Concurrent Orchestration in Haskell

John Launchbury and Trevor Elliott, Haskell '10, Galois Inc.

Colloquium presentation by Ruben de Gooijer

Department of Information and Computing Sciences
University of Utrecht, the Netherlands

November 3, 2011

$$
\begin{array}{l}
\mathit{either} :: \mathit{IO}\ a \to \mathit{IO}\ b \to \mathit{IO}\ (\mathit{Either}\ a\ b) \\
\mathit{either}\ p\ q = \mathbf{do} \\
\quad m \leftarrow \mathit{newEmptyMVar} \\
\quad \mathit{block}\ \$\ \mathbf{do} \\
\qquad \mathit{pId} \leftarrow \mathit{forkIO}\ (\mathit{unblock}\ \$\ p\ \texttt{>>=}\ \mathit{putMVar}\ m \circ \mathit{Left}) \\
\qquad \mathit{qId} \leftarrow \mathit{forkIO}\ (\mathit{unblock}\ \$\ q\ \texttt{>>=}\ \mathit{putMVar}\ m \circ \mathit{Right}) \\
\qquad r \leftarrow \mathit{takeMVar}\ m \\
\qquad \mathit{killThread}\ \mathit{pId} \\
\qquad \mathit{killThread}\ \mathit{qId} \\
\qquad \mathit{return}\ r
\end{array}
$$

What's wrong with plain Haskell? Can we do better?

Yes we can!

The authors have implemented the Orc language as a Haskell library.

Orc is a *concurrent scripting* DSL which allows concurrency problems to be expressed at a conceptually higher level using:

- primitive combinators
- and automated thread management

Code Example

Definition

Concurrent scripting or *orchestration*: the act of coordinating several external actions whose timing and interleaving is unpredictable.

For example:

$$\begin{array}{l} \textit{search} = \textbf{do} \\ \quad \textit{term} \leftarrow \textit{prompt}\ \texttt{"Enter a search term: "} \\ \quad \textit{cut}\ (\textit{bing term}\ <|>\ \textit{google term}) \end{array}$$

- External actions: prompt, bing, google
- Nondeterminism

Either Revisited

$$either\ p\ q = \\ \quad cut\ ((p \mathbin{>\!\!>\!\!=} return \circ Left) \mathbin{<\!|\!>} (q \mathbin{>\!\!>\!\!=} return \circ Right))$$

The programmer doesn't need to worry about:

- Locks
- Thread management
- Asynchronous exceptions

Overview

Presentation outline.

- The EDSL
- The Implementation
- Conclusion

The EDSL

The EDSL

From a bird's-eye-view Orc is a combination of three things:

- **Many-valued concurrency**
  Concurrent computations may produce zero or more results.
- **External actions (effects)**
  Calling webservices, making phone calls, launching missiles...
- **Managed resources**
  Threads are accounted for.

The EDSL

In Orc *external actions* are represented by *sites*.

Definition

Sites are external actions that may produce zero or more results, or halt.

If p is a site then *halt* is defined as:

- All sites called by `p` have either responded or halted.
- p will never call any more sites.
- p will never publish any more values.

To represent both *sites* and Orc computations a new type constructor is introduced:

$$\textbf{newtype}\ \mathit{Orc}\ a = \mathit{Orc}\ \{\ldots\}$$

A value of type *Orc a* *may* produce *zero or more results* of type a, and *may* have *many* effects.

The EDSL

To orchestrate *sites* Orc defines a small set of primitive combinators:

- sequential composition
- parallel composition
- pruning
- otherwise

The authors of Orc have shown that these primitives are sufficiently expressive to solve many concurrency problems.[1]

We'll now see how these primitives translate to Haskell...

[1] *Workflow Patterns in Orc, Cook, et al, Proceedings of the 8th International Conference, COORDINATION 2008, Coordination Models and Languages*

The EDSL

Sequential composition.

$$(\ \texttt{>>=} \ ) :: \mathit{Orc}\ a \rightarrow (a \rightarrow \mathit{Orc}\ b) \rightarrow \mathit{Orc}\ b$$

- Orc is made an instance of the Monad class
- The monad laws hold

We may use do-notation to express *sequential composition*.

