# A Semantics for the Essence of React

**Magnus Madsen**
Aarhus University, Denmark
`https://www.cs.au.dk/~magnusm/`
magnusm@cs.au.dk

**Ondřej Lhoták**
University of Waterloo, Canada
`https://plg.uwaterloo.ca/~olhotak/`
olhotak@uwaterloo.ca

**Frank Tip**
Northeastern University, USA
`https://www.franktip.org`
f.tip@northeastern.edu

## Abstract

Traditionally, web applications have been written as HTML pages with embedded JavaScript code that implements dynamic and interactive features by manipulating the Document Object Model (DOM) through a low-level browser API. However, this unprincipled approach leads to code that is brittle, difficult to understand, non-modular, and does not facilitate incremental update of user-interfaces in response to state changes.

React is a popular framework for constructing web applications that aims to overcome these problems. React applications are written in a declarative and object-oriented style, and consist of components that are organized in a tree structure. Each component has a set of properties representing input parameters, a state consisting of values that may vary over time, and a render method that declaratively specifies the subcomponents of the component. React's concept of reconciliation determines the impact of state changes and updates the user-interface incrementally by selective mounting and unmounting of subcomponents. At designated points, the React framework invokes lifecycle hooks that enable programmers to perform actions outside the framework such as acquiring and releasing resources needed by a component.

These mechanisms exhibit considerable complexity, but, to our knowledge, no formal specification of React's semantics exists. This paper presents a small-step operational semantics that captures the essence of React, as a first step towards a long-term goal of developing automatic tools for program understanding, automatic testing, and bug finding for React web applications. To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a well-behaved component and prove that well-behavedness is preserved by these operations.

**2012 ACM Subject Classification** Theory of computation → Operational semantics; Software and its engineering → Semantics

**Keywords and phrases** JavaScript, React, operational semantics, lifecycle, reconciliation

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2020.12

**Related Version** A technical report with proofs and additional examples is available [16].

**Funding** The third author was supported in part by National Science Foundation grant CCF-1715153. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

**Acknowledgements** The authors are grateful to the anonymous reviewers for their insightful comments.

## 1 Introduction

A web application is a program where the user-interface runs in a web browser. Traditionally, such applications have been written as HTML pages that contain embedded JavaScript code that implements dynamic and interactive features, such as input validation or data visualization, by manipulating the Document Object Model (DOM) through a low-level browser API. Using the DOM, the programmer can add, remove, or mutate HTML elements directly. While expressive, this unprincipled approach has several disadvantages. First, direct mutation of the DOM leads to brittle and difficult to understand code. Second, using this approach, it is difficult to design reusable user-interface components and libraries. Third, this approach does not easily lend itself to designs where the user-interface of a web application is updated *incrementally* in response to user input or new data received from a server. As a result, traditional web applications are often buggy and difficult to maintain [5, 6, 20, 21, 19].

The React framework [9] was developed to address these concerns. A React application does not manipulate the DOM directly but instead operates on a "virtual DOM", by constructing React components that are rendered incrementally as their properties and state change. Such components are written in a declarative and object-oriented programming style, where classes represent components, and reusing a component is as simple as creating an instance of a class. A *React application* is structured as a tree, where a root component represents the top-level element of the user-interface, and where a (possibly dynamically varying) set of subcomponents correspond to widgets within that page. A *React component* has three key constituents: (i) a set of *properties* representing input parameters needed to configure the component, (ii) an internal *state* consisting of values that may vary over time, and (iii) a `render` method that specifies how a component is rendered by returning a subtree composed of a mix of subcomponents and HTML elements. The process of creating and updating the user-interface of a React application is defined in terms of *mounting* and *unmounting* operations, corresponding to the addition and removal of subcomponents. A key feature of React is its concept of *reconciliation*, which entails determining those parts of a page that are affected by state changes and updating them incrementally by selectively mounting and unmounting subcomponents. At key points during this process (e.g., when components are mounted or unmounted), the React framework invokes *lifecycle* hooks—callback methods that enable programmers to perform actions, e.g., to fetch data from a remote server or to store data locally in `localStorage`.

Today, React is one of the world's most popular web frameworks. On StackOverflow, a popular question-and-answer forum for programmers, more than 181,554 questions are tagged reactjs. In comparison, the reactjs tag is more popular than the perl, scala, swing, or typescript tags. On GitHub, React is the fourth most starred repository, with more than 142,000 stars. On NPM, the package manager for Node.js, React has more than 20,000,000 downloads per month.

While React helps programmers structure their web application as a collection of modular components, it comes with its own set of challenges and bug patterns that new programmers must learn to avoid. For example, the intricate control- and dataflow can make it exceedingly difficult to understand how state changes in one component affect other components. As another example, the complex interplay between the component lifecycle methods and the reconciliation algorithm can be difficult to understand.

To enable the construction of tools for reasoning about the behavior of React applications, for automatic testing, and for bug finding, a precise understanding of the semantics of React is required. This paper establishes such an understanding, in the form of a formal semantics that captures the essence of React. Our semantics is based on $\lambda_{\text{js}}$ [11], and precisely models the key aspects of React:

(i) mounting and unmounting of components,
(ii) reconciliation of component descriptors and mounted components, and
(iii) the semantics of state changes.

To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a *well-behaved* component by imposing a ranking function on components, and requiring that the `render` method of a component only returns components of strictly smaller rank. We then prove that well-behavedness is preserved by these operations.

In this paper, we focus on the core of React version `16.x`. In the `16.x` series, React has undergone some changes in the supported lifecycle methods, but those are mostly orthogonal to our work. React 16.8 introduced *React hooks*, a new, optional mechanism for state management in a functional style that avoids the use of classes. To our knowledge, there is no plan to change or remove the current mechanism for state management, and is is unclear to what extent the community will be adopting React hooks. In the paper, we focus on traditional state management as used in current React applications.

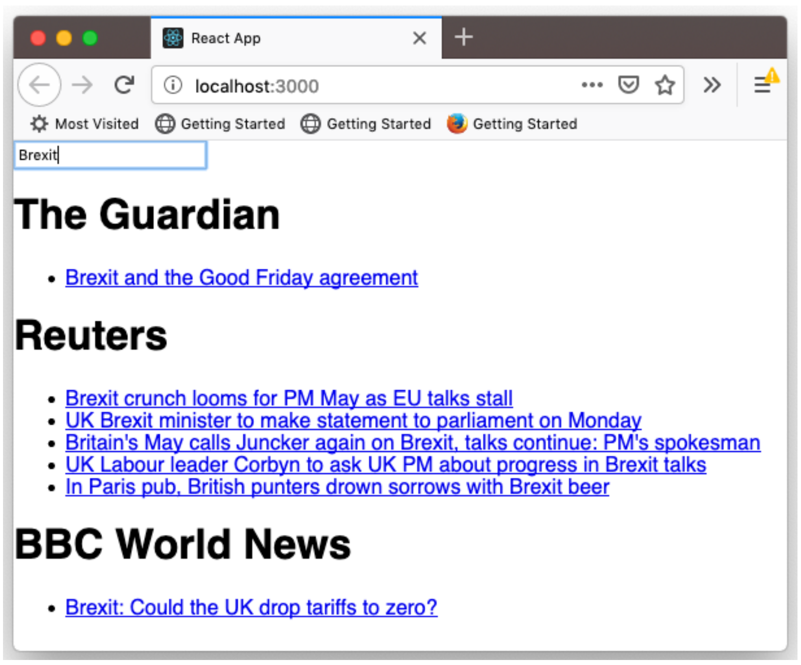
The remainder of this paper is organized as follows. Section 2 uses a small React application as an example to illustrate the key concepts and terminology associated with React. Section 3 presents a small-step operational semantics for the essence of React. Section 4 defines a well-behavedness property for components and demonstrates that well-behavedness is preserved by the key operations. Section 5 discusses how lifecycle hooks can be modeled. Related work is discussed in Section 6. Finally, Section 7 presents conclusions and directions for future work.

## 2 React

We will review the key concepts and terminology of React using a small React application that illustrates some of the typical requirements that a modern web developer must deal with. This includes fetching data and periodically receiving updates from the server, updating the browser's Document Object Model (DOM) to reflect the latest data, and filtering data based on user input. In pure JavaScript these steps can be difficult to manage, but React makes these steps easy to express.

Our example application receives RSS feeds from several news sites and, for each feed, displays the title of each news article. Clicking on the title will navigate the user to the full article on newspaper website. To focus on a specific news topic, the user may enter a keyword in the box at the top of the window to remove from the view any news items that do not contain the specified keyword. Figure 1 shows a screenshot of the application after the user has typed the word "brexit" in the box. The news feeds are polled every 5 seconds and the display is updated when existing news items disappear, and when additional news items appear. Note that such updates are performed *incrementally*, i.e., only the changed parts of the web page's DOM representation are updated and re-rendered.

In general, a *React application* is organized as as a tree of *React components*, each of which is self-contained UI widget that may be composed of *subcomponents*. React components are either instances of classes or they are HTML elements such as buttons or text fields. Our example application consists of a *root component* `App` that has 4 subcomponents: a text field and one subcomponent for each news feed, which is an instance of the `RssFeed` class. Each React component has three central constituents: A set of *properties*, an internal *state*, and a



The Guardian

- Brexit and the Good Friday agreement

Reuters

- Brexit crunch looms for PM May as EU talks stall
- UK Brexit minister to make statement to parliament on Monday
- Britain's May calls Juncker again on Brexit, talks continue: PM's spokesman
- UK Labour leader Corbyn to ask UK PM about progress in Brexit talks
- In Paris pub, British punters drown sorrows with Brexit beer

BBC World News

- Brexit: Could the UK drop tariffs to zero?

