Concurrent Orchestration in Haskell

John Launchbury
Trevor Elliott
foo :: (a -> s -> s) -> s -> Orc a -> Orc s
foo f s p = do a <- newMVarM s
               x <- p
               v <- takeMVarM a
               let w = f x v
               putMVarM a w
               return w

This code implements a well-known idiom — as we go on, try to figure out what it is...
• Concurrent scripting
• Laws
• Thread management
Testing Xen Virtual Machines

- Tester talks with each of the VMs concurrently
- Many possible behaviors are “correct” / “incorrect”
- Timeouts, VMs dying, etc.
- Subtle concurrency bugs in test framework
fplang :: Orc String
fplang = return “Haskell” <|> return “ML” <|> return “Scheme”
metronome :: Orc ()
metronome = return () <|> (delay 2.5 >> metronome)
quotes :: Query -> Query -> Orc Quote
quotes srcA srcB = do
    quoteA <- eagerly $ getQuote srcA
    quoteB <- eagerly $ getQuote srcB
    cut ( (return least <*> quoteA <*> quoteB)
            <|> (quoteA >>= threshold)
            <|> (quoteB >>= threshold)
            <|> (delay 25 >> (quoteA <|> quoteB))
            <|> (delay 30 >> return noQuote))

least x y = if price x < price y then x else y
threshold x = guard (price x < 300) >> return x

Need to book a ticket, under $300 if possible…
quote
queens = fmap show (extend []) <|> return ("Computing 8-queens...")

extend :: [Int] -> Orc [Int]
extend xs = if length xs == 8 then return xs else do
  j <- listOrc [1..8]
  guard $ not (conflict xs j)
  extend (j:xs)

conflict :: [Int] -> Int
conflict = ...

listOrc :: [a] -> Orc a
listOrc = foldr (<|>) stop . map return
*Main> printOrc (queens)
Ans = "Computing 8-queens...
Ans = "[5,7,1,3,8,6,4,2]"
Ans = "[5,2,4,7,3,8,6,1]"
Ans = "[6,4,2,8,5,7,1,3]"
Ans = "[5,3,8,4,7,1,6,2]"
Ans = "[4,2,7,3,6,8,5,1]"
:

*Main> printOrc (queens)
Ans = "Computing 8-queens...
Ans = "[4,2,7,3,6,8,5,1]"
Ans = "[6,4,7,1,8,2,5,3]"
Ans = "[3,6,8,1,4,7,5,2]"
Ans = "[3,6,4,2,8,5,7,1]"
Ans = "[2,7,3,6,8,5,1,4]"
:

Orc Example
baseball :: Orc (String, String)

baseball = do

    team <- prompt "Name a baseball team"
    \`after\` (12, return "Yankees")
    <|> prompt "Name another team"
    \`notBefore\` 10
    <|> (delay 8 >> return "Mariners")

    agree <- prompt ("Do you like " ++ team ++ "?"")
    \`after\` (20, guard (team /= "Mets") >> return "maybe")

    return (team, agree)
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
    `after` (12, return "Yankees")
  <|> prompt "Name another team"
    `notBefore` 10
  <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
    `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
baseball :: Orc (String, String)
baseball = do

  team <- prompt "Name a baseball team"
  `after` (12, return "Yankees")
  <|> prompt "Name another team"
  `notBefore` 10
  <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like " ++ team ++ "?"")
  `after` (20, guard (team /= "Mets") >> return "maybe")

  return (team, agree)
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
    `after` (12, return "Yankees")
  <|> prompt "Name another team"
    `notBefore` 10
  <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
    `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
  `after` (12, return "Yankees")
  <|> prompt "Name another team"
  `notBefore` 10
  <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like " ++ team ++ "?"")
  `after` (20, guard (team /= "Mets") >> return "maybe")

  return (team, agree)
foo :: (a -> s -> s) -> s -> Orc a -> Orc s
foo f s p = do a <- newMVarM s
    x <- p
    v <- takeMVarM a
    let w = f x v
    putMVarM a w
    return w
scan :: (a -> s -> s) -> s -> Orc a -> Orc s
scan f s p = do a <- newMVarM s
  x <- p
  v <- takeMVarM a
  let w = f x v
  putMVarM a w
  return w

% printOrc (scan (+) 0 $ listOrc [1,2,3,4,5])
scan :: (a -> s -> s) -> s -> Orc a -> Orc s
scan f s p = do a <- newMVarM s
    x <- p
    v <- takeMVarM a
    let w = f x v
    putMVarM a w
    return w

% printOrc (scan (+) 0 $ list0rc [1,2,3,4,5])
Ans = 1
Ans = 3
Ans = 6
Ans = 11
Ans = 15
%
### Layered Implementation

<table>
<thead>
<tr>
<th>Transition Semantics</th>
<th>Orc Scripts</th>
<th>Orc Monad</th>
<th>HIO Monad</th>
<th>IO Monad</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>multiple results</td>
<td>thread control</td>
<td>external effects</td>
</tr>
</tbody>
</table>

- **Layered implementation — layered semantics**
  - Properties at one level depend on properties at the level below

- **What properties should Orc terms satisfy?**
  - Hence, what properties should be built into HIO?

