill;
                     return (Just r) });
               (\ex -> Nothing)
       }
```

Fork a thread that sleeps and then throws an exception to its parent

Do the action, and then kill the timeout

The timeout won!

113

Semantics

Add a rule for **throwTo**

Make sure we replace the **innermost** "current action"

$$\frac{M \neq (N_1 \texttt{>>=} N_2) \qquad M \neq (\texttt{catch}\ N_1\ N_2)}{\{\mathrm{E}_1[\texttt{throwTo}\ t\ e]\}_s \mid \{\mathrm{E}_2[M]\}_t \rightarrow \{\mathrm{E}_1[\texttt{return ()}]\}_s \mid \{\mathrm{E}_2[\texttt{ioError}\ e]\}_t}$$

Replace "current action" in target thread with ioError

114

What have we achieved?

115

Motivation

Functional programming is SO much fun.

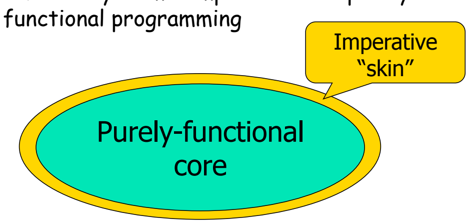
Plan of attack

1. Find an application
2. Try to write it in Haskell
3. Fail
4. Figure out how to fix Haskell
5. Abstract key ideas, write a paper
6. Repeat from (2)

116

What have we achieved?

- The ability to mix imperative and purely-functional programming



117

What have we achieved?

- ...without ruining either
- All laws of pure functional programming remain unconditionally true, even of actions

e.g. let x=e in ...x....x...

=

....e....e.....

118

What we have not achieved

- Imperative programming is as hard as it always was.

e.g. do { ...; x <- f 1; y <- f 2; ...}

?=?

do { ...; y <- f 2; x <- f 1; ...}

- ...but there's less of it!
- ...and actions are first-class values

119

Not covered in the lectures

...But in the notes

- Foreign language interfacing

120

What next?

- Write applications
- Real reasoning about monadic Haskell programs; proving theorems
- Alternative semantic models (trace semantics)
- More refined monads (the IO monad is a giant sin-bin at the moment)

121

What next?

http://research.microsoft.com/~simonpj
http://haskell.org

Have Lots More Fun!

122