Refactoring Asynchrony in JavaScript

Keheliya Gallaba
Quinn Hanam, Ali Mesbah, **Ivan Beschastnikh**

McGill

THE UNIVERSITY OF BRITISH COLUMBIA

Why JavaScript?

On the client

@WalmartLabs

ebay

IBM

node

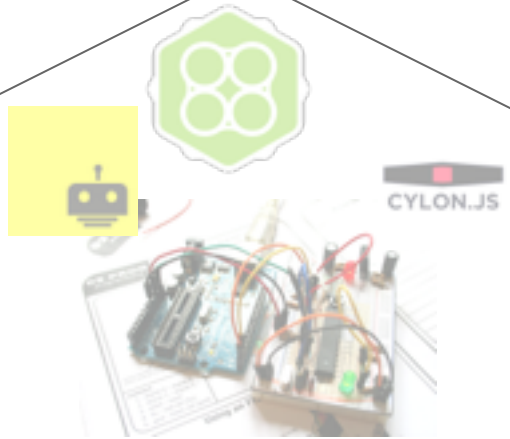
Microsoft

LinkedIn

PayPal

Joyent

redhat



Espruino

TESSEL

Why JavaScript?

On the client

@WalmartLabs
ebay
IBM
node
JS
Microsoft
LinkedIn
PayPal
Joyent

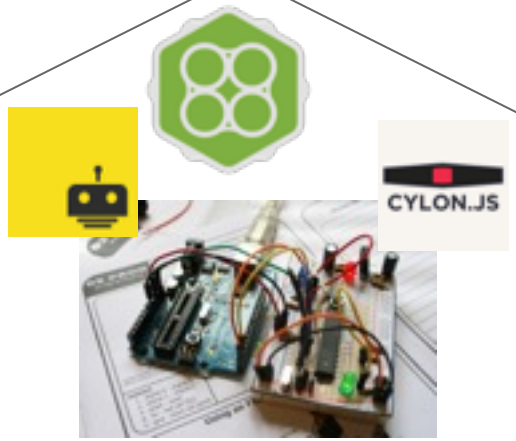
On the server

Why JavaScript?

On the client

@WalmartLabs
ebay
IBM
node
Microsoft
LinkedIn
PayPal
Joyent

On the server



Espruino
TESSEL

Even in hardware!

Why JavaScript?

JavaScript has become the modern lingua franca!

On the client

On the server

Even in hardware!

JavaScript is single-threaded

- Long running tasks / event handling are managed with API calls that use a **callback** to notify the caller of events
  - API call returns immediately
  - Callback invoked **asynchronously**

```
function buttonHandler(event){
    alert("Button Clicked.");
}

$("button").click(buttonHandler);
```

Used for:

- HTTP Request/Response
- File I/O in Node.js
- Mouse click/drag events in the browser

Callbacks make JavaScript responsive!

Our previous callbacks study at ESEM

138 popular **open source** JavaScript subject systems,

from **6** distinct categories,

with both **Client-side** & **Server-side** code.

Category	Subject systems	Client side	Server side	Total files	Total LOC
NPM Modules	86		✓	4,983	1,228,271
Web Apps.	16	✓	✓	1,779	494,621
Game Engines	16	✓	✓	1,740	1,726,122
Frameworks	8	✓		2,374	711,172
DataViz Libs.	6	✓		3,731	958,983
Games	6	✓		347	119,279
Total	**138**	✓	✓	**14,954**	**5,238,448**

Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript.
Gallaba, Mesbah, Beschastnikh. ESEM 2015

