Concurrent web service

```haskell
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

```
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
  - empty, or
  - holds a value of type t
```

What if two threads want to communicate? Or share data?

Example: keep a global count of how many client threads are running
- Increment count when spawning
- Decrement count when dying
Lazy functional programming for real

### Communication and sharing

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>takeMVar</strong></td>
<td>Block</td>
<td>Return contents, leave MVar empty</td>
</tr>
<tr>
<td><strong>putMVar</strong></td>
<td>Fill MVar</td>
<td>Block</td>
</tr>
</tbody>
</table>

#### Using MVars

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

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

#### Semantics

**Step 1:** elaborate the program state

- `P, Q, R := ...`
- `{M}`, A thread called `t`
- `{M}c`, An MVar called `m` containing `M`
- `{0}c`, An empty MVar called `m`

#### MVars as channels

An MVar directly implements:
- a shared data structure
- a one-place channel

#### Semantics

**Step 2:** a rule for forkIO

- `u \notin \{M, E\}`
- `{E[forkIO M]}t \rightarrow \rho_u(E[return u])` (FORK)

- Restrict the new thread name
- Return the thread name to the caller
- The new thread

#### Semantics

**Step 3:** rules for new, take, put

- `{E[takeMVar m]}c \rightarrow \rho_m(E[return M]) | \{M\}c` (FORK)
- `{E[putMVar M]}c \rightarrow \rho_m(E[return ()]) | \{M\}c`

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

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

A buffered channel

\[
\text{type Chan } a = (\text{MVar } (\text{Stream } a), \text{MVar } (\text{Stream } a))
\]

\[
\text{type Stream } a = \text{MVar } (\text{Item } a)
\]

\[
\text{data Item } a = \text{MItem } a (\text{Stream } a)
\]

A buffered channel

\[
\text{putChan :: Chan } a \rightarrow a \rightarrow \text{IO } ()
\]

\[
\begin{align*}
\text{putChan} & \mid \text{(read, write)} \mid \text{val} \\
& \text{= do } \{ \text{new_hole } \leftarrow \text{newEmptyMVar}; \\
& \quad \text{old_hole } \leftarrow \text{takeMVar write}; \\
& \quad \text{putMVar write new_hole}; \\
& \quad \text{putMVar old_hole (MkItem val new_hole)} \}
\end{align*}
\]

Summary

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

Exceptions
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!

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

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

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:

\[
\text{catch} : \text{IO a} \to (\text{IOError} \to \text{IO a}) \to \text{IO a}
\]

\[
\text{userError} : \text{String} \to \text{IOError}
\]

\[
\text{ioError} : \text{IOError} \to \text{IO a}
\]

\[
\text{catch} (\text{do} \{ h <- \text{openFile} "foo"; \text{processFile} h \}) \{ \text{w <- putStr "Oh dear"} \}
\]

Dynamic scope: exceptions raised in processFile are also caught

Semantics

Step 1: add a new evaluation context

\[
E ::= [] \mid E \gg M \mid \text{catch } E \text{ M}
\]

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

Step 2: add propagation rule for `ioError`

\[ \{E[\text{ioError } e \gg= M]\}_t \rightarrow \{E[\text{ioError } e]\}_t \]

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

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

Standard stack-unwinding implementation is possible

Step 3: add rules for `catch`

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

What to do if an exception is raised

What to do if an exception is not raised

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

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

Exceptions in values

Idea: embed exceptions in values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>throw</code></td>
<td>Exception -&gt; a</td>
</tr>
<tr>
<td><code>divide</code></td>
<td>Int -&gt; Int -&gt; Int</td>
</tr>
<tr>
<td><code>divide x y = if y==0 then throw DivZero else x/y</code></td>
<td></td>
</tr>
</tbody>
</table>

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.
Catching exceptions

New primitive for catching exceptions: BAD BAD!

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

Example

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

A well-known problem

What exception is raised by `*+`?

```haskell
(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

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

Semantics without exceptions

Denotations of Haskell types, \( \llbracket T \rrbracket \)

\[
\llbracket \text{Int} \rrbracket = M \mathbb{Z} \\
\llbracket t_1 \to t_2 \rrbracket = M \left( \llbracket t_1 \rrbracket \to \llbracket t_2 \rrbracket \right) \\
\llbracket (t_1, t_2) \rrbracket = M \left( \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \right)
\]

\( M \uparrow = \uparrow \cup \{ \bot \} \)

e.g. \( \llbracket \text{Int} \to \text{Int} \rrbracket = M(\mathbb{M} \text{Int} \to \mathbb{M} \text{Int}) \)

Semantics with exceptions

Denotations of Haskell types, \( \llbracket T \rrbracket \)

\[
\llbracket \text{Int} \rrbracket = M \mathbb{Z} \\
\llbracket t_1 \to t_2 \rrbracket = M \left( \llbracket t_1 \rrbracket \to \llbracket t_2 \rrbracket \right) \\
\llbracket (t_1, t_2) \rrbracket = M \left( \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \right)
\]

\( M \uparrow = \{ \text{Ok} x \mid x \in \uparrow \} \cup \{ \text{Bad} s \mid s \subseteq E \} \cup \{ \bot \} \)

e.g. \( \llbracket \text{Int} \to \text{Int} \rrbracket = M(\mathbb{M} \text{Int} \to \mathbb{M} \text{Int}) \)
Lazy functional programming for real

[Int] : Exceptional Haskell

- OK
- OK 0
- OK 1
- OK -1
- OK 2
- OK
- OK -2
- ⊥
- ...