The EDSL

For example:

$$\begin{aligned}&\textbf{do}\ x \leftarrow p\\ &\quad\ y \leftarrow q\ x\\ &\quad\ h\ x\ y\end{aligned}$$

The expression reads as nested iteration:

foreach x produced by p
    foreach y produced by $(q\ \ x)$
        produce all values of $(h\ \ x\ \ y)$

Nested iteration is similar to the List monad. But now:

- the ordering of results is nondeterministic
- all iterations are run concurrently

The EDSL

We can use `stop` to end the local thread of execution.

$$stop :: Orc\ a$$

For example:

$$\begin{aligned}&\textbf{do}\ x \leftarrow p\\&\quad\ \ y \leftarrow stop\\&\quad\ \ h\ x\ y\end{aligned}$$

which is equivalent to:

$$p \gg stop$$

Parallel composition.

$$<\!|\!> \;\; :: \mathit{Orc}\ a \rightarrow \mathit{Orc}\ a \rightarrow \mathit{Orc}\ a$$

For example:

$$p \;<\!|\!>\; q$$

- Executes p and q in parallel and returns all the results produced by both as the become available.
- Results of p and q may be arbitrarily interleaved.

The EDSL - Code Example

A simple program that combines parallel and sequential composition to be of a constant nuisance to John.

$$
\begin{aligned}
&\mathit{emailJohn} :: \mathit{Orc}\ ()\\
&\mathit{emailJohn} = \mathit{email}\ \texttt{"john.doe@undefined.com"}\\
&\quad <\!|\!>\ (\mathit{delay}\ 2\ >>\ \mathit{emailJohn})
\end{aligned}
$$

The EDSL

Managing concurrency.

$$eagerly :: Orc\ a \to Orc\ (Orc\ a)$$

For example:

$$eagerly\ p$$

- Execute p in a separate thread and return a handle to its first result.
- Once the first result is produced the redundant concurrency gets pruned.

$$\begin{aligned} cut\ p &= join \circ eagerly \\ &\equiv \mathbf{do}\ x \leftarrow eagerly\ p \\ &\qquad\quad x \end{aligned}$$

Fork-join:

$$
\begin{array}{l}
\mathit{sync} :: (a \to b \to c) \to \mathit{Orc}\ a \to \mathit{Orc}\ b \to \mathit{Orc}\ c \\
\mathit{sync}\ f\ p\ q = \mathbf{do} \\
\quad \mathit{po} \leftarrow \mathit{eagerly}\ p \\
\quad \mathit{qo} \leftarrow \mathit{eagerly}\ q \\
\quad \mathit{pure}\ f\ \ \texttt{<*>}\ \ \mathit{po}\ \ \texttt{<*>}\ \ \mathit{qo}
\end{array}
$$

The EDSL - Code Example

Fork-join:

$$
\begin{aligned}
&sync :: (a \to b \to c) \to Orc\ a \to Orc\ b \to Orc\ c\\
&sync\ f\ p\ q = \mathbf{do}\\
&\quad po \leftarrow eagerly\ p\\
&\quad qo \leftarrow eagerly\ q\\
&\quad pure\ f\ <\!\!*\!\!>\ po\ <\!\!*\!\!>\ qo
\end{aligned}
$$

$$
\begin{aligned}
&notBefore :: Orc\ a \to Float \to Orc\ a\\
&p\ `notBefore`\ w = sync\ const\ p\ (delay\ w)
\end{aligned}
$$

The EDSL

In Orc *sites* are lifted by the compiler or implemented externally.

$$\begin{aligned} \mathit{return} &:: a \rightarrow \mathit{Orc}\ a \\ \mathit{liftIO} &:: \mathit{IO}\ a \rightarrow \mathit{Orc}\ a \end{aligned}$$

In Haskell *sites* are simply liftings of either:

- pure (*return*) or
- *IO* computations (*liftIO*).