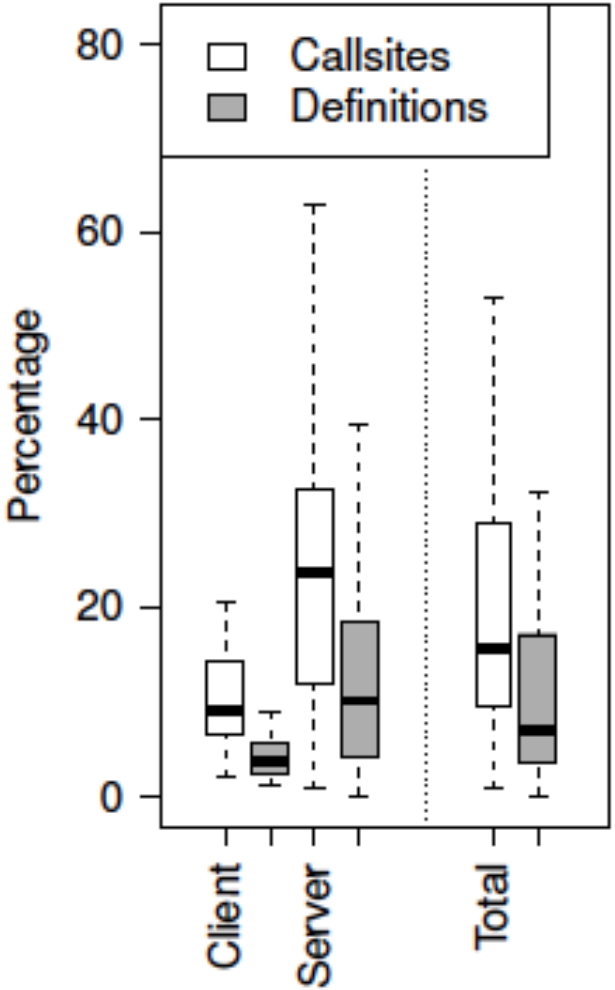
Callback prevalence

Callback-accepting function definitions

- On average, **10%** of all function definitions take callback arguments.
- They are more prevalent in server-side code **(10%)** than in client- side code **(4.5%)**.

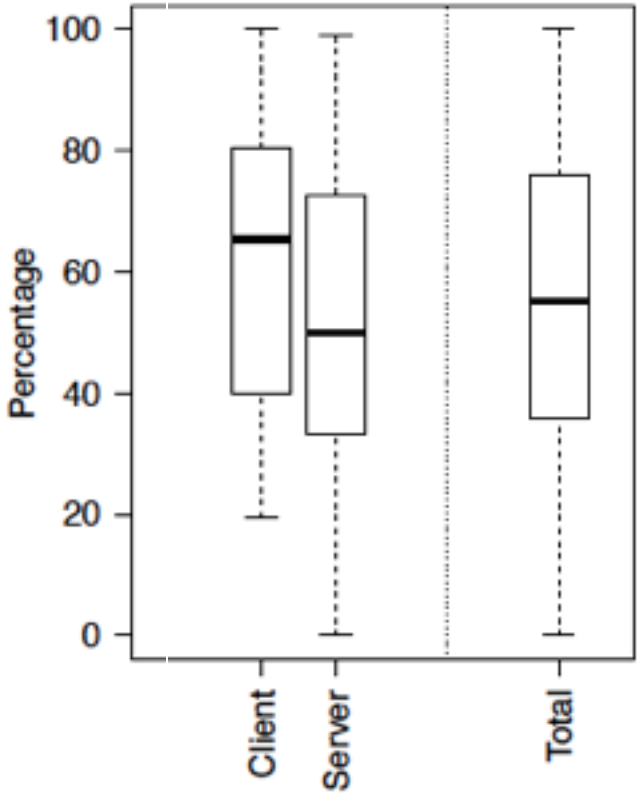
Callback-accepting function callsites

- **19%** of all function callsites take callback arguments.
- Callback-accepting function call-sites are more prevalent in server-side code **(24%)** than in client-side code **(9%)**.



Asynchronous Callbacks – Results

- More than half **(56%)** of all callbacks are Asynchronous.
- Asynchronous callbacks, on average, appear more frequently in client-side code **(72%)** than in server-side code **(55%)**.



