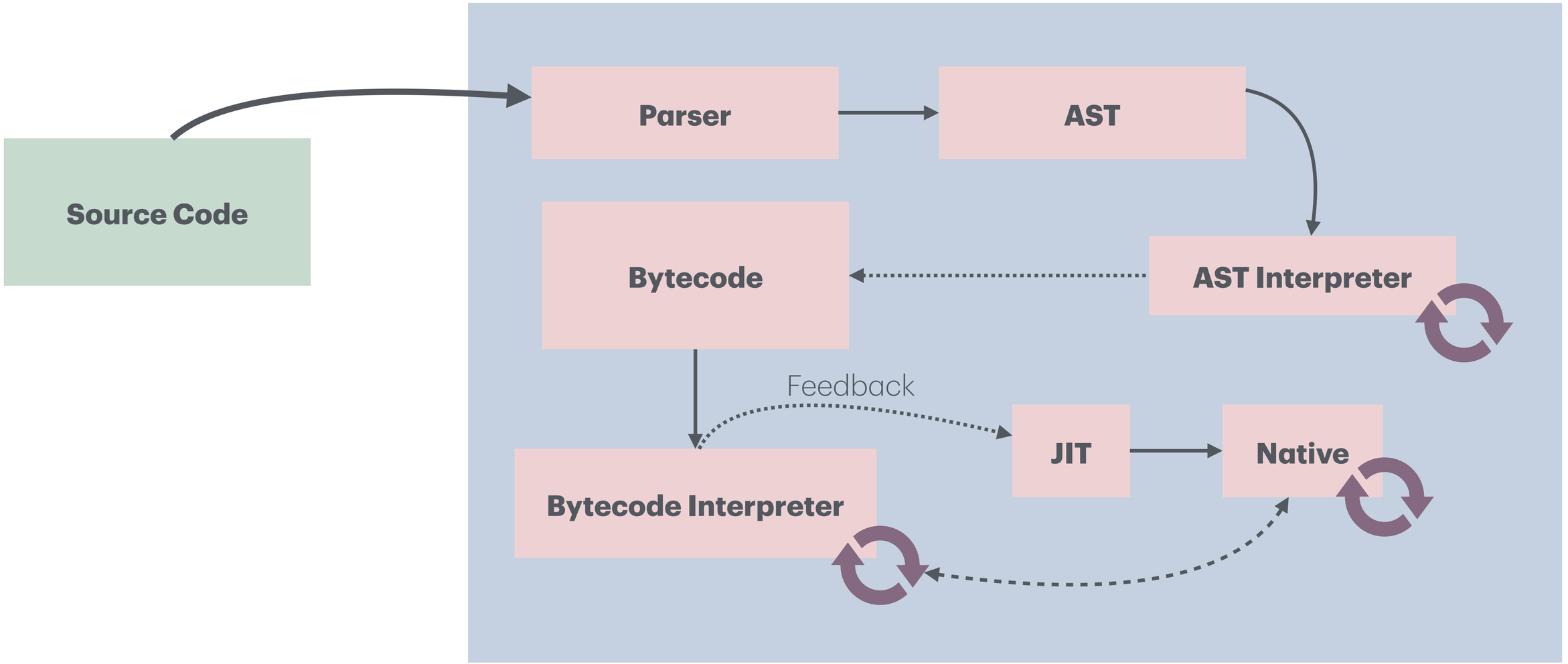
Why generating TAC for **JavaScript** is hard?

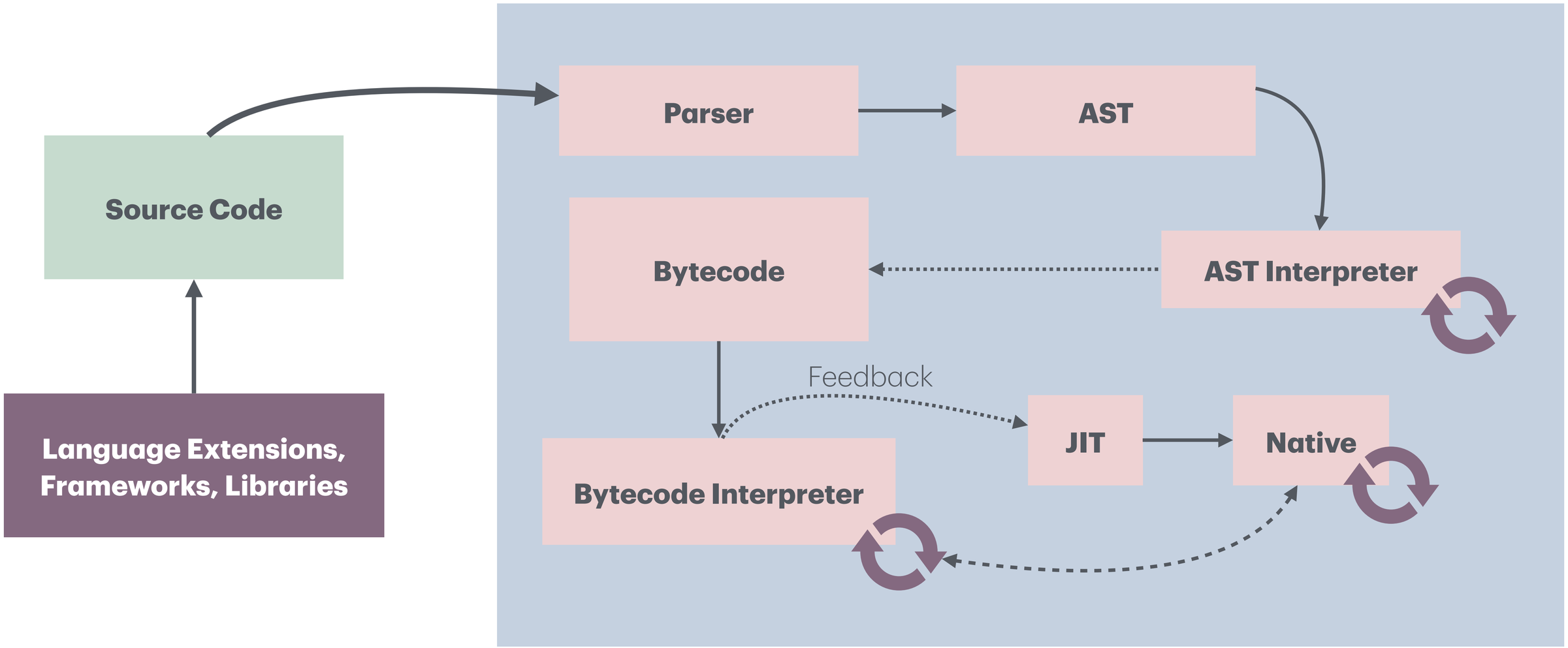
IICT, Bengaluru

Meetesh Kalpesh Mehta, Manas Thakur, 29th September '24

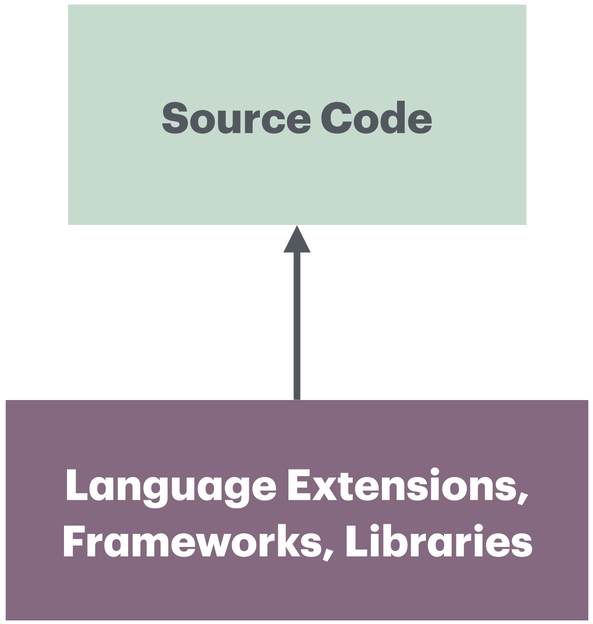
Managed Runtimes: JavaScript



Managed Runtimes: JavaScript



Managed Runtimes: JavaScript

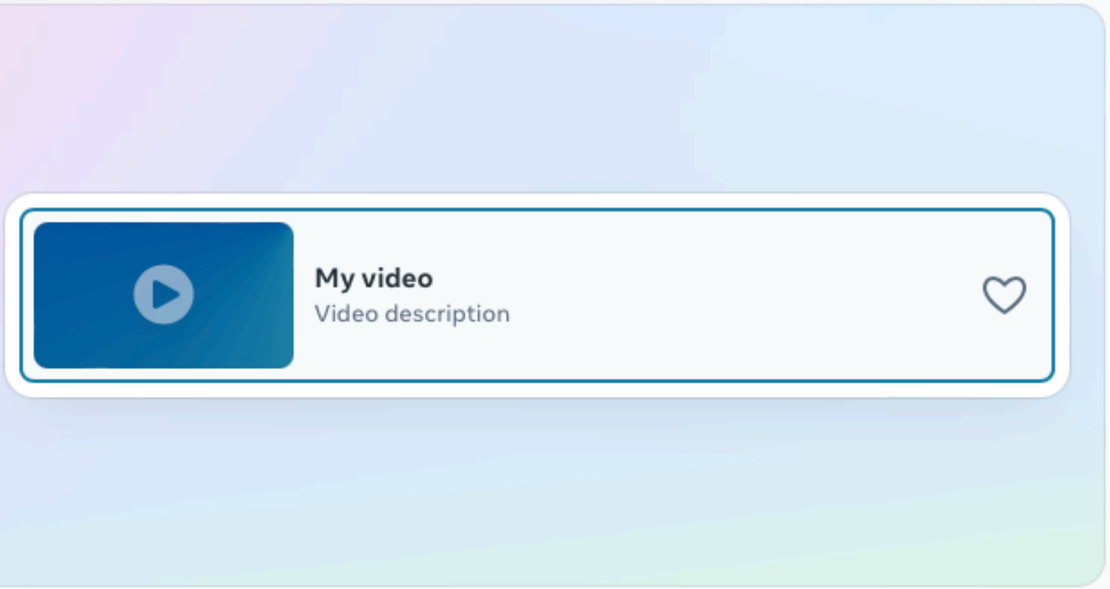


React: Functional UI’s

- Even though JavaScript is dynamic, not all code written in it really is.
- **React Forget compiler** performs memoization to speedup renders.

