Type Inference of Asynchronous Arrows in JavaScript

Eric Fritz   Tian Zhao

University of Wisconsin - Milwaukee

– 0 –
Outline

- JS Asynchronicity

– 0 –
Outline

- JS Asynchronicity
- Promises

– 0 –
Outline

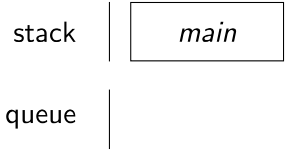
- JS Asynchronicity
- Promises
- Asynchronous Arrows

– 0 –
Outline

- JS Asynchronicity
- Promises
- Asynchronous Arrows
- Type Inference Strategy

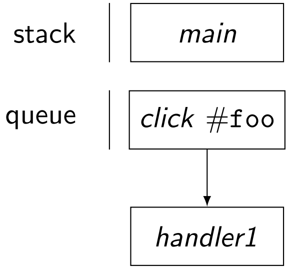
– 1 –
JS Asynchronicity

javascript event loop



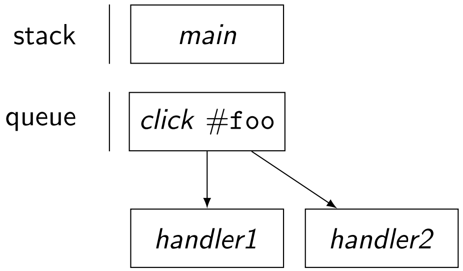
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



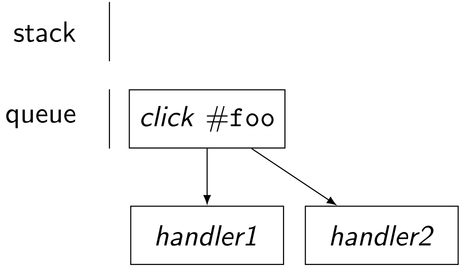
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



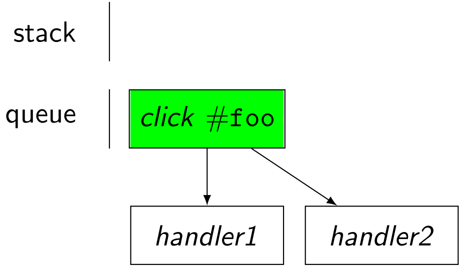
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



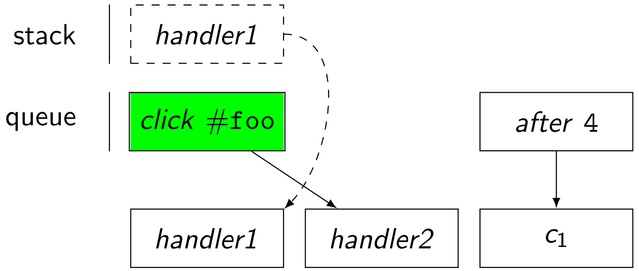
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



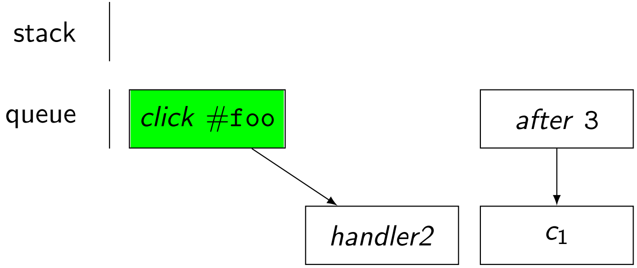
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



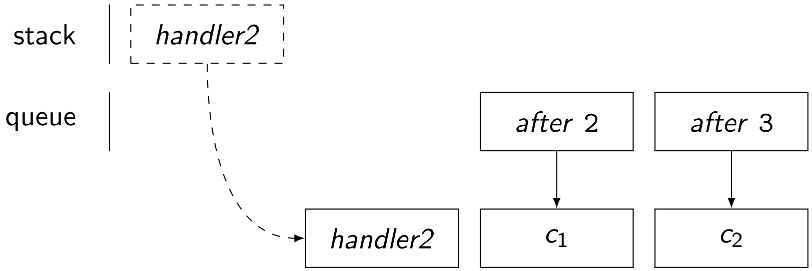
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



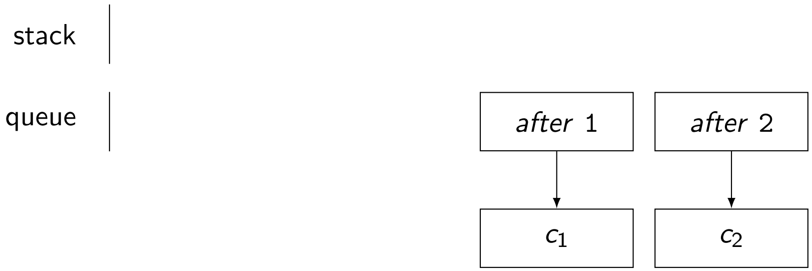
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