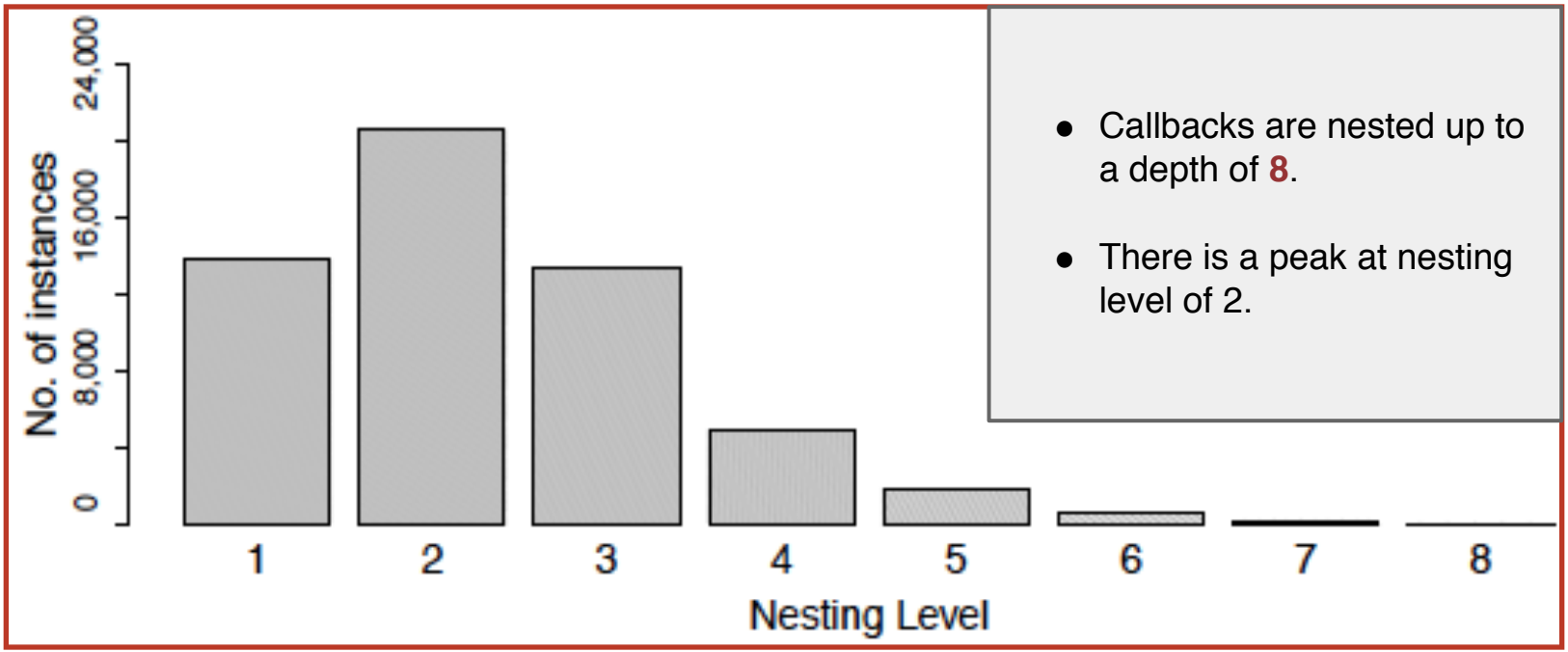
Callback nesting

Nesting Level

```
1  define('admin/general/dashboard', 'semver', function(semver
       ) {
2    var Admin = {};

4    $('#logout-link').on('click', function() {
5      $.post(RELATIVE_PATH + '/logout', function() {
6        window.location.href = RELATIVE_PATH + '/';
7      });
8    });

10   ...

12   $('.restart').on('click', function() {
13     bootbox.confirm('Are you sure you wish to restart
           NodeBB?', function(confirm) {
14       if (confirm) {
15         $(window).one('action:reconnected', function() {
16           app.alert({ alert_id: 'instance_restart', });
17         });

19         socket.emit('admin.restart');
20       }
21     });
22   });
23   return Admin;
24 });
```

Notorious for:
Callback hell aka Pyramid of Doom

Callback nesting



- Callbacks are nested up to a depth of **8**.
- There is a peak at nesting level of 2.

Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has a **convention**:

Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has a **convention**:

Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Error-first Protocol

We found
- **20%** of function definitions follow the error-first protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has a **convention**:

Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Async callbacks pain points

- Nesting degrades readability
- Asynchrony and nesting complicates control flow
- Informal error handling idiom of error-first protocol provides no guarantees
- Existing error-handling try/catch mech do not apply
- No at-most-once semantics for callback invocation

How can we help with this?

Promises to the rescue

Promises: a native language feature for solving the Asynchronous composition problem.

