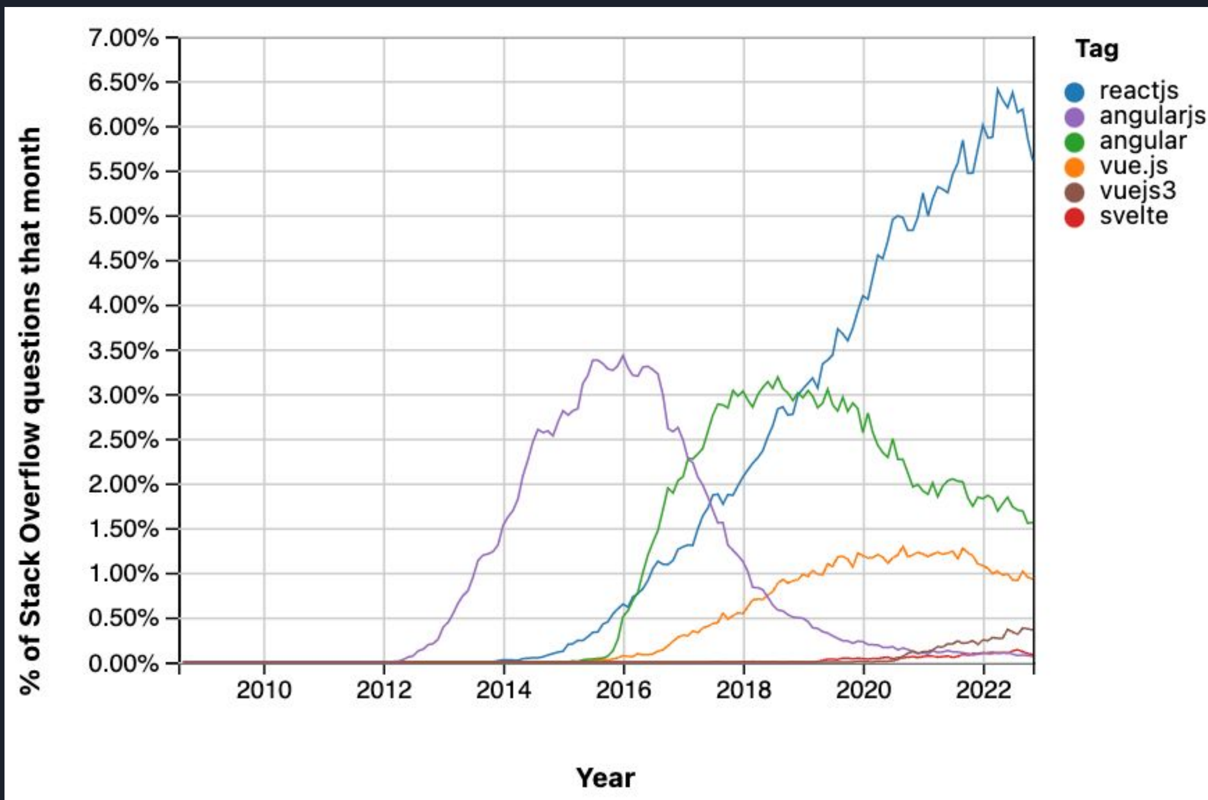
Harmonizing TypeScript and Haskell with
Functional Reactive Programming

Ryan Trinkle
Partner, Obsidian Systems
MuniHac 2023



[Frontend Frameworks Popularity]

Which **programming, scripting, and markup languages** have you done extensive development work in over the past year, and which do you want to work in over the next year? Stack Overflow Developer Survey 2023

Build better software, faster

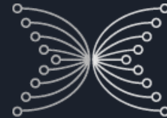
- Strong Types
- Functional Purity

⇒ Haskell

Obsidian Systems
https://obsidian.systems

- Started in 2015
- Haskell is our go-to language
- Lots of frontend dev
- Now 30 people