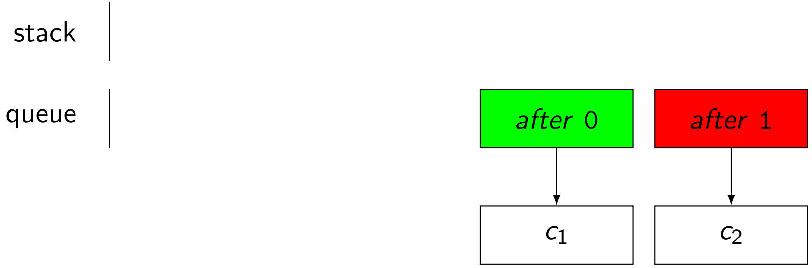
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



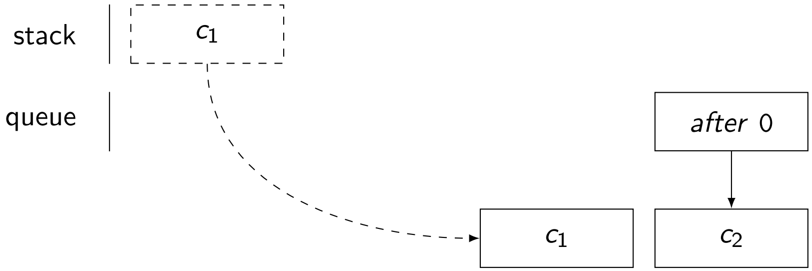
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



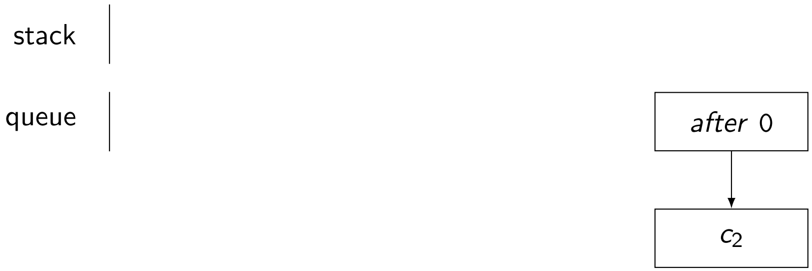
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



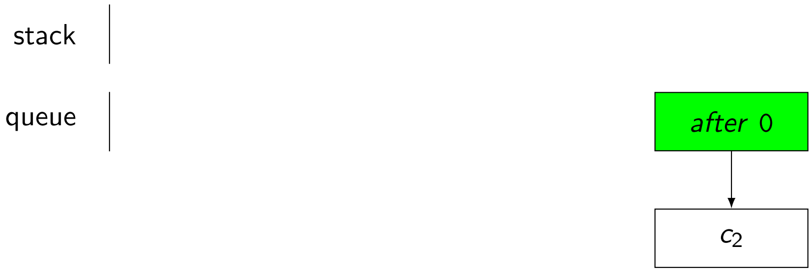
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



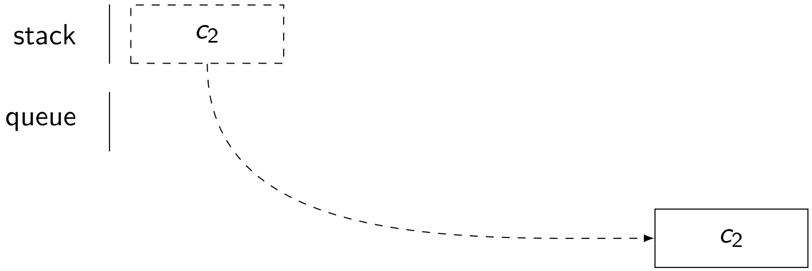
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



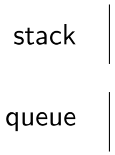
```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

javascript event loop



```
function handler1(ev) {
  setTimeout(() => $(this).css('color', '#f00'), 4000);
}

function handler2(ev) {
  setTimeout(() => $(this).css('color', '#00f'), 3000);
}

$('#foo').one('click', handler1);
$('#foo').one('click', handler2);
```

– 2 –
Promises - A Solution

– 2 –
Promises - A Solution

- Promises/A and A+ (2009)
- kriskowal/q (2010) & jQuery Deferred (2011)
- ECMAScript 6 Native Promises (2015)

Promise (File Parsing)