**Figure 1** Screenshot of our example React application. The screenshot shows the set of articles from news feeds from The Guardian, Reuters, and BBC World News after the user has entered the search term "Brexit".

`render` method. The properties are a form of input parameters typically used to configure the component. The state holds time-varying values, e.g., the values of input fields. The `render` method is used to draw the component by returning a subtree composed of a mix of subcomponents and HTML elements. In the case of our example application, the number of subcomponents is fixed because it depends on the number of news feeds being monitored, which is fixed. However, in general the component tree is not static, as a `render` method can vary the tree returned based on a component's properties and state. For example, one can easily imagine adding a feature that would allow the user to subscribe to additional news feeds, so that the number of components would vary dynamically as well.

The process of creating and updating the user-interface of a React application is defined in terms of *mounting* and *unmounting* operations. Here, mounting an application involves instantiating the class corresponding to its root component, rendering it by calling its `render` method, and recursively mounting its subcomponents. The React framework automatically takes care of all of this. To do so, React must be informed explicitly when state changes occur, by invoking the `setState` method with an object that specifies the state changes. When state changes occur, React will invoke the `render` method of the affected components to update the user-interface appropriately. However, changes are applied *incrementally*: React's *reconciliation* mechanism ensures that state changes do not require recomputation and re-rendering of the entire component tree, but only of those components affected by the state change. If a state change has the effect of removing a subcomponent, such a subcomponent is *unmounted*, i.e., the subcomponent is removed from its parent, and cleanup actions are performed as necessary. At designated points in the execution of a React application (e.g., prior to and upon completion of mounting and unmounting operations and when state changes occur), so-called *lifecycle methods* are invoked by the React framework.

Lifecycle methods are declared in classes corresponding to React components and can perform any programmer-specified action. Typically, lifecycle methods are used to initiate network requests to fetch data or initialize resources when a component is mounted, and to free resources when a component is unmounted.

Figure 2 shows the complete source code for the React application shown in Figure 1. The application consists of two classes: `App` (lines 7–27) and `RSSFeed` (lines 28–60). Each React class has a constructor method that is responsible for initializing the component's state. The state of the root component `App` includes a field[1] `filter` that will be used to determine which items should be selected from the newsfeeds. The constructor of the root component `App` (line 8–11) initializes the `filter` field to the empty string, reflecting that, by default, no news items should be filtered out. Next, on lines 12–16, a field `feeds` is initialized with an array that contains the URLs for the news feeds. Lines 17–23 show the `render` method for class `App`. In general, a `render` method of a React class produces a *component descriptor*, i.e., a declarative description of the component's subcomponents that are to be mounted/reconciled by React. Here, the `render` method returns a `<div>` tag containing a text field (line 19), and a sequence of `RSSFeed` components (lines 20–21). Line 19 specifies that entering or changing the text in the text field will cause the method `notifyChange` (lines 24–26) to be invoked with the current text as an argument. Lines 20–21 map a function over the `feeds` array to create an array of `RSSFeed` components, passing each field's URL and title, and the current value of the `filter` property. The `notifyChange` method on lines 24–26 specifies that React's `setState` function should be invoked, passing in an object with a property `filter` that is bound to the current value of the filter. This illustrates how React merges a form of declarative and object-oriented programming: The `render` methods return a declarative description of subcomponents that React then instantiates and maintains as objects.

The state of an `RSSFeed` component consists of a field `items` that represents the current items of the corresponding news feed, and the constructor (lines 29–32) initializes this field with an empty array. Next, the lifecycle methods `componentWillMount` (lines 33–36) and `componentWillUnmount` (lines 37–39) are defined. The former specifies that, when an `RSSFeed` component mounts, the `doUpdate` method should be invoked (line 34) immediately, and then invoked periodically (line 35) every 5 seconds. The latter specifies that, upon unmounting an `RSSFeed` component, the timer should be cleared (line 38). The `doUpdate` method (lines 40–46) asynchronously requests content from an RSS feed on line 42, using a proxy to enable cross-origin requests that would otherwise be disallowed due to browser's same-origin policy. The contents of the feed are parsed on line 43, and the `RSSFeed` component's state is updated on line 44 by invoking `setState` to set the `items` property to the news items in the feed's contents. The `RSSFeed` component's `render` method (lines 49–59) makes use of an auxiliary method `matchesKeyword` (lines 47–48) to determine if a given newsfeed item matches the filter if a filter is specified (if no filter is specified, all items match). The `render` method returns a `div` element (line 51) containing a title (line 52) and a list of news items extracted from the feed (line 55). The latter is constructed by filtering the `items` using the `matchesKeyword` function and creating an `li` (list item) tag for each item containing a hyperlink that is created using the title and URL obtained from the news feed.

We conclude from this example that React enables the construction of sophisticated interactive web applications, for which the user-interface is modular and incrementally maintained in response to property and state changes. React applications are remarkably

[1] In this paper, we will use the term "field" to refer to object fields and the term "property" to refer to the inputs provided to React components.

```
import React, {Component} from 'react';
import './App.css';

let Parser = require('rss-parser');
let parser = new Parser();

class App extends Component {
  constructor() {
       super();
       this.state = {filter: ""};
  }
  feeds = [
    { title: "The Guardian", url: "https://www.theguardian.com/world/zimbabwe/rss" },
    { title: "Reuters", url: "http://feeds.reuters.com/Reuters/worldNews"},
    { title: "BBC World News", url: "http://feeds.bbci.co.uk/news/world/rss.xml"}
  ];
  render = () => {
    return <div>
      <input type="text" onChange={this.notifyChange}/>
      { this.feeds.
          map((feed) => <RssFeed title={feed.title} url={feed.url} filter={this.state.filter}/>)}
    </div>
  }
  notifyChange = (e) => {
    this.setState({filter: e.target.value});
  }
}
class RssFeed extends Component {
    constructor() {
        super();
        this.state = {items: []};
    }
    componentWillMount = () => {
        this.doUpdate()
        this.timer = setInterval(this.doUpdate, 5000)
    }
    componentWillUnmount = () => {
        clearInterval(this.timer)
    }
    doUpdate = () => { // use "cors-anywhere" proxy to add CORS headers to the proxied request
        (async () => {
            let url = "https://cors-anywhere.herokuapp.com/" + this.props.url
            let feed = await parser.parseURL(url);
            this.setState({items: feed.items});
        })();
    }
    matchesKeyword = (newsItem) =>
        (this.props.filter === "") || newsItem.title.includes(this.props.filter);
    render = () => {
        return (
            <div className="feed">
                <h1>{this.props.title}</h1>
                <ul>
                  {this.state.items.filter(this.matchesKeyword).
                          map(item => <li><a href={item.link}>{item.title}</a></li>)}
                </ul>
            </div>
        );
    }
}
export default App;
```

**Figure 2** Example of a React web application.

concise due to a powerful combination of declarative and object-oriented programming. While the behavior of React applications can be understood in terms of a small number of key operations, thus far these operations have only been defined informally. To our knowledge, our paper presents the first approach that places the essence of React on a formal foundation.

## 3 Semantics

We now present a small-step operational semantics, named $\lambda_{\text{react}}$, that captures the essence of React. We formulate the semantics as an extension of the $\lambda_{\text{js}}$ calculus [11]. Using $\lambda_{\text{js}}$ as a foundation allows us to focus on the core issues without being distracted by complex JavaScript features such as prototype-based inheritance, dynamic property access, implicit coercions, and so on, which are handled by $\lambda_{\text{js}}$.

The calculus aims to capture three central aspects of React:

(i) the mounting and unmounting of components,
(ii) the reconciliation of component descriptors and mounted components, and
(iii) the semantics of state changes.

### 3.1 Design Decisions

We briefly outline the major design choices we made in the formulation of the semantics.

- React is a huge framework and our aim to distill it down to its essence. We want to describe the primary concepts of React in as little formalism as possible. It is *not* our goal to provide a complete formal specification of the entire React framework.
- React relies on classes, which are supported in EcmaScript 6, but $\lambda_{\text{js}}$ is based on an earlier version of EcmaScript. We believe that $\lambda_{\text{js}}$ could be extended with classes, but that is beyond the scope of the current paper. In the $\lambda_{\text{react}}$ semantics, we side-step this issue by direct modeling of the React constructs.
- We extend the syntax of $\lambda_{\text{js}}$ with explicit terms for mounting, unmounting, and reconciliation of components. In React, the programmer cannot use these terms directly; they are part of the internals.
- We model the registration of event listeners since they are the main driver of execution once a React application has started. We simulate the execution of these event listeners in a non-deterministic fashion with a special '$\bullet$' term that represents the event loop.
- React places a strong emphasis on performance. For the most part, we ignore such considerations, however our specification of object equivalence and merging does reflect these underlying concerns.
- We omit lifecycle hooks from the $\lambda_{\text{react}}$ semantics. Although they play an important role in any realistic React application they are not particularly interesting from a semantic point of view, and adding them would be straightforward, if tedious. Nevertheless, in Section 5 we give some ideas of how to incorporate lifecycle hooks into the semantics.