Without Promises

```
getUser('mjackson', function (error, user) {
  if (error) {
    handleError(error);
  } else {
    getNewTweets(user, function (error, tweets) {
      if (error) {
        handleError(error);
      } else {
        updateTimeline(tweets, function (error) {
          if (error) handleError(error);
        });
      }
    });
  }
});
```

With Promises

```
getUser('mjackson')
        .then(getNewTweets,null)
        .then(updateTimeline)
        .catch(handleError);
```

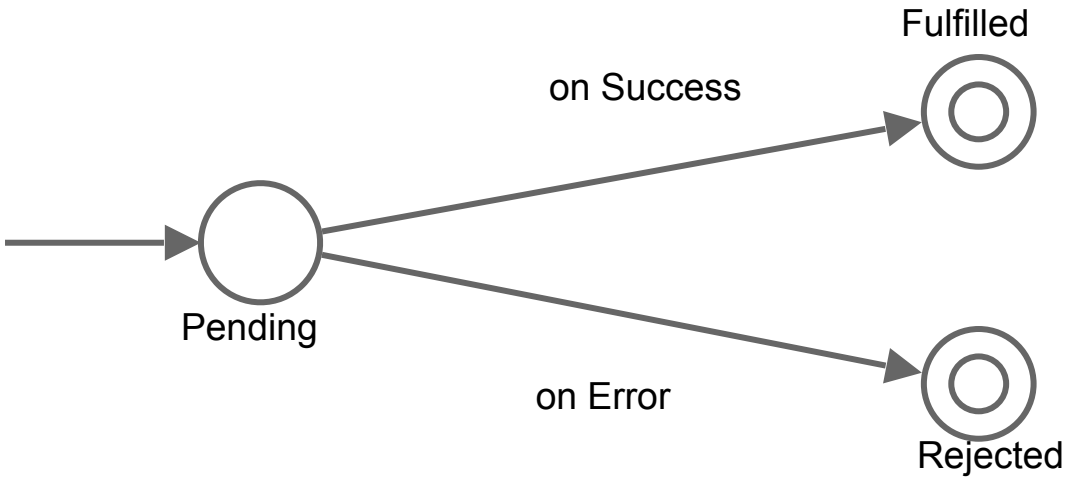
Promises as an alternative

```
Promise.then(fulfilledHandler, errorHandler)
```

Called when the promise is fulfilled

Called when a promise fails

A promise can be in one of three states:



Developers want to refactor Callbacks to Promises

- Finding Issues Related to Callbacks and Promises on Github 
- Search Query Used:

```
promise callback language:JavaScript
stars:>30 comments:>5 type:issue
```

- Results: 4,342 Issues
- Some Comments:

> *We've recently converted pretty large internal codebases from async.js to promises and the code became **smaller**, more **declarative**, and **cleaner** at the same time.*

> *They're fairly **simple** and **lightweight**, but they make it **easy** to build higher level async constructs.*

> *Personally I'm very pleased with the amount of additional **safety** and **expressiveness** I've gained by using promises.*

Do they refactor Callbacks to Promises?

- Finding Refactoring Pull Requests on Github
- Search Query Used:

```
Refactor promises language:Javascript
stars:>20 type:pr
```

- Results: 451 pull requests

> “It is not a question of whether doing it or not, but when!”

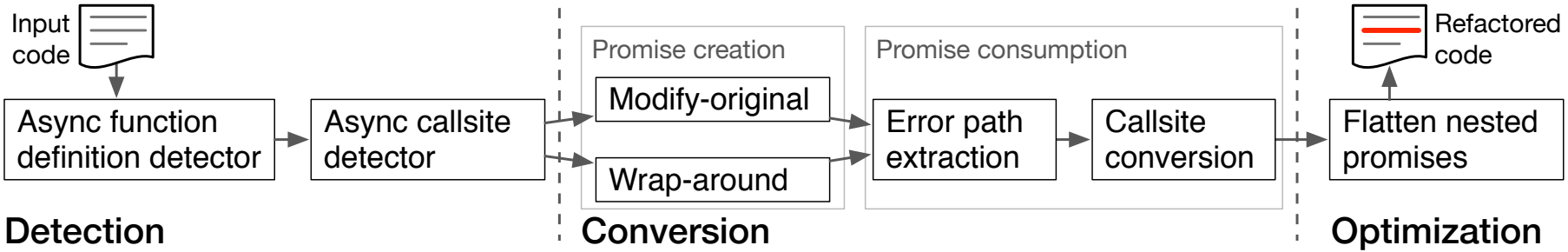
- Common style of refactoring is project-independent and amenable to automation

> But no mention or use of refactoring tools for this!