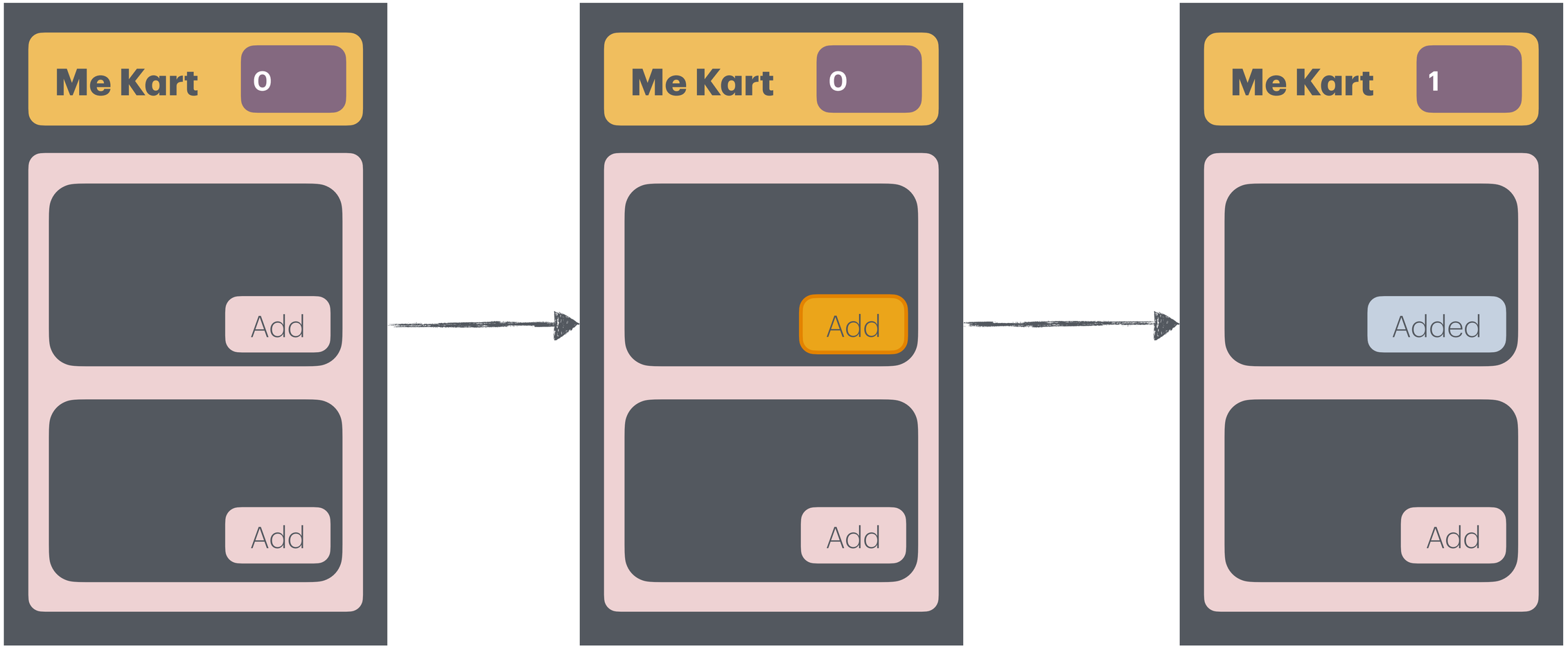
Video.js

```
function Video({ video }) {
  return (
    <div>
      <Thumbnail video={video} />
      <a href={video.url}>
        <h3>{video.title}</h3>
        <p>{video.description}</p>
      </a>
      <LikeButton video={video} />
    </div>
  );
}
```

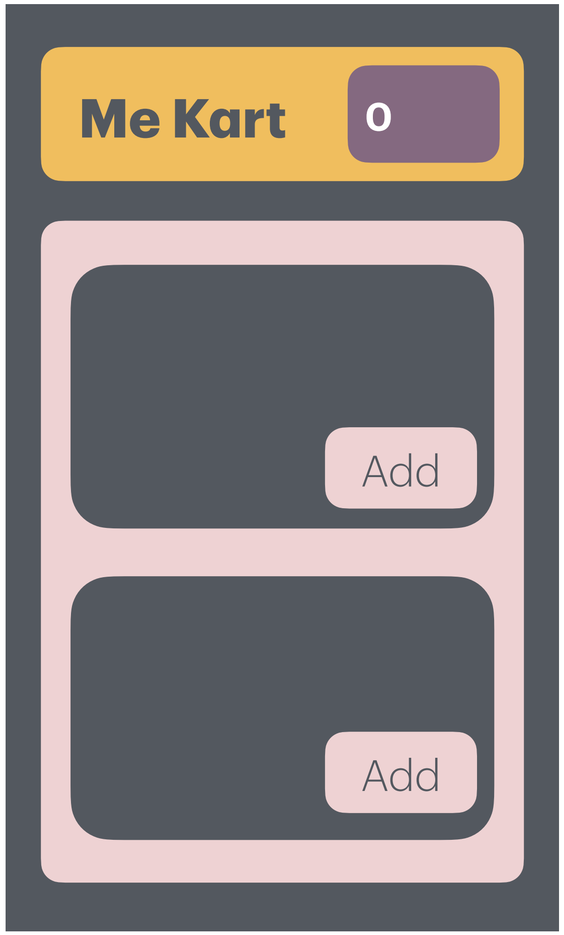


https://react.dev/

Managed Runtimes: Modelling UI

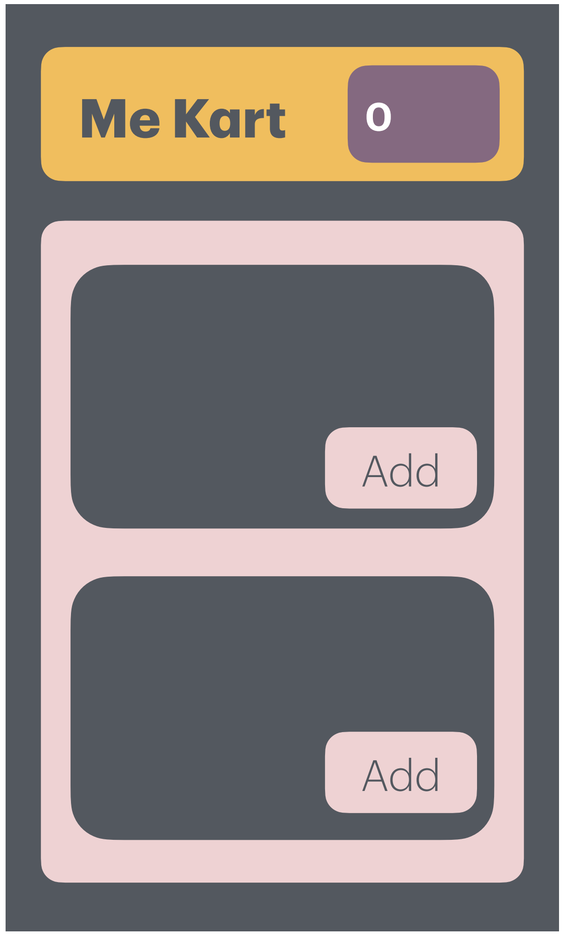


Managed Runtimes: Modelling UI



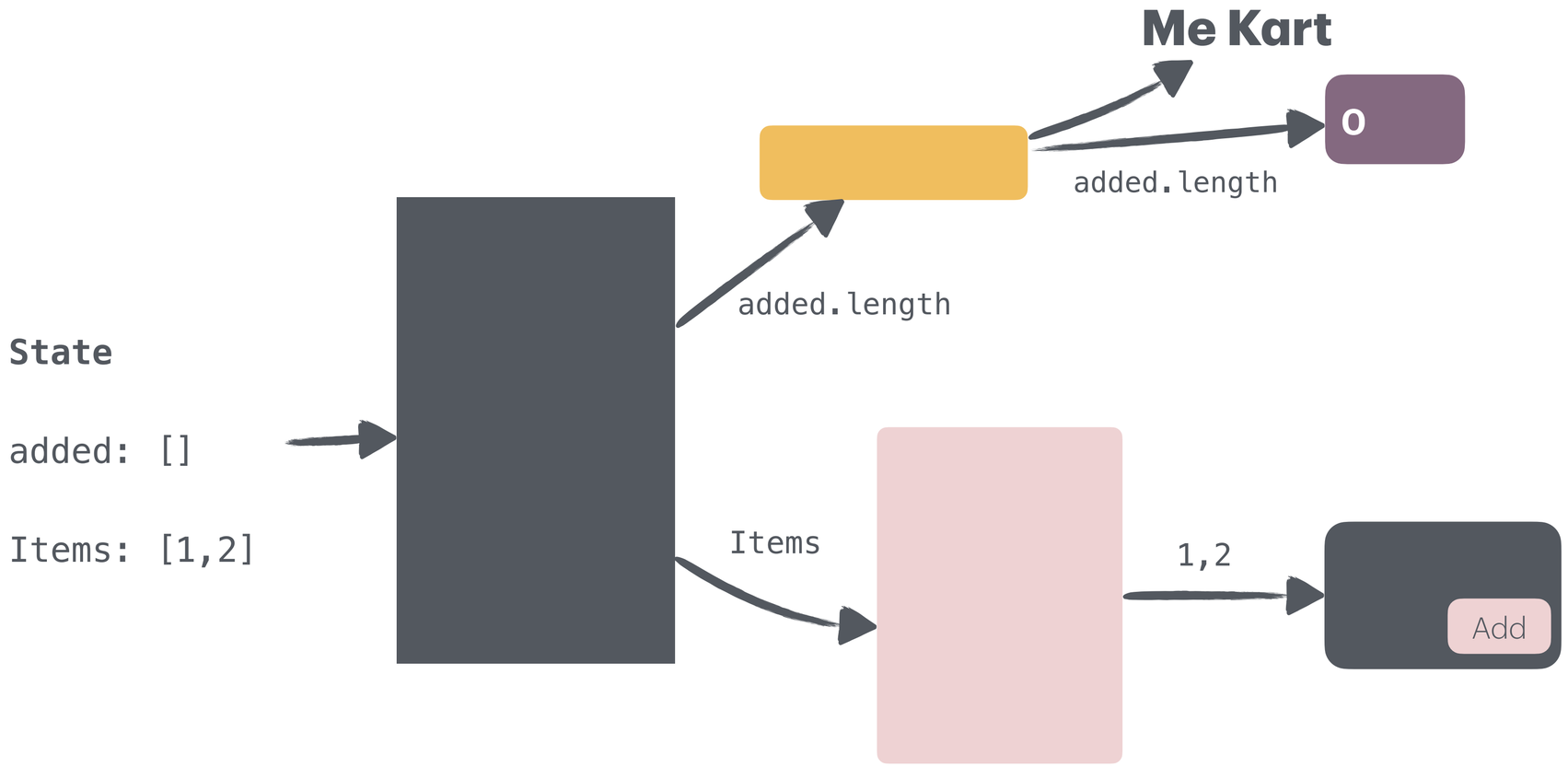

```
y       = f(x)
picture = f(state)
```