### 3.2 Components, Component Descriptors, and Mounted Components

A React component is a class that extends the `React.Component` class. Each React component has a set of *properties* and an internal *state*. The properties are a form of input parameters used to configure the component, whereas the state holds time-varying values.

Every component has a `render` method that returns fragments that are either React *component descriptors* or HTML elements. This tree fragment represents the "view" of the component and is used by React to “draw” the component in the DOM. For example, a component could return the HTML element `<h1>Hello</h1>`, which React would simply display. On the other hand, it could also return a component descriptor `<RssFeed title="..." url="..." />` whose view would depend on the `render` method inside `RssFeed`.

### Component Descriptors and Mounted Components

In React terminology, the process of creating a React component is called *mounting*. A mounted component is an object that is currently part of the virtual (and real DOM). When a component is taken out of the virtual DOM (and real DOM) it is *unmounted* and eventually garbage-collected. A *component descriptor* is tag-like structure that carries a name of a class and optionally several properties. A component descriptor can be turned into a mounted component. To illustrate these concepts, consider the following `render` method:

```
class App extends React.Component {
	render() {
		if (this.state.progress < 100) {
			return <ProgressBar value={this.state.progress} />
		} else {
			return <Game />
		}
	}
	...
}
```

Here the `render` method consists of an if-then-else statement. If the current progress, kept in the internal state of the `App` component, is less than `100` then the method returns the component descriptor `<ProgressBar value=... />` passing the current progress as a property. Otherwise, the method returns the `<Game />` component descriptor.

When the `App` component is mounted, its initial progress is zero. Hence the render method returns the `<ProgressBar />` descriptor. React then mounts and displays this component. Over time, the progress might change, as assets for the game are downloaded. When this happens, React will re-invoke the `render` method. Let us say that the progress changes from 0% to 20%. Before the change, React knows that the last component descriptor it mounted underneath `App` was:

```
<ProgressBar value=0 />
```

when the progress is changed, `render` returns the component descriptor:

```
<ProgressBar value=20 />
```

At this point, React observes that the two component descriptors are of the same type (`ProgressBar` and `ProgressBar`), but that one of the properties has changed. Since the components are the same, React simply updates the property `value` in the mounted component `ProgressBar` and calls its `render` method. This is called reconciliation.

Now, let us consider what happens when the progress reaches 100%. React knows that the last component descriptor it mounted underneath `App` was:

```
<ProgressBar value=20 />
```

when the progress is changed to 100%, `render` returns the component descriptor:

```
<Game />
```

At this point, React observes that the two component descriptors are *not* of the same type (ProgressBar and Game), hence it unmounts ProgressBar (destroying it) and then it mounts Game in its place and calls its render method.

In summary, a component descriptor is a value that is a static description of a component that can be turned into a mounted component React. The goal of React is to ensure that whatever component descriptors are returned by render, they are kept consistent with the currently mounted components, and that render is invoked whenever a change happens that might change its output.

## 3.3 Component State and Properties

Each React component has a set of properties and an internal state. The properties are accessed through the this.props field inside the component, whereas the state is accessed through this.state. Importantly, *neither field should be changed directly!* The properties of a component are always derived from the properties described in a component descriptor, e.g., <ProgressBar value=20 />. To change the state of a component, the programmer must explicitly call setState on the component. These two patterns ensure that React always knows when changes occur to the properties or state of a mounted component. If properties or state were to be changed through other means, the component might become out of sync with its visual representation in the virtual DOM (and real DOM).

In React, the data flow of properties and state can be quite complex. In general, the state of a component can be passed as a property to another component. However, it is also possible, to use a property as part of a component's own internal state (e.g., by passing it to setState), or to derive state from a property. For example, in the motivating example, the filter is a part of the *state* of the App component, but it is passed as a *property* to the RssFeed component.

## 3.4 Render and Child Components

The render method is at the heart of each React component. It determines the subcomponents of a component by returning component descriptors. A small, but important detail is that it only determines *one level* of components. For an example, consider the following program:

```
class Component extends React.Component {
	render() {
		return (
			<Subcomponent>
				<Button>Click Me</Button>
			</Subcomponent>
		);
	}
}
class Subcomponent extends React.Component {
	render() {
		return (<h1>Hello World!</h1>);
	}
}
```

Here, the render method of the Component class returns a <Subcomponent>...</Subcomponent> descriptor with a Button component descriptor inside it. Our intuition tells us that Subcomponent should have a button somewhere inside it, but this is not necessarily so. In fact, there is no guarantee that the Button component descriptor is ever mounted. To understand why, consider the render method of the Subcomponent. This method unconditionally returns an h1 tag. Hence if we were to render `Component` all we would see would be a `h1` tag. In React, nested component descriptors are simply treated as a special property called `children`. A component must explicitly refer to this property to use any component descriptors that may be nested within it. For example, we could change the subcomponent to:

```
class Subcomponent extends React.Component {
	render() {
		return (
			<div>
				<h1>Goodbye World!</h1>
				{this.props.children}
			</div>
		);
	}
}
```

In this case, the subcomponent would also mount the `Button` component descriptor passed by its parent component. Hence, in the semantics we will write component descriptors as `<C props.../>` ignoring any nested component descriptors, as these are simply passed as a special property called `children`.

## 3.5 Syntax of $\lambda_{\text{react}}$

We are now ready to present the syntax of $\lambda_{\text{react}}$. We assume a base language with support for objects and references such as $\lambda_{\text{js}}$ [11], shown in Figure 3.

### Values

We extend the values of $\lambda_{\text{js}}$ with two new central concepts: *component descriptors* and *mounted components*. Figure 4 and Figure 5 show the extended grammar of values in $\lambda_{\text{react}}$. A component descriptor is written as `<`$C$ *props*`/>` where $C$ is an identifier and *props* is an object literal, i.e., a set of key-value pairs. In React, $C$ is a class, but for our purposes it is sufficient that $C$ is an identifier. A mounted component is written as `<`$C$@$a$ *props*`/>` and is similar to a component descriptor, except that it is associated with an object in the heap stored at address $a$. A component descriptor is just that; a "dead" description, whereas a mounted component is a "live" object. We will write $\pi$ to refer to component descriptors and $\Pi$ to refer to mounted components. The mnemonic is that mounting a component descriptor changes it from $\pi$ to $\Pi$.

### Expressions

We extend the syntax of $\lambda_{\text{js}}$ with React constructs for mounting, unmounting, and reconciliation of components. Figure 6 shows the grammar of the new constructs. We briefly explain each new expression; their semantics are explained in-depth in the following subsection. $\textsc{Mount}(e)$ is used to mount a component descriptor. $\textsc{Unmount}(e)$ is used to unmount a mounted component. $\textsc{MountSeq}(e)$ and $\textsc{UnmountSeq}(e)$ are variants of these that operate on sequences of component descriptors and mounted components, respectively. $\textsc{Mounted}(e)$ and $\textsc{Unmounted}(e)$ are used to perform cleanup after a mount or unmount operation has completed. $\textsc{Reconcile}(e, e)$ is used to reconcile a component descriptor with a mounted component. Reconciliation, as will be explained, is the process of updating a mounted component with new data; either through an incremental re-render of the affected subcomponents or through unmounting/mounting. $\textsc{ReconcileSeq}(e, e)$ is similar, but reconciles a sequence of component descriptors with a sequence of mounted components.

$$
\begin{array}{rcll}
c \in \mathit{Cst} & = & \mathit{bool} \mid \mathit{num} \mid \mathit{str} \mid \texttt{null} \mid \texttt{undef} & \text{[constant]} \\
\\
v \in \mathit{Val} & = & c & \text{[literal]} \\
& \mid & a & \text{[address]} \\
& \mid & \{\mathit{str} : v \cdots\} & \text{[object]} \\
& \mid & \lambda\,(x\cdots)\;\; e & \text{[function]} \\
\\
e \in \mathit{Exp} & = & v & \text{[value]} \\
& \mid & x & \text{[variable]} \\
& \mid & e \,;\, e & \text{[sequence]} \\
& \mid & e = e & \text{[assignment]} \\
& \mid & \texttt{let}\;\; (x = e)\;\; e & \text{[binding]} \\
& \mid & e\,(e\cdots) & \text{[call]} \\
& \mid & e.f & \text{[field load]} \\
& \mid & e.f = e & \text{[field store]} \\
& \mid & \texttt{ref}\;\; e & \text{[address of]} \\
& \mid & \texttt{deref}\;\; e & \text{[value at]} \\
\\
x \in \mathit{Var} & = & \text{is a finite set of variable names.} \\
f \in \mathit{Fld} & = & \text{is a finite set of field names.} \\
a \in \mathit{Addr} & = & \text{is an infinite set of memory addresses.} \\
\lambda \in \mathit{Lam} & = & \text{is the set of all lambda expressions.}
\end{array}
$$

