choose Your Own Derivative (Extended Abstract)

Jennifer Paykin Antal Spector-Zabusky Kenneth Foner
University of Pennsylvania, USA
{jpaykin,antsals,kfoner}@seas.upenn.edu

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

Keywords selective choice; derivatives; concurrency; Haskell

Abstract

We discuss a generalization of the synchronization mechanism selective choice. We argue that selective choice can be extended to synchronize arbitrary data structures of events, based on a typing paradigm introduced by McBride: the derivatives of recursive data types. We discuss our work in progress implementing generalized selective choice as a Haskell library based on generic programming.

1. Introduction

Synchronization is the hardest part of concurrent programming. As envisioned by Reppy [4], selective choice is a synchronization mechanism that takes a list of events, executes them concurrently, and returns the value of the first event to complete. We extend selective choice to synchronize arbitrary data structures of events, based on a type-level derivative operator inspired by McBride’s derivatives of recursive data types [3].

Selective choice. Consider implementing a Haskell function timeout that takes an amount of time and an IO action evt, and runs evt 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.

We can extend this idea even further to arbitrary data structures. For example, consider an abortable IO action: a pair of a threaded IO action along with its thread ID (used to kill it).

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 takes a “prioritized” list of IO actions, runs them all until one completes, and then aborts all of the lower-priority IO actions that did not complete:

runUntilFirst :: [Abort a] -> IO (a, [Abort a])
runUntilFirst actions = do
  (lower, (tid,a), higher) <- chooseAbort actions
  putStrLn (show tid ++ " completed!"
  runM_ abort lower
  return (a, higher)

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, which we can apply to arbitrary data structures, including trees and mutually recursive types. 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, represented by 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 (IO a) (IO 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, + for Either, × for (,), and + for IO, we have

choose : (a + b) -> (a + b)
choose : (a × b) -> (a × b)

choose : (a + b) -> (a × b) + (a × b)
We implement \( \partial_x \) 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 \( \partial_x \) produces the type of one-hole contexts with holes for \( \omega \), or equivalently \( \text{IO} \) actions. This means that the derivative of an event \( \omega A \) (or equivalently \( \text{IO} a \)) is just its result type \( A \). The “hole” corresponds just to the \( \omega \) (the \( \text{IO} \)), and not the \( A \). This operation is heterogeneous in the sense that events in the same data structure may have different result types (such as \( \omega A \times \omega B \)). The two derivative operations, \( \partial_x \) and \( \partial_o \), are defined together in Fig. 1.

3. **Selective Choice**

In general, the type of \( \text{choose} \) is given by

\[
\text{choose} :: \text{Generic} \ a \Rightarrow a \rightarrow \text{IO} \ (\partial_{10} a)
\]

where \( \partial_{10} \) is a Haskell type family corresponding to \( \partial \) and \text{Generic} is a pre-defined type class that allows type-directed programming. The intended semantics is that \( \text{choose} \ a \) is an event that triggers when any event inside of \( a \) does. Using \text{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 \text{choose}.

The implementation is based on a helper function \textit{locations}, which finds every event in its argument; it returns a list containing, for each event \( e \), an event that triggers when \( e \) does, returning the corresponding one-hole context.

\[
\text{locations} :: \text{Generic} \ a \Rightarrow a \rightarrow \left[ \text{IO} \ (\partial_{10} a) \right]
\]

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

\[
\begin{align*}
\text{choose} :: & \text{Generic} \ a \Rightarrow a \rightarrow \text{IO} \ (\partial_{10} a) \\
\text{choose} v = & \text{do} \\
& \text{win} \leftarrow \text{newEmptyMVar} \\
& v' \leftarrow \text{spawnall} \ v \\
& \text{threads} \leftarrow \text{forM} \ (\text{locations} \ v') \ [f \ \text{result}] \\
& \text{forkIO} \ [\text{putMVar} \ \text{win} \leftrightarrow \text{result}]
\end{align*}
\]

3.1 **Implementation (Work In Progress)**

We currently have an in-progress Haskell implementation of \text{choose}, available at \url{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 \( \partial_{10} \); GADTs; explicit type application; and reified constraints. These allow us to state and prove the theorems that ensure values of type \( \partial_{10} a \) are well-formed.

We have fully implemented the unified derivative of types given in Fig. 1, including an implementation of McBride-style derivatives as a consequence. The runtime behavior of \text{locations} and \text{choose} is complete, but the type-level proofs of correctness are in progress.

3.2 **Extensions**

The \textit{locations} function mentioned above does not depend on any details of \text{IO} in particular; the implementation actually has the more general type

\[
(\text{Generic} \ a, \text{Functor} \ f) \Rightarrow a \rightarrow [f \ (\partial f t \ a)]
\]

In fact, the rules in Fig. 1 are independent of the semantics of \( \partial \). As part of ongoing work, we plan to generalize the behavior of \text{choose} to other monads, including:

1. Other concurrency monads; \text{choose} does not inherently depend on the specifics of \text{IO}.
2. Signals or events in graphical user interface libraries. For example, in Gtk2Hs \(^5\), \text{choose} could act as a signal combinator for forwarding messages (“signals”) throughout a DOM tree of widgets.
3. Parser combinators. Here, \text{choose} \( p \) would apply the first successful parser in \( p \), 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, \text{choose} \( g \) would randomly generate an element from one of the component generators of \( g \).

**Acknowledgments**

This work was supported by NSF Graduate Research Fellowship no. DGE-1321851, NSF award no. 1521523, Expeditions in Computing: The Science of Deep Specification; and NSF award no. 1319880, Rich Type Inference for Functional Programming.

**References**


---

\(^1\) A type composed only of recursion (\( \mu \)), sums, and products.

\(^2\) Note that the type signatures and code presented are simplified to illustrate the main idea; the implementation includes details about efficiency and more complicated typing features.