Concurrent Orchestration in Haskell

John Launchbury   Trevor Elliott

Galois, Inc.
{john,trevor} at galois.com

Abstract

We present a concurrent scripting language embedded in Haskell, emulating the functionality of the Orc orchestration language by providing many-valued (real) non-determinism in the context of concurrent effects. We provide many examples of its use, as well as a brief description of how we use the embedded Orc DSL in practice. We describe the abstraction layers of the implementation, and use the fact that we have a layered approach to demonstrate algebraic properties satisfied by the combinators.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Design, Languages

Keywords Concurrency, Haskell, Orc, DSL, Monad

1. Introduction

Concurrent programming continues to grow in importance, both because of the prevalence of multicore processors, and because of the distributed nature of internet and enterprise systems. Because different concerns dominate in the different settings, there will never be one single way to program these concurrent systems, just as sequential programming continues to benefit from a multiplicity of distinct approaches. Furthermore, because concurrent programming is far less mature than sequential programming, the area is still ripe for exploring new paradigms, identifying their strengths and weaknesses in different settings.

This exploration proceeds *par excellence* in the context of Haskell. Haskell (and, in particular, the premier Haskell compiler GHC) provides a very powerful and effective set of concurrency primitives together with relevant support in the run-time system. For many purposes, the level of abstraction of these primitives is appropriate but, as their designers state, while these primitives "are expressive ... we do not advocate programming with them directly; instead we hope to build a library of robust abstractions, layered on top of the primitives, that express common programming paradigms." [MPMR01]. True to that aim, the Haskell community now has higher level libraries, including software transactional memory (STM) [HMPH05] which provides one approach to the holy grail of composable concurrency (though STM also required additional support in the run-time system).

The mechanisms and libraries provided by GHC are now quite mature, and at Galois we have used concurrent Haskell to write many systems, including a webDAV server, network stacks, and a library of virtual machine infrastructure. Our experience is that Haskell is very effective for these purposes, allowing us to write intricately concurrent programs with surprising ease, and have them highly performant at run-time. Having said that, we still need to care about many details to get these concurrent programs right.

There are many concurrent applications where we would want to work at a yet higher level, not having to worry about forks, thread identifiers, or race conditions, etc. *Concurrent scripting* or *orchestration* is an example, by which we mean any situation where we wish to orchestrate multiple external (possibly remote) actions whose timing and interleaving is unpredictable, i.e. scripting in a setting where concurrency is prevalent, and indeed dominant. Having a robust and composable approach is highly desirable. For this we turned to the Orc domain-specific language for our inspiration [MC07, KQCM09]. Orc was introduced to address the challenges of highly concurrent scripting, with particular reference to internet programming. To echo an example from the Orc literature, we might wish to contact two airlines simultaneously seeking price quotes. If either quote comes back below a threshold price, say $300, then let's buy a ticket immediately. On the other hand if both quotes exceed the threshold, then buy the cheapest ticket. Additionally, buy a ticket if the other airline does not give a timely quote, or notify the user if neither airline provides a timely quote.

The original version of Orc is a stand-alone domain-specific language (DSL), with mechanisms enabling any Java class instance to be called. As with any DSL, there are advantages and disadvantages to providing it as a stand-alone language. On the one hand, syntax can be specialized to the task at hand, the error messages may be specifically designed, additional analysis tools can be provided, and new users are not challenged with learning more than the DSL itself. On the other hand, different benefits accrue from embedding the DSL in a host language (i.e. an EDSL). These benefits include the ability to mix and match the EDSL programs with other tasks, and a shorter learning curve as much is inherited from the host language. As the ideas of concurrent orchestration are largely non-specific to any particular language, we wanted to take the Orc ideas and adapt them to fit naturally within the Haskell setting.

Hence this paper! We describe an adaptation of Orc as an EDSL in Haskell. Many aspects of the embedding were so straightforward and natural that they would be scarcely worth writing about—in fact we have heard of a number of people who have each embedded portions of Orc into Haskell. But a few aspects turn out to be subtle and tricky, meriting a more careful examination, and these form the main contributions of the paper.

- The first concerns control of concurrency. In an Orc-like setting it is easy to generate abundant concurrency; what is more tricky is trimming and controlling the concurrency when it is no longer needed.

Haskell'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0252-4/10/09...$10.00

- A second aspect concerns laziness. The original Orc language uses concurrent non-strictness to manage synchronization. It is appealing to duplicate this in Haskell, but we have concluded that a different approach fits more properly in the context of Haskell. We relegate the lazy version as a design alternative.
- Third, Orc comes with a number of algebraic laws as part of its specification. We have made significant progress in proving that our *implementation* of the Orc EDSL satisfies these laws, including identifying the requirements we need to establish from the underlying concurrent Haskell foundation.
- Finally, the Haskell setting has provided a fertile ground to explore different choices of combinators, leading us to propose an alternative primitive from that chosen in the original Orc.

This paper contains quite a bit of code. We have been careful to show the code that is useful in explaining the ideas, and elide any that is simply boilerplate. All the code has been released as a Hackage library, and may be installed using cabal.

A note about naming. We use the name Orc to refer to both the original stand-alone DSL, and also to our embedding of the ideas in Haskell. Mostly the context makes clear which we mean, but we try to be explicit whenever there may be confusion.

2. The Orc Language

Orc is the combination of three things: many-valued concurrency, external actions (effects), and managed resources, all packaged in a high-level set of abstractions that feel more like scripting rather than programming.

Unsurprisingly, like most EDSLs, Orc is a monad. We introduce a type constructor Orc to represent Orc computations. An expression p::Orc A may perform many actions, and may produce many results of type A. Orc terms are constructed using the following primitive operators:

```
return  :: a -> Orc a
(>>=)   :: Orc a -> (a -> Orc b) -> Orc b
stop    :: Orc a
(<|>)   :: Orc a -> Orc a -> Orc a
(<+>)   :: Orc a -> Orc a -> Orc a
eagerly :: Orc a -> Orc (Orc a)
liftIO  :: IO a -> Orc a
runOrc  :: Orc a -> IO ()
```

As usual, the monad operators return and bind (>>=) allow us to use the do-notation to build Orc terms. Monads are often thought as sequential, but it will soon be clear that this is not the case here. A better intuition for the Orc monadic bind will be *nested iteration*, rather like in the list monad. Thus an expression like

```
do x <- p
   y <- q x
   h y x
```

is best read as a sequence of *"for each"* statements. In particular, *"for each* x *drawn from the execution of* p*, and then for each* y *drawn from the execution of* (q x)*, produce all the values created in the execution of* (h x y)*."* Thus the Orc monad sets up a kind of nested loop structure, except that the various "loops" are all run concurrently.

The stop operation finishes a local thread of operations; anything that was sequenced after a stop will not get executed, and no value will be returned from this particular computation. So, for example,

```
do x <- p
   y <- stop
   h y x
```

is just the same as

```
do x <- p
   stop
```

Formally, stop is a left-zero of bind (but not a right-zero, as we shall see later).

Now we come to the parallel operators. Orc provides three different kinds of parallelism. First, there is a parallel choice operator <|> (pronounced *par*), which has no intrinsic left-right bias—unlike the list monad. Rather, (p <|> q) will perform all the actions of (and return all the results of) both p and q, whenever they become available. This is true *non-determinism* in that the semantics of Orc does not specify any ordering between p and q.