**Figure 3** Syntax of $\lambda_{\text{js}}$.

$$
\begin{array}{rcl}
\pi \in \mathit{Component\ Descriptor} & = & \texttt{<}C\ \mathit{props}\,\texttt{/>} \\
\Pi \in \mathit{Mounted\ Component} & = & \texttt{<}C@a\ \mathit{props}\texttt{/>} \\
\mathit{props} & = & \{k_1 = v_1, \cdots, k_n = v_n\} \\
C & = & \text{is a set of identifiers.}
\end{array}
$$

**Figure 4** Component Descriptors and Mounted Components.

$$
v \in \mathit{Val} \quad = \quad \cdots \mid \pi \mid \Pi
$$

**Figure 5** Values of $\lambda_{\text{react}}$.

$$
\begin{array}{rcl}
e \in \mathit{Exp} & = & \cdots \\
& | & \mathsf{SetState}(e, e) \mid \textsc{Render}(e) \mid \textsc{ReRender}(e) \\
& | & \textsc{Mount}(e) \mid \textsc{MountSeq}(e) \mid \textsc{Mounted}(e, e) \\
& | & \textsc{Unmount}(e) \mid \textsc{UnmountSeq}(e) \mid \textsc{Unmounted}(e) \\
& | & \textsc{Reconcile}(e, e) \mid \textsc{ReconcileSeq}(e, e) \\
& | & \bullet
\end{array}
$$

**Figure 6** Syntax of $\lambda_{\text{react}}$.

$$
\begin{array}{rcl}
E & = & \square \\
& | & \mathsf{SetState}(E, e) \mid \mathsf{SetState}(v, E) \mid \textsc{Render}(E) \mid \textsc{ReRender}(E) \\
& | & \textsc{Mount}(E) \mid \textsc{MountSeq}(E) \mid \textsc{Mounted}(E, e) \mid \textsc{Mounted}(v, E) \\
& | & \textsc{Unmount}(E) \mid \textsc{UnmountSeq}(E) \mid \textsc{Unmounted}(E, e) \mid \textsc{Unmounted}(v, E) \\
& | & \textsc{Reconcile}(E, e) \mid \textsc{Reconcile}(v, E) \mid \textsc{ReconcileSeq}(E, e) \mid \textsc{ReconcileSeq}(v, E)
\end{array}
$$

**Figure 7** Evaluation Contexts for $\lambda_{\text{react}}$.

Finally, the $\bullet$ expression represents the event-loop, which marks when an event listener can be executed.

### Evaluation Context

We extend $\lambda_{\text{react}}$ with the evaluation contexts shown in Figure 7.

### Notation

We will write $\overline{a}$ for a sequence of addresses, $\overline{\pi}$ for a sequence of component descriptors, and $\overline{\Pi}$ for a sequence of mounted components. We will write the empty sequence as *Nil*. We use pattern matching $\pi :: \overline{\pi}$ to deconstruct a sequence into its head ($\pi$) and its tail ($\overline{\pi}$). Given a partial map $f : A \hookrightarrow B$, we write $f - a$ for the same map, but with the binding for $a$ removed. We write the empty map as $\varnothing$.

## 3.6 Runtime of $\lambda_{\text{react}}$

The runtime of $\lambda_{\text{react}}$ is conceptually similar to $\lambda_{\text{js}}$, but extended with several additional aspects to keep track of React components. Figure 8 shows the runtime of $\lambda_{\text{react}}$. A configuration $\chi \in \mathit{Configuration}$ is a 5-tuple $\langle \sigma, \delta, \zeta, \ell, e \rangle$ consisting of the heap $\sigma$, the component state map $\delta$, the component shape map $\zeta$, the listener map $\ell$, and an expression $e$. A heap $\sigma$ is a partial map from addresses to values. A component state map $\delta$ is a partial map from (component) addresses to objects. The component state map does *not* hold the current state of a component, but rather its *next* state which will become the current state through the process of reconciliation. (The current state of a component is always available through the `state` field on the component object.) A component shape map $\zeta$ is a partial map from (component) addresses to pairs of a mounted component and a sequence of addresses. The component shape map records the current "shape" of a mounted component along with its currently mounted subcomponents. Intuitively, the component shape map can be thought

$$\begin{aligned}
\sigma \in \mathit{Heap} &= \mathit{Addr} \hookrightarrow \mathit{Val}\\
\delta \in \mathit{ComponentState} &= \mathit{Addr} \hookrightarrow \mathit{Obj}\\
\zeta \in \mathit{ComponentShape} &= \mathit{Addr} \hookrightarrow \mathit{Mounted\ Component} \times (\mathit{Address})^\star\\
\ell \in \mathit{Listeners} &= \mathit{Addr} \hookrightarrow \mathcal{P}(\mathit{Lam})\\
\chi \in \mathit{Configuration} &= \mathit{Heap} \times \mathit{ComponentState} \times \mathit{ComponentShape} \times \mathit{Listeners} \times \mathit{Exp}
\end{aligned}$$

**Figure 8** Runtime of $\lambda_{\text{react}}$.

$$\frac{\begin{array}{c}\mathsf{keys}(o_1) = \mathsf{keys}(o_2)\\ \forall k \in \mathsf{keys}(o_1).\ o_1(k) \text{ is primitive} \Rightarrow o_1(k) = \mathord{=} o_2(k)\\ \forall k \in \mathsf{keys}(o_1).\ o_1(k) \text{ is reference} \Rightarrow o_1(k) === o_2(k)\end{array}}{o_1 \equiv o_2}\quad [\equiv\text{-}\textsc{Props}]$$

**Figure 9** React Object Equivalence.

of as the "Virtual DOM"; what the browser is currently displaying. A listener map $\ell$ is a partial map from (component) addresses to a set of lambda expressions. The map holds the currently registered event listeners associated with a mounted component. Finally, every configuration has an expression $e$.

## 3.7 Initial State

A $\lambda_{\text{react}}$ program consists of a single root component descriptor $\pi$ (e.g., <*App props*/>). We define a function, inject, to insert the component descriptor into an empty, initial configuration:

$$\mathsf{inject}(\pi) = \langle \varnothing, \varnothing, \varnothing, \varnothing, \textsc{Mount}(\pi); \bullet \rangle$$

The initial configuration starts with an empty heap ($\sigma = \varnothing$), an empty component state map ($\delta = \varnothing$), an empty component shape map ($\zeta = \varnothing$), and an empty map of listeners ($\ell = \varnothing$). The initial expression is $\textsc{Mount}(\pi); \bullet$ which will trigger a mount of the root component descriptor, recursively mounting its subcomponents, and registering event listeners on all mounted components. Once the root component is mounted, the expression enters the event loop, and begins to execute event listeners non-deterministically.

## 3.8 Semantics of Object Equality

A key React operation is to determine when two objects are equal. React uses this to determine when property- and state objects are unchanged during reconciliation, as discussed later. Figure 9 defines two objects to be equal if

(i) they share the same keys,
(ii) the values of primitive types are compared by equality, and
(iii) the references are compared using reference equality.

For example,

$$\{a : 21, b : 42\} \equiv \{a : 21, b : 42\}$$

$$\frac{\begin{array}{c}\forall k_i \in \mathsf{keys}(o_1).\ \ o_1(k_i) = v_i \\ \forall k'_i \in (\mathsf{keys}(o_2) - \mathsf{keys}(o_1)).\ \ o_2(k'_i) = v'_i \\ o_3 = \{k_1 : v_1, \cdots, k_n : v_n, k'_1 : v'_1, \cdots, k'_n : v'_n\}\end{array}}{o_1 \otimes o_2 = o_3} \quad [\otimes\text{-}\textsc{State}]$$

**Figure 10** The State Merge Operator ⊗.

and

$$\{a : 21, b : \ell\} \equiv \{a : 21, b : \ell\}$$

where $\ell$ is some address in the heap.

This shallow notion of equality can be checked efficiently, since we never have to recursively descend into the object structure.

## 3.9 Semantics of State Merges

Another key React operation is to merge two objects. Like equivalence, merging is also a shallow operation. Specifically, two objects are merged in a left-biased manner where the returned object is obtained by taking all keys and values from the left object and adding those keys and values from the right object that did not appear in the left object. Figure 10 captures this notion.

For example,

$$\{a : 21, b : 42\} \otimes \{b : 84\} = \{a : 21, b : 42\}$$

and

$$\{a : 21, b : 42\} \otimes \{c : 84\} = \{a : 21, b : 42, c : 84\}$$

Note that the procedure is *not* recursive:

$$\{o : \{a : 21\}\} \otimes \{o : \{b : 42\}\} = \{o : \{a : 21\}\}$$

which is common source of bugs.

Both object equality and merging play vital roles in the reconciliation of components.