Managed Runtimes: Modelling UI

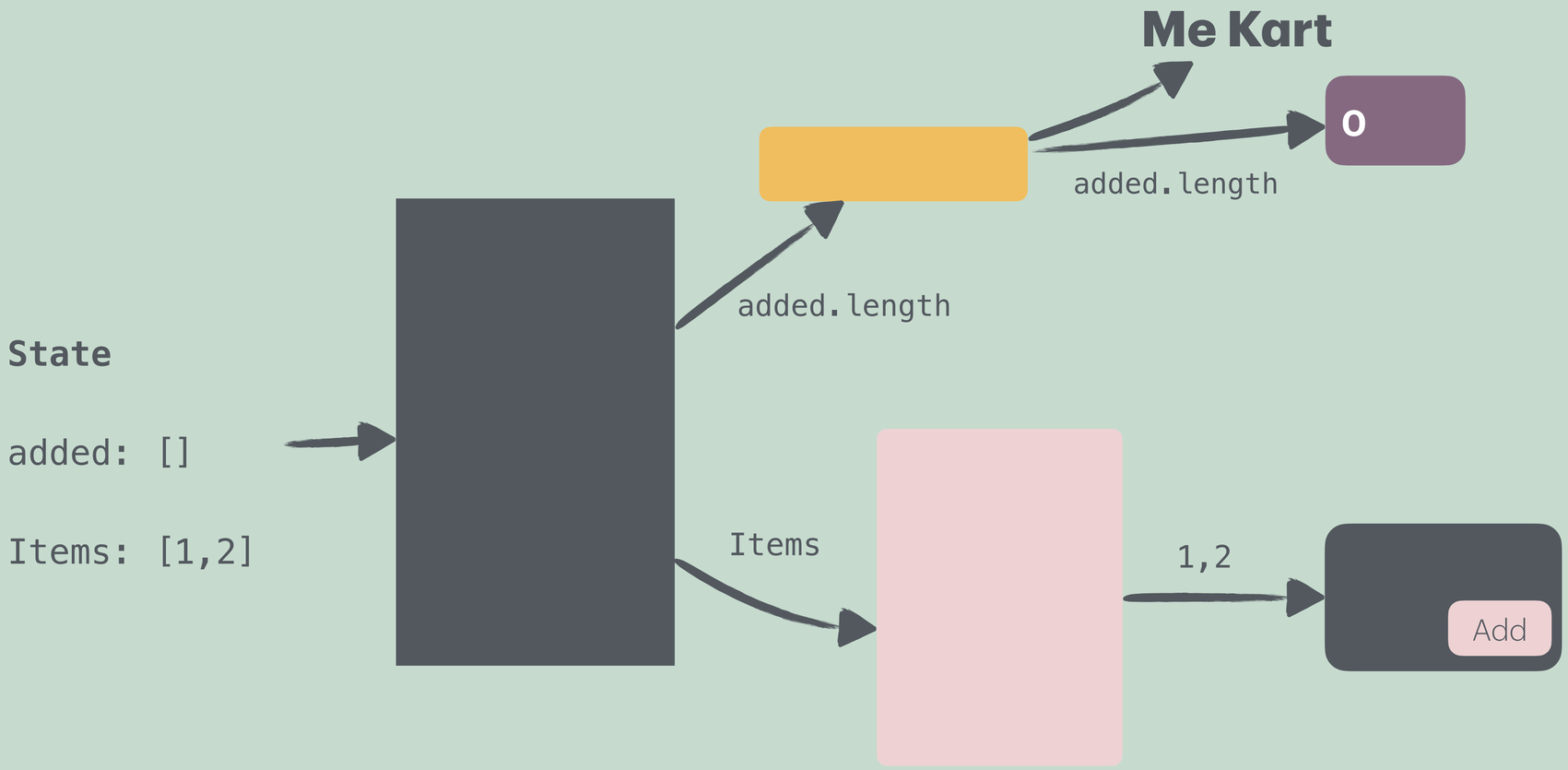


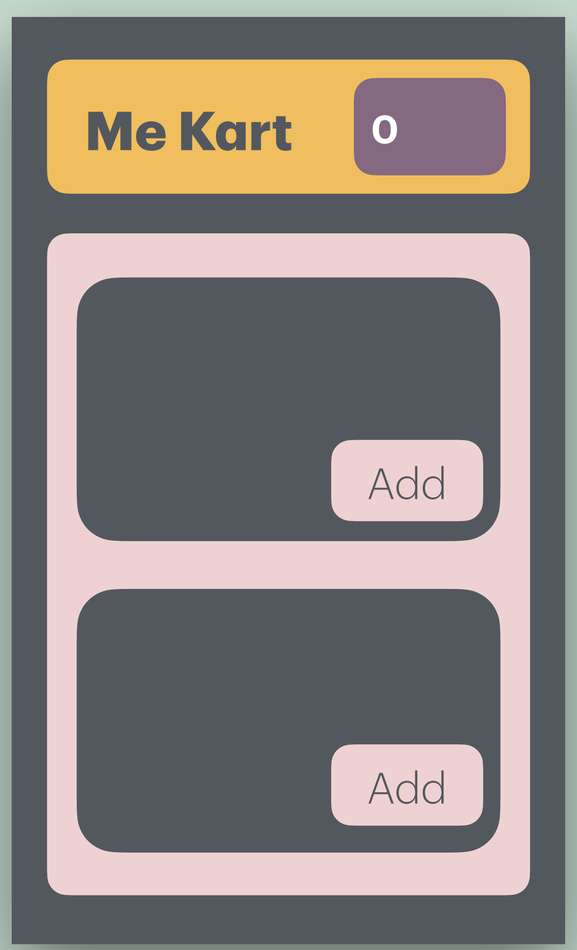

```
= function({ added: [], items: [1,2] })
```