```
readFile('config.json', function(err, text) {
  if (err) {
    // Handle Read Error
  } else {
    try {
      // Process
    } catch (err) {
      // Handle Parse Error
    }
  }
});
```

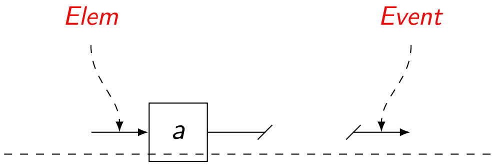
```
readFile('config.json')
  .catch(err => /* Handle Read Error */)
  .then(text => /* Process */)
  .catch(err => /* Handle Parse Error */);
```

– 3 –
Asynchronous Arrows

– 3 –
Asynchronous Arrows

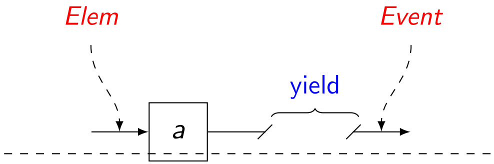
- Generalizing Monads to Arrows (Hughes 2000)
- Arrowlets (Khoo 2009)

arrows



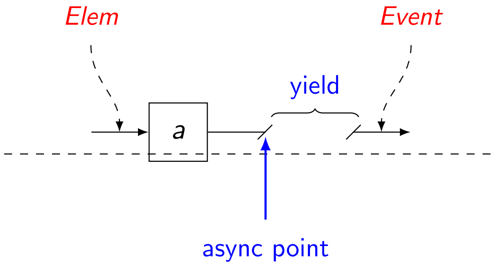
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



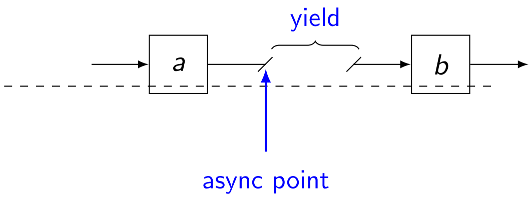
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



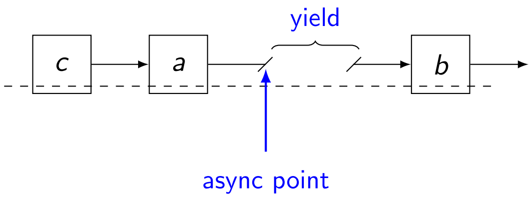
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



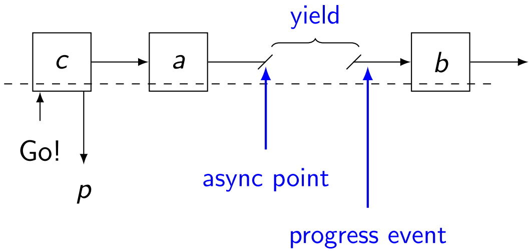
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



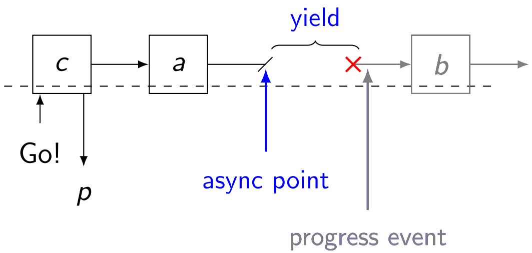
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



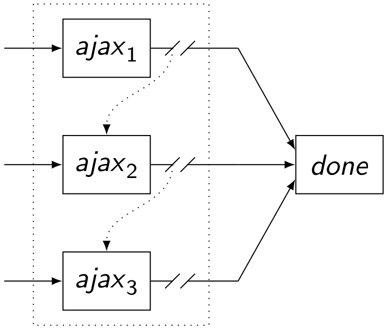
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

arrows



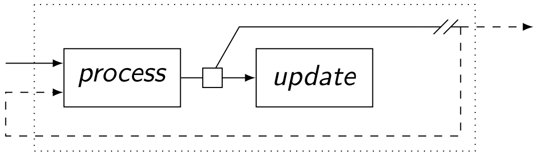
```
let a = new EventArrow('click');
let b = ...
let x = a.seq(b);

let c = new ElemArrow('#enter');
let y = c.seq(x);
let p = y.run();
...
p.cancel();
```

all combinator (parallel, unordered load)



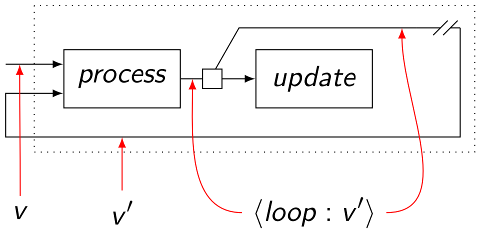
```
let done  = function(data1, data2, data3) { ... }
let ajax1 = new AjaxArrow(() => { url: '/item/1');
let ajax2 = new AjaxArrow(() => { url: '/item/2');
let ajax3 = new AjaxArrow(() => { url: '/item/3');

let loadAll = Arrow.all(ajax1, ajax2, ajax3).seq(done.lift());
```

repeat combinator (chunked array processing)



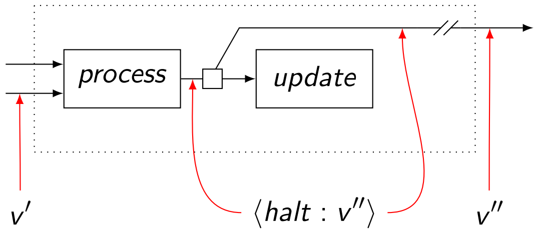
```
function processChunk(array, start) {
  for (let i = start; i < start + 100 && i < array.length; i++) {
    processElem(array[i]);
  }

  return start >= array.length
    ? Arrow.halt()                          // base case
    : Arrow.loop([array, start + 100]); // enable feedback
}

let processArray = processChunk.lift().tap(update).repeat();
```

repeat combinator (chunked array processing)



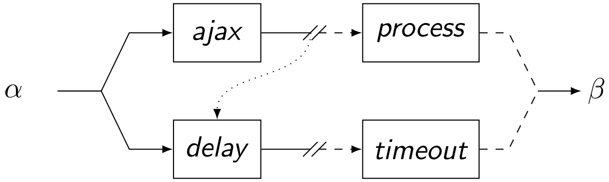
```
function processChunk(array, start) {
  for (let i = start; i < start + 100 && i < array.length; i++) {
    processElem(array[i]);
  }