Digital Asset

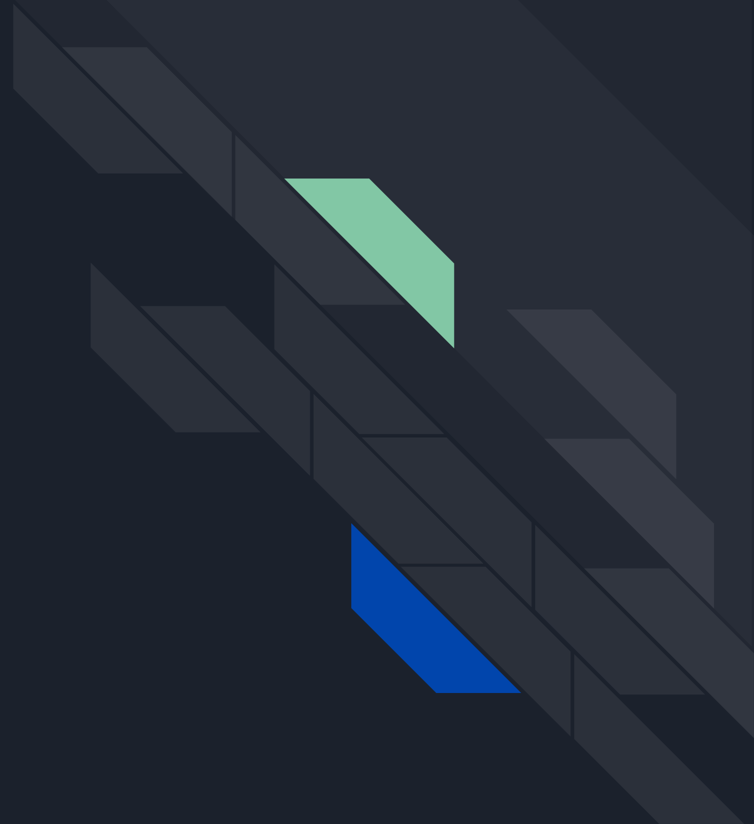


INPUT | OUTPUT

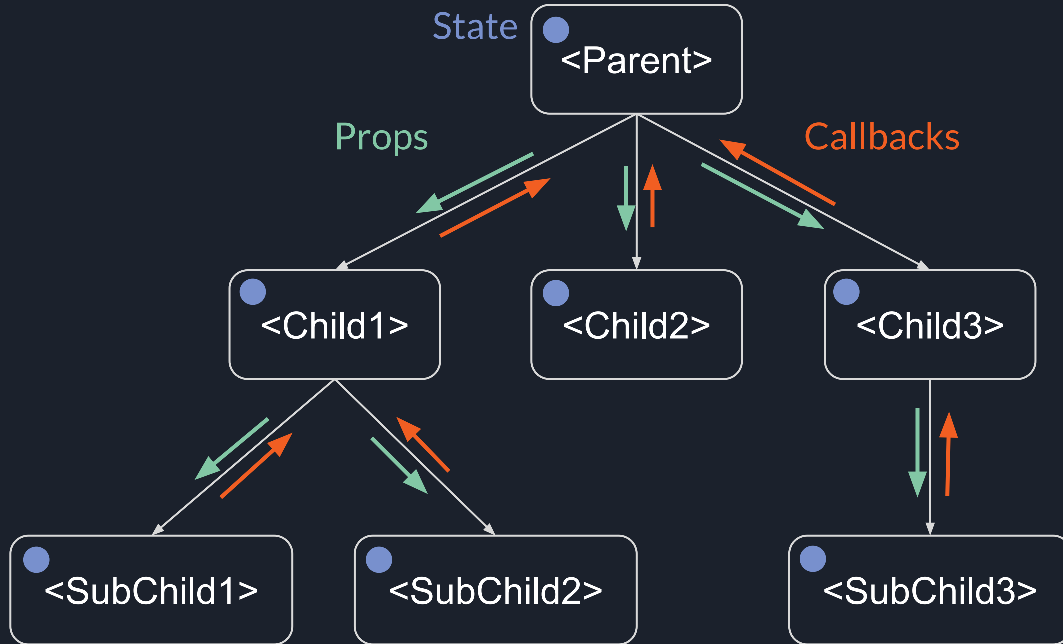
PDT PARTNERS

	JavaScript	TypeScript	Haskell
Easy to Learn			
Large Ecosystem			
Lambdas			
Types			
Purity			

A Brief Introduction to React



React Data Flow



React Props

```
function App() {
  return <SimplePropsTypescript v={10}/>;
}

function SimplePropsTypescript(props) {
  return (
     <>
     {JSON.stringify(props)}
     </>
  );
}
```

{"v":10}

React Callback and State

```
export function SimpleStateTypescript() {
  const [v, setV] = useState(0);
  const increment = useCallback(() => setV(v+1), [v]);
  return (
    <>
      <button onClick={increment}>+</button>
      {v.toString()}
    </>
  );
}
```