Managed Runtimes: Modelling UI

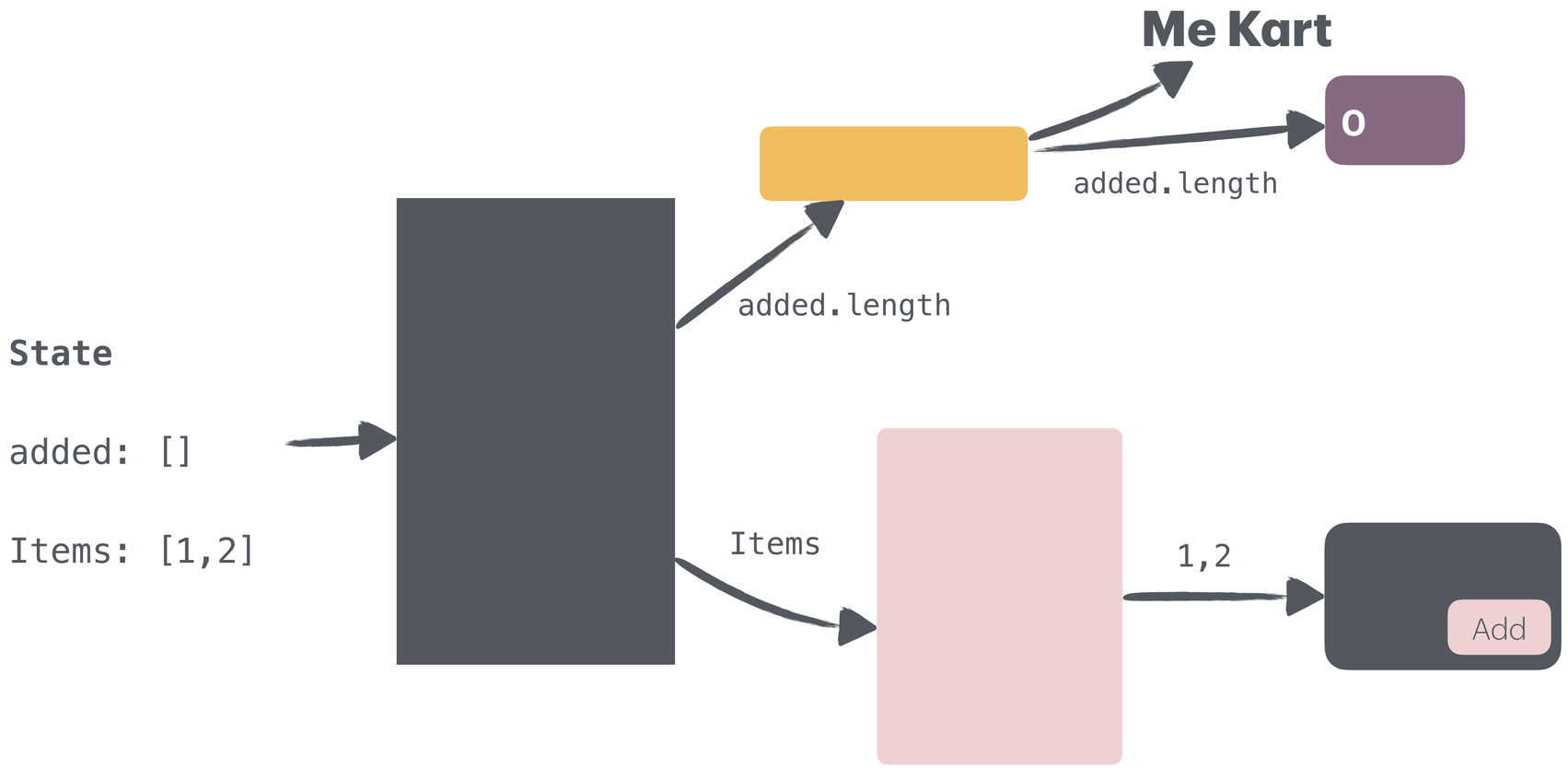


Managed Runtimes: Modelling UI

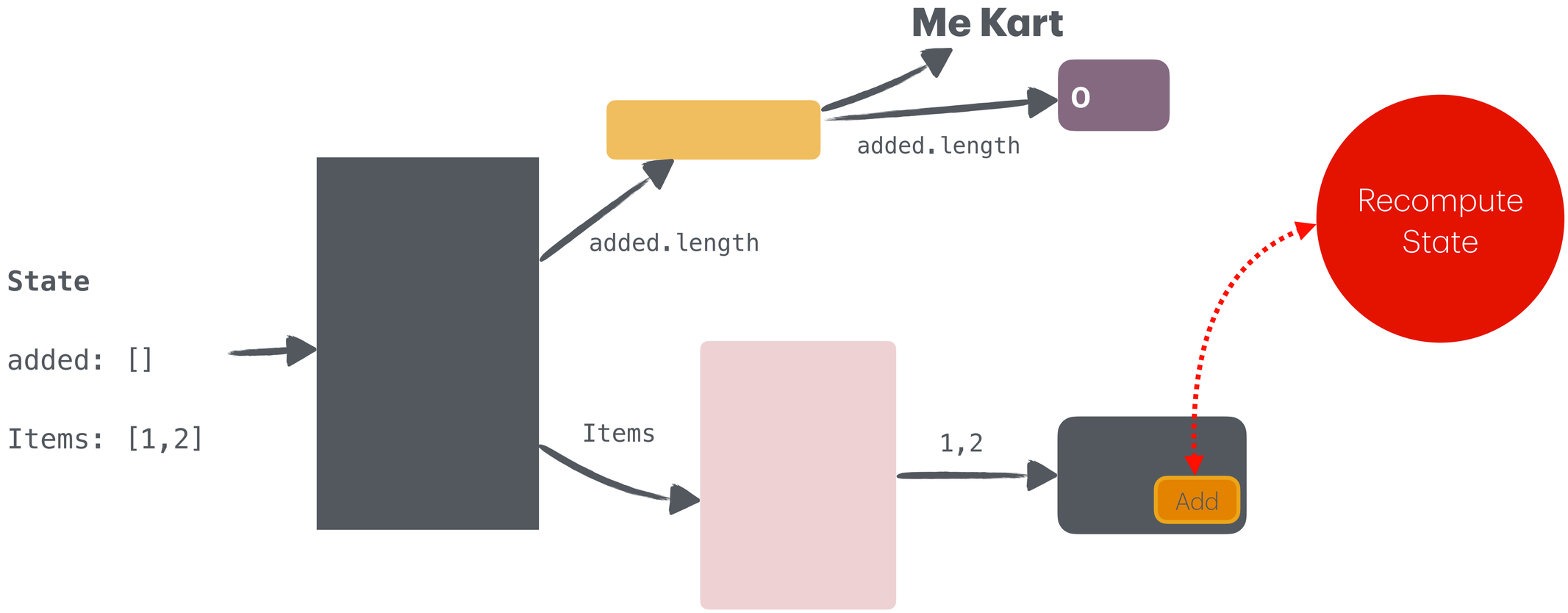




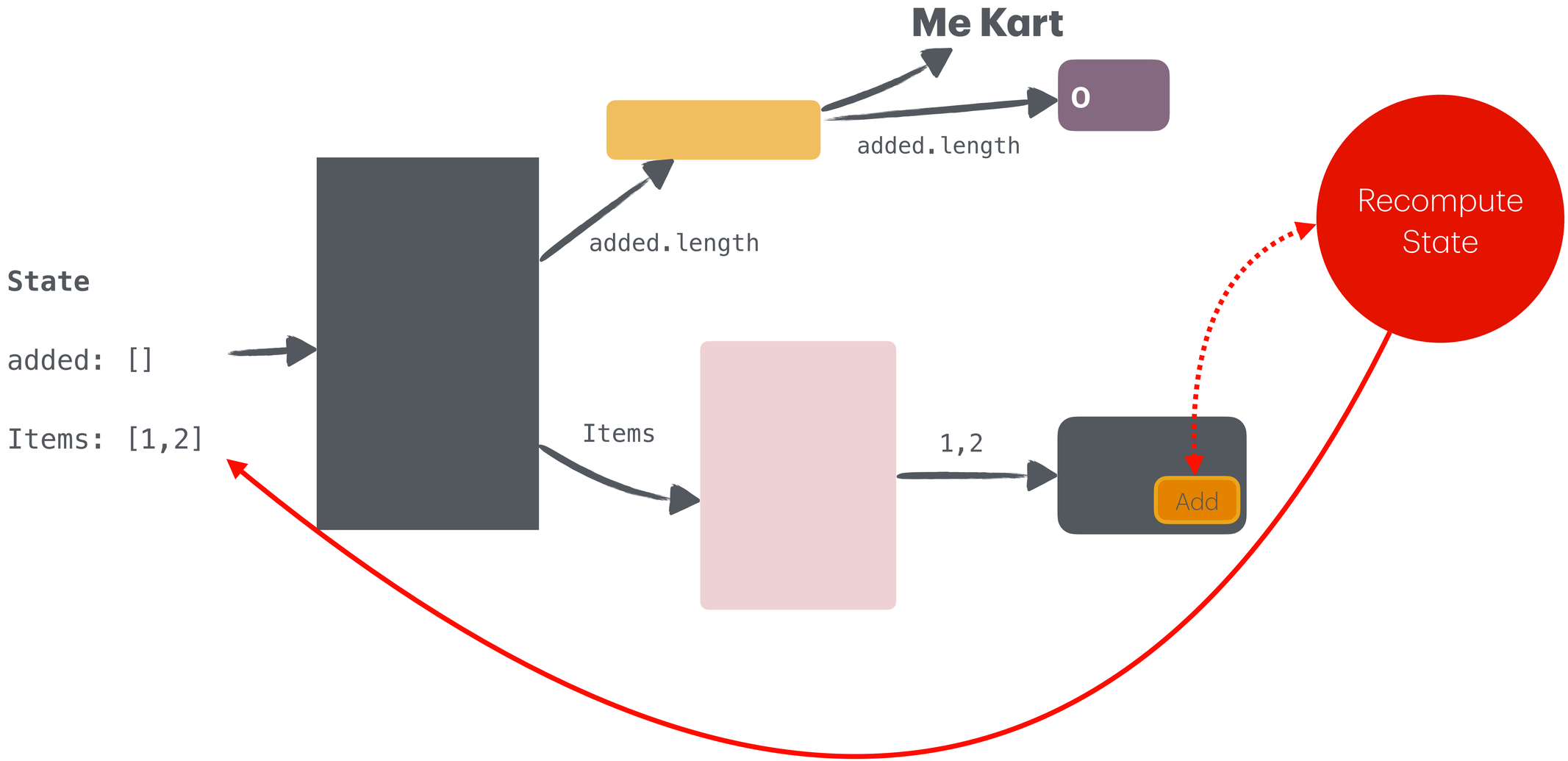
Managed Runtimes: Modelling UI



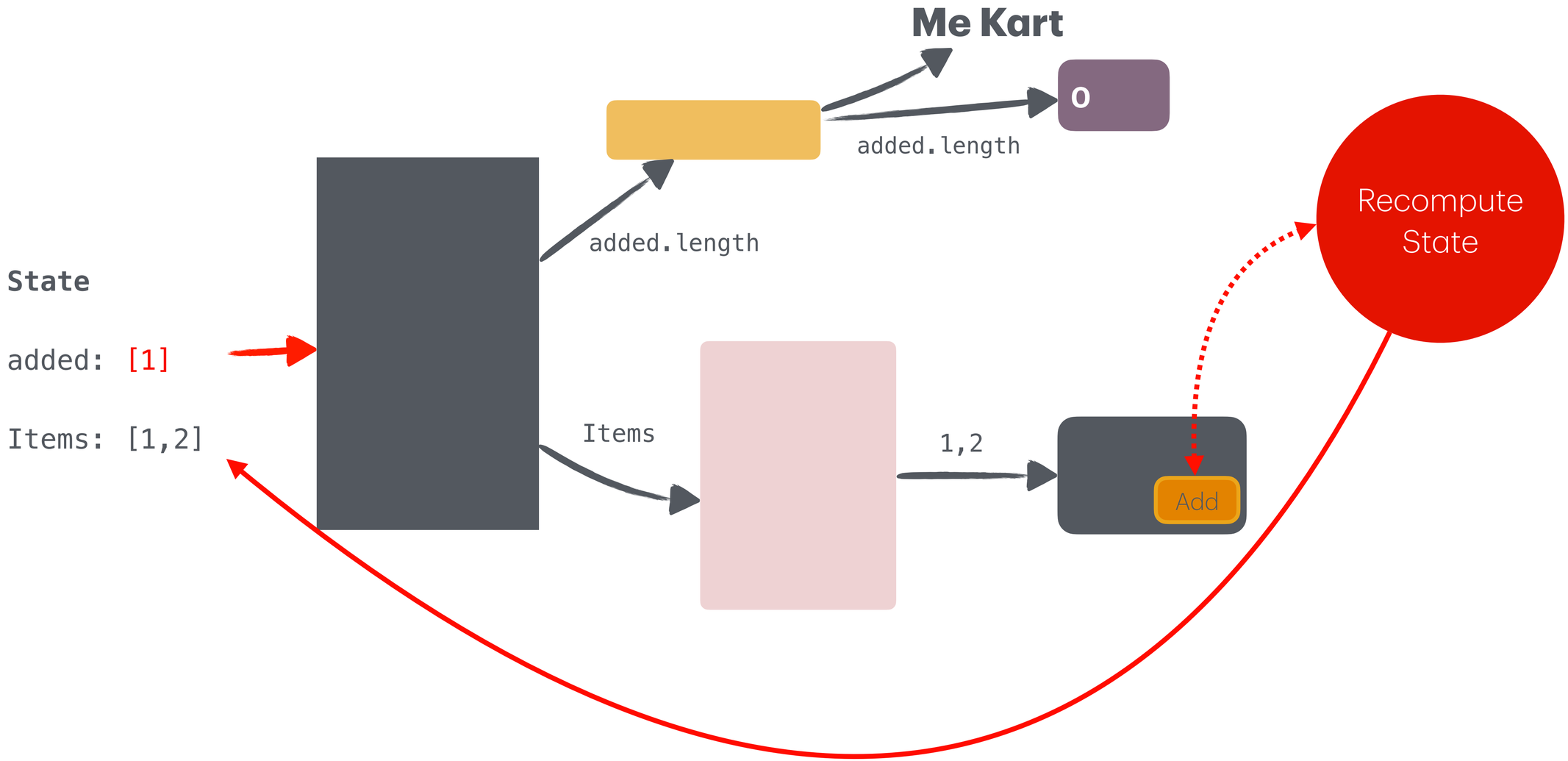
Managed Runtimes: Modelling UI



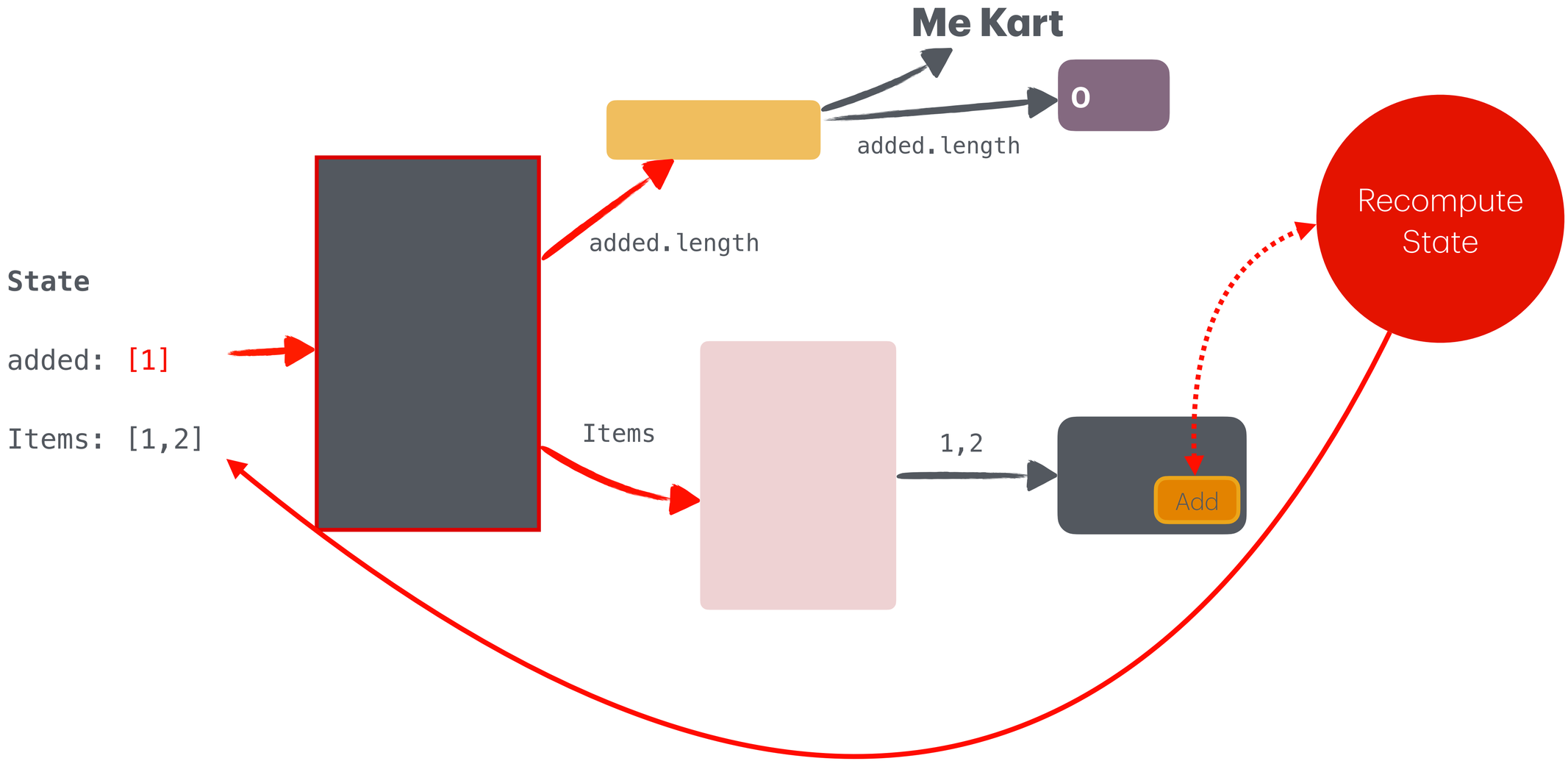
Managed Runtimes: Modelling UI