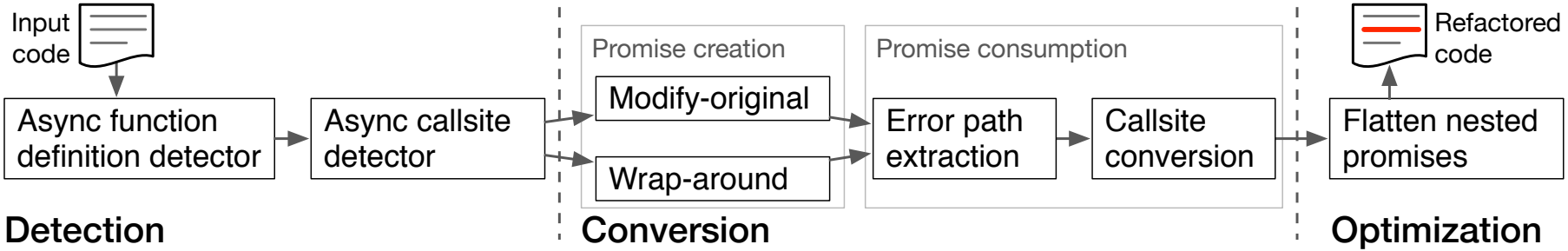
Goal and contribution of this work

Automated refactoring of asynchronous JavaScript callbacks into Promises

PromisesLand design



PromisesLand design



- Esprima for AST construction
- Estraverse to traverse the AST
- Escope for scope analysis
- Hackett and Guo points-to analysis + type inference

Many analyses are unsound/approximations (good enough in practice!)

Design - Detecting Functions with Asynchronous Callbacks

An example refactoring candidate

```
function f(cb) {
        .
        .
        .
    async(function cb_async(data) {
      if(error) cb(null, data);
      else cb(error, null);
    });
});
```

Function definition **f** is a candidate for refactoring if all of following is true.

- Accepts a callback **cb**
- **cb** is invoked inside a known Async API inside **f**
- **cb** is not invoked outside the Async API
- **f** does not have a return value

Some known **async** APIs

Category	Examples	Availability
DOM events	`addEventListener, onclick`	Browser
Network calls	`XMLHTTPRequest.open`	Browser
Timers (macro-Task)	`setImmediate(), setTimeout(), setInterval()`	Browser, Node.js
Timers (micro-task)	`process.nextTick()`	Node.js
I/O	APIs of `fs, net`	Node.js

(we use a whitelist of these)

Design - Transformation

```
function f(cb) {
                .
                .
                .
    async(function cb_async(data) {
      if(error) cb(null, data);
      else cb(error, null);
    });
}

function cb(error, data) {
  if(error) {
      // Handle error
  } else {
      // Handle data
  }
}

f(cb)
```

Refactoring candidate

Design - Transformation

```
function f(cb) {
                .
                .
                .
    async(function cb_async(data) {
      if(error) cb(null, data);
      else cb(error, null);
    });
}

function cb(error, data) {
  if(error) {
      // Handle error
  } else {
      // Handle data
  }
}

f(cb)
```

Refactoring candidate

```
function f() {
 return new Promise(function (resolve, reject){
   async(function cb_async(data}) {
     if(error) reject(null, data);
     else resolve(error, null);
  });
});
```

Modify Original

- Produces code similar to how developers would refactor
- Refactors only some instances

Design - Transformation

```
function f(cb) {
                .
                .
                .
    async(function cb_async(data) {
      if(error) cb(null, data);
      else cb(error, null);
    });
}

function cb(error, data) {
  if(error) {
      // Handle error
  } else {
      // Handle data
  }
}

f(cb)
```

Refactoring candidate

```
function f() {
 return new Promise(function (resolve, reject){
   async(function cb_async(data}) {
     if(error) reject(null, data);
     else resolve(error, null);
  });
});
```

OR

```
function f_new() {
  return new Promise(function (resolve, reject) {
    f(function(err,data){
      if(err !== null)
        return reject(err);
      resolve(data);
    });
  });
}
```

Modify Original

- Produces code similar to how developers would refactor
- Refactors only some instances

Wrap-around

- Transforms most instances
- Produces code that can be more complex than the original
- Good if Async is in libraries