Haskell Package: `react`

Write React components in Haskell

```
component
  :: (props -> Hook Element)
  -> m (Component props ())
```

```
function SimplePropsTypescript(props) {
  return (
     <>
     {JSON.stringify(props)}
     </>
  );
}

simplePropsHaskell = component $ \props -> do
  propsJson <- liftJSM $ valToJSON props
  pure $ createFragment $ fromString $ fromJSString propsJson
```

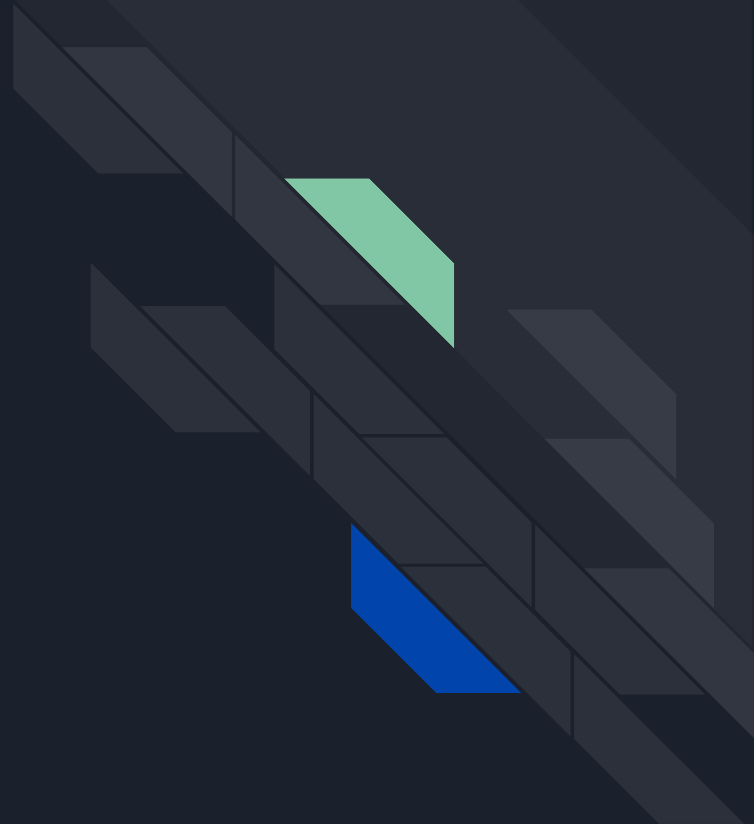
```
useState
  :: a
  -> Hook (a, a -> JSM ())
```

Haskell Components: Simple State TS

```
export function SimpleStateTypescript() {
  const [v, setV] = useState(0);
  const increment = useCallback(() => setV(v+1), [v]);
  return (
    <>
      <button onClick={increment}>+</button>
      {v.toString()}
    </>
  );
}
```

```
simpleStateHaskell = component $ \_ -> do
  (v, setV) <- useState (0 :: Int)
  increment <- useCallback (\_ _ _ -> setV (v + 1)) (Just [toJSVal v])
  pure $ createFragment
    [ createElement "button" ("onClick" =: increment) ["+"]
    , fromString $ show v
    ]
```