Secondly, there is a biased-choice operator <+> (pronounced *and then*). In contrast to <|>, an expression of the form (p <+> q) will perform all the actions of (and return all the results of) p, and only when p is completely done will it then go on to perform all the actions of (and return all the results of) q. It may be a misnomer to list <+> as a parallel operator. We do so because its form is almost identical to <|>, overlooking the fact that it imposes an ordering on its arguments.

In both of these operators, p and q are independent of each other. Unless they share a common state element, they will not affect or communicate with each other. The third form of parallelism provides explicitly for communication. The Orc term (eagerly p) will fire up p in a parallel thread, and immediately return a handle for accessing the first result that p produces. The handle is itself an Orc term, so can be used anywhere an Orc term (of the appropriate type) would be used.

The final two operators relate the Orc monad and the IO monad. Any IO operation can be lifted into the Orc monad using liftIO, and it will behave just like it does in IO, returning the single result that it would as an IO operation. The liftIO function is actually the overloaded method of the MonadIO class, making Orc an instance of MonadIO, and thus providing access to any functions defined over that class. In particular, by replacing the standard module Control.Concurrent with the overloaded replacement module Control.Concurrent.MonadIO we can use MVars and the like directly in Orc, as the module gives these accessor functions the following overloaded types:

```
newEmptyMVar :: MonadIO io => io (MVar a)
takeMVar  :: MonadIO io => MVar a -> io a
putMVar   :: MonadIO io => MVar a -> a -> io ()
```

Some later examples will take advantage of this.

The runOrc function works in the other direction to liftIO, allowing an Orc computation to be executed within the IO monad. Note that there is no canonical way to reduce the many results of an Orc computation into the single result that would be required of an IO computation. Discarding the results is canonical, however, so this is what the primitive does. We will later be able to produce a result of (IO [a]) built in terms of this.

We often use a version of runOrc called printOrc that prints each output on a separate line. It can be defined in Orc as follows.

```
printOrc :: Show a => Orc a -> IO ()
printOrc p = runOrc $ do
    x <- p
    liftIO $ putStrLn ("Ans = " ++ show x)
```

That is, for each value x obtained from the Orc computation p, print a line displaying the answer. Note that the order that the results are printed is dependent on the order that they are produced; the results do not necessarily print in a predictable order.

2.1 Examples

To give a flavor of the Orc monad in practice, we'll walk though a series of examples. First, a trivial one. If we define,

```
fplang =  do
  w <- return "Haskell" <|> return "ML"
         <|> return "Scheme"
  return (w ++ " is great!")
```

then executing it (in the interactive Haskell environment GHCi) proceeds as follows:

```
*Main> printOrc fplang
Ans = "Haskell is great!"
Ans = "Scheme is great!"
Ans = "ML is great!"
```

where the order of the answers is somewhat indeterminate. Now consider a slightly richer example.

```
metronome = return () <|> (delay 2 >> metronome)
```

In parallel, `metronome` both returns a value `()`, and starts to wait 2 seconds before doing the whole thing all over again. The `delay` function is obtained just by lifting the IO `threadDelay` operation into the Orc monad (and we choose to use fractional seconds rather than microseconds as our unit of time).

```
delay :: (RealFrac a) => a -> Orc ()
delay w = liftIO $ threadDelay (round (w * 1000000))
```

Here's what we get when we print the result:

```
*Main> printOrc metronome
Ans = ()
Ans = ()
^CInterrupted.
```

where each line was produced a couple of seconds after the previous one. Note that the `<|>` operator is actually an overloaded operator from the standard `Alternative` class, of which `Orc` is an instance. Additionally, `Orc` can be made instances of other standard classes, including the `MonadPlus` and `Applicative` classes, which provides some useful standard combinators for free, such as:

```
guard :: Bool -> Orc ()
pure  :: a -> Orc a
<*>   :: Orc (a->b) -> Orc a -> Orc b
<$>   :: (a->b) -> Orc a -> Orc b
```

Depending on its boolean argument, the `guard` function acts either as `stop` or `(return ())`. The `<*>` operator provides function application between Orc valued computations. The `pure` function lifts values (and hence also functions) into the Orc monad, and `<$>` acts like application (`$`) lifted into the monad. These latter two are each (sometimes helpful) renaming of `return` and `fmap` respectively. We will see `guard` in use in the 8 Queens example.

```
queens =  extend []
      <|> return ("Computing 8-queens...")

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

liftList :: [a] -> Orc a
liftList = foldr (<|>) stop . map return
```

The argument to the `extend` function represents a partial solution to the problem by recording the row positions of the queens in some initial number of columns. Then, for each value of `j` from 1 to 8 (explored in some indeterminate order), we consider whether the position `j` will conflict with the previous partial solution. We omit the code for the conflict testing function as it is a standard boolean test and not Orc-specific.

Note that the body of extend is very similar to what one would write within the list monad to solve the same problem. That is no coincidence, as Orc may be seen as the merger of the list monad with the IO monad (except that the order of results is indeterminate).

In the case of 8 Queens, the output produced is:

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

We immediately ran it again and got the following:

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

Note that the order of the results is different because there is genuine non-determinism going on here. Note also that in each case the first answer given happens to be "Computing 8-queens". There is nothing in the semantics that says it will be the first answer, but operationally it is likely to be the first answer because it can be produced so quickly. If we wanted to ensure the ordering, we could have written:

```
queens =  return ("Computing 8-queens...")
      <+> extend []
```

using the sequentializing operator `<+>`.

Our next examples demonstrate the interplay of effects and concurrency. First, scan. On lists, a `scan` function passes over a list calculating and returning all the partial `foldl` or `foldr` results (depending which scan function we define). The corresponding function in Orc will accumulate the partial `fold` results in whatever order the values become available. We use a `TVar` within Orc to store the ongoing accumulator, having written an atomic modify operation in STM to increment it[1]. The code is as follows:

```
scan :: (a -> s -> s) -> s -> Orc a -> Orc s
scan f s p = do
  accum <- newTVar s
  x <- p
  (w,w') <- modifyTVar accum (f x)
  return w'
```

where the type of `modifyTVar` is

```
modifyTVar :: MonadIO io =>
              TVar a -> (a -> a) -> io (a,a)
```

[1] Just like MVars, overloaded versions of TVar accessor functions are available on Hackage, in this case in the module Control.Concurrent.STM.MonadIO. This allows us to have direct access to TVars from Orc and IO (and from any other monad in the MonadIO class).

Note that at first blush scan looks like linear sequential code. But recall that we need to read the monadic <- as *for each*. As most of these lines are simply liftings from IO, they will produce just a single answer anyway, but the line x <- p could produce zero, one, or many answers. So the code reads as follows: create an accumulator containing the initial value s, and then for each value x produced from p atomically modify the accumulator by combining its value with x (through a partial application of f), and then return the new accumulated value.

In a similar style, but this time using the bias of the <+> operator, we can write a function which counts how many results another Orc program produces.

```
count :: Orc a -> Orc (Either a Int)
count p = do
    accum <- newTVar 0
    (do x <- p
        modifyTVar accum (+1)
        return $ Left x)
      <+>
        (do c <- readTVar accum
            return $ Right c)
```

For each value x produced by p, we increment the accumulator, and then return a tagged version of the value x. Once everything in the inner do is completed, we will read the accumulator, and return that value too (appropriately tagged).