  return start >= array.length
    ? Arrow.halt()                          // base case
    : Arrow.loop([array, start + 100]); // enable feedback
}

let processArray = processChunk.lift().tap(update).repeat();
```

repeat combinator (chunked array processing)



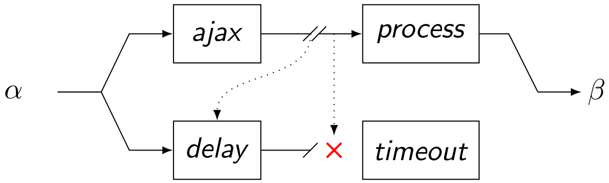
```
function processChunk(array, start) {
  for (let i = start; i < start + 100 && i < array.length; i++) {
    processElem(array[i]);
  }

  return start >= array.length
    ? Arrow.halt()                          // base case
    : Arrow.loop([array, start + 100]); // enable feedback
}

let processArray = processChunk.lift().tap(update).repeat();
```

any combinator (asynchronous timeout)



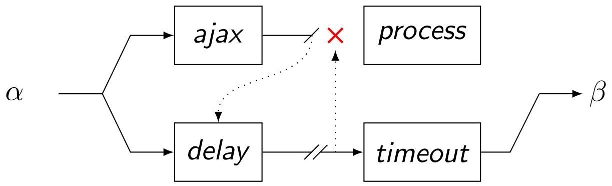
```
let process = ...
let timeout = ...

let fetch = Arrow.any(
  new AjaxArrow(() => { url: '/items' }).seq(process),
  new DelayArrow(30*1000).seq(timeout),
);
```

any combinator (asynchronous timeout)



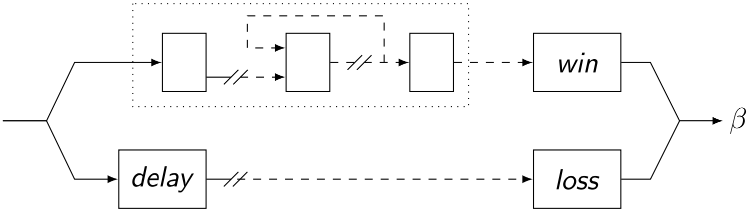
```
let process = ...
let timeout = ...

let fetch = Arrow.any(
  new AjaxArrow(() => { url: '/items' }).seq(process),
  new DelayArrow(30*1000).seq(timeout),
);
```

any combinator (asynchronous timeout)



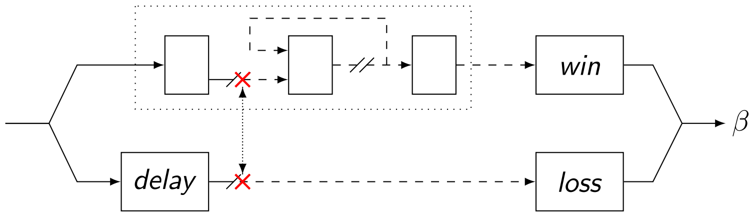
```
let process = ...
let timeout = ...

let fetch = Arrow.any(
  new AjaxArrow(() => { url: '/items' }).seq(process),
  new DelayArrow(30*1000).seq(timeout),
);
```

noemit combinator (complex asynchronous timeout)



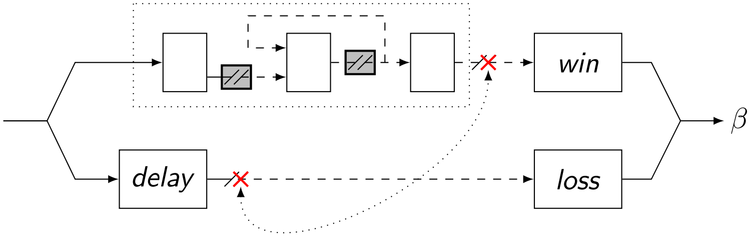
```
let win  = ...
let loss = ...
let play = ...
    play = noemit(play);

let timedGame = Arrow.any(
  play.seq(win),
  new DelayArrow(30*1000).seq(loss),
);
```

noemit combinator (complex asynchronous timeout)



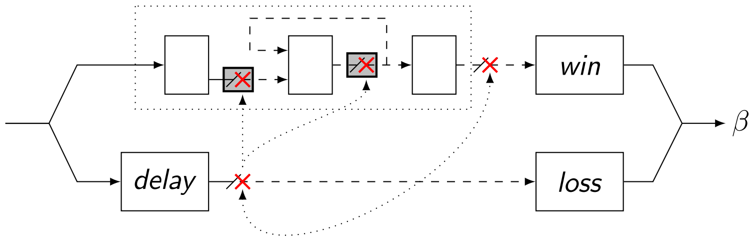
```
let win  = ...
let loss = ...
let play = ...
    play = noemit(play);

let timedGame = Arrow.any(
  play.seq(win),
  new DelayArrow(30*1000).seq(loss),
);
```

noemit combinator (complex asynchronous timeout)



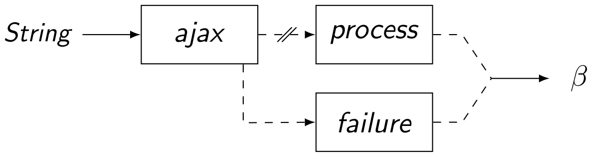
```
let win  = ...
let loss = ...
let play = ...
    play = noemit(play);