## 3.10 Semantics of Mounting and Unmounting

We now discuss the process of mounting and unmounting components. A component is mounted when a $\lambda_{\text{react}}$ application starts and it may cause subcomponents to be mounted, and so on recursively. Components are also mounted and unmounted as part of the reconciliation process, which will be described later. Figure 11 shows the semantics of mounting and unmounting components. We now discuss each evaluation rule in greater detail:

### [E-Mount]

The rule states that to mount a component descriptor $\pi$ = <$C$ *props*/> the following steps are taken: A fresh address $a \notin \text{dom}(\sigma)$ is chosen. An object is stored at that address in the heap with a copy of the *props* object. The component state map $\delta$ is updated with a binding

$$\frac{\pi = \texttt{<}C\ props\ \texttt{/>} \quad a \notin \mathrm{dom}(\sigma) \\ \sigma' = \sigma[a \mapsto \{props : props\}] \quad \delta' = \delta[a \mapsto \{\}] \quad \ell' = \ell[a \mapsto \mathsf{listenersOf}(props)]}{\langle \sigma, \delta, \zeta, \ell, \textsc{Mount}(\pi)\rangle \to \langle \sigma', \delta', \zeta, \ell', \textsc{Mounted}(\texttt{<}C@a\, props\texttt{/>}, \textsc{MountSeq}(\textsc{Render}(\pi)))\rangle} \ (\text{E-}\textsc{Mount})$$

$$\frac{\zeta' = \zeta[a \mapsto (\texttt{<}C@a\, props\texttt{/>}, \overline{a})]}{\langle \sigma, \delta, \zeta, \ell, \textsc{Mounted}(\texttt{<}C@a\, props\texttt{/>}, \overline{a})\rangle \to \langle \sigma, \delta, \zeta', \ell, a\rangle} \ (\text{E-}\textsc{Mounted})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{MountSeq}(Nil)\rangle \to \langle \sigma, \delta, \zeta, \ell, Nil\rangle} \ (\text{E-}\textsc{Mount-Seq-1})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{MountSeq}(\pi :: \overline{\pi})\rangle \to \langle \sigma, \delta, \zeta, \ell, \textsc{Mount}(\pi) :: \textsc{MountSeq}(\overline{\pi})\rangle} \ (\text{E-}\textsc{Mount-Seq-2})$$

$$\frac{\zeta(a) = (\texttt{<}C@a\, props\texttt{/>}, \overline{a})}{\langle \sigma, \delta, \zeta, \ell, \textsc{Unmount}(a)\rangle \to \langle \sigma, \delta, \zeta, \ell, \textsc{UnmountSeq}(\overline{a}); \textsc{Unmounted}(a)\rangle} \ (\text{E-}\textsc{Unmount})$$

$$\frac{\ell' = \ell - a}{\langle \sigma, \delta, \zeta, \ell, \textsc{Unmounted}(a)\rangle \to \langle \sigma, \delta, \zeta, \ell', Nil\rangle} \ (\text{E-}\textsc{Unmounted})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{UnmountSeq}(Nil)\rangle \to \langle \sigma, \delta, \zeta, \ell, Nil)\rangle} \ (\text{E-}\textsc{Unmount-Seq-1})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{UnmountSeq}(a :: \overline{a})\rangle \to \langle \sigma, \delta, \zeta, \ell, \textsc{Unmount}(a)\,;\, \textsc{UnmountSeq}(\overline{a})\rangle} \ (\text{E-}\textsc{Unmount-Seq-2})$$

**Figure 11** Semantics of mounting and unmounting components.

for $a$, binding it to the empty object (since the next pending state is currently empty). The event listeners are extracted from the *props* object and registered in the event listener map $\ell$. These steps are sufficient to mount the component, but we must also recursively mount its subcomponents as determined by its `render` method.

We achieve this by having the mount expression reduce to:

$$\textsc{Mounted}(\texttt{<}C@a\, props\texttt{/>}, \textsc{MountSeq}(\textsc{Render}(\pi)))$$

which can be understood as follows: The inner $\textsc{Render}(\pi)$ will reduce to a sequence of subcomponent descriptors. We will then mount each of these in turn. This will reduce to a sequence of mounted component addresses $\overline{a}$. Finally, the $\textsc{Mounted}(\texttt{<}C@a\, props\texttt{/>}, \overline{a})$ expression will register that the mounted components addresses $\overline{a}$ are subcomponents of the current component $a$.

## [E-Mounted]

The rule states that the expression $\textsc{Mounted}(\texttt{<}C@a\, props\texttt{/>}, \overline{a})$ reduces to the address $a$ of the mounted component $\texttt{<}C@a\, props\texttt{/>}$ with the component shape map $\zeta$ updated to reflect the current shape of the component $a$ and that $\overline{a}$ are the current subcomponents of $a$. Intuitively, once the `Mounted` expression is evaluated, the component $a$ and its subcomponents have been fully mounted, and we have recorded their shape so that, in the future when we re-render the component, we are able to compare the current shape to the component descriptors returned by `render`.

### [E-Mount-Seq-1]

The rule states that mounting the empty sequence of component descriptors results in the empty sequence of mounted component addresses.

### [E-Mount-Seq-2]

The rule states that to mount a sequence of component descriptors $\pi :: \overline{\pi}$ we mount the first component $\pi$ and then we mount the remaining component descriptors $\overline{\pi}$. Mounting a component descriptor $\pi$ returns a mounted component address $a$ and since our goal is to produce a sequence of mounted component addresses, we prepend the result of mounting $\pi$ with the result of mounting the remaining component descriptors $\overline{\pi}$. Thus, `MountSeq` always reduces to a sequence of mounted component addresses.

For example, if we mount the two component descriptors:

$$\langle \sigma, \delta, \zeta, \ell, \textsc{MountSeq}(\texttt{<}\mathit{TextField\ props1}\texttt{/>} :: \texttt{<}\mathit{Button\ props2}\texttt{/>} :: \mathit{Nil}) \rangle$$

we obtain a new configuration with the two mounted components:

$$\langle \sigma', \delta', \zeta', \ell', a_1 :: a_2 :: \mathit{Nil} \rangle$$

where

$$\zeta'(a_1) = \texttt{<}\mathit{TextField@}a_1\ \mathit{props1}\texttt{/>} \quad \text{and} \quad \zeta'(a_2) = \texttt{<}\mathit{Button@}a_2\ \mathit{props2}\texttt{/>}$$

### [E-Unmount]

The rule states that to unmount a mounted component address $a$, we must first unmount its subcomponents, which are known from the component shape map $\zeta$, and afterwards we can consider the component $\Pi$ to be unmounted.

### [E-Unmounted]

The rule states that once the subcomponents of a component $a$ have been unmounted, all listeners are removed from the event listener map $\ell$. We do *not* remove the address $a$ from the heap $\sigma$ since there could still be a reference to the component object somewhere nor do we remove it from the component shape map $\zeta$. A garbage collector can be used to clean these maps, if desired.

### [E-Unmount-Seq-1]

The rule states that unmounting the empty sequence of mounted component addresses results in the empty sequence.

### [E-Unmount-Seq-2]

The rule states that to unmount a sequence of mounted component addresses $a :: \overline{a}$ we must unmount the first component $a$ and then we can unmount the remaining components $\overline{a}$.

It is easy to see how the structure of the [E-Mount], [E-Mounted], [E-Mount-Seq-1], and [E-Mount-Seq-2] mirror the structure of [E-Unmount], [E-Unmounted], [E-Unmount-Seq-1], and [E-Unmount-Seq-2]. Mounting is essentially a recursive traversal of a tree that is gradually being computed by the `render` methods. Unmounting is the reverse process, using the component shape map to recursively remove subcomponents.

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{ReconcileSeq}(\mathit{Nil}, \mathit{Nil})\rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \mathit{Nil}\rangle} \quad (\textsc{RC-Empty})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{ReconcileSeq}(\overline{\pi}, \mathit{Nil})\rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \textsc{MountSeq}(\overline{\pi})\rangle} \quad (\textsc{RC-Extend})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{ReconcileSeq}(\mathit{Nil}, \overline{a})\rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \textsc{UnmountSeq}(\overline{a})\rangle} \quad (\textsc{RC-Truncate})$$

$$\frac{}{\langle \sigma, \delta, \zeta, \ell, \textsc{ReconcileSeq}(\pi :: \overline{\pi}, a :: \overline{a})\rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \textsc{Reconcile}(\pi, a) :: \textsc{ReconcileSeq}(\overline{\pi}, \overline{a})\rangle} \quad (\textsc{RC-Sequence})$$

$$\frac{\pi = \texttt{<}C_1\ \mathit{nextProps}\,\texttt{/>} \quad \zeta(a) = (\texttt{<}C_2@a\,\mathit{prevProps}\texttt{/>}, \_) \quad C_1 \neq C_2}{\langle \sigma, \delta, \zeta, \ell, \textsc{Reconcile}(\pi, a)\rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \textsc{Unmount}(a)\,;\,\textsc{Mount}(\pi)\rangle} \quad (\textsc{RC-Diff-Root})$$

$$\frac{\begin{array}{c}\pi = \texttt{<}C\ \mathit{nextProps}\,\texttt{/>} \quad \zeta(a) = (\texttt{<}C@a\,\mathit{prevProps}\texttt{/>}, \overline{a}) \quad \mathit{nextState} = \delta(a) \\ o = \sigma(a) \quad o' = o[\texttt{props} \mapsto \mathit{nextProps}][\texttt{state} \mapsto \mathit{nextState}] \quad \sigma' = \sigma[a \mapsto o']\end{array}}{\begin{array}{c}\langle \sigma, \delta, \zeta, \ell, \textsc{Reconcile}(\pi, a)\rangle \rightarrow \\ \langle \sigma', \delta, \zeta, \ell, \textsc{Reconciled}(\texttt{<}C@a\,\mathit{nextProps}\texttt{/>}, \textsc{ReconcileSeq}(\textsc{ReRender}(a), \overline{a}))\rangle\end{array}} \quad (\textsc{RC-Same-Root})$$

$$\frac{\zeta' = \zeta[a \mapsto (\texttt{<}C@a\,\mathit{props}\texttt{/>}, \overline{a})]}{\langle \sigma, \delta, \zeta, \ell, \textsc{Reconciled}(\texttt{<}C@a\,\mathit{props}\texttt{/>}, \overline{a})\rangle \rightarrow \langle \sigma, \delta, \zeta', \ell, a\rangle} \quad (\textsc{RC-Reconciled})$$

**Figure 12** Semantics of reconciliation

## 3.11 Semantics of Reconciliation

The purpose of reconciliation is to merge a component descriptor (or sequence of component descriptors) with a mounted component (or sequence of mounted components). At a high level, there are two broad cases to consider: (i) a mounted component is updated with new properties and state, and (ii) a mounted component is replaced by another component. We introduce two expressions: $\textsc{Reconcile}(e, e)$ and $\textsc{ReconcileSeq}(e, e)$ to model reconciliation. The former reconciles a component descriptor with a mounted component address, whereas the latter deals with sequences of component descriptors and mounted component addresses. Figure 12 shows the semantics of reconciliation. We now discuss each rule in greater detail:

### [RC-Empty]

The rule states that reconciliation of the empty sequence of component descriptors with the empty sequence of mounted component addresses simply results in the empty sequence of mounted component addresses.