Design - Transformation

```
function f(cb) {
                .
                .
                .
    async(function cb_async(data) {
      if(error) cb(null, data);
      else cb(error, null);
    });
}

function cb(error, data) {
  if(error) {
     // Handle error
  } else {
     // Handle data
  }
}

f(cb)
```

reject

resolve

Refactoring candidate

```
function f() {
 return new Promise(function (resolve, reject){
   async(function cb_async(data}) {
     if(error) reject(null, data);
     else resolve(error, null);
  });
});
```

OR

```
function f_new() {
  return new Promise(function (resolve, reject) {
    f(function(err,data){
      if(err !== null)
        return reject(err);
      resolve(data);
    });
  });
}
```

Modify Original

- Produces code similar to how developers would refactor
- Refactors only some instances

Wrap-around

- Transforms most instances
- Produces code that can be more complex than the original
- Good if Async is in libraries

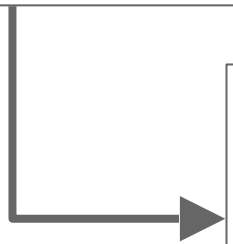
Design - Transforming the Call Site

```
f(cb);
```

```
f().then(onSuccess, onError);
```

Design - Flattening Promise Consumers

```
getLocationDataNew("jackson").then(function (details) {
  getLongLatNew(details.address, details.country).then(function (longLat) {
    getNearbyATMsNew(longLat).then(function (atms) {
      console.log('Your nearest ATM is: ' + atms[[0]]);
    });
  });
});
```



```
getLocationDataNew("jackson").then(function (details) {
  return getLongLatNew(details.address, details.country);
}).then(function (longLat) {
  return getNearbyATMsNew(longLat);
}).then(function (atms) {
  return console.log('Your nearest ATM is: ' + atms[[0]]);
});
```

An example modify-original refactoring

```
- function addTranslations(translations, call){
    translations = JSON.parse(translations);
    fs.readdir(dirname + '/../client/src/
        translations/',
              function (err, pofiles) {
      if (err) {
-       return callback(err);
      }
      var vars = [[]];
      pofiles.forEach(function (file) {
        var loc = file.slice(0, -3);
        if ((file.slice(-3) === '.po') && (loc !==
            'template')) {
          vars.push({tag: loc, language:
              translations[[loc]]});
        }
      });
-     return callback(vars);
    });
  }
-  addTranslations(trans, jobComplete);
```