let timedGame = Arrow.any(
  play.seq(win),
  new DelayArrow(30*1000).seq(loss),
);
```

noemit combinator (complex asynchronous timeout)



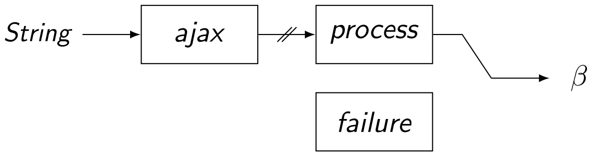
```
let win  = ...
let loss = ...
let play = ...
    play = noemit(play);

let timedGame = Arrow.any(
  play.seq(win),
  new DelayArrow(30*1000).seq(loss),
);
```

try (ajax failure)



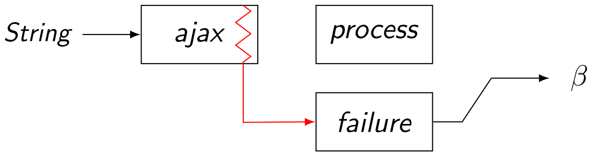
```
let process = ...
let failure = ...

let fetch = Arrow.try(
  new AjaxArrow(url => { 'url': url })
  process,
  failure,
);
```

try (ajax failure)



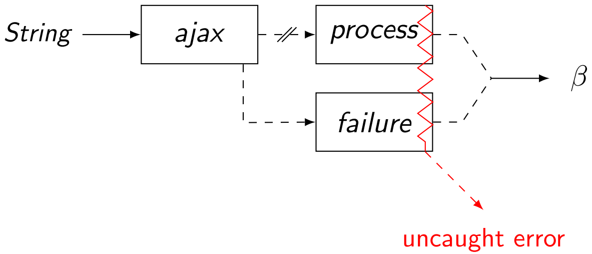
```
let process = ...
let failure = ...

let fetch = Arrow.try(
  new AjaxArrow(url => { 'url': url })
  process,
  failure,
);
```

try (ajax failure)



```
let process = ...
let failure = ...

let fetch = Arrow.try(
  new AjaxArrow(url => { 'url': url })
  process,
  failure,
);
```

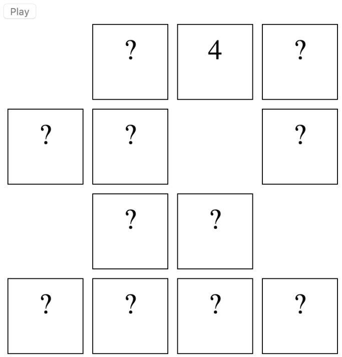
try (ajax failure)



```
let process = ...
let failure = ...

let fetch = Arrow.try(
  new AjaxArrow(url => { 'url': url })
  process,
  failure,
);
```

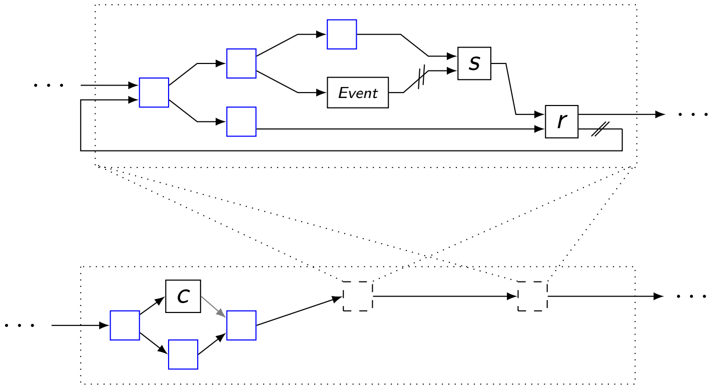
– 4 –
Type Inference Strategy

Memory (Sample Application)



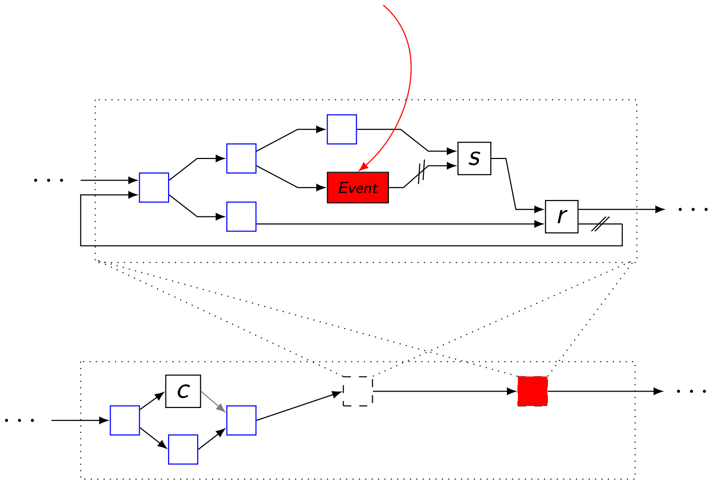
```
var selectOne = select.lift()
  .on('click')
  .whileTrue();

var round = Arrow.id()
  .tap(clear)
  .seq(selectOne)
  .seq(selectOne)
  .seq(validate.lift())
  .carry()
  .wait(500)
  .tap(freeze)
  .wait(500);