Pure Functional Programming



Referential transparency

Value is determined entirely by its definition

✅	x = 1 + 2	y = x * 3	let z = f(z)
❌	x.modify()	free(y)	z := z + 1

Equational Reasoning

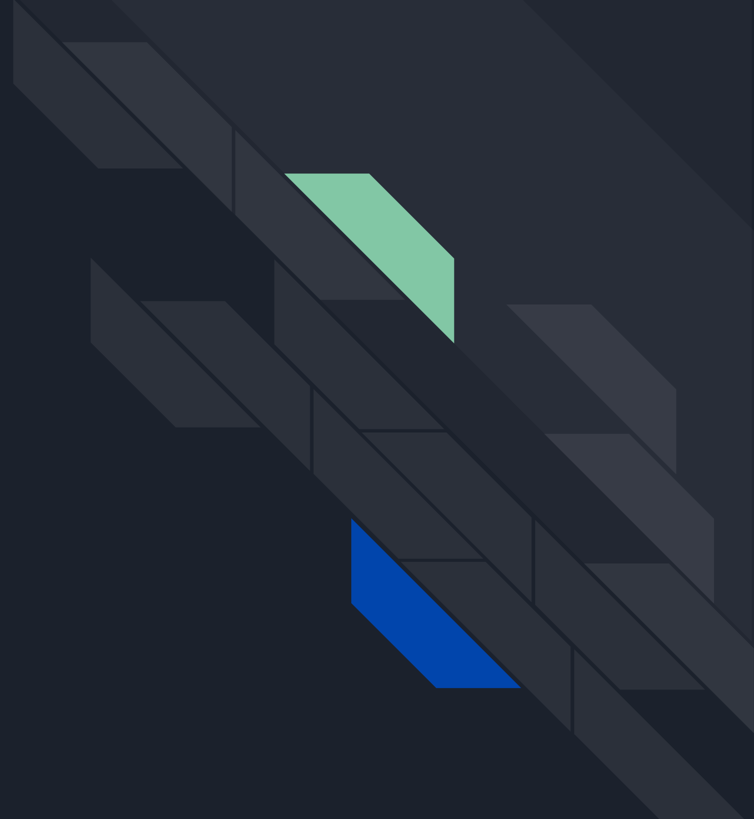
Reorder

```
let y = f x; z = g x ⇔ let z = g x; y = f x
```

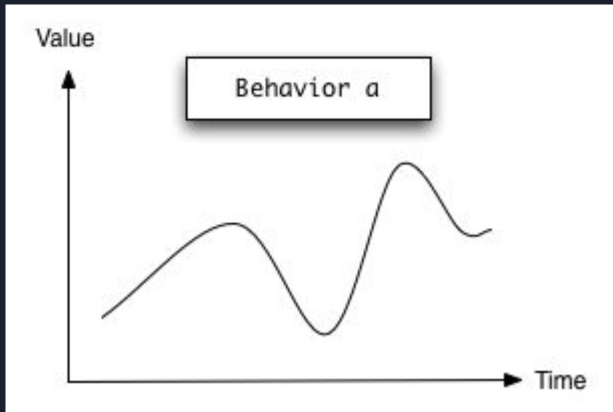
Deduplicate

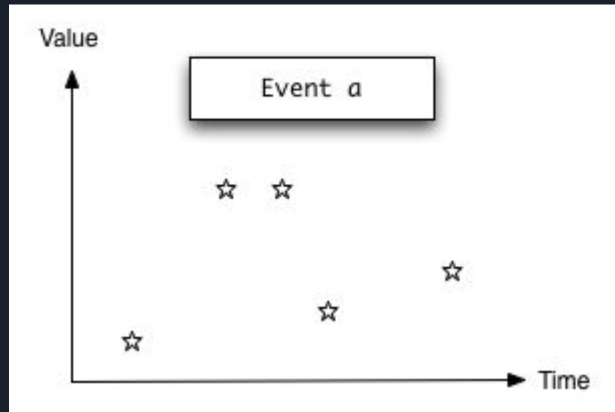
```
let y = f x; z = f x in [y, z] ⇔ let y = f x in [y, y]
```

How can we make React pure?