For example:

$$\mathit{google}\ \mathit{term} = \mathit{liftIO}\ \$\ \mathit{httpGet}\ (\texttt{"google.com?s="} + t)$$

The EDSL

The canonical way to run Orc computations is using `runOrc`.

$$runOrc :: Orc\ a \rightarrow IO\ ()$$

For example:

$$runOrc\ \$\ google\ \texttt{"But what does it mean?"}$$

Note: the unit type of IO.

The EDSL - Code Example

All combinators thus far inherit non-termination from their arguments.

As a remedy the Orc library provides a *timeout* combinator.

$$\mathit{timeout} :: \mathit{Float} \to a \to \mathit{Orc}\ a \to \mathit{Orc}\ a$$
$$\mathit{timeout}\ n\ a\ p = \mathit{cut}\ (p\ \texttt{<|>}\ (\mathit{delay}\ n\ \texttt{>>}\ \mathit{return}\ a))$$

$$> \mathit{timeout}\ 3\ \texttt{"No results."}\ (\mathit{google}\ \texttt{"Haskell"})$$

The Implementation

The Implementation

Orc is implemented by stacking monads on top of each other.

The implementation stack (bottom to top).

- **The IO monad**
  External actions (effects)
- **The hierarchical IO monad (HIO)**
  Managed resources, i.e. automated thread management
- **The Orc monad**
  Multiple results

All layers are made instances of *MonadIO*.
To move up a layer use *liftIO*.

The Implementation - HIO

- **The goal**
  Implement automated thread management for Orc computations.

The Implementation - HIO

- **The goal**
  Implement automated thread management for Orc computations.

- **The problem**
  The concurrency introduced inside the *IO* monad is not accounted for.

  We are left without a handle to kill redundant threads.

The Implementation - HIO

- **The goal**
  Implement automated thread management for Orc computations.

- **The problem**
  The concurrency introduced inside the *IO* monad is not accounted for.

  We are left without a handle to kill redundant threads.

- **The solution**
  Extend the *IO* monad with an environment that can be used to do the bookkeeping of running threads.

Computations in the HIO monad are represented by:

$$\textbf{newtype}\ \mathit{HIO}\ a = \mathit{HIO}\ \{\mathit{inGroup} :: \mathit{Group} \to \mathit{IO}\ a\}$$

isomorphic to *ReaderT Group IO a*.

$$\begin{aligned}
&\textbf{type}\ \mathit{Group} && = (\mathit{TVar}\ \mathit{NumberOfActiveThreads},\\
& && \quad \mathit{TVar}\ \mathit{Inhabitants})\\
&\textbf{data}\ \mathit{Inhabitants} && = \ \ldots\\
&\textbf{data}\ \mathit{Entry} && = \mathit{Thread}\ \mathit{ThreadId}\\
& && \mid\ \mathit{Group}\ \mathit{Group}
\end{aligned}$$

The Implementation - HIO

$$
\begin{array}{l}
\textbf{instance}\ Monad\ HIO\ \textbf{where} \\
\quad return\ x = HIO\ \$\ \lambda w \rightarrow return\ x \\
\quad m\ \mathbin{>\!\!>\!\!=}\ k = HIO\ \$\ \lambda w \rightarrow \textbf{do} \\
\qquad\qquad\qquad x \leftarrow m\ \text{`}inGroup\text{`}\ w \\
\qquad\qquad\qquad k\ x\ \text{`}inGroup\text{`}\ w
\end{array}
$$

Think Monad Reader.

The Implementation - HIO

We use an overloaded version of *forkIO* such that we may do some bookkeeping at the spawning and killing of threads.

```
instance HasFork HIO where
  fork hio = HIO $ λw → block $ do
    increment w
    fork (block (do tid ←   myThreadId
                            register (Thread tid) w
                            unblock (hio `inGroup` w))
          `finally`
            decrement w)
```

Note: block is deprecated, replaced by mask.