var game = Arrow.id()
  .tap(round)
  .seq(cardsLeft.lift())
  .whileTrue();

var play = initialize.lift()
  .wait(1000)
  .seq(game)
  .seq(won.lift());
```

Memory (Topology)

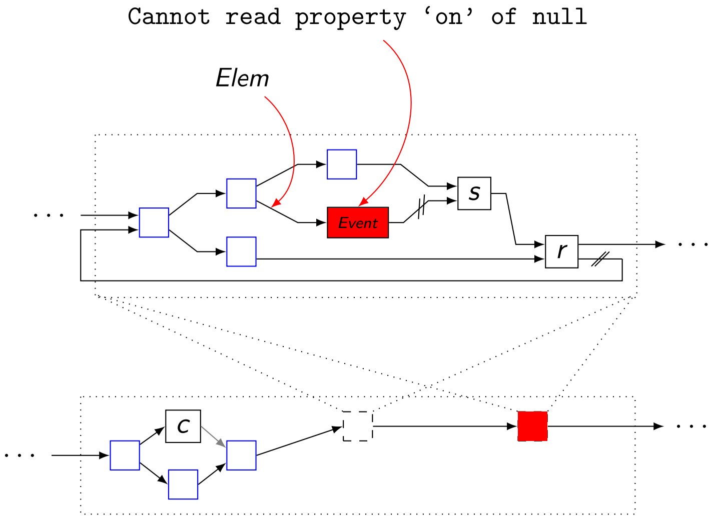


Memory (Topology)

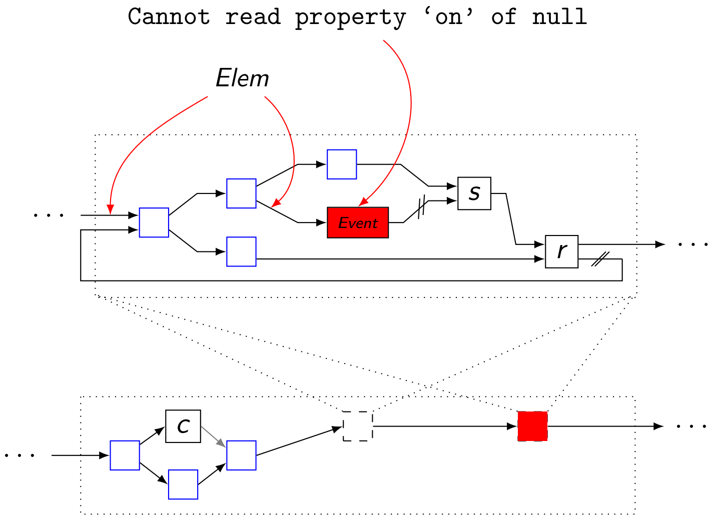
```
Cannot read property 'on' of null
```



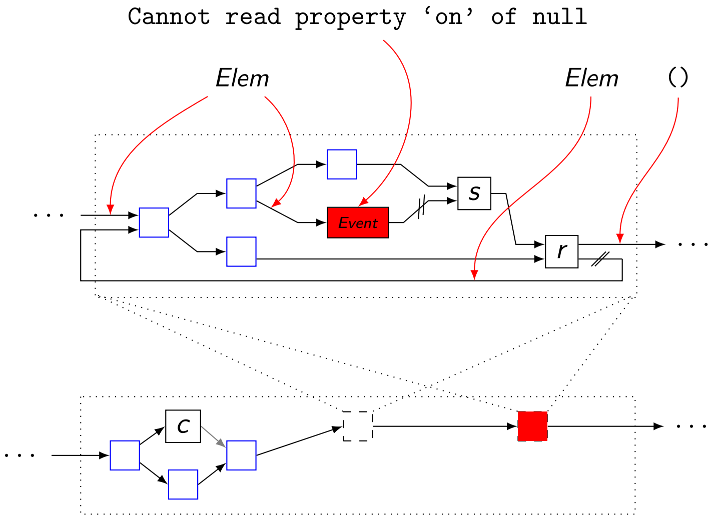
Memory (Topology)



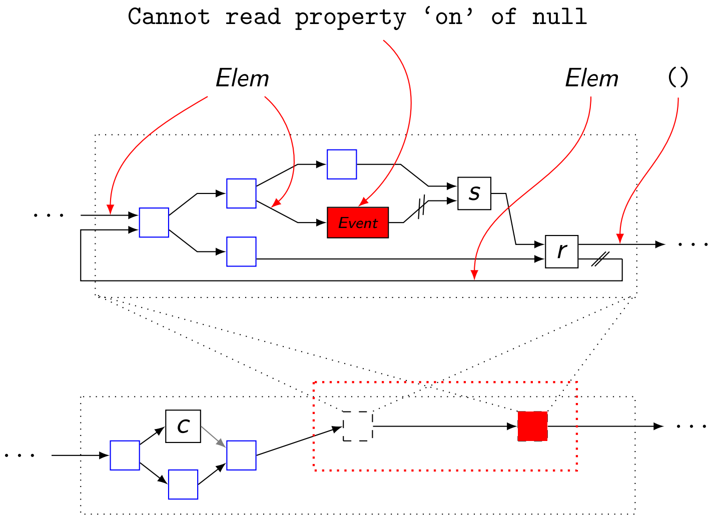
Memory (Topology)



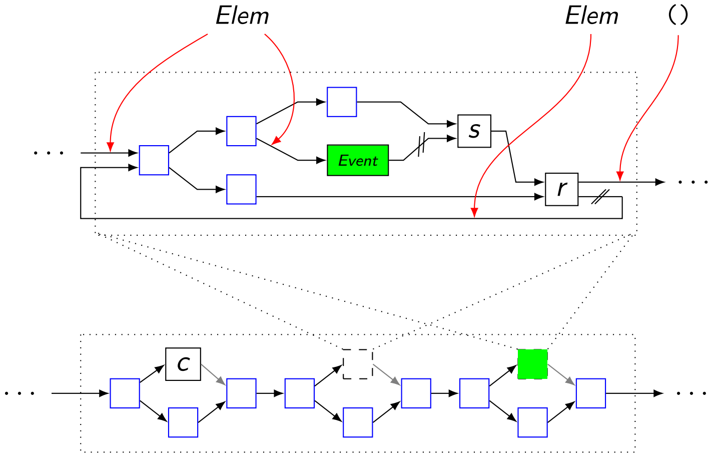
Memory (Topology)



Memory (Topology)



Memory (Topology)



arrow type

$$\widetilde{\tau} ::= \tau_{in} \rightsquigarrow \tau_{out} \setminus (\boxed{C}, \boxed{E})$$

- Set of subtype constraints $\tau \leq \tau$
- Set of possible exceptions types

arrow type

$$\tilde{\tau} ::= \tau_{in} \rightsquigarrow \tau_{out} \setminus (\boxed{C}, \boxed{E})$$

- Set of subtype constraints $\tau \leq \tau$
- Set of possible exceptions types

- Examples
  - $\mathit{Elem} \rightsquigarrow (\mathit{Event}, \mathit{Bool})$
  - $\alpha \rightsquigarrow \beta \setminus (\{\alpha \leq \beta\}, \emptyset)$
  - $\mathit{String} \rightsquigarrow [\mathit{Number}] \setminus (\emptyset, \{\mathit{AjaxError}, \mathit{ValidationError}\})$

combinator type (constraint example)

$$\frac{a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i),\ C_i' = \{\alpha \le \tau_i,\ \tau_i' \le \beta\}}{\texttt{any}(a_1, \ldots, a_n) : \boxed{\alpha} \rightsquigarrow \boxed{\beta} \setminus (\bigcup C_i' \cup \bigcup C_i,\ \boxed{\bigcup E_i})}$$