Semantics

\[ [e_1 + e_2] = \text{OK } (m + n) \]
\[ \text{if OK } m = [e_1] \]
\[ \text{OK } n = [e_2] \]
\[ = \text{Bad } (S([e_1]) \cup S([e_2])) \]
otherwise

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

Payoff: \( [e_1 + e_2] = [e_2 + e_1] \)

Whoa! What about getException?

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

\[ \text{getException} :: a \rightarrow \text{ExVal } a \]
\[ \text{data ExVal } a = \text{OK } a | \text{Bad } \text{Exception} \]

Solution 1: choose any. But that makes getException non-deterministic. And that loses even \( \beta \)-reduction!

\[ \text{let } x = \text{getException } e \text{ in } x = x \]
\[ = \text{True} \]

Verdict: Cure worse than disease.

Using the IO monad

Solution 2: put getException in the IO monad:

\[ \text{evaluate} :: a \rightarrow \text{IO } a \]

\[ \text{evaluate} \text{ evaluates its argument:} \]
- 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

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)

\[ \text{evaluate} :: a \rightarrow \text{IO } a \]

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
Semantics

Add rules for evaluate

\[ E[M] = \text{Ok} \quad \forall V \quad M \neq V \]
\[ \{ E[Evaluate M] \}_{E} \quad \rightarrow \quad \{ E[\text{Return} \ V] \}_{E} \]

\[ E[M] = \text{Bad} \quad e \in S \]
\[ \{ E[Evaluate M] \}_{E} \quad \rightarrow \quad \{ E[\text{Err} \ e] \}_{E} \]

Watch out!

We've just changed what values look like!

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

Ordinary value

Watch out!

Exceptional value

\[ E[M] = \text{Ok} \quad M \neq V \]
\[ \{ E[M] \}_{E} \quad \rightarrow \quad \{ E[V] \}_{E} \] (FUN1)

\[ E[M] = \text{Bad} \quad e \in S \]
\[ \{ E[M] \}_{E} \quad \rightarrow \quad \{ E[\text{Err} \ e] \}_{E} \] (FUN2)

Quiz

What does each of these programs do?

\[ a_1, a_2, a_3, a_4, a_5 :: \text{IO} () \]
\[ a_1 = \text{do} \{ x <- \text{evaluate} 4; \text{print} \ x \} \]
\[ a_2 = \text{do} \{ \text{evaluate} (\text{head} []); \text{print} \ "no" \} \]
\[ a_3 = \text{do} \{ \text{return} (\text{head} []); \text{print} \ "yes" \} \]
\[ a_4 = \text{do} \{ x <- \text{evaluate} [1 \div 0]; \text{print} \ (\text{length} \ x) \} \]
\[ a_5 = \text{do} \{ x <- \text{evaluate} [1 \div 0]; \text{print} \ (\text{head} \ x) \} \]

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

Asynchronous exceptions
Asynchronous exceptions

A flexible form of asynchronous exception:

\[
\begin{align*}
\text{throw} & : \text{Exception} \to \text{IO a} \\
\text{throwTo} & : \text{ThreadId} \to \text{Exception} \to \text{IO a}
\end{align*}
\]

Timeouts

\[
\text{timeout} :: \text{Int} \to \text{IO a} \to \text{IO (Maybe a)}
\]

\[
\text{timeout} \ n \ a = \text{do} \{ t \leftarrow \text{myThreadId} ; \\
\quad s \leftarrow \text{forkIO} \{ \text{do} \{ \text{sleep n} ; \\
\quad \text{throwTo} \ t \ \text{TimeOut} \} \} ; \\
\quad \text{catch} \{ \text{do} \{ x \leftarrow a ; \\
\quad \text{throwTo} \ s \ \text{Kill} ; \\
\quad \text{return} \ (\text{Just x}) \} ; \\
\quad \{ \text{ex \to Nothing} \} \}
\}
\]

Semantics

Add a rule for throwTo

\[
\begin{align*}
M \neq (N_1 >> N_2) & \quad M \neq (\text{catch } N_1 \ N_2) \\
(E,(\text{throwTo } t \ e)) \cdot (E,(\text{return } ())) & \rightarrow (E,[(\text{killError } e)])
\end{align*}
\]

What have we achieved?

- The ability to mix imperative and purely-functional programming

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)

What have we achieved?

- The ability to mix imperative and purely-functional programming
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.....

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

Not covered in the lectures
- ...But in the notes
- Foreign language interfacing

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)

What next?
- http://research.microsoft.com/~simonpj
- http://haskell.org

Have Lots
More Fun!