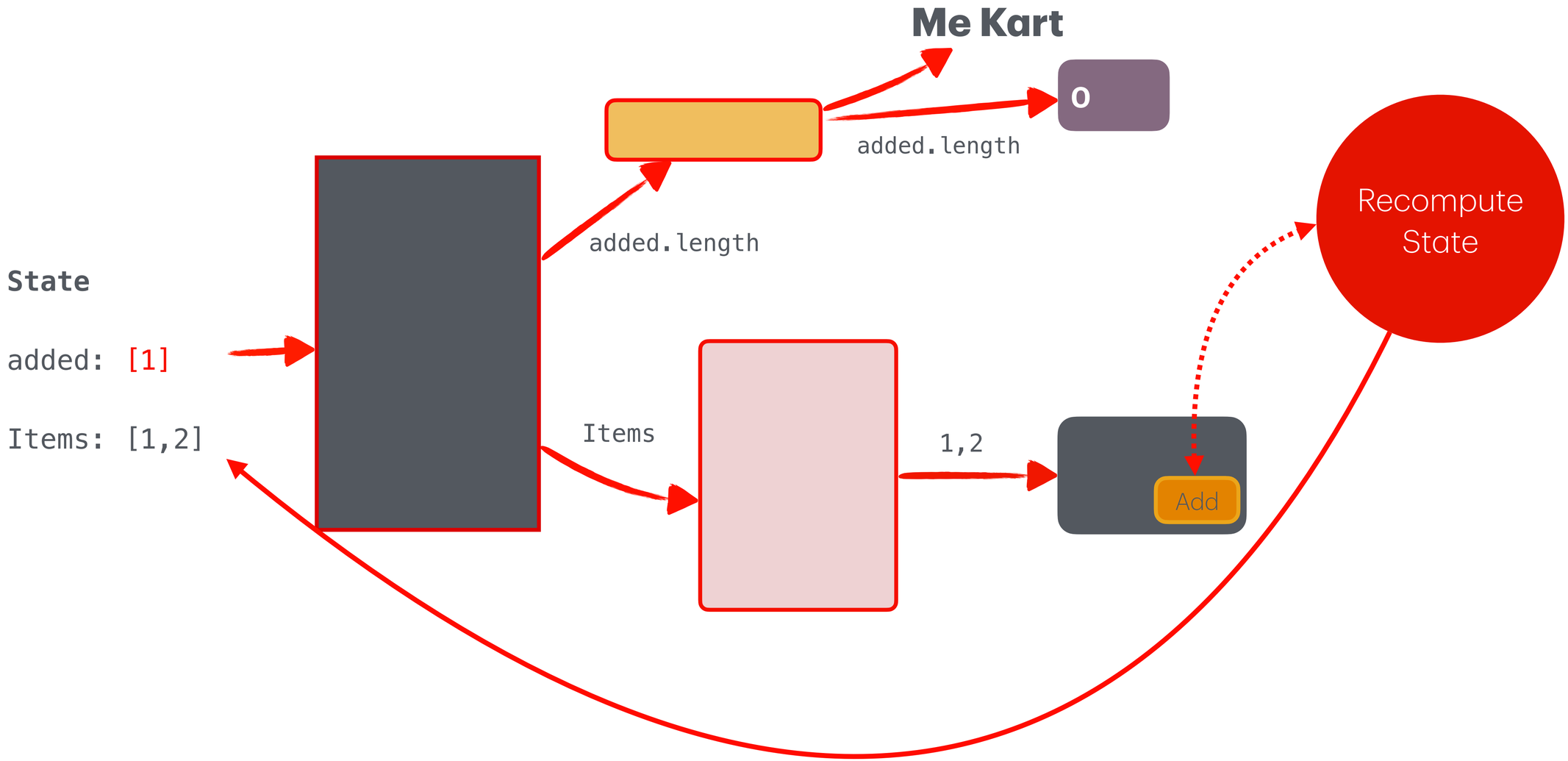
Managed Runtimes: Modelling UI



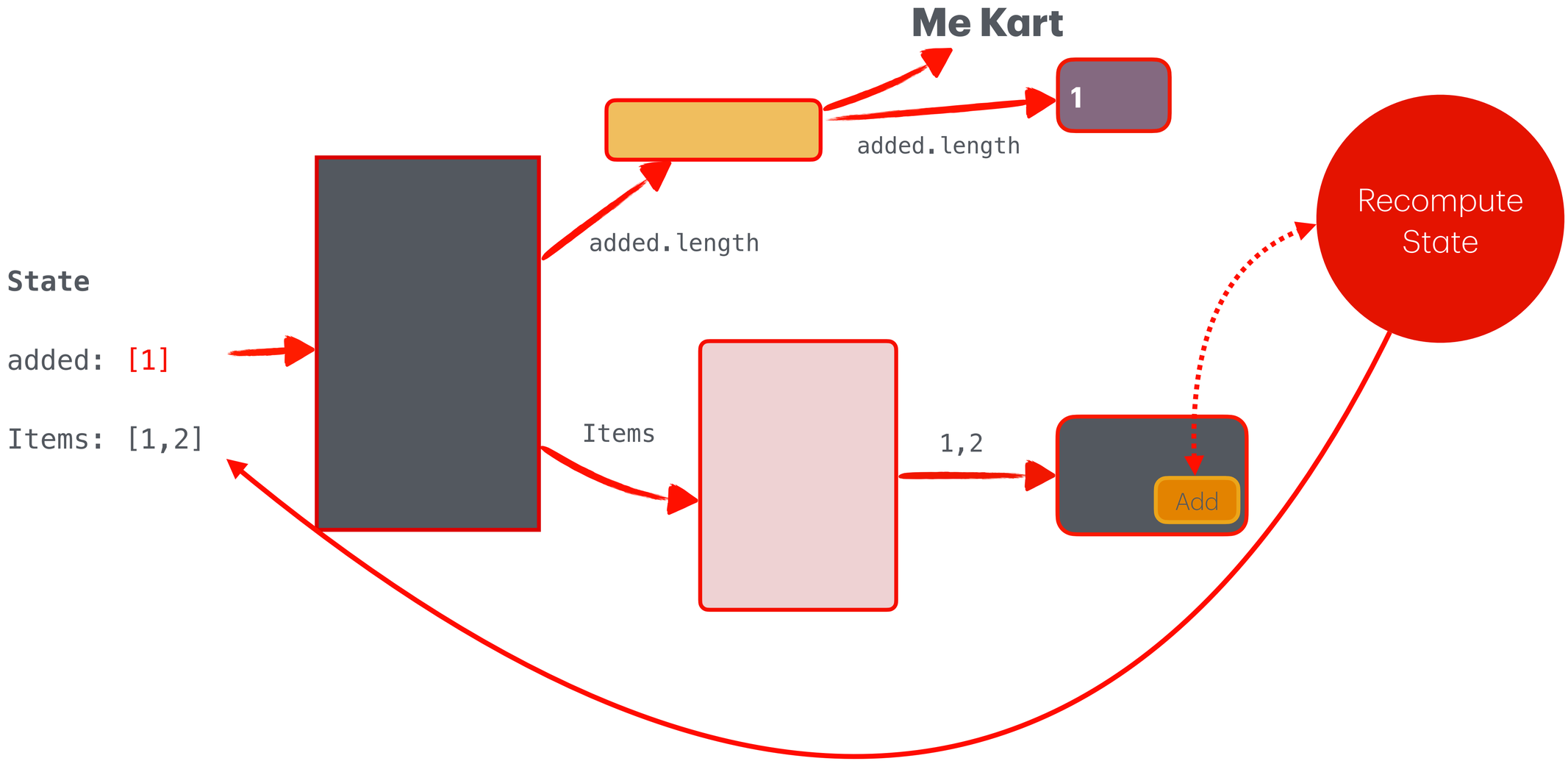
Managed Runtimes: Modelling UI



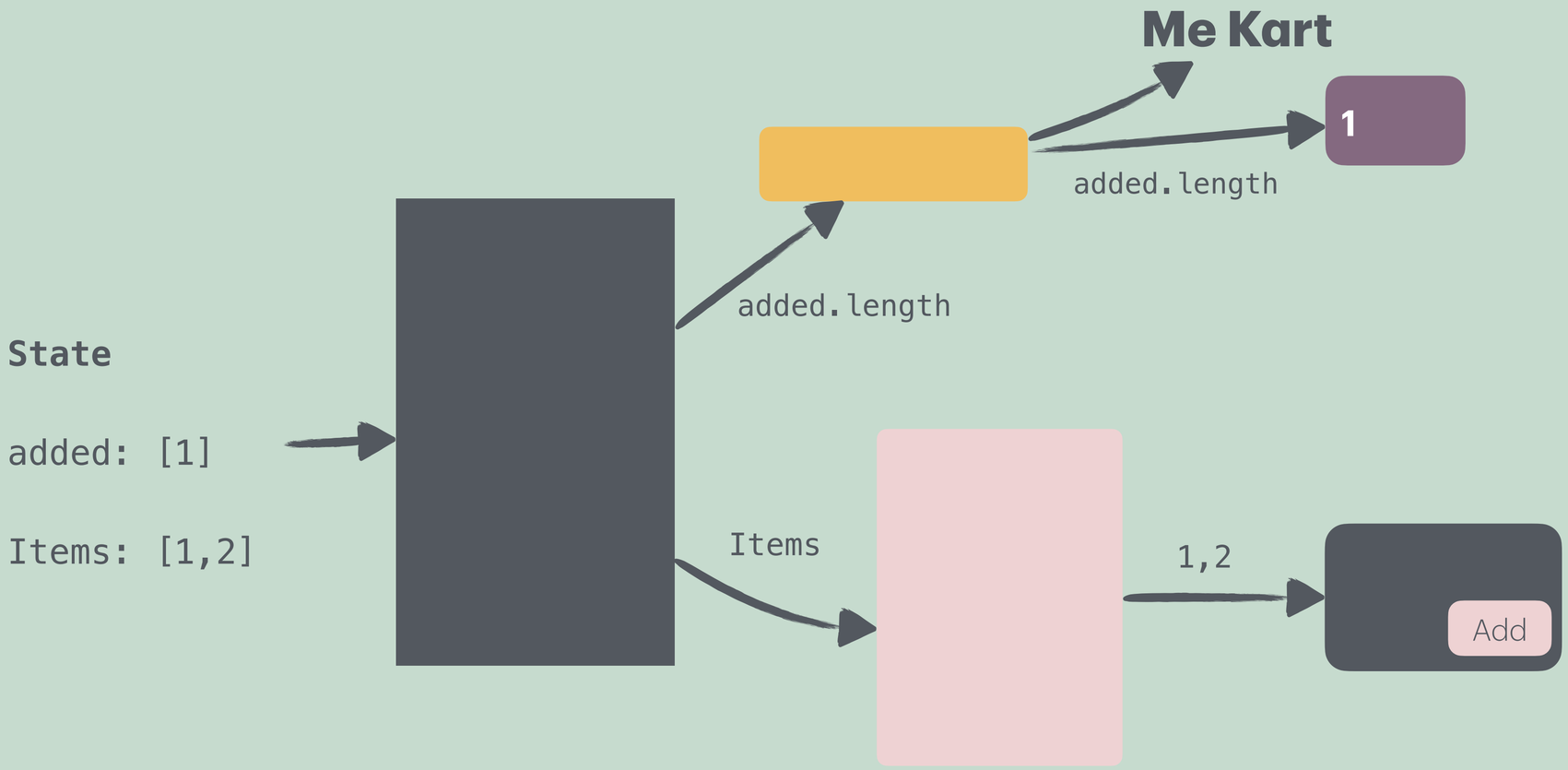
Managed Runtimes: Modelling UI

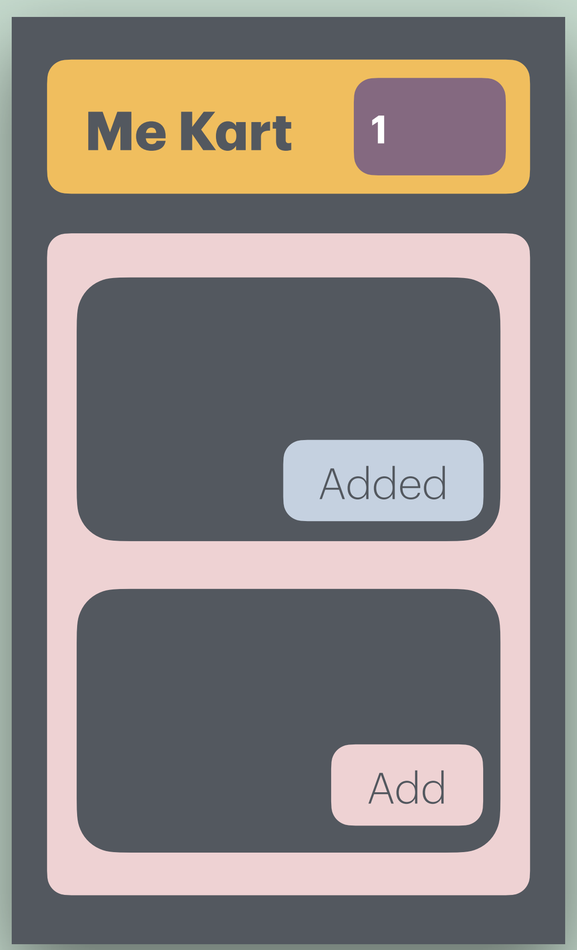


Managed Runtimes: Modelling UI

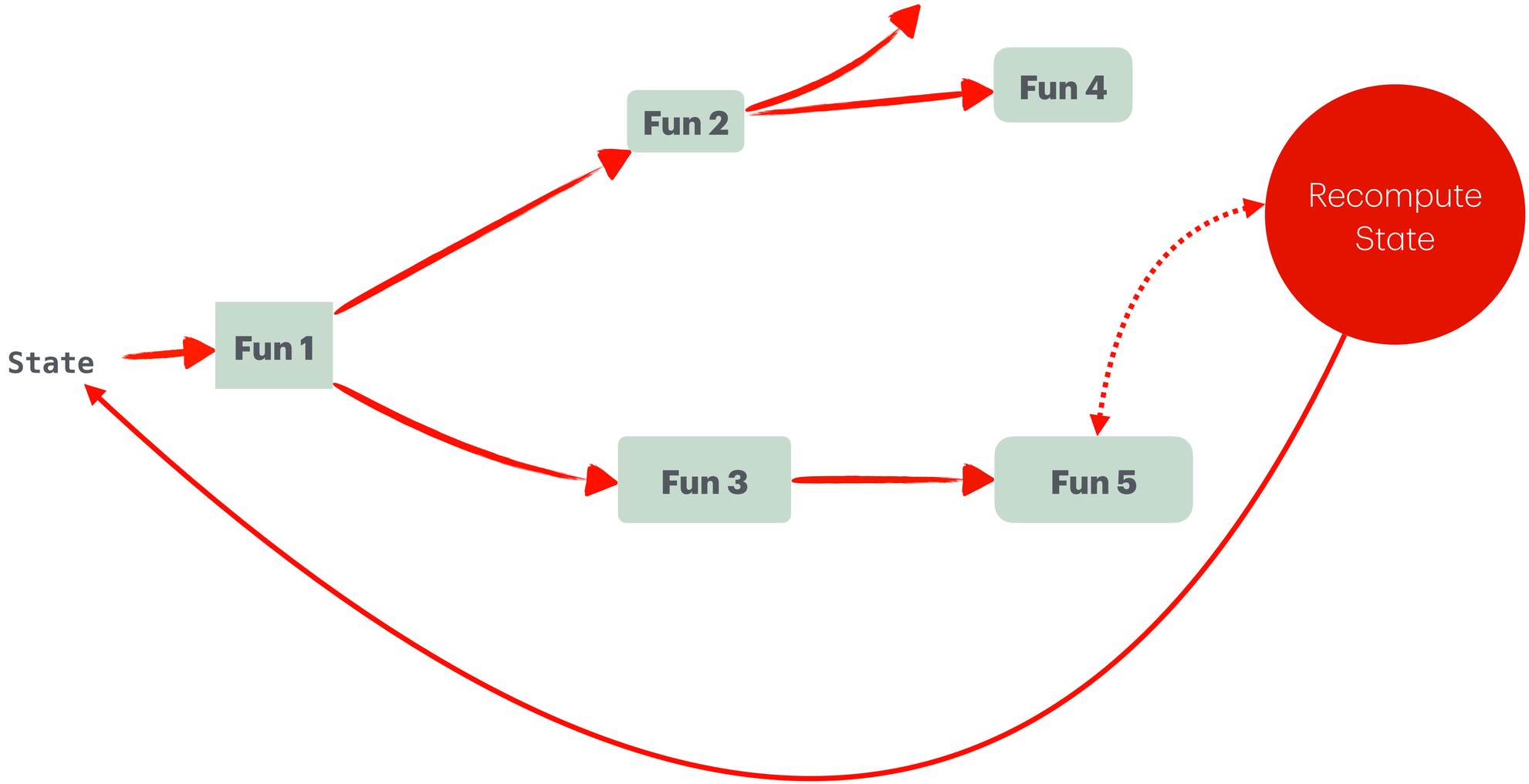


Managed Runtimes: Modelling UI





Managed Runtimes: De-sugaring



Managed Runtimes: De-sugaring

```
const App = ({state}) => <Fun1 state={state} />

const Fun1 = ({state}) => <div>
  <Fun2 len={state.len} />
  <Fun3 items={state.items} />
</div>

const Fun2 = ({len}) => <div>
  Me Kart
  <Fun4 len={len} />
</div>

const Fun3 = ({items}) => <div>
  {items.map(i => <Fun5 item={i} />)}
</div>
```