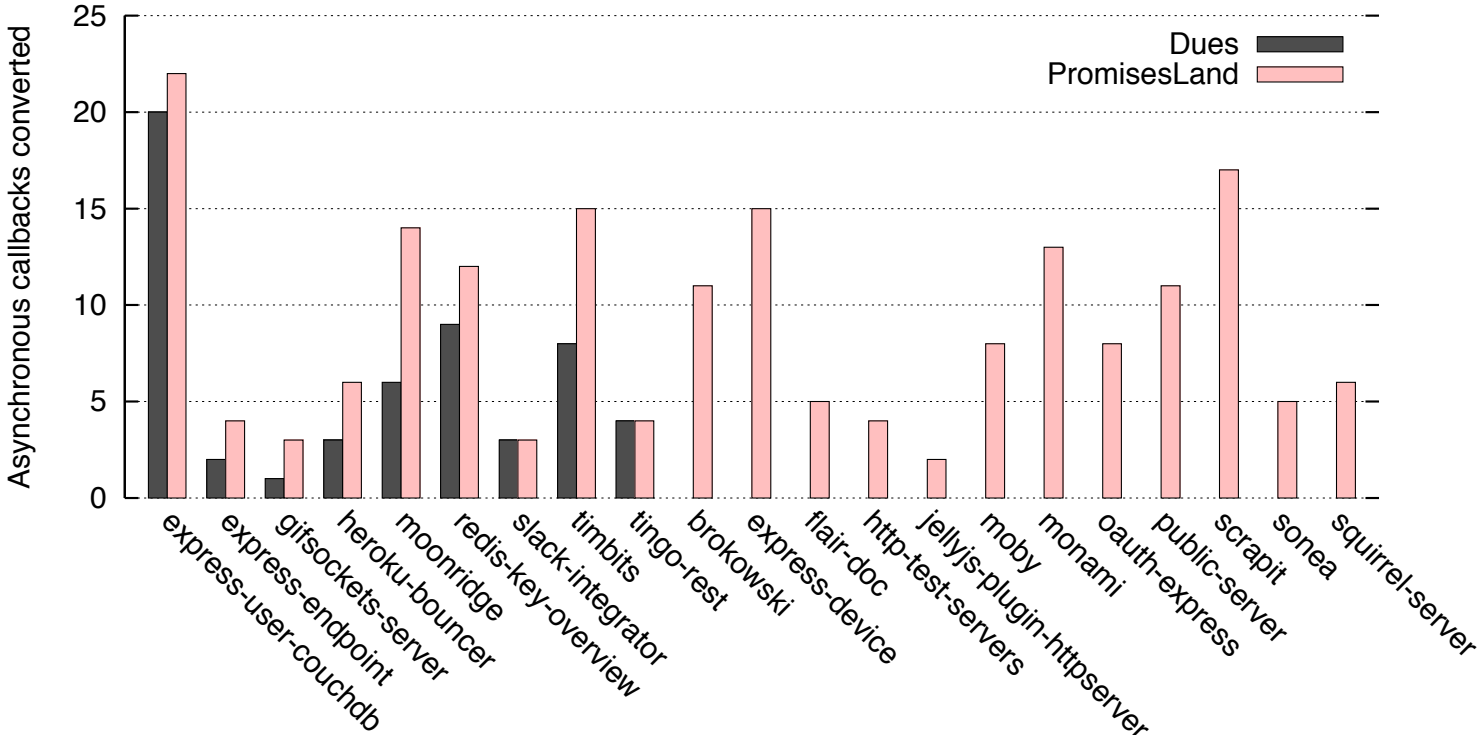
Listing 7. An example of an asynchronous callback before refactoring to promises – from KiwiIRC #581.

```
+ function addTranslations(translations){
+   return new Promise(function (resolve, reject){
      translations = JSON.parse(translations);
      fs.readdir(dirname + '/../client/src/
          translations/',
                function (err, pofiles) {
        if (err) {
+         return reject(err);
        }
        var vars = [[]];
        pofiles.forEach(function (file) {
          var loc = file.slice(0, -3);
          if ((file.slice(-3) === '.po') && (loc
              !== 'template')) {
            vars.push({tag: loc, language:
                translations[[loc]]});
          }
        });
+       return resolve(vars);
      });
+   });
  }
+ addTranslations(trans).then(jobComplete);
```

Listing 8. An example of an asynchronous callback after refactoring to promises using *Modify-original* strategy.