The Implementation - HIO

HIO computations can be run inside the IO monad.

$$
\begin{aligned}
&\mathit{runHIO} :: \mathit{HIO}\ b \rightarrow \mathit{IO}\ ()\\
&\mathit{runHIO}\ \mathit{hio} = \mathbf{do}\\
&\quad w \leftarrow \mathit{newPrimGroup}\\
&\quad r \leftarrow \mathit{hio}\ `\mathit{inGroup}`\ w\\
&\quad \mathit{isZero}\ w\\
&\quad \mathit{return}\ ()
\end{aligned}
$$

isZero: wait until all threads in the *Group* have *halted*.

The Implementation - Orc Monad

- **The goal**
  Allow the definition of Orc computations that produce zero or more results. For example:

$$((p \ \texttt{<|>} \ q) \ \texttt{>>=} \ k) :: \mathit{Orc} \ a$$

- **The problem**
  As the results of p and q become available we need to somehow pass them along to the rest of the program k.

- **The goal**
  Allow the definition of Orc computations that produce zero or more results. For example:

  $$((p \;\texttt{<|>}\; q) \;\texttt{>>=}\; k) :: \mathit{Orc}\ a$$

- **The problem**
  As the results of `p` and `q` become available we need to somehow pass them along to the rest of the program k.

- **The solution**
  Write functions that produce results in *continuation-passing-style* (CPS).

  Enables functions to pass their results to the future of the program.

Short detour into CPS...

$$multiply :: Int \to Int \to Int$$
$$multiply\ x\ y = x * y$$

Short detour into CPS...

$$multiply :: Int \to Int \to Int$$
$$multiply\ x\ y = x * y$$

Written in *continuation-passing-style*:

$$multiply :: Int \to Int \to (Int \to r) \to r$$
$$multiply\ x\ y\ k = k\ (x * y)$$

The Implementation - Orc Monad

Short detour into CPS...

$$multiply :: Int \to Int \to Int$$
$$multiply\ x\ y = x * y$$

Written in *continuation-passing-style*:

$$multiply :: Int \to Int \to (Int \to r) \to r$$
$$multiply\ x\ y\ k = k\ (x * y)$$

Usage:

$$> multiply\ 3\ 4\ (putStrLn \circ show)$$
$$> 12$$

Short detour into CPS...

$$multiply :: Int \to Int \to Int$$
$$multiply\ x\ y = x * y$$

Written in *continuation-passing-style*:

$$multiply :: Int \to Int \to (Int \to r) \to r$$
$$multiply\ x\ y\ k = k\ (x * y)$$

Usage:

$$> multiply\ 3\ 4\ (putStrLn \circ show)$$
$$> 12$$

The **current** computation $multiply\ 3\ 4$ decides what it does with the **future** of the computation $putStrLn \circ show$!

The Implementation - Orc Monad

The implementation of the Orc type constructor.

$$\textbf{newtype}\ \mathit{Orc}\ a = \mathit{Orc}\ \{\mathit{withCont} :: (a \to \mathit{HIO}\ ()) \to \mathit{HIO}\ ()\}$$

- Newtype abstracts from explicit continuation passing
- The Orc constructor is not exposed. Only primitive combinators are allowed access to the current continuation
- The representation of the future of the program is not polymorphic. In the end Orc computations should always run inside *HIO*.
- Isomorphic to $\mathit{ContT}\ ()\ \mathit{HIO}\ a$.

The Implementation - Orc Monad

Hiding the *current continuation* inside the Orc type.