Here's another variant: this collects the values produced by an Orc computation, and delivers them as a list when all are completed.

```
collect :: Orc a -> Orc [a]
collect p = do
    accum <- newTVar []
    (do x <- p
        modifyTVar accum (x:)
        stop)
      <+>
        readTVar accum
```

Note that (collect p) will only return a result if p itself has only finitely many results, and also completes all its execution in a finite time. Similarly, count will return a Right value—the count—only when and if its argument completes.

We can also program a variant of <+>, which we write as <?> (pronounced *or else*). An expression of the form (p<?>q) will perform all the actions of (and return all the results of) p, and only if p produced no answers will it then go on to perform all the actions of (and return all the results of) q.

```
(<?>) :: Orc a -> Orc a -> Orc a
p <?> q = do
    tripwire <- newEmptyMVar
    (do x <- p
        tryPutMVar tripwire ()
        return x)
      <+>
      (do triggered <- tryTakeMVar tripwire
          case triggered of
            Nothing -> q
            Just _  -> stop)
```

For any value x produced by p, we set the tripwire variable. Once p has completely finished, we then try to read the tripwire variable to see if it was triggered. If not, we execute q, otherwise we simply stop this alternative action.

The <?> operator is provided as a primitive in the standalone Orc DSL. We choose to provide <+> instead as it is quite complicated to define <+> in terms of <?>, whereas it is pretty straightforward the other way around.

3. Managed Concurrency

With <|> and <+>, we have a non-deterministic multi-valued monad that includes IO actions. These parallel operations are superb at generating concurrency, but not so good at limiting it when it is no longer required. This is where eagerly comes into its own. The purpose of the eagerly combinator is twofold: it sparks off computations early, and it cuts down the set of results to a single result (the first one that happens to be produced).

Many times we only care about the latter capability. For this we can use eagerly to define a cut combinator, which returns just the first answer its Orc argument produced.

```
cut:: Orc a -> Orc a
cut = join . eagerly
```

or for those less comfortable with monad magic,

```
cut p = do
  ox <- eagerly p
  ox
```

i.e. fire up the Orc expression p to get a handle to the trimmed single result, and then immediately wait for that handle to deliver. Only one value result will be produced.

Using cut, we can specify a simple timeout combinator, as follows:

```
butAfter :: Orc a -> (Float, Orc a) -> Orc a
p ‘butAfter‘ (t,def) = cut (p <|> (delay t >> def))
```

The butAfter combinator sets up a race. If the Orc expression p returns a result before the time t is expired, then it will be the first result, and so the only one allowed through the cut. Any other computations initiated in p will be terminated, along with the delay and the default computation. On the other hand, if the delay turns out to conclude sooner than the Orc expression, and if def then produces a value first, it will be the sole value returned. Note that if ever p terminates without producing any results, the default computation def will execute (after the delay has completed).

To see the other role of eagerly (that of sparking a parallel computation), consider parallel or—if either argument is true, we want to return true even if the other argument has not declared a result. Parallel or is not definable in the sequential lambda calculus, but in Orc we define it as follows:

```
parallelOr :: Orc Bool -> Orc Bool -> Orc Bool
parallelOr p q = do
  ox <- eagerly p
  oy <- eagerly q
  cut (  (ox >>= guard >> return True)
     <|> (oy >>= guard >> return True)
     <|> (pure (||) <*> ox <*> oy))
```

Both p and q are sparked as computations, and ox and oy are bound to Orc valued computations that will return the first values p and q produce. Within the cut, we attempt three different computations in parallel, corresponding to the three cases of parallel or. The first two wait on the results of p and q respectively, and if True, return True immediately. The third case applies the standard *or* function (||) to the results of both p and q, which covers the case when both are False. Whichever of these computations is the first to succeed becomes the single result of the whole parallelOr function.

The sync combinator shows a more general use of eagerly. It captures the idea of fork-join. The function sync launches two Orc computations in parallel, and then waits for a result to come back from both before continuing. In this case we have parameterized over the function used to combine the results.

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

Just as in `parallelOr`, `(pure f)` lifts the function `f` into the Orc monad, whereupon `<*>` applies it to each argument as their values become available. Here's an easy use of `sync`:

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

Unlike `delay` which delays the *start* of a computation, `notBefore` delays the *result* of the computation (though like `delay` it also returns just a single value).

We now have enough machinery to do the example described in the introduction. Assume that we have functions

```
getQuote :: Query -> Orc Quote
price :: Quote -> Int
```

which, respectively, attempt to obtain each individual quote (with an HTTP query, for example), and to extract the price of the quote. Then we can code up the logic of the query simply as follows.

```
quotes :: Query -> Query -> Orc Quote
quotes srcA srcB = do
  quoteA <- eagerly $ getQuote srcA
  quoteB <- eagerly $ getQuote srcB
  cut (  (pure 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
```

The two quotes are launched eagerly, and then whichever of the various clauses in the cut is completed first, that's what the result will be.

4. Semantics

With any DSL, it is useful to provide laws to help the user understand the behavior without having to think operationally. Providing laws also helps ensure that the design is clean. Unsurprisingly given its origin as a process calculus, the Orc language stipulates a set of laws.

A number of the Orc laws are just the monad laws (which incidentally provides yet more evidence that a monadic formulation of Orc is very natural):

LAW 1 (Monad Laws). *For all* `x`*,* `k`*,* `p`*, and* `h`*,*
Left-Return: `(return x >>= k) = k x`
Right-Return: `(p >>= return) = p`
Bind-Associativity: `((p >>= k) >>= h) = (p >>= (k >=> h))`

Note that the `>=>` operator is monadic (Kleisli) composition. That is, `k >=> h = \x -> k x >>= h`. In writing laws such as these, we assume the variables to act as if they were `let`-bound variables in Haskell (so we don't have to worry about variable capture), and of the appropriate type.

The main value of the monad laws is that they allow flexible use of the `do`-notation. We can abstract a sub-portion of a sequence of Orc operations, and understand them in isolation from the rest of the context.

The combinators `stop` and `<|>` satisfy laws as follows:

LAW 2 (Par Laws). *For all* `k`*,* `p`*,* `q`*, and* `r`*,*
Left-Zero: `(stop >>= k) = stop`
Stop-Identity: `p <|> stop = p`
Par-Commutativity: `p <|> q = q <|> p`
Par-Associativity: `p <|> (q <|> r) = (p <|> q) <|> r`
Par-Bind: `((p<|>q) >>= k) = ((p>>=k) <|> (q>>=k))`

Note that these are not the same as the laws typically suggested for the `MonadPlus` class. In particular, Orc calls for the parallel combinator to be commutative, a property violated in many instances of the `MonadPlus` class, including the classic instance, List, in which `<|>` is ++ (list append).

On the other hand, a commonly suggested law for the `MonadPlus` class is missing from the list here, namely the Right-Zero law:

```
p >> stop = stop  {- Not true -}
```

We do not want this law to hold in Orc, as we want the *effects* of `p` to occur, even though it produces no value results. A key point here is that Orc is not only a multiple-value monad, but is equally a concurrent-effect monad. Here's an example where we might use the `(p >> stop)` paradigm.

```
hassle = (metronome>>email("Simon","Hey!")>>stop)
            ‘butAfter‘ (60, return ())
```

Over the course of a minute, this will send Simon an irritating email every 2 seconds. Note that we pipe the result of the `email` operator into `stop` in order to ensure that the `butAfter` combinator continues to wait until the timeout is achieved. We saw a similar use of `stop` in the `collect` example earlier.