Functional Reactive Programming



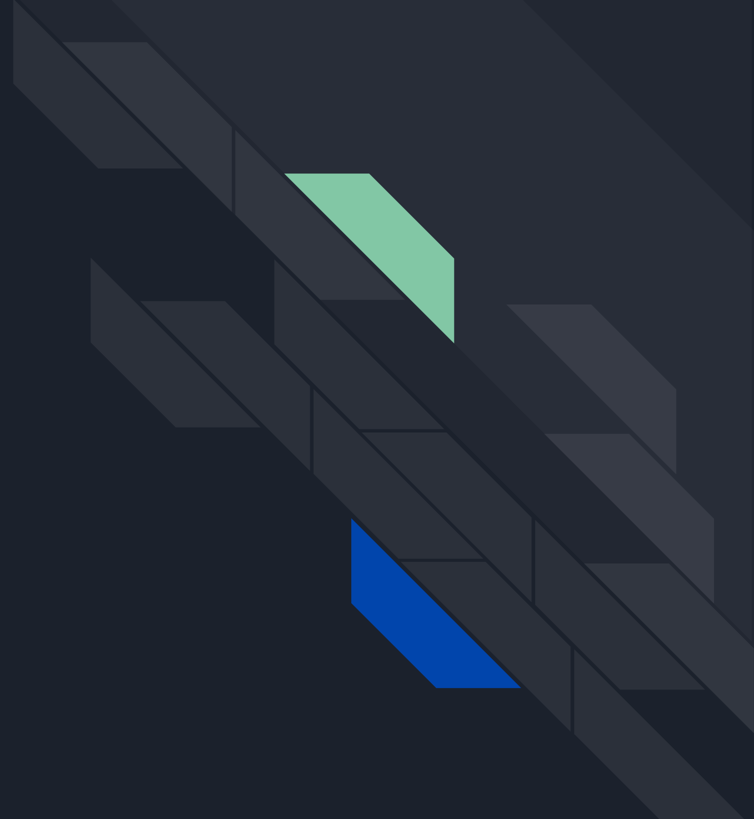


Are they pure?

Yes!

See reactive-banana on Hackage. Links in talk description.

Functional Reactive Programming provides *pure* time-varying values



What changes?

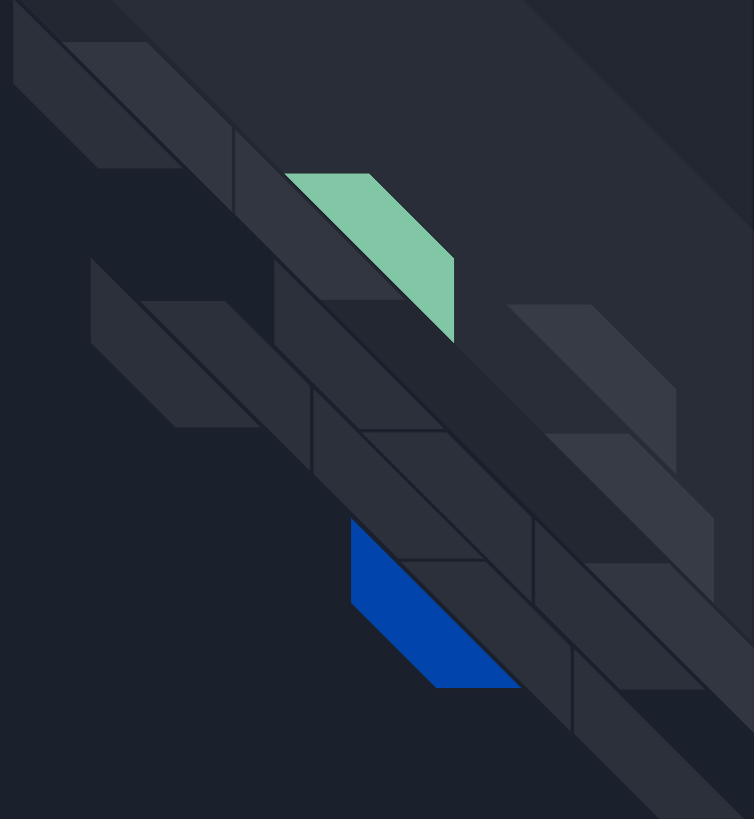
```
# React
const [clicked, setClicked] = useState(false);
...
<button onClick={() => setClicked(true)}>
```

```
# Reflex
clicked <- hold False (True <$ onClick)
...
onClick <- button "+"
```

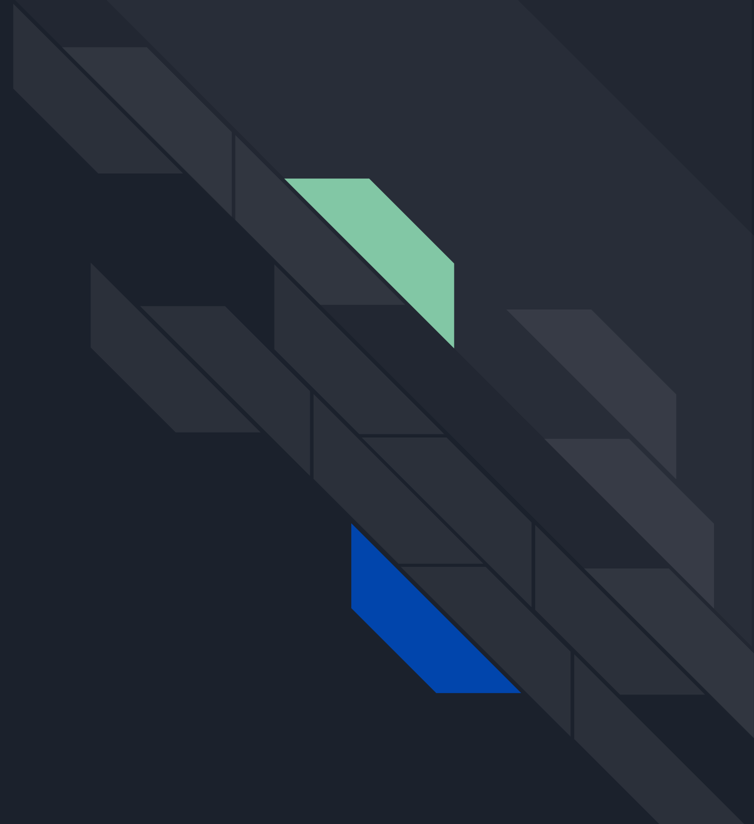
What changes?