### [RC-Extend]

The rule states that reconciliation of a sequence of component descriptors $\overline{\pi}$ with the empty sequence of mounted component addresses *Nil* results in each of the $\overline{\pi}$ component descriptors being mounted. For example, if we were to reconcile the component descriptors:

```
<RssFeed title="..."/> :: <RssFeed title="..."/>
```

with the empty sequence of mounted component addresses then we would simply mount the two `<RssFeed>`s. This is a common occurrence when the `render` method of a component returns additional component descriptors. The rule is called extend because it *extends* a sequence of subcomponents with additional components.

### [RC-Truncate]

The rule states that reconciliation of the empty sequence of component descriptors with a sequence of $\overline{a}$ of mounted component addresses results in each of the $\overline{a}$ components being unmounted. For example, if we were to reconcile the empty sequence of component descriptors with the sequence of mounted component addresses:

$$a_1 :: a_2 :: \mathit{Nil}$$

where

$\zeta(a_1)$ = `<a1@RssFeed title="..."/>`  $\zeta(a_2)$ = `<a2@RssFeed title="..."/>`

then the mounted components $a_1$ and $a_2$ would be unmounted. The rule is called truncate because its *truncates* a sequence of subcomponents. Intuitively, this is the dual of the [RC-Extend] rule.

### [RC-Sequence]

The rule states that reconciliation of a sequence of component descriptors $\pi :: \overline{\pi}$ with a sequence of mounted component addresses $a :: \overline{a}$ requires pairwise reconciliation, i.e., we have to reconcile $\pi$ with $a$ and then reconcile the rest of the two sequences $\overline{\pi}$ and $\overline{a}$. For example, if we were to reconcile the sequence of component descriptors:

```
<RssFeed title="The Guardian"/> :: <RssFeed title="Reuters"/>
```

with the sequence of mounted component addresses:

$$a_1 :: a_2 :: \mathit{Nil}$$

where

$\zeta(a_1)$ = `<a1@RssFeed title="BBC World"/>`  $\zeta(a_2)$ = `<a2@RssFeed title="Reuters"/>`

the first component descriptor would be reconciled with the first mounted component $a_1$ and similarly the second component descriptor would be reconciled with the second mounted component $a_2$. In this case, the first component $a_1$ would be re-rendered and recursively reconciled since one of its properties changed.

We now turn to the more interesting question of how to reconcile a single component descriptor with a single mounted component. As stated previously, there are two cases to consider: (i) a mounted component is being updated with new properties and state, or (ii) a mounted component is being replaced by another component. We begin with the latter.

### [RC-Diff-Root]

The rule states that reconciliation of a component descriptor $\pi$ = `<`$C_1$ *nextProps*`/>` with a mounted component address $a$ where $\zeta(a)$ = `<`$a$@$C_2$ *prevProps*`/>` and where the descriptor and the mounted components have different kinds, i.e., $C_1 \neq C_2$, is a two-step process. First, the currently mounted component $C_2$ is unmounted, which as we have seen, will recursively unmount its subcomponents. Second, the $C_1$ component descriptor is mounted.

$$\frac{\pi = \texttt{<}C\ props\texttt{/>} \quad \langle \sigma, \delta, \zeta, \ell, \mathsf{props.render()} \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \overline{\pi} \rangle}{\langle \sigma, \delta, \zeta, \ell, \textsc{Render}(\pi) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \overline{\pi} \rangle} \quad (\text{E-}\textsc{Render})$$

$$\frac{\zeta(a) = (\texttt{<}C@a\ props\texttt{/>}, \_) \quad \langle \sigma, \delta, \zeta, \ell, \mathsf{props.render()} \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \overline{\pi} \rangle}{\langle \sigma, \delta, \zeta, \ell, \textsc{ReRender}(a) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \overline{\pi} \rangle} \quad (\text{E-}\textsc{ReRender})$$

**Figure 13** Semantics of Rendering and Re-Rendering

For example, if we were to reconcile the component descriptor:

`<Alert color="secondary">Submitted!</Alert>`

with the mounted component address $a$ where:

$\zeta(a)$ = `<a@Button color="primary">Submit</Button>`

The mounted `Button` component would be unmounted, and the `Alert` component descriptor would be mounted in its place. One component being replaced by another component is a common occurence in the implementation of form dialogs and page navigation.

### [RC-Same-Root]

This rule states that reconciliation of a component descriptor $\pi$ = `<`$C$ *nextProps*`/>` with a mounted component address $a$ where $\zeta(a)$ = `<`$C$@$a$ *prevProps*`/>` and the descriptor and mounted component have the same kind requires multiple steps: The `props` field of the component object is updated to *nextProps*. Similarly, the `state` field is updated to the value of the next state as specified by the component state map $\delta$. Finally, the expression reduces to the expression:

$\textsc{Reconciled}(\Pi, \textsc{ReconcileSeq}(\textsc{ReRender}(a), \overline{a}))$

since we must re-render the component and reconcile the returned component descriptors (which could have changed) with the currently mounted subcomponents $\overline{a}$. Once this is done, we must update the component shape map of $a$ to store its newly mounted / reconciled subcomponents, hence we wrap the result in Reconciled which is similar to Mounted and Unmounted.

A variant of the [RC-Same-Root] rule, closer to React semantics, would use React's object equivalence to determine whether the *prevProps* and *nextProps* are equivalent, and whether *nextState* and the current state are equivalent. If all were found to be equivalent then there is no need to do anything, and we could simply skip the updates and re-rendering. This is a performance consideration, hence we have omitted it from the rules.

### [Rc-Reconciled]

The rule states that once reconciliation is complete for the mounted component address $a$ with (possibly new) subcomponents $\overline{a}$ then we update the component shape map $\zeta$ to store the subcomponents and then return the component address $a$ itself.

## 3.12 Semantics of Rendering

The semantics of rendering, shown in Figure 13, are straightforward. As mentioned earlier, there are two types of rendering: rendering a component descriptor for the first time and re-rendering an already mounted component.

$$\frac{\mathit{nextState} = \delta(a) \qquad \delta' = \delta[a \mapsto \mathit{newState} \otimes \mathit{nextState}] \qquad \zeta(a) = (\texttt{<}C@a\ \mathit{props}\texttt{/>}, \overline{a})}{\langle \sigma, \delta, \zeta, \ell, \mathsf{SetState}(a, \mathit{newState}) \rangle \rightarrow \langle \sigma, \delta', \zeta, \ell, \textsc{Reconciled}(a, \textsc{ReconcileSeq}(\textsc{Render}(a), \overline{a})) \rangle} \quad [\text{E-}\textsc{Set-State}]$$

**Figure 14** Semantics of Set-State

$$\frac{a \in \mathit{Addr} \qquad \lambda \in \ell(a)}{\langle \sigma, \delta, \zeta, \ell, \bullet \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \lambda(a); \bullet \rangle} \quad [\text{E-}\textsc{Loop}]$$

**Figure 15** Semantics of Events

### [E-Render] and [E-ReRender]

The [E-Render] rule states that to render a component descriptor we invoke the render method of the props object. We assume that such a method always exists on props. The [E-ReRender] is similar but for a mounted component where we use the address $a$ to retrieve the component from the component shape map $\zeta$ and then we call its render method of its props object.

An interesting observation about [E-Render] and [E-ReRender] is that, once a component has been mounted, overwriting its props.render will not have any effect, since the props object itself is stored in the component shape map $\zeta$. While this may seem overly complicated (and to some extent it is), it (i) is consistent with actual React semantics, and (ii) it allows us to prove key properties of $\lambda_{\text{react}}$. Specifically, in these proofs, we need to know that the render method is not suddenly changed underneath us. Note that calling `render` by itself has no effect in our semantics; it is only when it is called from within e.g., MountSeq and ReconcileSeq that mounting or reconciliation is triggered.

As mentioned earlier, the `render` method must return a sequence of component descriptors. Each component descriptor carries its own properties with a render method inside it. For this process to terminate, at some point a component will not have *any* subcomponents and simply return the empty sequence of component descriptors.