Comparison again prior work

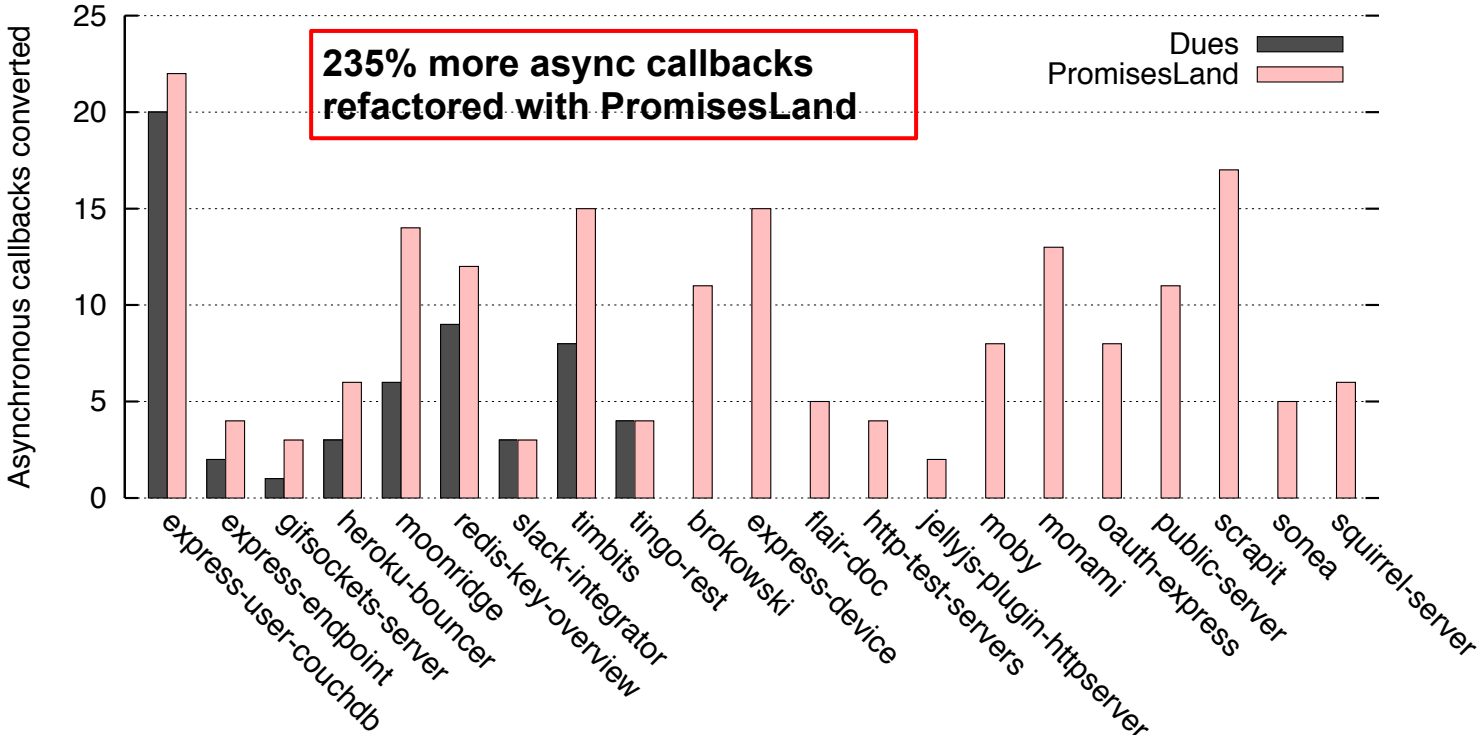
- Subject Systems: 21 NPM Modules
- Compared against Dues prior work by Brodu et al.



Comparison again prior work

- Subject Systems: 21 NPM Modules
- Compared against Dues prior work by Brodu et al.

Instances converted with each strategy:
- Modify-original: 73
- Wrap-around: 115



Evaluation - Detection Accuracy

Subject System	LOC (JS)	Detected Instances	Refactored Instances	Precision (%)	Recall (%)
heroku-bouncer	947	7	6	100	85.7
moonridge	3,760	19	14	100	73.6
timbits	1,226	17	15	100	88.2
tingo-rest	238	4	4	100	100
Total	6,171	39	47	100 (avg)	82.9 (avg)

Precision:

$$\frac{\text{asynchronous callbacks among the detected refactoring candidates}}{\text{refactoring candidates that tool detects}}$$

Recall:

$$\frac{\text{asynchronous callbacks that tool detects}}{\text{asynchronous callbacks that exist in the subject system}}$$

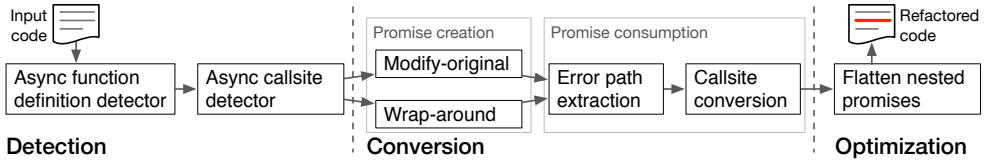
Evaluation - Performance

Time taken at each phase per instance in seconds

Phase	Min	Max	Mean	Median
Async Function Detection	0.12	1.00	0.51	0.50
Promise Creation Conversion	0.10	0.49	0.29	0.29
Promise Consumption Conversion	0.11	0.47	0.27	0.30
Optimization and Re-writing	0.14	0.95	0.61	0.58
All Phases	0.97	2.57	1.69	1.64

All subject systems were refactored in under 3 seconds.

Conclusion



- Callbacks ubiquitous in JavaScript
- Async callbacks challenging: readability, complex control flow, error handling..
- **Our contribution:**
  - PromisesLand to refactor async callbacks to promises
  - Runs in < 3 seconds on large applications
  - Refactors 235% more callbacks than prior work

http://salt.ece.ubc.ca/software/promisland