Callback	⟹	RecursiveDo
`clicked :: Bool`	⟹	`clicked :: Dynamic Bool`
Function Re-runs	⟹	Function Runs Once
`setClicked` may be called elsewhere	⟹	`clicked` cannot be affected by other code
Impure	⟹	Pure

Callback Hell

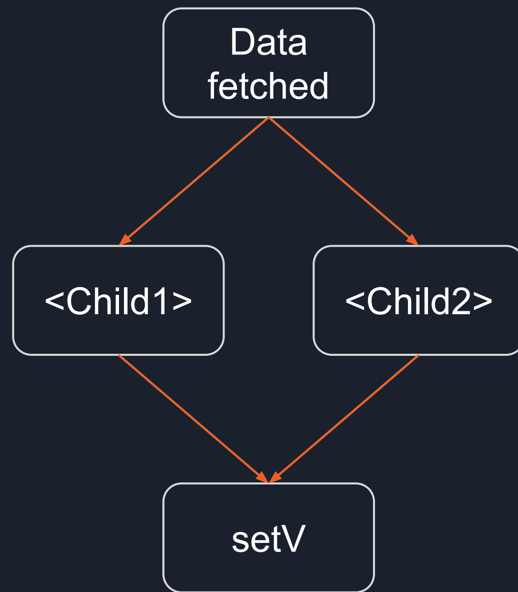


Callbacks *only* do side-effects



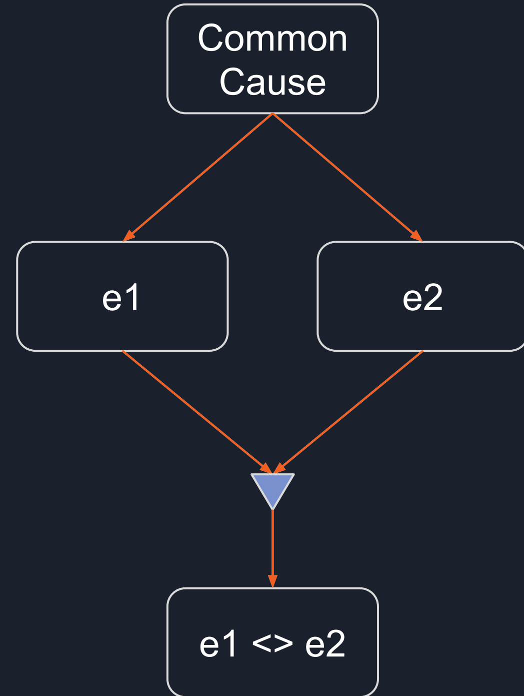
Callback Hell: Diamond

```
function Diamond() {
  const [v, setV] = useState('');

  return (
    <div>
      <Child1 gotData={() => setV(1)} />
      <Child2 gotData={() => setV(2)} />
      <Display data={v} />
    </div>
  );
}
```



Callback Hell: FRP's Solution

- Explicitly combine simultaneous events



Haskell Package: `reflex-react`

Write React components with Reflex-DOM

```
reflexComponent
  :: ...
  -> (Dynamic props -> Widget ())
  -> m (Component JSVal ())
```

```
function SimplePropsTypescript(props) {
  return (
     <>
     {JSON.stringify(props)}
     </>
  );
}

simplePropsReflex = reflexComponent "div" valToJSON $ \props -> do
  dynText $ fmap fromJSString props
```

