`choose` Your Own Derivative

(Extended Abstract)

Jennifer Paykin   Antal Spector-Zabusky   Kenneth Foner

University of Pennsylvania

{jpaykin,antals,kfoner}@seas.upenn.edu

Categories and Subject Descriptors   D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords   selective choice; derivatives; concurrency; Haskell

1. Introduction

Synchronization is the hardest part of concurrent programming. In this extended abstract we discuss a generalization of the synchronization mechanism *selective choice*. As envisioned by Reppy [4], selective choice takes a list of *events*, executes them concurrently, and returns the value of the first event to complete. We argue that selective choice can be extended to synchronize arbitrary data structures of events, based on a typing paradigm introduced by McBride [3]: the derivatives of recursive data types. We discuss our work in progress on implementing generalized selective choice as a Haskell library based on generic programming.

Selective choice.   Consider implementing a Haskell function `timeout time evt` that takes an `IO` action `evt` and runs it for at most `time` microseconds. This function can be implemented in terms of the primitive `threadDelay :: Int -> IO ()` by executing `evt` and `threadDelay time` concurrently and recording which one finishes first. The concurrency is provided by a function `chooseEither :: IO a -> IO b -> IO (Either a b)` (similar to `waitEither` in the Haskell Async library [2]).

```
timeout :: Int -> IO a -> IO (Maybe a)
timeout time evt = do
  x <- chooseEither evt (threadDelay time)
  case x of Left  a  -> pure $ Just a
            Right () -> pure Nothing
```

The `chooseEither` function is an instance of a more general mechanism called *selective choice*, which is a pattern for executing some number of events concurrently and recording which one happens first. Frequently, selective choice comes in the form `chooseAny :: [IO a] -> IO a`, returning the value of the first `IO` action in a list to trigger. However,

`chooseAny` does not indicate *which* choice was made; to find that out we might prefer an operation `chooseList` of type

```
[IO a] -> IO ([IO a], a, [IO a])
```

that returns not only the aforementioned value, but also the remaining `IO` actions in the list.

We can extend this idea even further by focusing on the context of an arbitrary data structure. For example, consider an abortable `IO` action: a pair of a threaded `IO` action along with its thread ID.

```
type Abort a = (ThreadId, IO a)
abort :: Abort a -> IO ()
```

We can define a selective choice operator `chooseAbort` that waits for the first `IO` action in a list of `Abort a` values:

```
[Abort a] -> IO ([Abort a],(ThreadId,a),[Abort a])
```

Using `chooseAbort` we can easily implement a function that aborts all of the `IO` actions that were not chosen:

```
runUntilFirst :: [Abort a] -> IO a
runUntilFirst ls = do
  (prev, (tid,a), post) <- chooseAbort ls
  mapM_ abort (prev ++ post)
  putStrLn (show tid ++ " completed!")
  return a
```

In the rest of this document we describe how to generalize the type of `chooseEither`, `chooseList`, and `chooseAbort` into a single type-directed function `choose`. The result type of `choose` is based on the *derivative* of the input type, in the sense of McBride's one-hole contexts [3].

2. Derivatives and One-Hole Contexts

The `choose` function takes a data structure, which may contain arbitrary `IO` actions to be run concurrently, and selects a single action inside that structure – specifically, the next action to complete.

For example, `choose` over a disjunction of two actions, `Either (IO a) (IO b)`, is just an action returning a disjunction of results, `IO (Either a b)`. On the other hand, `choose` over a pair of actions, `(IO a, IO b)`, produces a more complicated action of type `IO (Either (a, IO b) (IO a, b))`. Here, either the `IO a` action completed and the `IO b` is still in progress (the `(a, IO b)` case), or vice versa (the `(IO a, b)` case). For both sums and products, the result type of `choose` is reminiscent of the sum and product rules for the derivative operation in calculus. This is easiest to see in a more type-theoretic notation: writing $+$ for `Either`, $\times$ for `(,)`, and $\diamond$

$$\partial_x\, x = 1 \qquad\qquad \partial_x \diamond t = 0$$
$$\partial_\alpha\, y = 0 \qquad\qquad \partial_\diamond \diamond t = t$$

$$\partial_\alpha\, 0 = 0 \qquad\quad \partial_\alpha(s + t) = \partial_\alpha s + \partial_\alpha t$$
$$\partial_\alpha\, 1 = 0 \qquad\quad \partial_\alpha(s \times t) = \partial_\alpha s \times t + s \times \partial_\alpha t$$

$$\partial_x(\mu x.\, t) = 0$$
$$\partial_\alpha(\mu y.\, t) = \mu z. \left(\partial_\alpha t \,\middle|\, y = \mu y.\, t\right) + \left(\partial_y t \,\middle|\, y = \mu y.\, t\right) \times z$$

Figure 1. The derivative of a regular type with respect to α, which is either a type variable (x) or an event ($\diamond$).

for $\mathtt{IO}$, we have

$$\mathtt{choose} : (\diamond A + \diamond B) \to \diamond(A + B)$$
$$\mathtt{choose} : (\diamond A \times \diamond B) \to \diamond\big((A \times \diamond B) + (\diamond A \times B)\big)$$

McBride [3] defines a derivative operation ∂_x on types, where x is a type variable, and shows that it gives the type of one-hole contexts for any regular data type[1]. Here, x specifies the type of values that fill the hole.

