A semantics for React

Sam Lindley and Toby Ueno

The University of Edinburgh

Huawei Joint Lab Meeting, Hangzhou, April 2026

ReactFX: reactive programming with effect handlers

Project start date: September 2025

PI — Sam Lindley

PhD student — Toby Ueno

ReactFX: reactive programming with effects and handlers

Research objectives

- **Foundations**: unify synchronous effects from the programming language with asynchronous events from the environment
- **Implementations**: experiment with research languages, e.g., Links, Koka, OCaml
- **Case studies**: React-style web applications, spreadsheets, etc.
- **Effect typing**: exploit for optimisation and modularity
- **Incremental updates**: use effect handlers to abstract over incremental updates to virtual DOM
- **Pre-emptive concurrency:** synergy with Cangjie implementation of effect handlers

Understanding React

- What are components?
  - They look like stateful functions — *in what sense are they functions?*
- Goal: **translate** React into something more well-understood
  - **Expressivity** result [Felleisen, 1991]
  - What programming language features are required?
- We define the following:
  - λ_{react} : a simplified React-like source language (inspired by React-tRace [Lee, Ahn, Yee, OOPSLA 2025])
  - λ_{ref} : a canonical target language with ML-style references
  - a **translation** $[\![-]\!]$ from λ_{react} into λ_{ref}

Core language

$$\text{Values } v, w ::= x \mid () \mid s \mid \lambda x.e \mid [] \mid v :: w$$
$$\mid \text{text } v \mid \text{tag}(v, w) \mid \text{attr}(v, w) \mid \text{onClick } v$$
$$\text{Computations } e ::= \text{return } v \mid \text{let } x = e \text{ in } e' \mid v\ w$$
$$\mid \text{case } v\ \{[] \Rightarrow e \mid x :: xs \Rightarrow e'\}$$
$$\text{Types } A, B ::= \mathbf{1} \mid \text{String} \mid A \to B \mid \text{List } A \mid \text{View} \mid \text{Attr}$$

- Fine-grain call-by-value
- Types & constructors for HTML

Source language: λ_{react}

Values $v, w ::= \cdots \mid \mathsf{comp}\{b\} \mid \mathsf{set}@_p^\ell$

Bodies $b ::= e \mid \mathsf{let}\ x = e\ \mathsf{in}\ b \mid \mathsf{let}\ x, x_{\mathsf{set}} = \mathsf{useState}^\ell\ v\ \mathsf{in}\ b$

Types $A, B ::= \cdots \mid A \xrightarrow{E} B$

Effect $E ::= \mathsf{Pure} \mid \mathsf{Mut}$

Record $\mathcal{R} ::= \emptyset \mid \mathcal{R}, \ell : A$

Syntactically force useState to occur only at component top-level!

React versus λ_{react}

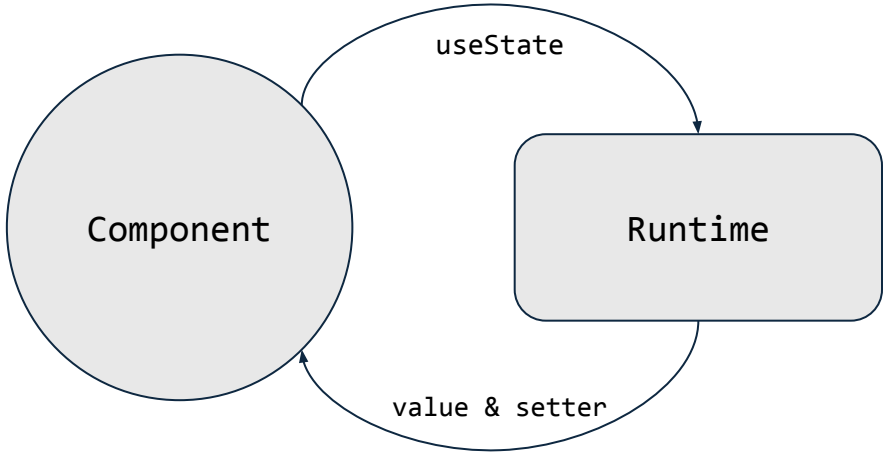
```
const App = z => {
  const [x, x_set] = useState(z);
  const y = x * x;
  return (
    <p onClick={() => {x_set (x + 1)}>
      {y}
    </p>
  )
}
```

```
let App = λz.comp {
  let x, x_set = useState z in
  let y = x * x in
  return (
    <p onClick={λ().{x_set (x + 1)}>
      {y}
    </p>
  )
}
```

λ_{react} design

- Looks & acts like React
- **Abstract away** from details
- Focus on “essence” of React
  - Components
  - State
  - Runtime

```
> ('b' + 'a' + + 'a' + 'a').toLowerCase()
< 'banana'
```



λ_{react} is type-safe!

```
const App = z => {
  const [x, x_set] = useState(z);
  x_set (x + 1);
  const y = x * x;
  return (
    <p onClick={() => {x_set (x + 1)}>
      {y}
    </p>
  )
}
```


Infinite re-render!