## 3.13 Semantics of State Changes

The semantics of state changes, shown in Figure 14, are also straightforward:

### [E-Set-State]

The rule states that if the mounted component $a$ is passed an object *newState* with some new state then we must retrieve the *nextState* from the $\delta$ map and merge the current next state with the new state. Finally, we must trigger a reconciliation wrapped in a Reconciled since changing the state of a component could change what is returned by its `render`.

## 3.14 Semantics of Events

As mentioned earlier, the properties of a component descriptor may contain fields that correspond to various event listeners. For example, if there is a field `onClick` then it should be registered as an event listener when the component descriptor is mounted (and unregistered when the component is unmounted). In a real React application, such event listeners are executed in response to user events. In the $\lambda_{\text{react}}$ semantics, we add a rule, shown in Figure 15, which states that once we are in the event loop $\bullet$ then we may select any (component) address $a$ and pick any of its event listeners $\lambda \in \ell(a)$, execute it, and then return to the event loop.

In the semantics, as well as in real React applications, execution of an event listener may invoke `setState` which in turn may cause a component to re-render and trigger the process of reconciliation. Thus, at a high-level, the execution of a React application can be understood as an initial mount (as defined by the inject function) followed by a sequence of reconciliations caused by calls to `setState` from event listeners.

## 4 Properties of $\lambda_{\text{react}}$

We want to show that mounting, unmounting, and reconciliation terminate. However, in general, these processes may not terminate if the user-defined `render` function is badly behaved. Trivially, if `render` does not terminate then mounting a component descriptor will not terminate. But even if we assume that `render` terminates, it could return a list of "recursive" component descriptors. That is, the `render` function of a component descriptor <$C$ props/> could return a list that includes $C$ itself. This would cause an execution where an infinite tree of component descriptors is mounted (which obviously never terminates).

To overcome these issues, we define the notion of a *well-behaved* component. Simply put, the `render` function of a well-behaved component must always return a list of component descriptors where each comment descriptor is strictly "smaller" than the component itself. Under the assumption that components are well-behaved, we can prove properties about mounting, unmounting, and reconciliation.

We now formalize the notions of rank and well-behavedness:

### 4.1 Definitions

▶ **Definition 1** (Rank). *A ranking function rank : Identifier → Nat is a map from identifiers (component names) to natural numbers.*

▶ **Definition 2** ($k$-Well-Behaved Expressions). *An expression $e$ is $k$* well-behaved *if it evaluates to a list of component descriptors $\overline{\pi}$ such that for each component descriptor $\pi_i$ = <$C$ props/> in the list it is the case that rank$(C) < k$. If $k = 0$ then $e$ must evaluate to the empty list.*

▶ **Definition 3** ($k$-Well-Behaved Component Descriptors). *A component descriptor $\pi$ = <$C$ props/> is $k$ well-behaved if rank$(C) = k$ and the render function props.render is $k$ well-behaved.*

That is to say, the render function can only return component descriptors with a strictly lower rank than the rank of the component.

▶ **Definition 4** ($k$-Well-Behaved Mounted Components). *A mounted component $\Pi$ = <$C$@$a$ props/> is $k$ well-behaved if rank$(C) = k$ and the render function props.render is $k$ well-behaved.*

That is to say, the `render` function can only return component descriptors with a strictly lower rank than the rank of the mounted component.

▶ **Definition 5** (Well-Behaved Component Shape Maps). *A component shape map $\zeta$ is well-behaved if:*

- *For every $a \in$ dom$(\zeta)$ where $\zeta(a) = (\Pi, \overline{a})$, $\Pi$ is $k$ well-behaved for some $k$ and for every address $a_i \in \overline{a}$, $\zeta(a_i) = (\Pi', \_)$, $\Pi'$ is $k'$ well-behaved for some $k'$ where $k' < k$. That is to* *say, every mounted component is well-behaved, its children are well-behaved, and they have strictly lower rank.*

- *For every pair of addresses $a_1$ and $a_2$ with $a_1 \neq a_2$ it is the case that if $\zeta(a_1) = (\_, \overline{a_1})$ and $\zeta(a) = (\_, \overline{a_2})$ then the two lists $\overline{a_1}$ and $\overline{a_2}$ have disjoint elements. That is to say, every mounted component has exactly one parent.*

*As before, if $k = 0$ then the children $\overline{a}$ of a mounted component must be the empty list.*

## 4.2 Theorems

We can now state the main theoretical results of the paper.

▶ **Theorem 6** (Mount Preserves Well-Behavedness). *If $\pi$ is a $k$ well-behaved component descriptor and $\zeta$ is a well-behaved component shape map then:*

$$\langle \sigma, \delta, \zeta, \ell, E[\textsc{Mount}(\pi)]\rangle \to^\star \langle \sigma', \delta', \zeta', \ell', E[a]\rangle$$

*and:*

- *$\zeta'$ is a well-behaved component shape map,*
- *$\zeta'(a)$ is $k$ well-behaved mounted component, and*
- *$a$ is* not *the child of any mounted component, i.e. there does not exist an address $a_2$ such that $\zeta(a_2) = (\_, \overline{a_2})$ where $a \in \overline{a_2}$.*

▶ **Corollary 7** (Inject is Well-Behaved). *If $\pi$ is a $k$ well-behaved component then:*

$$\mathit{inject}(\pi) \to^\star \langle \sigma, \delta, \zeta, \ell, \bullet\rangle$$

*where $\zeta$ is well-behaved.*

▶ **Theorem 8** (Unmount Preserves Well-Behavedness). *If $a$ is an address in $\mathit{dom}(\zeta)$ and $\zeta$ is a well-behaved component shape map then:*

$$\langle \sigma, \delta, \zeta, \ell, E[\textsc{Unmount}(a)]\rangle \to^\star \langle \sigma, \delta, \zeta, \ell', E[\mathit{Nil}]\rangle$$

▶ **Theorem 9** (Reconciliation Preserves Well-Behavedness). *If $\pi$ is a $k$ well-behaved component descriptor, $\zeta$ is a well-behaved component shape map, $a \in \mathit{dom}(\zeta)$, $\zeta(a) = (\Pi, \_)$, $\Pi$ is $k'$ well-behaved then*

$$\langle \sigma, \delta, \zeta, \ell, E[\textsc{Reconcile}(\pi, a)]\rangle \to^\star \langle \sigma', \delta', \zeta', \ell', E[a']\rangle$$

*and $\zeta'$ is well-behaved and $\zeta'(a')$ is $k$ well-behaved.*

▶ **Lemma 10** (ReconcileSeq Preserves Well-Behavedness). *If $\zeta$ is a well-behaved component shape map, $\overline{\pi} = \pi_1, \cdots, \pi_n$, each $\pi_i$ is $k_i$ well-behaved, $\overline{a} = a_1, \cdots, a_m$, and each $a_i \in \mathit{dom}(\zeta)$ then*

$$\langle \sigma, \delta, \zeta, \ell, E[\textsc{ReconcileSeq}(\overline{\pi}, \overline{a})]\rangle \to^\star \langle \sigma', \delta', \zeta', \ell', E[a'_1, \cdots, a'_n]\rangle$$

*and $\zeta'$ is well-behaved and every $\zeta'(a'_i)$ is $k_i$ well-behaved.*

The detailed proofs of these properties are available in a separate technical report [16].

In summary, we have proved that as long as the `render` functions terminate and the component descriptors form a hierarchy that rules out infinite component trees, then the processes of mounting, unmounting, and reconciling components all terminate. The theorems show that these restrictions expressed in terms of React programs are reflected in the runtime state of these programs, and are preserved in the runtime state by all the operations. The theorems also show that these restrictions are sufficient to ensure termination of each of the operations that manipulate components.

| Lifecycle Hook | Use `setState`? | Deprecated? |
|---|---|---|
| `constructor`(*props*) | No | No |
| `componentDidMount`() | Yes | No |
| `componentDidUpdate`(*prevProps*, *prevState*, *snapshot*) | Yes$^\star$ | No |
| `componentWillReceiveProps`(*nextProps*) | Yes | Yes |
| `componentWillMount`() | Yes | Yes |
| `componentWillUnmount`() | No | No |
| `componentWillUpdate`(*nextProps*, *prevProps*) | No | Yes |
| `shouldComponentUpdate`() | - | No |
| `getDerivedStateFromProps`() | n/a | No |
| `getSnapshotBeforeUpdate`(*prevProps*, *prevState*) | - | No |

**Table 1** Summary of React Lifecycle Hooks. ($\star$ only under certain conditions.)

## 5 Lifecycle Hooks

Lifecycle hooks are an important part of React. A lifecycle hook is a callback executed by React in response to changes to a component's properties and state, and when it is mounted or unmounted. Lifecycle hooks are frequently used to acquire (and release) resources, to retrieve data over the internet, and so on. Table 1 shows an overview of the lifecycle hooks available in React. As the figure shows, the design of lifecycle hooks has gone through several iterations, and some lifecycle hooks are now deprecated. Another important aspect of lifecycle hooks is whether they are allowed to call `setState`. This turns out to be quite tricky, because it is easy to accidentally construct infinite loops where a lifecycle hook calls `setState` which in turn triggers a lifecycle hook, and so on. This is source of bugs in React.