The Par-Bind law also deserves a mention. By extrapolation, this law tells us that code which follows a bind (`k`, here) gets re-executed for each value that is produced by the left hand argument. That is, all its effects are re-performed, and all its results are re-returned. But what about the dual law? That is, are

1. `p >>= (\x -> h x <|> k x)`
2. `(p >>= h) <|> (p >>= k)`

equal? The answer is a resounding No. In general these two expressions are quite different. In the first term, the effects of `p` are performed once, whereas they may be performed twice (or more) in the second term.

The laws for `<+>` are similar: `<+>` is associative, and has `stop` as a left and right identity. However, it is not commutative, and neither does it satisfy a corresponding version of the *Par-Bind* law (as the subsequent computations may re-arrange the order in which the results are produced).

4.1 Eagerly Laws

For the pruning operator `eagerly` we have again taken the corresponding laws from Orc, and translated them into our monadic Haskell setting.

The first we consider is called *Distributivity over* `>>`. Translating into our setting, we would express this property as follows: for all (let-bound variables) `q`, `k`, and `h`,

```
eagerly q >>= (k >=> h)
= (eagerly q >>= k) >>= h
```

Now we can see that this law is just an instance of bind associativity, where `(eagerly q)` substitutes for `p` in the expression of the law. It is this property that demonstrates that `eagerly` can be just a combinator, and does not need to have its own binding construct (as it does in the original Orc DSL).

The next law about `eagerly` is a weak dual to the Par-Bind law we saw earlier.

LAW 3 (Par-Eagerly). *For all* p, k, *and* h,

```
eagerly p >>= (\x -> k x <|> h)
= (eagerly p >>= k) <|> h
```

Extrapolating from this law, we learn that later computations (h in this case) do not wait for an Orc term guarded by eagerly to produce a result before being fired up themselves. That is, the expression (eagerly p) places p outside of a sequential flow of control, to be performed concurrently according to some undetermined schedule. The next law says something similar:

LAW 4 (Eagerly-Swap). *For all* p, q, *and* k,

```
do y <- eagerly p
   x <- eagerly q
   k x y
= do x <- eagerly q
     y <- eagerly p
     k x y
```

It doesn't matter in what order the eager computations are launched: the effects and the result will be the same. Of course, what actually gets produced in each case on any given run will depend on undetermined scheduling choices.

Finally, in the original Orc setting, there is a law called Elimination of Where. In our setting, this corresponds to lifting IO operations into the Orc monad using liftIO, so the law becomes:

LAW 5 (Eagerly-IO). *For all* p, m,

```
eagerly (liftIO m) >> p = (liftIO m >> stop) <|> p
```

Like *Par-Eagerly*, this law translates a sequential use of eagerly into a parallel usage, demonstrating the non-blocking concurrent nature of eagerly. Again, just because we pipe any value result into stop (the Orc equivalent of /dev/null) doesn't mean that we want to lose the effects of m. A stronger law, in which liftIO m is replaced by an arbitrary Orc term q seems plausible at first, i.e. that

```
(eagerly q >> p) = ((q >> stop) <|> p)
```

Unfortunately, this is false. It expresses well that q is done concurrently, but misses the fact that eagerly also trims its argument to produce a single result only, and kills any remaining effects. No such trimming is done with (q >> stop). To capture this we would need the law to have the form

```
(eagerly q >> p) = ((cut q >> stop) <|> p)
```

but then we haven't fully eliminated eagerly from the right hand side.

```
import Orc
import System.Random

data Bidder = Bidder
 { name  :: String
 , logic :: Item -> Price -> Orc Price }
type Item  = String
type Price = Int

auction :: Item -> Price -> [Bidder] -> Orc ()
auction item price members = do
  (bid,bidder) <- cut (seekBid item price members)
  continue item bid bidder members

continue :: Item -> Price -> Bidder -> [Bidder] ->
            Orc ()
continue item price bidder members = do
  liftIO $ putStrLn (name bidder++
                     " bids $"++show price)
  mb <- (Just <$> seekBid item price members)
         ‘butAfter‘ (5, return Nothing)
  case mb of
    Nothing -> purchase item price bidder
    Just (bid’,bidder’) ->
      continue item bid’ bidder’ members

seekBid :: Item -> Price -> [Bidder] ->
           Orc (Price, Bidder)
seekBid item price members
  = foldr (<|>) stop
      [consider item price m | m <- members]

consider :: Item -> Price -> Bidder ->
            Orc (Price, Bidder)
consider item price member = do
  bid <- logic member item price
  guard (bid > price)
  return (bid,member)

purchase :: Item -> Price -> Bidder -> Orc ()
purchase item price bidder = do
  liftIO $ putStrLn (name bidder++" wins "
            ++item++" for $"++show price)
```

Figure 1. The “Orction”

5. Example

Let's work a slightly larger example to see Orc blended with other parts of Haskell. Imagine that we are holding an auction. We start by selecting an item and determining a good opening price for it, assemble a group of bidders who are interested in competing to purchase this item. We start the auction by asking the group for something above the initial asking price. After someone makes a bid, we give everyone the opportunity to raise the price, accepting a higher bid if it arrives within a time limit. Once no additional bids are received, we stop the bidding, awarding the item to the bidder with the highest bid.

The code for this is in Figure 1. The main function is auction. We start the auction by giving all of the members of the auction opportunity to place an initial bid via the seekBid function. We apply cut in order to take only the quickest response. Next, the auction enters a phase of repeated bidding, requesting that members of the auction make bids, and timing out after 5 seconds if no bid is received. This timeout is accomplished with the use of the butAfter combinator, allowing a Nothing to be put in place of the bid if none of the bidders decide to act in time. If there is no response to the latest bid, the current highest bidder will be awarded the item, and the computation terminates. If there was a new bid, the auction will continue with the whole bidding process repeating again.

5.1 Orc in Practice

One of the interesting aspects of the auction example is that the elements of concurrency are all very short-lived. There is very little deeply concurrent backtracking-style computation of the form found in 8-queens, for example. In our experience, different applications call for quite different blends of deep and shallow concurrency, along with other kinds of effects.

The original motivating application for Orc at Galois was writing concurrent tests for virtual machines running in the Xen hypervisor. These machines need to be prompted to talk to each other, and their communications had to be monitored to see if they were correct. At any time there is the possibility that one of these machines will die, or fail to start up in the first instance. Again, many of the test scripts show quite shallow parallelism, but having it available and well integrated with IO actions was very important.

Our first implementation of Orc had some subtle concurrency bugs which meant that the test harness would sometimes hang, maybe after running all the tests, or maybe not. Of course, we know *now* that it was a bug in the Orc implementation; at the time it was very difficult to find out what was wrong. Fixing the test framework was what prompted this more rigorous examination of Orc.

6. Implementing the `Orc` Monad

There are a number of different ways to implement Orc. In earlier versions we used resumptions over the IO monad, but now have a much more efficient implementation using continuations over the IO monad. The result is disarmingly simple, partly because we hide some of the resource abstraction one level down. We define:

```
newtype Orc a = Orc {(#) :: (a -> IO ()) -> IO ()}
```

We will later change the IO monad to an IO monad with an environment, but considering it as IO for now will be sufficient.