λ_{react} is type-safe!

```
const App = z => {
  const [x, x_set] = useState(z);
  x_set (x + 1);
  const y = x * x;
  return (
    <p onClick={() => {x_set (x + 1)}>
      {y}
    </p>
  )
}
```

```
let App = λz.comp {
  let x, x_set = useState z in
  let () = x_set (x + 1) in
  let y = x * x in
  return (
    <p onClick={λ().{x_set (x + 1)}>
      {y}
    </p>
  )
}
```

λ_{react} is type-safe!

Error: Mutation effect introduced in component body!

```
let App = λz.comp {
  let x, x_set = useState z in
  let () = x_set (x + 1) in
  let y = x * x in
  return (
    <p onClick={λ().{x_set (x + 1)}>
      {y}
    </p>
  )
}
```

$$\frac{\Gamma \vdash v_1 : A \xrightarrow{\text{Mut}} \mathbf{1} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \mathbf{1}\ !\ \text{Mut}}\ \text{TE-App}$$

$$\frac{\Gamma \vdash e : A\ !\ \text{Pure} \qquad \Gamma, x : A \vdash \hat{e} : \mathcal{R}}{\Gamma \vdash \text{let } x = e \text{ in } \hat{e} : \mathcal{R}}\ \text{TB-Let}$$

Target language: λ_{ref}

$$\text{Values } v, w ::= \cdots \mid r$$

$$\text{Computations } e ::= \cdots \mid \text{ref } v \mid !v \mid v := w$$

$$\text{Types } A, B ::= \cdots \mid \text{Ref } A$$

λ_{react} to λ_{ref} by example

```
λz.comp {
  let x, x_set = useState z in
  let y = x * x in
  return (
    <p onClick={λ().{x_set (x + 1)}>
      {y}
    </p>
  )
}
```

```
λz.λ().
  let x = return z in
  let x_l = ref(x) in
  let x_set = λw.x_l := w in
  return λ().
    let x = !x_l in
    let x_set = λw.x_l := w in
    let y = x * x in
    return (
      <p onClick={λ().{x_set (x + 1)}>
        {y}
      </p>
    )
```

Initial

Successive

λ_{react} to λ_{ref}

- Key insight: components have *two* different interpretations!
  - When *first* run, useState **initializes** a reference
  - On *successive* runs, useState **retrieves** that reference

$$[\![\mathsf{let}\ x, x_{\mathsf{set}} = \mathsf{useState}^{\ell} v\ \mathsf{in}\ b]\!]^{\mathsf{Init}} \approx \mathsf{let}\ x = \mathsf{return}\ [\![v]\!]\ \mathsf{in}$$
$$\mathsf{let}\ x_{\ell} = \mathsf{ref}(x)\ \mathsf{in}$$
$$\mathsf{let}\ x_{\mathsf{set}} = \lambda y.x_{\ell} := y\ \mathsf{in}$$
$$[\![b]\!]^{\mathsf{Init}}$$

λ_{react} to λ_{ref}

- Key insight: components have *two* different interpretations!
  - When *first* run, useState **initializes** a reference
  - On *successive* runs, useState **retrieves** that reference

$$[\![\mathsf{let}\ x, x_{\mathsf{set}} = \mathsf{useState}^{\ell} v\ \mathsf{in}\ b]\!]^{\mathsf{Succ}} \approx \mathsf{let}\ x = !x_{\ell}\ \mathsf{in}$$
$$\mathsf{let}\ x_{\mathsf{set}} = \lambda y.x_{\ell} := y\ \mathsf{in}$$
$$[\![b]\!]^{\mathsf{Succ}}$$

What next for λ_{react}?

$$m, e \mapsto e', m'$$

$$\Longrightarrow$$

$$[\![m]\!], [\![e]\!] \mapsto^* [\![e']\!], [\![m']\!]$$

- Proof of operational correspondence
  - Show our translation is “good” (almost done)

- Possible extensions: capture more of React
  - useEffect?
  - Reconciliation?
  - Custom hooks

Future directions: beyond λ_{react}

- How do we **react** with the outside world?
  - λ_{react} assumes discrete, synchronous events
  - What about continuous signals?
    - Draw on **functional reactive programming**
- How do effect handlers fit into this picture?
  - Connecting effect handlers and React
  - Asynchronous effects [Ahman and Pretnar, 2024]
  - …