- **Unresolved question: what laws should the basic operations of the IO monad satisfy?**
type Orc a = (a -> HIO ()) -> HIO ()

return x = \k -> k x

p >>= h = \k -> p (\x -> h x k)

p <|> q = \k -> fork (p k) >> q k

stop = \k -> return ()

runOrc p = p (\x -> return ())
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x

p >>= h = \k -> p (\x -> h x k)

p <|> q = \k -> fork (p k) >> q k
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x

p >>= h = \k -> p (\x -> h x k)

p <|> q = \k -> fork (p k) >> q k
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x

p >>= h = \k -> p (\x -> h x k)

p <|> q = \k -> fork (p k) >> q k
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x

p >>= h = \k -> p (\x -> h x k)

p <|> q = \k -> fork (p k) >> q k
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
    r <- newEmptyMVarM
    forkM (p (putMVarM r))
    k (\k' -> readMVarM r >>= k')

• Give p a continuation that will store its result
• Return the “value” that accesses that result for the then current continuation
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
    r <- newEmptyMVarM

    forkM (p `saveOnce` (r ))
    k (\k' -> readMVarM r >>= k')

saveOnce :: Orc a -> (MVar a ) -> HIO ()
p `saveOnce` (r ) = do
    putMVarM r x

• Give p a continuation that will store its result (but once only even if duplicated)
• Return the “value” that accesses that result for the then current continuation
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
    r <- newEmptyMVarM
    forkM (p `saveOnce` (r ))
    k (\k’ -> readMVarM r >>= k’)

saveOnce :: Orc a -> (MVar a) -> HIO ()
p `saveOnce` (r ) = do
    ticket <- newMVarM ()
    p (\x -> takeMVarM ticket >> putMVarM r x)

- Give p a continuation that will store its result (but once only even if duplicated)
- Return the “value” that accesses that result for the then current continuation
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
    r <- newEmptyMVarM
    e <- newLocality
    local e $ forkM (p `saveOnce` (r,e))
    k (\k' -> readMVarM r >>= k')

saveOnce :: Orc a -> (MVar a,Locality) -> HIO ()
p `saveOnce` (r,e) = do
    ticket <- newMVarM ()
    p (\x -> takeMVarM ticket >> putMVarM r x >> close e)

• Give p a continuation that will store its result (but once only even if duplicated)
• Return the “value” that accesses that result for the then current continuation
• Thread management can be carried over too
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
    po <- eagerly p
    qo <- eagerly q
    return f <*> po <*> qo

notBefore:: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)

- Entering the handle waits for the result
- Synchronization
- cut
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
  po <- eagerly p
  qo <- eagerly q
  return f <*> po <*> qo

notBefore :: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)

cut :: Orc a -> Orc a
cut p = do
  po <- eagerly p
  po
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
  po <- eagerly p
  qo <- eagerly q
  return f <*> po <*> qo

notBefore:: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)

cut:: Orc a -> Orc a
cut p = do
  po <- eagerly p
  po

• Entering the handle waits for the result
• Synchronization
• cut
Orc Laws

Left-Return: \((\text{return } x >>= k) = k x\)

Right-Return: \((p >>= \text{return}) = p\)

Bind-Associativity: \(((p >>= k) >>= h) = (p >>= (k >>= h))\)

Stop-Identity: \(p <|> \text{stop} = p\)

Par-Commutativity: \(p <|> q = q <|> p\)

Par-Associativity: \(p <|> (q <|> r) = (p <|> q) <|> r\)

Left-Zero: \((\text{stop} >>= k) = \text{stop}\)

Par-Bind: \(((p <|> q) >>= k) = ((p >>= k) <|> (q >>= k))\)
Non-Laws

Bind-Par?: \[ p \implies (\lambda x \rightarrow h x \langle \mid \rangle k x) = (p \implies h) \langle \mid \rangle (p \implies k) \]

Right-Zero?: \[ p \implies \text{stop} = \text{stop} \]
Non-Laws

**Bind-Par?:**  \[ p \ggg (\lambda x \rightarrow h x \left<\leftarrow\right> k x) = (p \ggg h) \left<\leftarrow\right> (p \ggg k) \]