The functor and monad instance definitions for Orc are just the standard continuation instances, where the answer type is itself a monad. We use the record selector # as an infix operator to apply an Orc term to its continuation. Thus:

```
instance Functor Orc where
  fmap f p = Orc $ \k -> p # (k . f)

instance Monad Orc where
  return x = Orc $ \k -> k x
  p >>= h  = Orc $ \k -> p # (\x -> h x # k)
  fail _   = stop

stop :: Orc a
stop = Orc $ \_ -> return ()
```

In the bind (>>=) we execute p with the continuation that will take its result (x), and pass it to the h function—which is itself handed the continuation of the the whole expression, namely k. The `fail` method says what happens when pattern matching fails in the do-notation. In this case, we simply finish the thread, discarding any computations that may have been scheduled in the continuation.

The plan for p<|>q is that both p and q are executed (with any effects they have), passing any results they produce to the computations that follow them. We model many values being returned by calling the continuation repeatedly. Thus,

```
par :: Orc a -> Orc a -> Orc a
par p q = Orc $ \k -> do
    fork (p # k)
    fork (q # k)
    return ()

instance Alternative Orc where
  empty = stop
  (<|>) = par

instance MonadIO Orc where
  liftIO io = Orc $ \k -> (liftIO io >>= k)
```

We can optimize the definition of <|> to avoid the second `fork`, and instead execute (q#k) in the current thread; but for now we will work with the symmetric version as it makes the examination of the laws more straightforward.

Also, as noted earlier, we will be using a monad other than just IO, so we introduce `fork` as an overloaded operator, which is simply `forkIO` on the IO monad. Similarly, in `liftIO`, we lift any IO computation into Orc by executing the computation and applying the continuation to the result. This >>= is in the IO monad, or rather in the IO-like monad that we will later use instead of IO (hence the inner `liftIO`).

The `eagerly` combinator launches its Orc argument in a separate forked thread, and immediately returns with a single value that is itself an Orc computation. This result computation, when executed, will return just the first result of the original Orc argument. Here's a simplified definition:

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = Orc $ \k -> do
    res <- newEmptyMVar
    fork (p ‘saveOnce‘ res)
    k (liftIO $ readMVar res)

saveOnce :: Orc a -> MVar a -> IO ()
p ‘saveOnce‘ r = do
    ticket <- newMVar ()
    p # \x -> (takeMVar ticket >> putMVar r x)
```

The function `eagerly` executes p in a forked process with a continuation that writes p's result in an `MVar`. It then invokes its own continuation on a simple Orc process that reads the result from p when it becomes available. As we may need to access the result value many times (recall the ticketing function `quotes`, for example), we use `readMVar` to allow the result to be read many times, rather than using `takeMVar` which would block after the first access. As p may well invoke its continuation many times, we have to make sure that only the first of the writes succeeds, so we use the `MVar ticket` as a gating operation.

7. Thread Leaks

We now turn our attention to the IO substrate on which the Orc combinators are built. We have plenty of opportunities for creating work, whether through the parallel construct <|> or with the `eagerly` combinator, but we have no particular capability for controlling and shutting down work when it is no longer needed. In `quotes`, for example, if the B-source delivers an acceptable quote, we have no need to continue analyzing the A-source quote, nor continuing with the timeout computations, as their results cannot affect the outcome of the composite query. In these cases they are simple computations, so perhaps it's not a problem, but in general the alternative computations could represent an arbitrary amount of work, creating a multiplicity of threads perhaps, none of which are required. This is what might be styled a *thread-leak*: when unneeded threads are not closed properly, and the number of unused threads grow with time.

Fortunately, the Orc combinators provide sufficient guidance about the programmer's intent to allow us to build in automatic thread management. The Orc programmer can avoid thinking about thread management to about the same extent that a functional programmer can avoid having to think about space management. That is, for most purposes, the Orc programmer can just assume that the implementation does The Right Thing. But just as with space, there are times when the threads themselves become the critical resource, and then the Orc programmer will need to give more careful thought as to how many threads are being created and when they are being retired. This kind of advanced thread management is the topic of current research, and beyond the scope of this paper.

About the only change we will make to introduce automatic thread management is to change the monad underlying Orc.

```
newtype Orc a = Orc {(#)::(a -> HIO ()) -> HIO ()}
```

We introduce a hierarchical IO monad, HIO, which is just the IO monad augmented with an environment that tracks the current *thread group*. Whenever a new thread is forked, we will register its thread identifier with the current thread group, so that when the computations of a group are no longer needed, they can all be killed *en masse*. We will also track how many threads are active within the group, which will allow us to tell when a group has finished naturally. We will need this capability to define <+>.

```
newtype HIO a    = HIO {inGroup :: Group -> IO a}

type Group       = (TVar Int, TVar Inhabitants)
data Inhabitants = Closed | Open [Entry]
data Entry       = Thread ThreadId
                 | Group Group

newPrimGroup    :: IO Group
register        :: Entry -> Group -> IO ()
killGroup       :: Group -> IO ()

increment, decrement, isZero :: Group -> IO ()

instance MonadIO HIO where
  liftIO io = HIO $ \_ -> io
```

As the type declarations indicate, groups contain both thread identifiers and sub-groups, providing a hierarchical structure to the groups. They also include a count of the number of their active threads.

To co-opt the earlier definition of par we make HIO an instance of the HasFork class, by providing a definition of fork in which a freshly forked thread will register itself within the current group, and then go on to execute its body in that same group.

```
instance HasFork HIO where
  fork hio = HIO $ \g -> block $ do
    increment g
    fork (block (do tid <- myThreadId
                    register (Thread tid) g
                    unblock (hio ‘inGroup‘ g))
          ‘finally‘
            decrement g)
```

We use some GHC-specific aspects of thread implementation here. The block function prevents the thread registration code from being interrupted by an asynchronous exception, but once we enter the body of the thread (hio executing in group g) we use unblock to reenable exceptions. When the thread terminates (either naturally by running out of code, or through being killed with an exception), the decrement code is executed, to record that there is one fewer thread in the group.

Note that, unlike some approaches, we don't automatically generate a new sub-group for each forked thread. We tried that at first, but our experience was that it is an unhelpful conflation of ideas. In fact, we concluded that the concept of *fork* and the concept of *group* are quite distinct, and should be handled separately. There are echoes here with Scheme's *custodians*, but we are not nearly so comprehensive [FFKF99].

Keeping the Group type abstract, we can define accessor functions for groups. The newGroup function creates a new sub-group within the existing group. The associated function local sets the current group environment within the HIO monad, close shuts down all the threads in the group (and sub-groups) at the end, and finished hangs until the group has completed (again, either naturally or through being killed).