From explicit CPS:

$$\begin{aligned} &multiply &&:: Int \to Int \to (Int \to r) \to r \\ &multiply\ x\ y\ k &&= k\ (x * y) \end{aligned}$$

to implicit CPS using Orc:

$$\begin{aligned} &multiply &&:: Int \to Int \to Orc\ Int \\ &multiply\ x\ y &&= Orc\ \$\ \lambda k \to k\ (x * y) \end{aligned}$$

The Implementation - Orc Monad

The implementation of the primitive combinators.

$$return\ x = \lambda k \to k\ x$$

$$p\ \ {>}{>}{=}\ \ h = \lambda k \to p\ (\lambda x \to h\ x\ k)$$

$$p\ \ {<}|{>}\ \ q = \lambda k \to fork\ (p\ k)\ \ {>}{>}\ \ q\ k$$

$$stop = \lambda k \to return\ ()$$

$$runOrc\ p = runHIO\ \$\ p\ (\lambda x \to return\ ())$$

Equational reasoning.

$$
\begin{aligned}
& \mathit{return}\ \texttt{"Hello"}\ \texttt{>>=}\ \mathit{const}\ \mathit{stop} \\
\equiv\quad & \{\ \text{definition of}\ \texttt{>>=}\ \} \\
& \lambda k \rightarrow \mathit{return}\ \texttt{"Hello"}\ (\lambda x \rightarrow (\mathit{const}\ \mathit{stop})\ x\ k) \\
\equiv\quad & \{\ \text{definition of}\ \mathit{return}\ \} \\
& \lambda k \rightarrow (\lambda k \rightarrow k\ \texttt{"Hello"})\ (\lambda x \rightarrow (\mathit{const}\ \mathit{stop})\ x\ k) \\
\equiv\quad & \{\ \text{beta-reduction}\ \} \\
& \lambda k \rightarrow (\lambda x \rightarrow (\mathit{const}\ \mathit{stop})\ x\ k)\ \texttt{"Hello"} \\
\equiv\quad & \{\ \text{beta-reduction}\ \} \\
& \lambda k \rightarrow (\mathit{const}\ \mathit{stop})\ \texttt{"Hello"}\ k \\
\equiv\quad & \{\ \text{definition of}\ \mathit{const}\ \} \\
& \lambda k \rightarrow \mathit{stop}\ k
\end{aligned}
$$

The Implementation - Orc Monad

$$
\begin{aligned}
&\mathit{eagerly} \quad :: \mathit{Orc}\ a \rightarrow \mathit{Orc}\ (\mathit{Orc}\ a)\\
&\mathit{eagerly}\ p = \mathit{Orc}\ \$\ \lambda k \rightarrow \mathbf{do}\\
&\quad \mathit{result} \quad \leftarrow \mathit{newEmptyMVar}\\
&\quad w \qquad\quad \leftarrow \mathit{newGroup}\\
&\quad \mathit{local}\ w\ \ \$\ \ \mathit{fork}\ (p\ \text{`}\mathit{saveOnce}\text{`}\ (\mathit{result}, w))\\
&\quad k\ (\mathit{liftIO}\ \$\ \mathit{readMVar}\ \ \mathit{result})
\end{aligned}
$$

The Implementation - Orc Monad

$$
\begin{aligned}
&\mathit{eagerly} \quad :: \mathit{Orc}\ a \to \mathit{Orc}\ (\mathit{Orc}\ a)\\
&\mathit{eagerly}\ p = \mathit{Orc}\ \$\ \lambda k \to \mathbf{do}\\
&\quad \mathit{result} \leftarrow \mathit{newEmptyMVar}\\
&\quad w \leftarrow \mathit{newGroup}\\
&\quad \mathit{local}\ w\ \$\ \mathit{fork}\ (p\ `\mathit{saveOnce}`\ (\mathit{result}, w))\\
&\quad k\ (\mathit{liftIO}\ \$\ \mathit{readMVar}\ \mathit{result})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{saveOnce} \quad :: \mathit{Orc}\ a \to (\mathit{MVar}\ a, \mathit{Group}) \to \mathit{HIO}\ ()\\
&p\ `\mathit{saveOnce}`\ (r, w) = \mathbf{do}\\
&\quad \mathit{ticket} \leftarrow \mathit{newMVar}\ ()\\
&\quad p \# \lambda x \to\\
&\qquad (\mathit{takeMVar}\ \mathit{ticket} >\!> \mathit{putMVar}\ r\ x >\!> \mathit{close}\ w)
\end{aligned}
$$

readMVar: is atomic only if there are **no** other producers for the *result* MVar.