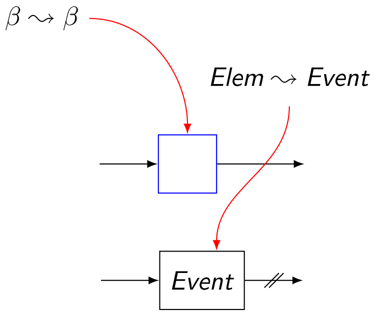
- All arrows receive the same input
- Any arrow may produce the combinator's result
- Any arrow may still throw exception from any arrow

combinator type (exceptions example)

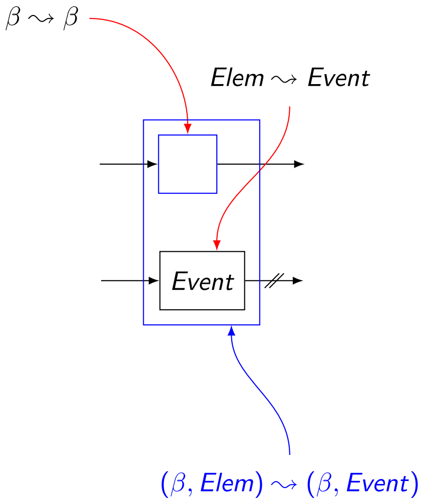
$$\frac{a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C' = \{\tau_1' \leq \tau_2,\ \tau_2' \leq \beta,\ \tau_3' \leq \beta\} \cup \{\boxed{\tau \leq \tau_3 \mid \tau \in E_1}\}}{\texttt{try}(a_1, a_2, a_3) : \tau_1 \rightsquigarrow \boxed{\beta} \setminus (C' \cup \bigcup C_i,\ \boxed{E_2 \cup E_3})}$$

- Exception handler *might* be called
- Exceptions cannot *leak* from a_1
- Exception handler must accept all errors produced by a_1

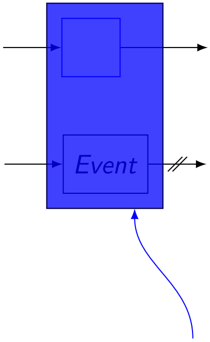
Type Inference



Type Inference

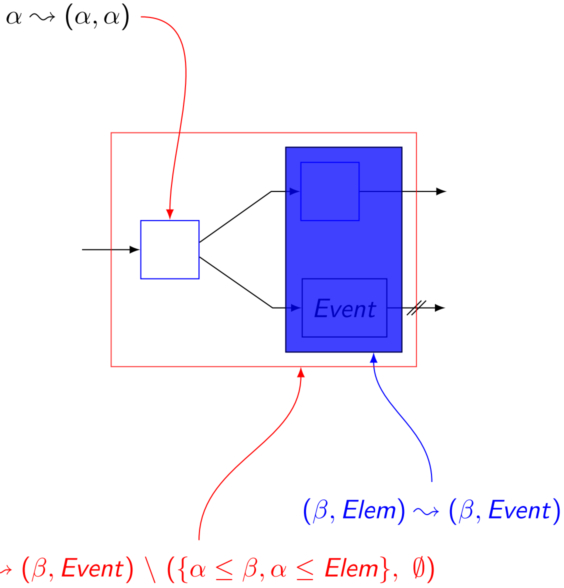

$\beta \rightsquigarrow \beta$
$\mathit{Elem} \rightsquigarrow \mathit{Event}$
Event
$(\beta, \mathit{Elem}) \rightsquigarrow (\beta, \mathit{Event})$

Type Inference



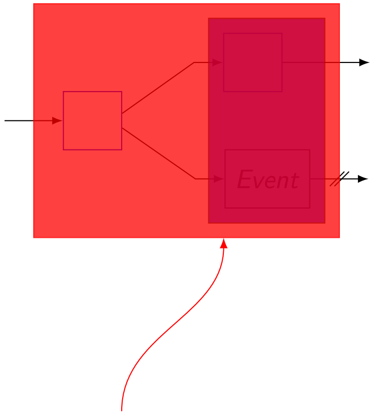
$(\beta, \mathit{Elem}) \rightsquigarrow (\beta, \mathit{Event})$

Type Inference



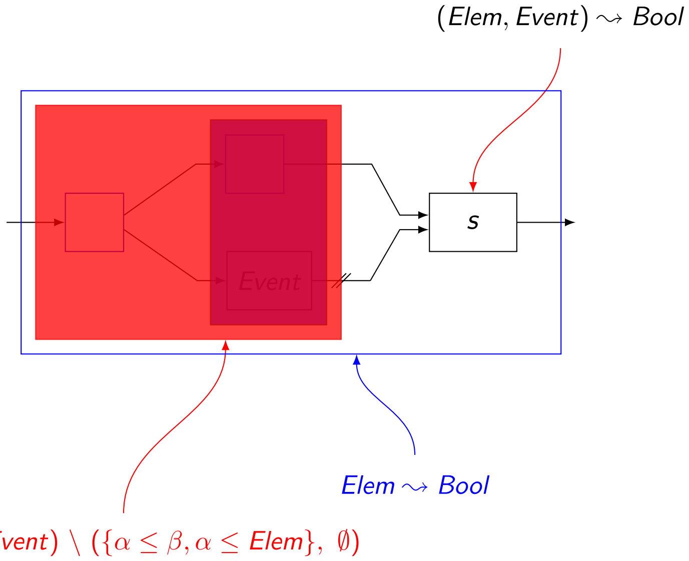
$\alpha \rightsquigarrow (\alpha, \alpha)$
Event
$(\beta, \mathit{Elem}) \rightsquigarrow (\beta, \mathit{Event})$