Managed Runtimes: De-sugaring

```
const App = ({state}) => createElement(“Fun1”, { state : state }, [])

const Fun1 = ({state}) =>
   createElement(“div”, {},
    [
      createElement(“Fun2”, { len : state.len }),
      createElement(“Fun3”, { items : state.items }),
    ]
   )

const Fun2 = ({len}) =>
   createElement(“div”, {},
    [
      createTextElement(“Me Kart”),
      createElement(“Fun4”, { len : len }),
    ]
   )

const Fun3 = ({items}) =>
   createElement(“div”, {},
    [
      ...items.map(I => createElement(“Fun5”, { item : i }, []))
    ]
   )
```

Why JavaScript Works

- Fast (enough) + useful abstractions
- Mature language infrastructure (standard tests, parsers, working groups)
- Incredible backwards compatibility (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard tests, parsers, working groups)
- Incredible backwards compatibility (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard tests, parsers, working groups)
- Incredible backwards compatibility (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard tests, parsers, working groups)
- Incredible backwards compatibility (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard **tests**, parsers, working groups)
- Incredible backwards compatibility (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard **tests**, parsers, working groups)
- Incredible **backwards compatibility** (babel tests language compatibility all the way back to node@2015)
- Very well maintained specification
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard **tests**, parsers, working groups)
- Incredible **backwards compatibility** (babel tests language compatibility all the way back to node@2015)
- Very well maintained **specification**
- Wide adoption

Why JavaScript Works

NodeJS: Single Threaded Execution Model

React: Functional UI design

React Native: hybrid development

- Fast (enough) + **useful abstractions**
- Mature language infrastructure (standard **tests**, parsers, working groups)
- Incredible **backwards compatibility** (babel tests language compatibility all the way back to node@2015)
- Very well maintained **specification**
- Wide **adoption**

In a nutshell

```
Expression(Expression, Expression, ...);
```

```
((1,2,3,() => { console.log("Hello World") }))()
```

In a nutshell

```
Expression(Expression, Expression, ...);
```

```
Callee(Identifier, Identifier, …);
```

```
((1,2,3,() => { console.log("Hello World") }))()
```

```
let t$1 = 1
let t$2 = 2
let t$3 = 3
let t$4 = function() { console.log("Hello World"); }
let t$5 = (t$1, t$2, t$3, t$4)
let t$6 = t$5()
```

Motivating Example - 1

```
let a = (1,2,3,() => {})

console.log(a.name)
```

Motivating Example - 1

```
let a = (1,2,3,() => {})

console.log(a.name)
```

```
let t$1 = 1
let t$2 = 2
let t$3 = 3
let t$4 = () => {}

let a = (t$1, t$2, t$3, t$4)

console.log(a.name)
```

Motivating Example - 1

```
let a = (1,2,3,() => {})

console.log(a.name)
```

```
let t$1 = 1
let t$2 = 2
let t$3 = 3
let t$4 = () => {}

let a = (t$1, t$2, t$3, t$4)

console.log(a.name)
```

Motivating Example - 1

```
let a = (1,2,3,() => {})

console.log(a.name)
```

```
let t$1 = 1
let t$2 = 2
let t$3 = 3
let t$4 = () => {}

let a = (t$1, t$2, t$3, t$4)

console.log(a.name)
```

Not Semantics Preserving

Output: ‘’

Output: ‘t$4’

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");

let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

Hii

Motivating Example - 2

```
var yieldSet, C, iter;
function* g() {
  C = class {
    [(console.log("Hii"), "pokemon")] =  "Pikachu"

    get [yield]() { return 'Ash Ketchum'; }
  };
}
iter = g();
iter.next();
iter.next("name");
```

```
let myObj = new C()
console.log(myObj.pokemon)
console.log(myObj.name)
```

```
C = class {
  "pokemon": "Pikachu"
  get name() {
   return 'Ash Ketchum'
  }
}
```

Hii

Pikachu

Ash Ketchum

JavaScript Code is Fragile to Change

- Side-effect prone nature

JavaScript Code is Fragile to Change

- Side-effect prone nature

```
o.f = 10    =/=    o.f = 10
                   let t$1 = o.f
```

JavaScript Code is Fragile to Change

- Side-effect prone nature
- Complex semantics

JavaS ange

- Side-effect
- Complex s

meetesh06 3 weeks ago

I am looking at this testcase: test262

I see that babel fails this test on the website; I don't know how the babel test pipeline works so I am not sure what to make of it! (would love to know how these tests apply to babel; as I am trying to create a s2s transformer aimed at transforming code to a subset of JS)

Source

```
// ...
await /x.y/g;
// ...
```

Parsed AST

```
BINOP -- Left: Identifier -- 'await' // <- Non module level code treats top level await as an identifier?? is t
            |
            |-- Right: Member Expression...
```

Transformed Code

```
let js3$5 = await; // <- ;(
let js3$6 = x.y;
let js3$4 = js3$5 / js3$6;
let js3$7 = g;
let js3$3 = js3$4 / js3$7;
```

Is it incorrect to assume that **top-level awaits imply module mode** for the parsed code, or are there cases when this assumption is incorrect?

↑ 1

Answered by **nicolo-ribaudo** 3 weeks ago

await is a valid identifier outside of modules, so as you discovered that file can be parsed in two different ways:

JavaScript Code is Fragile to Change

- Side-effect prone nature
- Complex semantics
- Things silently break

JavaS ge

- Side-effect
- Complex se
- Things siler

```
let o = {
  val10: 10,
  m: function() { return this.val10 }
}
let r1 = o.m() // ===> 10
```

```
let o = {
  val10: 10,
  m: function() { return this.val10 }
}

let t$1 = o.m
let r1 = t$1() // ===> undefined
```

JavaScript Code is Fragile to Change

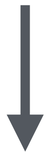
- Side-effect prone nature
- Complex semantics
- Things silently break
- Strange semantics

JavaScript Code is Fragile to Change

- Side-effec
- Complex s
- Things sil
- Strange s

```
var z = 3;
let temp = delete delete z

// temp is 'true'
```



```
var z = 3;
let t1 = delete z
let temp = delete t1

// temp is 'false'
```

JavaScript Code is Fragile to Change

- Side-effec
- Complex s
- Things sil
- Strange s

```
var z = 3;
let temp = delete delete z
  |
let temp = delete ( delete  z )
  |
let temp = delete ( false )
  |
let temp = delete false          <- RValue
  |
let temp = true


var z = 3;
let t1 = delete z
  |
let t1 = false

let temp = delete t1
  |
let temp = delete t1             <- LValue
  |
let temp = false
```

JavaScript Code is Fragile to Change

- Side-effec
- Complex s
- Things sil
- Strange s

```
var z = 3;
let temp = delete delete z
  |
let temp = delete ( delete  z )
  |
let temp = delete ( false )
  |
let temp = delete false          <- RValue
  |
  |
  |
let t1 = false

let temp = delete t1
  |
let temp = delete t1             <- LValue
  |
let temp = false
```

> **Note:** The syntax allows a wider range of expressions following the `delete` operator, but only the above forms lead to meaningful behaviors.

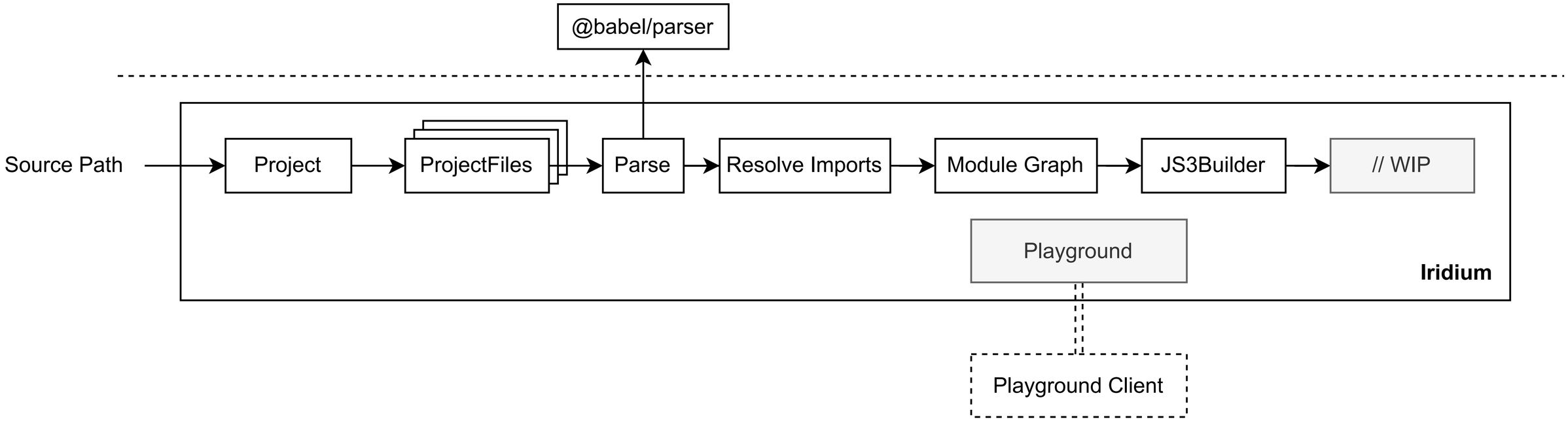
JavaScript Code is Fragile to Change

- Side-effect prone nature
- Complex semantics
- Things silently break
- Strange semantics

Iridium

- Slightly more manageable subset of the language
- Basic utilities needed for Analysis
- Model High Level Language Features