```
newGroup :: HIO Group
newGroup = HIO $ \g -> do
    g’ <- newPrimGroup
    register (Group g’) g
    return g’

local :: Group -> HIO a -> HIO a
local g p = liftIO (p ‘inGroup‘ g)

close :: Group -> HIO ()
close g = liftIO $ killGroup g

finished :: Group -> HIO ()
finished g = liftIO $ isZero g
```

These functions provide the capability we require for removing thread leaks from eagerly, which we now redefine as follows:

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = Orc $ \k -> do
    res <- newEmptyMVar
    g <- newGroup
    local g $ fork (p ‘saveOnce‘ (res,g))
    k (liftIO $ readMVar res)

saveOnce :: Orc a -> (MVar a,Group) -> HIO ()
p ‘saveOnce‘ (r,g) = do
    ticket <- newMVar ()
    p # \x -> (takeMVar ticket >> putMVar r x
               >> close g)
```

The execution of p takes place within a fresh sub-group w. The first time p returns a result (i.e. invokes its continuation), the group is closed down, and all ongoing work is terminated. The group infrastructure is sufficient for us to now define <+> also, as follows:

```
(<+>) :: Orc a -> Orc a -> Orc a
p <+> q = Orc $ \k -> do
    g <- newGroup
    local g $ fork (p # k)
    finished g
    q # k
```

Here, the new group w is not used to prematurely shut the work down, but rather to scope what work is active and identify when it all completes.

To blend well with this framework, users of liftIO should take into account the possibility that their IO operations will be summarily killed off, and include appropriate bracketing or finalizers to close down any resources they control [MPMR01].

8. Demonstrating the Orc Laws

We wanted to explore to what extent our implementation satisfied the laws provided earlier. Unfortunately, we cannot do formal proofs as the foundation of concurrent Haskell has not stabilized sufficiently: the published transition semantics for concurrency and asynchronous exceptions [MPMR01] are not what GHC currently implements. We have a draft semantics for Orc in the same style, but until the underlying semantics stabilizes, it is hard to say what the Orc extensions would mean.

What we have done instead, is reduce the laws on Orc to laws that we expect the underlying monad to satisfy (whether IO or HIO). We currently have justifications for almost all the monad and Orc laws, but had to assume certain properties from the underlying concurrency layer to do so.

In the case of IO, it will be hit and miss whether they will be satisfied. In the case of HIO, we have an opportunity to build the monad so that it can satisfy the appropriate laws, perhaps with proof obligations to the user when additional IO actions are lifted into Orc. We will look at a couple of examples.

First, the monad laws themselves may be shown by simple equational reasoning. For example, the associative monadic law goes as follows (we have dropped the `Orc` constructor—and `#` deconstructor—to make the presentation simpler):

```
(p >>= g) >>= h
= \k -> (\k’ -> p (\y -> g y k’)) (\x -> h x k)
= \k -> p (\y -> g y (\x -> h x k))
= \k -> p (\y -> (g y >>= h) k)
= \k -> p (\y -> (g >=> h) y k)
p >>= (g >=> h)
```

This same reasoning works for any continuation monad—there is nothing Orc-specific here.

For the Par-Commutativity law (`p <|> q = q <|> p`), simple equational reasoning again shows us exactly what we need to know—in this case, what key property we need of the underlying system (again we drop the `Orc` constructor):

```
p <|> q
= \k -> do
    fork (p k)
    fork (q k)
    return ()
= {Fork-Swap}
  \k -> do
    fork (q k)
    fork (p k)
    return ()
= q <|> p
```

The key step requires the following equivalence

Law 6 (Fork-Swap). *For all* `ioA` *and* `ioB` *of type* `HIO ()`*,*

```
fork ioA >> fork ioB = fork ioB >> fork ioA
```

At one level, it is hard to imagine any true concurrent system violating this law. Indeed, speaking loosely for a moment, this law might be able to be taken as the *definition* of real (rather than simulated) concurrency. By *real concurrency* we intuit it to be where the concurrent operations are acting in distinct and unsynchronized clock or time domains. On the other hand, the law will only be true subject to some appropriate equivalence where the underlying thread machinery is guaranteed not to be visible.

The Par-Associativity law shows a similar pattern:

```
(p <|> q) <|> r
= \k -> do
    fork ((p <|> q) k)
    fork (r k)
    return ()
= \k -> do
    fork (fork (p k) >> q k)
    fork (r k)
    return ()
{Fork-Floating}
= \k -> do
    fork (p k)
    fork (q k)
    fork (r k)
    return ()
{Fork-Empty}
= \k -> do
    fork (p k)
    fork (q k)
    fork (r k)
    fork (return ())
    return ()
{Fork-Floating}
= \k -> do
    fork (p k)
    fork (do fork (q k)
             fork (r k)
             return ())
    return ()
= \k -> do
    fork (p k)
    fork ((q <|> r) k)
    return ()
= p <|> (q <|> r)
```

We needed two lemmas for moving code in and out of threads, and for eliminating null threads.

Law 7 (Fork-Floating). *For all* `p` *and* `q`
`fork (fork q >>= p) = (fork q >>= (fork . p))`

Law 8 (Fork-Empty). *For all* `p`
`fork(return()) >> p = p`

Assuming these laws about the concurrency layer allows us to to do most of our reasoning about the Orc combinators at the level of Haskell code, rather than having to do low-level concurrency proofs directly. This was very helpful. In fact, moving towards an algebraic theory of threads seems quite promising as a generally applicable proof technique.

9. Design Alternatives

9.1 Redoing Eagerly

The original Orc DSL has explicit roles for both laziness and strictness. The primitive value-operators are all strict in their arguments, but just about everything else is non-strict. In particular, the pruning relies explicitly on laziness: the single value result of the eager computation is bound lazily, and the subsequent Orc computation will pause only at a point that the value of the previous computation is actually required (e.g. by a strict primitive function).

Given Haskell’s laziness, it was very appealing to build a corresponding design in Haskell. We defined a combinator

```
val :: Orc a -> Orc a
```

that executes its Orc argument, returning immediately with a pointer to a sparked thunk that contains the single (trimmed) result of the computation.

```
val :: Orc a -> Orc a
val p = Orc $ \k -> do
    res <- newEmptyMVar
    w <- newGroup
    local w $ fork (p ‘saveOnce‘ (res,w))
    k (unsafePerformIO $ readMVar res)
```

The definition is identical to the definition of `eagerly`, except that we replace the `liftIO` with `unsafePerformIO`. Despite being “unsafe”, this is a very mild use of `unsafePerformIO`, akin to its use within the GHC function `unsafeInterleaveIO`.

Here’s the parallel-or example redone using `val`

```
parallelOr p q = do
  x <- val p
  y <- val q
  cut (  (guard x >> return True)
     <|> (guard y >> return True)
     <|>  publish (x || y))
```

In this formulation, x and y are bound to lazy thunks that will evaluate to the boolean values themselves, rather than the Orc computations we had in the previous version. So we can apply guard and || directly to these values, and not have to do the application within the Orc monad.

However, for this to work, we need to introduce a new function publish,

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

The publish function is a hyperstrict form of return, hence the use of deepseq from the NFData class. A result is returned only after its argument is completely evaluated. Had we used return in the parallelOr example instead, then the expression return (x||y) would have immediately succeeded, despite the values of x and y not being available, and the parallel nature of the computation would be lost.

Similarly, consider redoing the quotes function from earlier:

```
quotes :: Query -> Query -> Orc Quote
quotes srcA srcB = do
  resultA <- val $ getQuote srcA
  resultB <- val $ getQuote srcB
  cut (   publish (least resultA resultB)
     <|> (threshold resultA)
     <|> (threshold resultB)
     <|> (delay 25 >>
            publish resultA <|> publish resultB)
     <|> (delay 30 >> return noQuote))
```

Again, we have to be quite careful about when we have to force evaluation and when we don't need to.