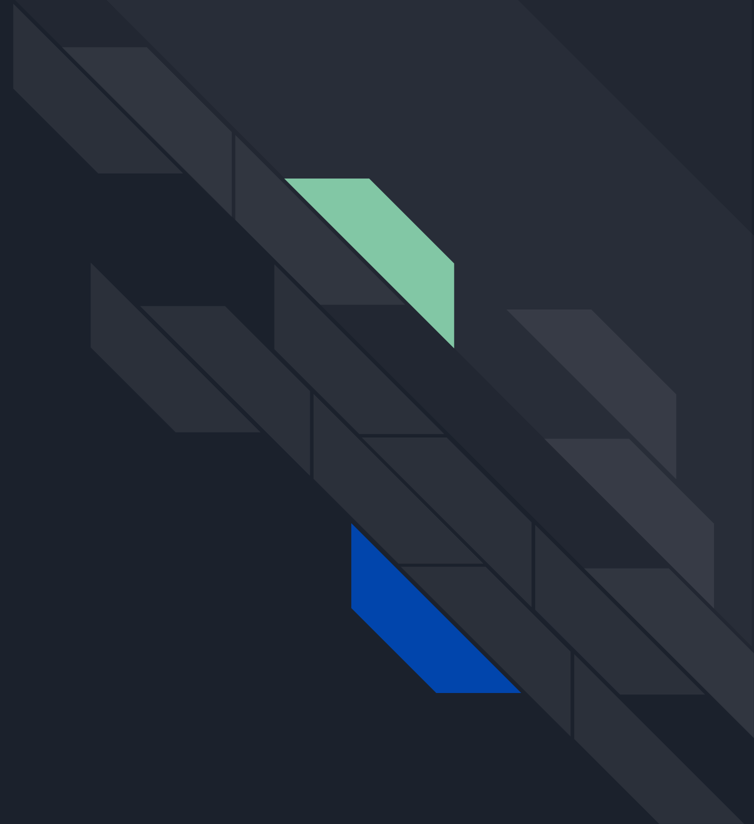
```
export function SimpleStateTypescript() {
  const [v, setV] = useState(0);
  const increment = useCallback(() => setV(v+1), [v]);
  return (
    <>
      <button onClick={increment}>+</button>
      {v.toString()}
    </>
  );
}
```

Haskell Components: Simple State HS

```
simpleStateReflex = reflexComponent "div" valToJSON $ \_ -> do
  clicked <- button "+"
  numClicks <- count clicked
  display numClicks
```

React	Reflex
Props	FRP-based regular function arguments
`useState`	`holdDyn`
`useReducer`	`foldDyn`
`useEffect`	`performEvent`
`useMemo/useCallback`	Unnecessary
`useContext`	ReaderT monad transformer

Getting Started



```
# Install next-haskell (per-project)
npm install --save next-haskell

# Add to next.config.js
const withHaskell = require('next-haskell');
/** @type {import('next').NextConfig} */
const nextConfig = withHaskell({});
module.exports = nextConfig;
```

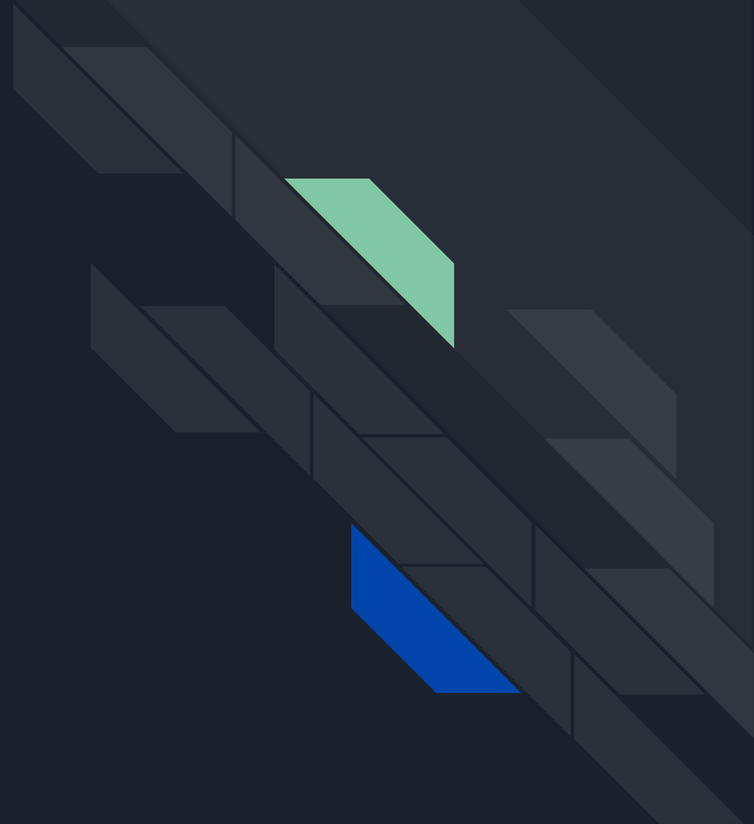
```
# Install Nix (system-wide)
sh <(curl -L https://nixos.org/nix/install) --daemon

# Set up Nix caches in /etc/nix.conf
binary-caches = https://cache.nixos.org https://nixcache.reflex-frp.org
binary-cache-public-keys =
cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=
ryantrinkle.com-1:JJiAKaRv9mWgpVAz8dwewnZe0AzzEAzPkagE9SP5NWI=
binary-caches-parallel-connections = 40

# Unpack our example Haskell component
curl -L https://github.com/obsidiansystems/reflex-react-example/tarball/master |
tar -xz --strip-components=1 --wildcards '*/components/haskell'
```