$\alpha \rightsquigarrow (\beta, \mathit{Event}) \setminus (\{\alpha \leq \beta, \alpha \leq \mathit{Elem}\},\ \emptyset)$

Type Inference



$\alpha \rightsquigarrow (\beta, \textit{Event}) \setminus (\{\alpha \leq \beta, \alpha \leq \textit{Elem}\},\ \emptyset)$

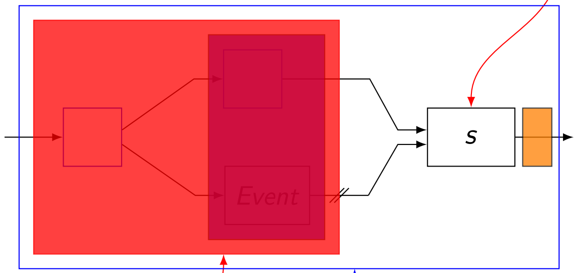
Type Inference



$(\mathit{Elem}, \mathit{Event}) \rightsquigarrow \mathit{Bool}$
s
Event
$\mathit{Elem} \rightsquigarrow \mathit{Bool}$
$\alpha \rightsquigarrow (\beta, \mathit{Event}) \setminus (\{\alpha \leq \beta, \alpha \leq \mathit{Elem}\},\ \emptyset)$

Type Inference

$(\mathit{Elem}, \mathit{Event}) \rightsquigarrow \mathit{Bool}$



$\mathit{Elem} \rightsquigarrow \mathit{Bool}$

$\alpha \rightsquigarrow (\beta, \mathit{Event}) \setminus (\{\alpha \leq \beta, \alpha \leq \mathit{Elem}\},\ \emptyset)$

Common Questions

- Developer Burden
    - Annotations are relatively infrequent

Common Questions

- Developer Burden
  - Annotations are relatively infrequent
- Performance Overhead
  - Tolerable in practice (types are minimized)
  - Type inference is pluggable - can (and should) be disabled

Common Questions

- Developer Burden
  - Annotations are relatively infrequent
- Performance Overhead
  - Tolerable in practice (types are minimized)
  - Type inference is pluggable - can (and should) be disabled
- Why Not Promises?
  - No *time* to type-check Promises

– Thank You –