Stepping back, it is certainly pleasant to be able to use a function like least directly, and not have to lift it into the monad in order to extract the results from the pruned computation. The solution is cute, but also ultimately unpredictable as it relies on knowing when expressions are evaluated. In the design we adopted, we have all the same capabilities and with more control and predictability, at the cost of a slightly more explicit stepping between the two worlds.

9.2 Other Combinators

As we program with the Orc combinators, we find that various patterns of use crop up repeatedly. For example, we may want to execute an Orc expression, allowing it to perform its effects continually, until some termination condition arises, such as a response from a remote request. We can capture this pattern as follows:

```
onlyUntil :: Orc a -> Orc b -> Orc b
p ‘onlyUntil‘ done = cut (silent p <|> done)

silent :: Orc a -> Orc b
silent p = p >> stop
```

That we don't care about the value results of p is shown by the use of silent, where the results are fed into stop. On the other hand, as soon as done returns any result, the cut will shut down the expression, including any subtasks of p.

Figure 2 shows a use of onlyUntil, this in the context of a fairly intricate Orc combinator. The combinator takeOrc is rather like the list function take: it returns the first n results its argument produces, and then terminates the computation. Two MVars are used to communicate between two parallel Orc operations, one of which is running the Orc argument p, the other of which is counting and transmitting the results. The same technique applies within the definition of zipOrc, we just have to double up the communications.

```
takeOrc :: Int -> Orc a -> Orc a
takeOrc n p = do
    vals <- newEmptyMVar
    end  <- newEmptyMVar
    echo n vals end
       <|> silent (sandbox p vals end)

dropOrc :: Int -> Orc a -> Orc a
dropOrc n p = do
    countdown <- newTVar n
    x <- p
    join $ atomically $ do
      w <- readTVarSTM countdown
      if w==0 then return $ return x
        else do
          writeTVarSTM countdown (w-1)
          return stop

zipOrc :: Orc a -> Orc b -> Orc (a,b)
zipOrc p q = do
    pvals <- newEmptyMVar
    qvals <- newEmptyMVar
    end   <- newEmptyMVar
    zipp pvals qvals end
      <|> silent (sandbox p pvals end)
      <|> silent (sandbox q qvals end)

--------------------------------------
-- Auxilliary definitions

sandbox :: Orc a -> MVar (Maybe a) ->
           MVar () -> Orc ()
sandbox p vals end
  = ((p >>= (putMVar vals . Just))
     <+> putMVar vals Nothing)
    ‘onlyUntil‘ takeMVar end

echo :: Int -> MVar (Maybe a) -> MVar () -> Orc a
echo 0  _   end = silent (putMVar end ())
echo j vals end = do
    mx <- takeMVar vals
    case mx of
      Nothing -> silent (putMVar end ())
      Just x  -> return x <|> echo (j-1) vals end

zipp :: MVar (Maybe a) -> MVar (Maybe b) ->
        MVar () -> Orc (a,b)
zipp pvals qvals end = do
  mx <- takeMVar pvals
  my <- takeMVar qvals
  case mx of
    Nothing -> silent (putMVar end ()
                      >> putMVar end ())
    Just x  -> case my of
      Nothing -> silent (putMVar end ()
                        >> putMVar end ())
      Just y  -> return (x,y)
               <|> zipp pvals qvals end
```

Figure 2. List-like combinators defined within Orc

The dropOrc function is able to use a simpler technique, because it has no need to prematurely terminate the Orc computation that is producing the results. Note the use of readTVarSTM, which is our name for the TVar read operation within the STM monad itself (we reserve the less explicit readTVar for reading TVars in any MonadIO instance).

We shall close this section by noting a couple of neat relationships. The cut combinator uses the trimming power of eagerly but not the concurrency. That trimming power is also provided by takeOrc. Therefore we have that

```
cut = takeOrc 1
```

Similarly, the synchronization function uses both the trimming and the parallel execution provided by eagerly, with a synchronization on the timing of the results. Thus,

```
sync (,) p q = cut (zipOrc p q)
```

10. Related Work

10.1 The Orc DSL

The original Orc DSL is a concurrent, impure, functional language [K08]. It started life as a process calculus, but has evolved to become a fully fledged scripting language, designed to interact closely with Java.

The primitives in Orc are called *sites*. These are (effectful) functions that are typically defined outside of Orc. Examples include Email(addr,mess) or Prompt("Name:") etc. In addition, basic arithmetic functions are also considered sites.

There were originally three main combinators in Orc: sequential composition, parallel composition, and pruning. The fourth combinator, for biased choice, has recently been added.

Sequential composition is written with an >> operator. If the result of a previous expression needs to be named, the programmer may place an identifier or pattern in the midst of the operator.

```
A(b,c) >x> C(x)
```

This syntax strongly echoes the two variations on the monadic bind operator in Haskell, either discarding or naming the result in a sequence of operations.

Parallel composition is written with a bar operator (|). It causes the two expressions to be executed in parallel. Thus, the expression 1 | 2 will cause execution to branch, with any successive operations being done twice—once for the value 1 and once for the value 2.

To take a simple example, finding all the combinations of numbers from 1 to 10 that add up to 11 is written in Orc as follows

```
def Iterate(l) = l >h:t> (h | Iterate(t))
val ls = [1,2,3,4,5,6,7,8,9,10]
Iterate(ls) >x> Iterate(ls) >y>
    if(x+y = 10) >> (x,y)
```

Note the use of pattern matching in the midst of the sequential combinator to separate the head and tail of the list.

The prune combinator (called *Where* in early versions of Orc) is written <<. This operator behaves syntactically like the sequential composition operator, but operationally has some key differences. First, in the expression F << G, both arguments execute in parallel and, second, only the first successful result from G is passed to F, and other results are discarded. This is useful when dealing with functions that could potentially produce infinite data, or for timing out long-running computation.

As the Iterate example showed above, Orc programs are built by writing functions, using recursion in order to provide loop-like behavior. Whereas sites are all strict, functions (and combinators) are not. The non-strictness allows control structures to be coded up as functions. For example, the function:

```
def Timeout(n,default,m)
       = v <v< ((Rtimer(n) >> default) | m)
```

makes it possible to stop long-running jobs, and provide a default value back in the event of timeout. It's worth noting that if the default value never produces a value itself, then the long-running job will continue to run, even after the timeout has occurred.

Bringing these concepts together, here is an example from the Orc distribution.

```
def isPrime(n) =
  def primeat(i) =
   val b = i * i <= n
      if(b) >> (n % i /= 0) && primeat(i+1)
    | if(~b) >> true
  primeat(2)

def Metronome(i) = i | Rtimer(500)>>Metronome(i+1)

Metronome(2) >n> if(isPrime(n)) >> n
```

This example uses a user-defined recursive function Metronome to generate a sequence of increasing values starting at 2. Each of these values (n) are checked for primality, and if so are returned as the result of the computation. As the Metronome function uses the bar operator, each prime candidate will be checked in parallel, with a 500ms delay introduced by the call to Rtimer having been provided to pace the output to a human scale.