Getting Started: Simple Component

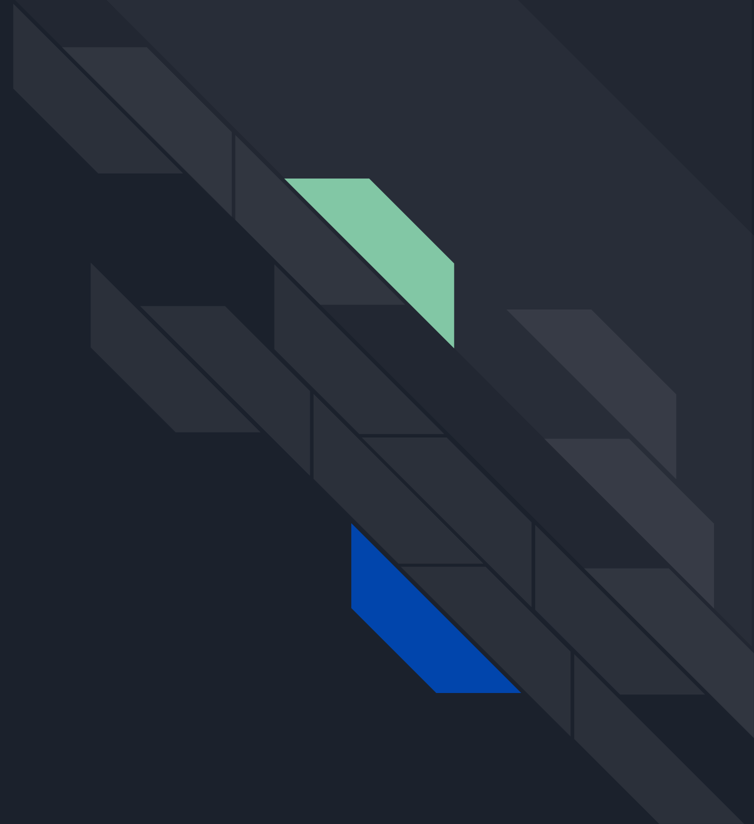
```
{-# LANGUAGE TemplateHaskell #-}
import React

showJsonProps :: ReaderT React JSM (Component JSVal ())
showJsonProps = component $ \props -> do
  propsJson <- liftJSM $ valToJSON props
  pure $ fromString $ fromJSString propsJson

mainExportsToJS
  [ ‘showJsonProps
  ]
```

```
‘use client’;
import Haskell from './haskell/haskell-components.cabal';

function SomeComponent(props) {
  return (<>
    <Haskell.showJsonProps someProp=”test” />
  </>);
}
```

Questions?



Packages



- Write React components in Haskell
- Designed for React 18
- Supports the most common built-in hooks
- Builds with GHC for development (fast reload)
- Builds with GHCJS for production (deployable JS)

- Provides FRP interface to React
- Uses reflex-dom, not ReactDOM

NPM Package: @ryantrinkle/`haskell-loader`

- A webpack loader for Haskell
- Allows you to import `.cabal` files just like `.js` files
- The cabal file must have a single executable, which exports values to JavaScript when it runs
- In development, runs with ghcid and jsaddle-warp
- In production, compiles with GHCJS

- Sets up your next.config.js to use @ryantrinkle/`haskell-loader`