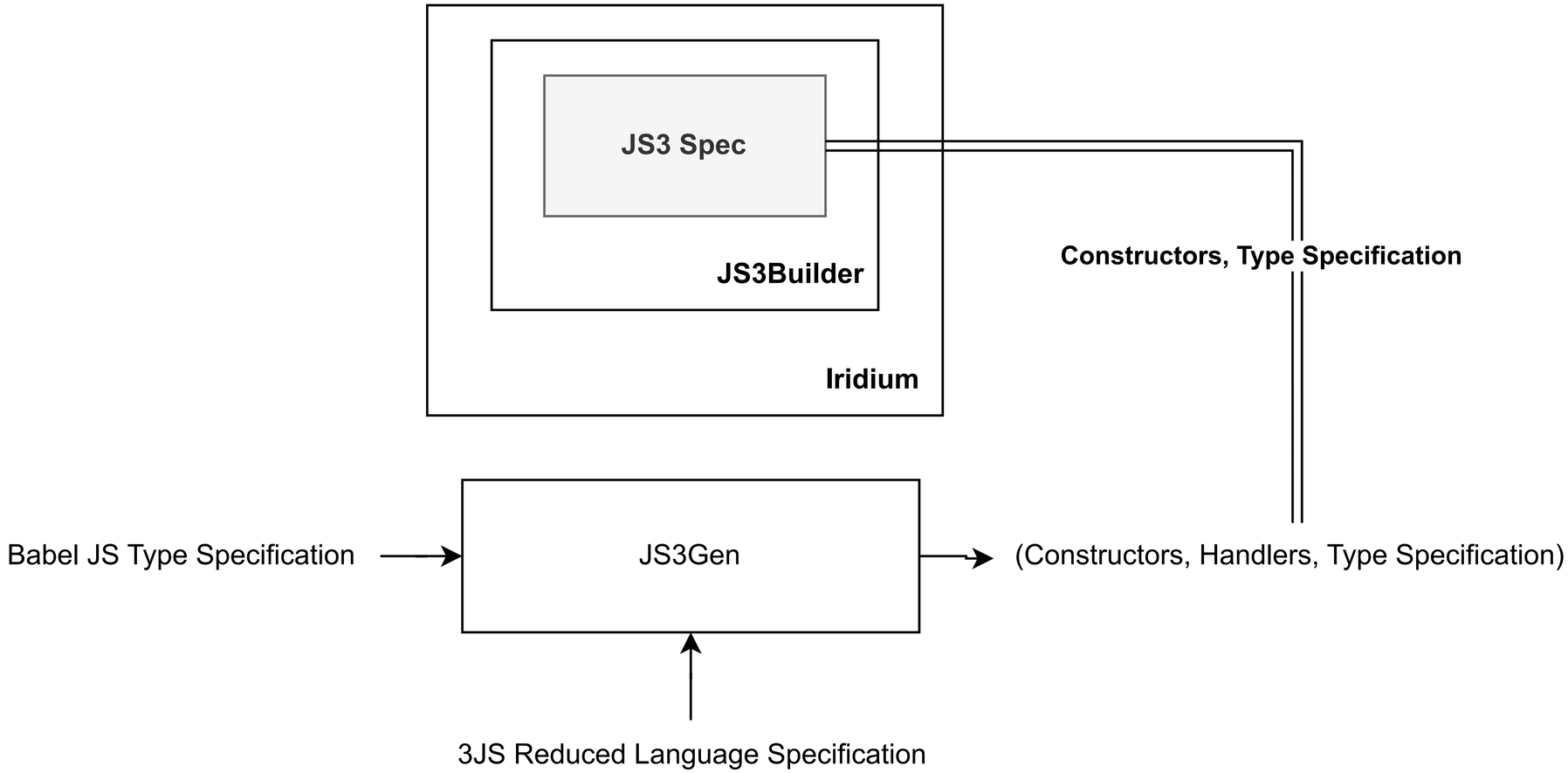
Iridium (Block Diagram)



JS3 Gen

- Make the process of subsetting language less error prone.
- Speed up development by generating stubs.

JS3 Gen (Block Diagram)



Why Even Perform Static Analysis?

- Modelling and optimising High Level concepts in hard.

Why Even Perform Static Analysis?

- Modelling and optimising High Level concepts in hard.

```cpp
#include <functional>
#include <iostream>

std::function<int(void)> foo() {
  std::function<int(void)> fun = []() { return 11; };
  return fun;
}

int main(int argc, char *argv[]) {
  int res = foo()();
  std::cout << res << std::endl;
  return 1;
}
```

```cpp
#include <functional>
#include <iostream>

int main(int argc, char *argv[]) {
  std::cout << 11 << std::endl;
  return 1;
}
```

Why Even Perform Static Analysis?

- Modelling and optimising High Level concepts in hard.

```cpp
#include <functional>
#include <iostream>

std::function<int(void)> foo() {
  std::function<int(void)> fun = []() { return 11; };
  return fun;
}

int main(int argc, char *argv[]) {
  int res = foo()();
  std::cout << res << std::endl;
  return 1;
}
```

X86_64 Assembly

```
Clang (O0): 3637 LOC , 148 fns
Clang (O1): 300  LOC , 11 fns
Clang (O2): 290  LOC , 11 fns
Clang (O3): 290  LOC , 11 fns
```

Conclusion

- JavaScript **may very well be impossible** to statically analyse.

Conclusion

- JavaScript **may very well be impossible** to statically analyse.
- But there are **millions of lines** of code that run JavaScript.

Conclusion

- JavaScript **may very well be impossible** to statically analyse.
- But there are **millions of lines** of code that run JavaScript.
- Many of these are **TOTALLY PURE** functions

Conclusion

- JavaScript **may very well be impossible** to statically analyse.
- But there are **millions of lines** of code that run JavaScript.
- Many of these are **TOTALLY PURE** functions

Safe to Execute in Parallel

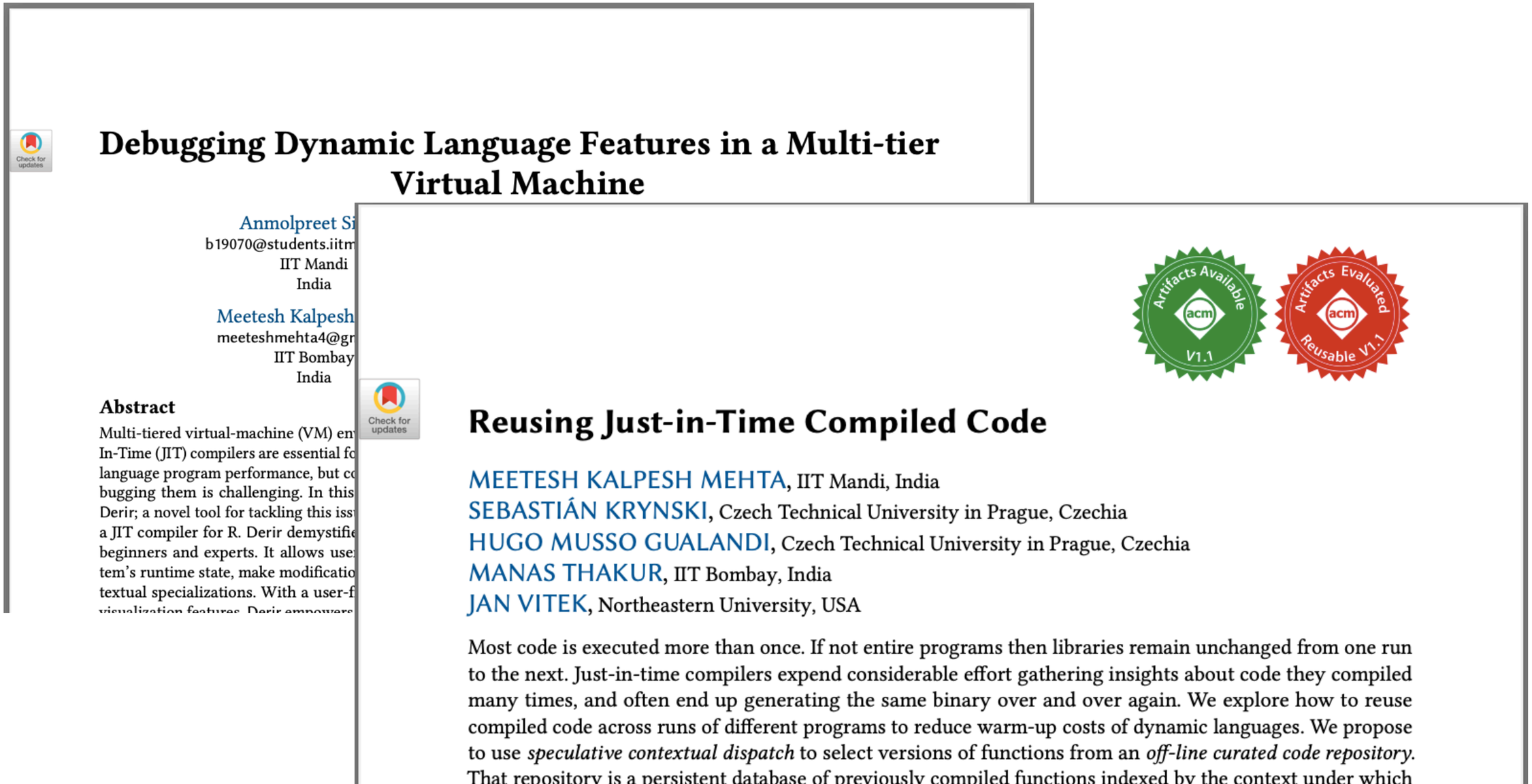
No Side Effects

Thunkable

Relevant Works

- **Call Graphs**
  - [Indirection-Bounded Call Graph Analysis, ECOOP-24]
  - [Correlation tracking for points-to analysis of javascript, ECOOP-12]
- **Analysing React Code**
  - React Forget Compiler - 2023
- **Tree Shaking / Code Splitting**
  - Webpack - 2014
- **Static Inference/Compilation**
  - [JavaScript AOT Compilation,DLS-18]
  - [Type Inference for Static Compilation of JavaScript, OOPSLA-16]

Experience with R



Debugging Dynamic Language Features in a Multi-tier Virtual Machine

Anmolpreet Si
b19070@students.iitm
IIT Mandi
India

Meetesh Kalpesh
meeteshmehta4@gr
IIT Bombay
India

Abstract

Multi-tiered virtual-machine (VM) en
In-Time (JIT) compilers are essential fo
language program performance, but co
bugging them is challenging. In this
Derir; a novel tool for tackling this iss
a JIT compiler for R. Derir demystifie
beginners and experts. It allows use
tem's runtime state, make modificatio
textual specializations. With a user-f
visualization features. Derir empowers

Reusing Just-in-Time Compiled Code

MEETESH KALPESH MEHTA, IIT Mandi, India
SEBASTIÁN KRYNSKI, Czech Technical University in Prague, Czechia
HUGO MUSSO GUALANDI, Czech Technical University in Prague, Czechia
MANAS THAKUR, IIT Bombay, India
JAN VITEK, Northeastern University, USA

Most code is executed more than once. If not entire programs then libraries remain unchanged from one run to the next. Just-in-time compilers expend considerable effort gathering insights about code they compiled many times, and often end up generating the same binary over and over again. We explore how to reuse compiled code across runs of different programs to reduce warm-up costs of dynamic languages. We propose to use *speculative contextual dispatch* to select versions of functions from an *off-line curated code repository*. That repository is a persistent database of previously compiled functions indexed by the context under which