As for the laws, in the original Orc the distributivity law is expressed as follows: if H is x-free, then

```
((K >> H) <x< Q)=(K <x< Q) >> H
```

Here K, H, and Q range over Orc syntactic terms (hence the need to discuss freeness of variables). Similarly, the law called Elimination of Where states that if Q is x-free, for site M

```
(Q <x< M) = Q |(M >> stop)
```

By reimplementing the Orc process calculus in Haskell, we provide Haskell programmers with the flexibility of the Orc calculus directly. But the Haskell embedding introduces new dimensions too. First of all, this is the first strongly typed implementation of Orc, with all the usual benefits a type system provides. Moreover, the Haskell distinction between *value* and *computation* makes it much easier for a user to write new combinators than when the two worlds are merged. Also, building on the monadic framework provides access to existing libraries of monad abstractions which can be re-used in Orc directly.

10.2 Other Related Work

There have been a number of previous partial implementations of Orc in Haskell. One recent effort [CB09] was the result of a senior undergraduate project. Like in this paper, the implementation used a monad to structure the sequential binding operation. Unlike here, though, the implementation had significant communication overhead between threads and, more significantly, did not provide an implementation of the critical Orc pruning construct *Where*, which we modeled in two ways with eagerly and val.

The instances of MonadPlus that are most similar to the Orc implementation here are those given in the LogicT library [KSFS05]. The presentation is different, and the underlying structure for the top level monad is more like resumptions than continuations. More significantly, the LogicT work places particular emphasis on the backtracking aspect of the monad. In particular, the various implementations substituted alternative backtracking strategies, particularly as a mechanism for search. One way to view this paper is an exploration of what happens to the ideas of LogicT when the underlying monad is deliberately concurrent and effectful, and how those effects are managed in the presence of pruning.

Another relevant approach is the `MonadLib` library of monad transformers that allow complex monads to be constructed from relatively simple layers [D08]. The `ChoiceT` transformer resembles the Orc monad in that it allows for choice points to be introduced, though the evaluation of these choices is left-biased.

Functional reactive programming (FRP) is an arrows-based paradigm for reactive programming that has been used in a variety of settings, including animated graphics and robotics [EH97]. In contrast to the explicit concurrency of Orc, input is viewed as a time-varying stream of events and/or values. It is not yet clear what the expressive tradeoff is between FRP and Orc, but they seem to be contrasting approaches to similar kinds of domains.

The mechanism we used for managing and controlling the forking of concurrent processes was a simple version of Scheme custodians [FFKF99]. A custodian is responsible for managing threads, ports, sockets, and so on, and whenever a thread or port is created, it is handed to the current custodian for management. Our notion of *current group* in HIO is similar, though all we manage are the threads. When a custodian is shut down, it closes all the resources it manages, and terminates its threads. Moreover, a closed custodian cannot manage any new objects. Similarly here, any attempt to register a thread with a closed group will cause the thread to die. The success of custodians suggests we ought to consider going further with our groups so that they too can manage objects other than threads, for example by registering finalizers when `liftIO` is used to lift more primitive IO operations into Orc.

11. Conclusion

In our experience with practical examples, the concurrency portions of Orc form a small part of the overall programs we write, and we need a rich language to do the other parts. The original Orc effort seems to have observed the same phenomenon, as it now comes with a large expression language to complement the concurrency parts. This experience suggests that Orc really should be thought of as a calculus that exists within the context of other languages, rather than a language in its own right. It is very naturally an embedded DSL.

Here we have embedded Orc in Haskell, but in principle it could be embedded in other languages too. The main elements of Haskell we exploited were monads (to be able to work with effectful terms as first-class objects), definitional laziness (to write control structures, including recursive control structures), higher-orderness (to provide continuations), and concurrency (to be concurrent (!)). Languages lacking any of these elements may be able to simulate them sufficiently to provide something comparable; macros, for example, go a long way in replacing laziness as a way to write control structures.

One place we did not use laziness was the very place where the original Orc DSL did—in the Where pruning construct. Instead we opted for the `eagerly` combinator, rather than `val` (which does echo the original Orc choice directly). The reason we chose this design is that Haskell works hard to make the order of evaluation something that the programmer doesn't need to think about, except when working on performance improvements. In fact, the major driver for monads was a desire to ensure that the choice and order of computational effects did not depend on the particular order of evaluation chosen by the compiler. For Orc to use an explicit control of laziness (using `publish`) to control which concurrent computations will continue performing their effects would be somewhat contrary to this deep design philosophy. So we went with `eagerly` which is more honest with respect to the monad structure, and with relatively little syntactic overhead. In other languages it may be appropriate to make the other choice.

Orc is a concurrency EDSL, but so is STM. Both are mini-languages that are sprinkled around mostly-IO code, and both try to remove some of the complexity that is present in a fully-fledged concurrent setting. This is an ongoing and major challenge, and there is much still to be understood here. For us, limiting the complexity of the underlying thread infrastructure was a critical step to undertake the task of establishing the laws.

There will be other concurrency EDSLs over time as new needs emerge, and they too will be built on top of IO one way or another. We now advocate overloading IO operations on MVars etc. to make for a smoother integration (using the `MonadIO` class). In our experience, `MVars` are best used for unidirectional communication between threads whereas `TVars` shine when used as shared state elements with atomic operations—in our preliminary performance measurements `MVars` degrade dramatically when in high contention, especially over multiple cores, whereas `TVars` are robust across many different configurations.

As regards the Orc calculus itself, we are left wondering whether `eagerly` can be factored into two separate components: the `cut` (which limits work but is not parallel); and an eager memo operator (which sparks the work, and returns a reusable handle to all the results). The purpose of this factorization would be to enable more laws, perhaps even a complete axiomatization of Orc.

Acknowledgments

Many people have helped us in this endeavor, and we offer them our sincere thanks. Notable among them are Andy Adams-Moran, Magnus Carlsson, William Cook, Iavor Diatchki, Kathleen Fisher, Simon Marlow, Eric Mertens, and Don Stewart. We are particularly indebted to Simon Peyton Jones for suggesting using explicit continuations as a way to simplify the implementation.

References

[CB09] M. D. Campos, and L. S. Barbosa, *Implementation of an Orchestration Language as a Haskell Domain Specific Language*. Electronic Notes in Theoretical Computer Science, Volume 255, Nov 2009

[D08] I. Diatchki. *MonadLib* http://www.purely-functional.net/monadLib

[EH97] C. Elliott and P. Hudak, *Functional Reactive Animation.* ACM Conference on International Conference on Functional Programming (ICFP), 1997.

[FFKF99] M. Flatt, R. Findler, S. Krishnamurthi, and M. Felleisen *Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)*. In: ACM SIGPLAN International Conference on Functional Programming, ICFP 1999

[HMPH05] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. *Composable Memory Transactions*. ACM Conference on Principles and Practice of Parallel Programming (PPoPP), 2005.

[K08] D. Kitchin, *A User's Guide To Orc.* http://orc.csres.utexas.edu/userguide.pdf

[KQCM09] D. Kitchin, A. Quark, W. Cook and J. Misra. *The Orc Programming Language*. Proceedings of FMOODS/FORTE, Springer Verlag, LNCS 5522, 2009.

[KSFS05] O. Kiselyov, C. Shan, D. Friedman, and A. Sabry. *Backtracking, interleaving, and terminating monad transformers*. In: ACM SIGPLAN international conference on Functional programming (ICFP), 2005.

[MC07] J. Misra and W. Cook. *Computation Orchestration: A Basis for Wide-Area Computing*. Journal of Software and Systems Modeling, March 2007

[MPMR01] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. *Asynchronous Exceptions in Haskell.* In ACM Conference on Programming Languages Design and Implementation, PLDI 2001.