In our setting, the analogous derivative $\partial_\diamond$ produces the type of one-hole contexts with holes for *events*, or equivalently *IO actions*. This means that the derivative of an event $\diamond A$ (or equivalently $\mathtt{IO\ a}$) is just its result type A. The hole is filled with the time the event occurred. This operation is heterogeneous in the sense that events in the same data structure may have different result types (such as $\diamond A \times \diamond B$).

The two derivative operations, ∂_x and $\partial_\diamond$, are defined together in Fig. 1.

3. Selective Choice

In general, the type of $\mathtt{choose}$ is given by

```
choose :: Generic a => a -> IO (∂ɪₒ a)
```

where $\partial_{\mathtt{IO}}$ is a Haskell type family corresponding to $\partial_\diamond$ and `Generic` is a pre-defined type class that allows type-directed programming. The intended semantics is that $\mathtt{choose\ a}$ is an event that triggers when any event inside of $\mathtt{a}$ does. Using $\mathtt{choose}$ minimizes the need for the user to deal with low-level concurrency primitives; the only place synchronization is needed is in the implementation of $\mathtt{choose}$.

The implementation is based on a helper function `locations` that finds all the events inside its argument. It returns a list containing, for each event $\mathtt{e}$ in the argument, an event that triggers when $\mathtt{e}$ does and that returns the corresponding one-hole context.

```
locations :: Generic a => a -> [IO (∂ɪₒ a)]
```

We implement $\mathtt{choose}$ using lower-level synchronization primitives (in this case, GHC's `MVars`) to determine which of these one-hole contexts triggered first.[2] The implementation also uses a helper function `spawnall :: a -> IO a` that spawns a new thread for every IO action in its argument and allows these threads to execute concurrently. The bulk of the effort in implementing $\mathtt{choose}$ goes into defining `locations`.

```
choose :: Generic a => a -> IO (∂ɪₒ a)
```

---

[1] A possibly-recursive type composed only of sums and products.

[2] Note that the type signatures and code presented are simplified to illustrate the main idea; the real implementation includes details about efficiency and more complicated typing features.

```
choose v = do
  win     <- newEmptyMVar
  v'      <- spawnall v
  threads <- forM (locations v') $ \result ->
               forkIO $ putMVar win =<< result
  readMVar win
```

3.1 Implementation (Work In Progress)

We currently have an in-progress Haskell implementation of $\mathtt{choose}$, available at `https://github.com/antalsz/choose-your-own-derivative`. This implementation makes extensive use of advanced GHC type-level programming features, including: `GHC.Generics`, to perform structural analysis on types; data type promotion, for basic dependently-typed programming; type families, or type-level functions, including $\partial_{\mathtt{IO}}$; GADTs; explicit type application; and reified constraints. These allow us to state and prove the theorems that ensure values of type $\partial_{\mathtt{IO}}$ A are well-formed.

3.2 Extensions

The `locations` function mentioned above does not depend on any details of $\mathtt{IO}$ in particular. In fact, the rules in Fig. 1 are independent of the semantics of $\diamond$. As part of ongoing work, we plan to generalize the behavior of `locations` and $\mathtt{choose}$ to other monads, including:

1. Other concurrency monads. Haskell has a rich library ecosystem, and $\mathtt{choose}$ does not inherently depend on the specifics of $\mathtt{IO}$.

2. Signals or events in graphical user interface libraries. For example, in Gtk2Hs [5], $\mathtt{choose}$ could act as a *signal combinator* for forwarding messages ("signals") throughout a DOM tree of widgets.

3. Parser combinators. Here, $\mathtt{choose\ p}$ would apply the first successful parser in $\mathtt{p}$, and return both the result of that parse and the remaining parsers. This allows, for example, parsing unordered lists of tokens.

4. Random generators, as in QuickCheck [1]. Here, $\mathtt{choose\ g}$ would nondeterministically generate an element from one of the component generators of $\mathtt{g}$.

Acknowledgments

References

[1] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[2] Simon Marlow. `async`: Run $\mathtt{IO}$ operations asynchronously and wait for their results. `http://hackage.haskell.org/package/async-2.1.0`, January 2016.

[3] Conor McBride. The derivative of a regular type is its type of one-hole contexts. Extended Abstract, 2001.

[4] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[5] Axel Simon, Duncan Coutts, et al. `gtk`: Binding to the Gtk+ graphical user interface library. `http://hackage.haskell.org/package/gtk-0.14.5`, June 2016.

2016/6/24