**Right-Zero?:**  \[ p \gg stop = stop \]

\[ p \ `\text{until}` \ done = \text{cut} (\text{silent } p \left<\leftarrow\right> \text{done}) \]

\[ \text{silent } p = p \gg stop \]
Non-Laws

**Bind-Par?:**
\[
p >>= (\ x \to \ h \ x \ <|> \ k \ x) = (p >>= h) <|> (p >>= k)
\]

**Right-Zero?:**
\[
p >> \text{stop} = \text{stop}
\]

\[
p \text{ `until` done = cut (silent p <|> done)}
\]

\[
silent \ p = p >> \text{stop}
\]

\[
hassle = (\text{metronome} >> \text{email}("Simon","Hey!"))
\]
\[
\ `\text{until}`
\]
\[
(delay 60 >> \text{return} ())
\]
Eagerly Laws

Eagerly-Par: \[ \text{eagerly } p >>= (\lambda x \to k \ x \ <|> h) = (\text{eagerly } p >>= k) <|> h \]

Eagerly-Swap:
\[
\begin{align*}
do \ y \leftarrow \text{eagerly } p & = do \ x \leftarrow \text{eagerly } q \\
x \leftarrow \text{eagerly } q & \quad y \leftarrow \text{eagerly } p \\
\text{return } (x,y) & \quad \text{return } (x,y)
\end{align*}
\]

Eagerly-IO: \[ \text{eagerly } (\text{ioOrc } m) >> p = (\text{ioOrc } m >> \text{stop}) <|> p \]
The implementation of `val` (the alternative that uses lazy thunks) is almost identical:

\[
\begin{align*}
\text{val} & : \text{Orc} \ a \rightarrow \text{Orc} \ a \\
\text{val} \ p & = \ \lambda k \rightarrow \text{do} \\
& \hspace{1cm} r \leftarrow \text{newEmptyMVarM} \\
& \hspace{1cm} e \leftarrow \text{newLocality} \\
& \hspace{1cm} \text{local} \ e \ \$ \ \text{forkM} \ (p \ \text{`saveOnce`} \ (r,e)) \\
& \hspace{1cm} k \ (\text{unsafePerformIO} \ \$ \ \text{readMVarM} \ r)
\end{align*}
\]

\[
\begin{align*}
\text{saveOnce} & : \text{Orc} \ a \rightarrow (\text{MVar} \ a, \text{Locality}) \rightarrow \text{HIO} \ () \\
p \ \text{`saveOnce`} \ (r,e) & = \ \text{do} \\
& \hspace{1cm} \text{ticket} \leftarrow \text{newMVarM} () \\
& \hspace{1cm} p \ (\lambda x \rightarrow \text{takeMVarM} \ \text{ticket} \ \gg \ \text{putMVarM} \ r \ x \ \gg \ \text{close} \ e)
\end{align*}
\]

- The implementation of `val` (the alternative that uses lazy thunks) is almost identical.
quotesVal :: Query -> Query -> Orc Quote
quotesVal srcA srcB = do
  quoteA <- val $ getQuote srcA
  quoteB <- val $ getQuote srcB
  cut ( publish (least quoteA quoteB)
       <|> (threshold quoteA)
       <|> (threshold quoteB)
       <|> (delay 25 >> (publish quoteA <|> publish quoteB))
       <|> (delay 30 >> return noQuote))

publish :: NFData a => a -> Orc a
publish x = deepseq x $ return x

- Good: use the lazy values directly
- Bad: have to be careful about evaluation
HIO Monad

- Don’t want the programmer to have to do explicit thread management
  - Nested groups of threads
- Want richer equational theory than IO
  - e.g. by managing asynchronous exceptions
- Don't want the programmer to have to do explicit thread management
  - Nested groups of threads
- Want richer equational theory than IO
  - e.g. by managing asynchronous exceptions

newtype HIO a = HIO {inGroup :: Locality -> IO a}

type Group   {- abstract -}
data Entry  = Thread ThreadId
                  | Group Group

newGroup  :: IO Group
register  :: Entry -> Group -> IO ()
killGroup :: Group -> IO ()
first :: Int -> Orc a -> Orc a
first n p = do
    vals <- newEmptyMVarM
    end <- newEmptyMVarM
    echo n vals end
        <|> silent (generate p vals end)

generate p vals end =
    (p >>= putMVarM vals) `until` takeMVarM end

echo n vals end = loop n
    where loop 0 = silent $ putMVarM end ()
            loop n = do x <- takeMVarM vals
                        return x <|> loop (n-1)

- Use MVars to communicate
- Use `until` to kill-off work when finished
**Standard function:**

```haskell
filterM _ []     =  return []
filterM p (x:xs) =  do
  b <- p x
  ys  <- filterM p xs
  return (if b then x:ys else ys)
```

`baz :: [a] -> Orc [a]`
`baz xs = filterM pred xs`

`pred x = return False <|> return True`