The Implementation - Orc Monad

In Orc *eagerly* is lazy in the binding of its results.

$$
\begin{aligned}
&\mathit{eagerlyLazy} :: \mathit{Orc}\ a \to \mathit{Orc}\ a \\
&\mathit{eagerlyLazy}\ p = \mathit{Orc}\ \$\ \lambda k \to \mathbf{do} \\
&\quad \ldots \\
&\quad k\ (\mathit{unsafePerformIO}\ \$\ \mathit{readMVar}\ \mathit{res})
\end{aligned}
$$

- Good: we may use lazy values directly.
- Bad: the programmer needs to be careful about evaluation order.

The authors deemed the non-lazy *eagerly* to be more inline with a core Haskell philosophy: *the programmer should not be concerned with the evaluation order of expressions.*

Conclusion

- Hiding thread management and orchestration inside a concurrent scripting language makes it easier to write concurrent programs.
- Orc naturally exists within a general purpose language.
  core combinators are a small part of the overall program.
- Haskell proofs to be a good choice for embedding Orc.
  higher-order functions, laziness, powerful type system, monads, concurrency
- A non-lazy semantics for *eagerly* fits better in Haskell.

Conclusion

Future.

- Implement more advanced thread management.
- The authors have made significant progress on proving the algebriac laws of Orc and identified the laws required from Concurrent Haskell. But there still is work to be done.
- Investigate splitting up *eagerly* into a part that limits work and an eager memo part which returns a reusable handle to all results.
- It would be interesting to see how Orc relates to the other concurrency approaches out there!

Related Work

- *Reo: a channel-based coordination model for component composition.*
- *Coordination Models Orc and Reo Compared*
- *Translating Orc features into Petri nets and the Join calculus.*
- π-calculus / Pict language
- Actor Model

End

Thank you. Questions?

The library is available on hackage:
`http://hackage.haskell.org/package/orc`

more combinators more fun

Analogous to the list *scanl* function. However, now the order in which the combining function is applied is nondeterministic.

$$
\begin{aligned}
&scan :: (a \to s \to s) \to s \to \mathit{Orc}\ a \to \mathit{Orc}\ s \\
&scan\ f\ s\ p = \mathbf{do} \\
&\quad accum \leftarrow newTVar\ s \\
&\quad x \leftarrow p \\
&\quad (w, w') \leftarrow modifyTVar\ accum\ (f\ x) \\
&\quad return\ w'
\end{aligned}
$$

Parallel or.

$$
\begin{array}{l}
\mathit{parallelOr} :: \mathit{Orc}\ \mathit{Bool} \rightarrow \mathit{Orc}\ \mathit{Bool} \rightarrow \mathit{Orc}\ \mathit{Bool} \\
\mathit{parallelOr}\ p\ q = \mathbf{do} \\
\quad \mathit{ox} \leftarrow \mathit{eagerly}\ p \\
\quad \mathit{oy} \leftarrow \mathit{eagerly}\ q \\
\quad \mathit{cut}\ (\quad\ (\mathit{ox}\ \texttt{>>=}\ \mathit{guard}\ \texttt{>>}\ \mathit{return}\ \mathit{True}) \\
\qquad\quad \texttt{<|>}\ (\mathit{oy}\ \texttt{>>=}\ \mathit{guard}\ \texttt{>>}\ \mathit{return}\ \mathit{False}) \\
\qquad\quad \texttt{<|>}\ (\mathit{pure}\ (\vee)\ \texttt{<*>}\ \mathit{ox}\ \texttt{<*>}\ \mathit{oy}))
\end{array}
$$