The semantics of $\lambda_{\text{react}}$ does not include lifecycles, but we can extend it to accommodate them. For example:

- The `componentWillMount()` method is invoked immediately *before* a component is mounted. In the semantics, this corresponds to the [E-Mount] rule, which we could update to trigger a call to `componentWillMount()`.
- The `componentDidMount()` method is invoked immediately *after* a component has been mounted. In the semantics, this corresponds the [E-Mounted] rule, which we could update to trigger a call to `componentDidMount()` immediately before returning the mounted component.
- The `componentWillUnmount()` method is invoked immediately *before* a component is unmounted. In the semantics, this corresponds to the [E-Unmount] rule, which we could update to trigger a call to `componentWillUnmount()`. There is no corresponding `componentDidUnmount()` because changes should not be made to an unmounted component.
- The `componentDidUpdate()` method is invoked immediately *after* a component has been updated. In the semantics, this corresponds approximately to the [Rc-Same-Root] reconciliation rule, which we could update to trigger a call to `componentDidUpdate()`.

Note that many of the lifecycle hooks receive the previous properties, the previous state, the new properties, and/or the new state. Since the $\lambda_{\text{react}}$ semantics meticulously models properties and state, these objects are readily available.

## 6 Related Work

We are not aware of any prior work on formally defining the semantics of React. In this related work section, we focus on previous research on formally specifying the semantics of JavaScript, and on related frameworks for defining user-interfaces declaratively.

### Semantics of JavaScript

Many proposals have been made for a formal semantics of JavaScript. Herman and Flanagan [12] presented an implementation of an interpreter for EcmaScript 4 written in ML. Being an interpreter, the specification was executable. However, EcmaScript 4 was never adopted as a standard. Maffeis et al. [18] presented the first small-step operational semantics for JavaScript as a basis for formalization of security properties in web applications.

Guha et al. [11] presented $\lambda_{\text{js}}$, a minimal semantics for JavaScript. A key aspect of their work is to formalize a semantics that is as small as possible, while still being expressive enough to allow compilation of all JavaScript constructs into it. In this way, $\lambda_{\text{js}}$ supports all ugly features of JavaScript, such as prototype-based inheritance, dynamic property access, and implicit coercions.

Gardner et al. [10] presented a program logic based on separation logic for reasoning about a large subset of the ECMAScript 3 language. The subset under consideration includes features such as prototype inheritance and the `with` construct, which interacts with JavaScript's scoping rules in intricate ways.

Park et al. [22] presented KJS, a complete formalization of ECMAScript 5.1 implemented in the K Framework [22]. Being specified in the K framework, the semantics is executable and has been tested against all 2,782 tests in the ECMAScript 5.1 conformance test suite. By specifying all of JavaScript, and executing all test cases, the authors were able to find evaluation rules not covered by any existing test, add tests for these rules, and then running them on different browsers, which ultimately revealed several implementation bugs.

Bodin et al. [7], presented JSCert, a formal semantics for the ECMAScript 5 version of JavaScript that is formalized and proven correct using the Coq proof assistant. Their work also includes a reference interpreter, JSRef, that can be used to execute test cases and compare results against standard JavaScript interpreters. As is typical in formalizations, JSCert excludes a number of pragmatic details such as certain native library functions, and relies on an external parser to implement `eval`. Also, the `for-in` construct has not been formalized because the standard defines it very loosely. Bodin's dissertation [8] explored the challenges associated with the formalization in greater detail.

The semantics of asynchronous JavaScript has been tackled by several authors. Madsen et al. [17] proposed an extension of $\lambda_{\text{js}}$ that models events, event listeners, and the event loop. Based on this semantics, the authors developed a static analysis to discover simple bugs in event-driven JavaScript programs. Loring et al. [14] and Madsen et al. both [15] proposed semantics to specify the behavior of JavaScript promises. Later work by Alimadadi et al. [4] presented a tool for finding bugs in promise-based JavaScript code based on [15].

### Other Frameworks

A discussion about the design of React and how it evolved can be found in CACM [1]. Since the introduction of React, many other framework have appeared that emulate its declarative and object-oriented programming style. React Native [13] lets programmers write native mobile applications using JavaScript and React. React Native uses the React model, but with the UI components of the underlying OS, e.g. iOS or Android. Preact [2] is a light-weight "close to the metal" React-style library with a focus on performance. Preact aims to provide the thinnest possible "virtual DOM" on top of the real DOM. Vue.js [3] is another React-like library with a focus on the view-layer and on easy integration with other existing libraries.

We believe our semantics offers a solid foundation for understanding and potentially modeling these React-inspired libraries. We think that the popularity of React and the adoption of the "React model" by many other frameworks is a sign of its importance for the future of web development.

## 7 Conclusions and Future Work

React is a framework that enables programmers to write web applications in a declarative and object-oriented style that facilitates reuse. Each component of a React application has a set of properties representing input parameters, a state consisting of values that may vary over time, and a `render` method that specifies its subcomponents. When state changes occur, React's reconciliation mechanism determines their impact and updates the user-interface incrementally by mounting, unmounting, or reconciling subcomponents selectively. At designated points in this process, the React framework invokes lifecycle methods that enable programmers to perform actions outside the framework such as acquiring and releasing resources. Since these mechanisms exhibit considerable complexity, programmers would benefit from program analyses and tools that can reason precisely about React programs.

It is our long-term goal to develop program understanding and bug finding tools for React applications. To our knowledge, this paper presents the first formal specification of a semantics that captures the essence of React, thus establishing a foundation for such tools. Our small-step operational semantics extends the $\lambda_{\text{js}}$ calculus [11] and models three key concepts of React: (i) mounting and unmounting of components, (ii) reconciliation of component descriptors and mounted components, and (iii) the semantics of state changes. To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a well-behaved component by imposing a ranking function on components, and requiring that the `render` method of a component only returns components of strictly smaller rank. We then prove that well-behavedness is preserved by these operations.

For future work, we plan to conduct a case study to identify and classify common bug patterns in React web applications. Then, with a formal semantics of React in place, we will develop static analysis techniques to detect instances of these bug patterns in React applications. Another avenue for future work is the development of a type system that is sufficiently expressive to capture the lifecycle of React components and ensure that properties and state are not accessed or modified incorrectly.

## References

1 React: Facebook's functional turn on writing JavaScript. *Commun. ACM*, 59(12):56–62, 2016. URL: `https://doi.org/10.1145/2980991`, `doi:10.1145/2980991`.

2 Preact: Fast 3kB alternative to React with the same modern API. `https://preactjs.com/`, 2019. Accessed: 2019-12-19.

3 Vue.js: The progressive JavaScript framework. `https://vuejs.org/`, 2019. Accessed: 2019-12-19.

4 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous JavaScript programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018. URL: `https://doi.org/10.1145/3276532`, `doi:10.1145/3276532`.

5 Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010. URL: `https://doi.org/10.1109/TSE.2010.31`, `doi:10.1109/TSE.2010.31`.

6 SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. SAFEWAPI: Web API misuse detector for web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 507–517, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2635868.2635916`, `doi:10.1145/2635868.2635916`.

7 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014. URL: `https://doi.org/10.1145/2535838.2535876`, `doi:10.1145/2535838.2535876`.

8 Michel Bodin. *Certified semantics and analysis of JavaScript.* PhD thesis, Université de Rennes, 2017.

9 Facebook, Inc. React: A JavaScript library for building user interfaces. `https://www.reactjs.org/`, 2019. Accessed: 2019-12-19.

10 Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 31–44, 2012. URL: `https://doi.org/10.1145/2103656.2103663`, `doi:10.1145/2103656.2103663`.

11 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010. URL: `https://doi.org/10.1007/978-3-642-14107-2_7`, `doi:10.1007/978-3-642-14107-2\_7`.

12 David Herman and Cormac Flanagan. Status report: specifying JavaScript with ML. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52. ACM, 2007.

13 Facebook Inc. React native: Learn once, write anywhere. `https://facebook.github.io/react-native/`, 2019. Accessed: 2019-12-19.

14 Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 51–62, 2017. URL: `https://doi.org/10.1145/3133841.3133846`, `doi:10.1145/3133841.3133846`.

15 Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017. URL: `http://doi.acm.org/10.1145/3133910`, `doi:10.1145/3133910`.

16 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A semantics for the essence of react. Technical Report CS-2020-03, University of Waterloo, 2020. URL: `https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2020-03.pdf`.

17 Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 505–519, 2015. URL: `https://doi.org/10.1145/2814270.2814272`, `doi:10.1145/2814270.2814272`.

18 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008. URL: `https://doi.org/10.1007/978-3-540-89330-1_22`, `doi:10.1007/978-3-540-89330-1\_22`.

**19** Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Trans. Software Eng.*, 43(2):128–144, 2017. URL: `https://doi.org/10.1109/TSE.2016.2586066`, `doi:10.1109/TSE.2016.2586066`.

**20** Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 325–335, 2015. URL: `https://doi.org/10.1109/ICSE.2015.52`, `doi:10.1109/ICSE.2015.52`.

**21** Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 566–577, 2017. URL: `https://doi.org/10.1109/ASE.2017.8115667`, `doi:10.1109/ASE.2017.8115667`.

**22** Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 346–356, 2015. URL: `https://doi.org/10.1145/2737924.2737991`, `doi:10.1145/2737924.2737991`.