Scope

Deian Stefan

(adopted from my & Edward Yang’s CSE242 slides)
Substitution model

• Way of giving semantics to the \( \lambda \)-calculus
  
  ➤ E.g., \((\lambda x. f \times x) (\lambda y. z) \rightarrow_\beta f (\lambda y. z) (\lambda y. z)\)

• Translate this knowledge to JavaScript functions
  
  ➤ \((x => f(x)(x)) (y => z) \rightarrow_\beta f(y => z)(y => z)\)
Substitution model

• Why would you, in practice, not really want to do function application in this way for a language like JavaScript?
  ➤ It’s super slow! Why?
  ➤ It’s actually nonsensical sometimes! When?
Substitution gone wrong

• Consider variable mutation in JavaScript:

```javascript
let y = 1;
let z = 0;                  ...
z++;                    →β. 0++;
console.log(z);          ...
```

• There is nothing wrong with substitution per say

  ➢ It’s symbolic evaluation/computation

  ➢ Problem is JavaScript has mutation and not amendable to symbolic evaluation
What can we do?

\( \lambda \)-calculus

environment model

machine model
The environment model (by example)

• Anatomy of a scope
• First-order functions
• Free variables
• High-order functions (bonus)
Anatomy of a scope

• What’s the point of a scope (e.g., block scope)?
Anatomy of a scope

• Recall our previous example:

```javascript
let y = 1;
let z = 0;
z++;
console.log(z);
```

• In this model, we associate an environment (activation record) with the code we’re executing

➤ Environment contains entries of all variables in scope

➤ Environment/stack ptr: points to cur activation record
Anatomy of a scope

• Recall our previous example:

```javascript
let y = 1;
let z = 0;
z++;
console.log(z);
```

<table>
<thead>
<tr>
<th>Environment ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

• In this model, we associate an environment (activation record) with the code we’re executing

➤ Environment contains entries of all variables in scope

➤ Environment/stack ptr: points to cur activation record
Anatomy of a scope

• In the environment model, we can distinguish between values and locations
  ➤ r-values: plain old values; we can reason about them using substitution semantics
  ➤ l-values: refer to locations where r-values are stored; they persist beyond single expressions.

• Why is this important?
  ➤ It tells us the kind of values operators like ++ must take. A: r-values.  B: l-values
Anatomy of a scope

• What’s the process for executing \( z++ \):

```javascript
let y = 1;
let z = 0;
z++;
console.log(z);
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
</tr>
</tbody>
</table>

• Algorithm:
  - Find the current environment
  - Check to see if variable being reference is in env: if so, mutate!
Anatomy of a scope

• What’s the process for executing `console.log(z)`

```javascript
let y = 1;
let z = 0;
z++; console.log(z);
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>y</strong></td>
<td></td>
</tr>
<tr>
<td><strong>z</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

• Algorithm:

➤ Find the current environment

➤ Check to see if variable being reference is in env: if so, read it!
Anatomy of a scope

• This sounds slow!
  ➤ It is!
  ➤ But remember: this is not the machine model, this is still an abstract model!

• Not too far off from machine model
  ➤ In x86, you dereference %esp to figure out where stack is and use offset to that location
  ➤ In JavaScript, you often do table lookup to find location of variables
The environment model (by example)

• Anatomy of a scope ✓
• First-order functions
• Free variables
• High-order functions (bonus)
First-order functions

- Consider activation record when calling function:

  ```javascript
  function fact(n) {
    if (n <= 1) {
      return 1;
    } else {
      return n * fact(n-1);
    }
  }
  fact(3);
  ```

- What else do we need to keep track of?
First-order functions

Consider activation record when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}

fact(3);
```

What else do we need to keep track of?
More bookkeeping

- The parts of an activation record when calling function
  - **control link**: records where to switch the environment pointer to when we finish evaluating in this scope.
  - Do we need this for block scopes too? A: yes, B: no
  - **return value**: l-value where the return value of function should be stored
  - **parameters**: l-value for each formal parameter
  - **local variables**: l-values for each let+const declaration
More bookkeeping

• Anything else?

  ➤ Yes! Typically activation records will store the return address where to resume ode execution — we’ll talk about this in the control flow lecture
Let’s look at how evaluation works

• Consider activation records when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return n * fact(n-1);
    }
}
fact(3);
```

• Do we keep the activation records on the stack after evaluation?
  A: yes, B: no
Let’s look at how evaluation works

- Consider activation records when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
fact(3);
```

- Do we keep the activation records on the stack after evaluation?
  A: yes, B: no
Let’s look at how evaluation works

- Consider activation records when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}

fact(3);
```

- Do we keep the activation records on the stack after evaluation?
  A: yes, B: no
Let’s look at how evaluation works

• Consider activation records when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}

fact(3);
```

• Do we keep the activation records on the stack after evaluation?
A: yes, B: no
Let’s look at how evaluation works

• Consider activation records when calling function:

```javascript
function fact(n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
fact(3);
```

• Do we keep the activation records on the stack after evaluation?
  A: yes, B: no
Let’s look at how evaluation works

• Consider activation records when calling function:

```javascript
function fact(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}

fact(3);
```

• Do we keep the activation records on the stack after evaluation?
  A: yes, B: no
The environment model *(by example)*

• Anatomy of a scope ✓
• First-order functions ✓
• Free variables
• High-order functions (bonus)
Free variables

• Consider activation records when calling f:

```javascript
let x = 1;
function f() {
  console.log(x)
}
f();
```

• Should we lookup x via the control link?
  ➤ A: yes
  ➤ B: no
Free variables

- Consider activation records when calling g:

```javascript
let x = 1;
function f() {
    console.log(x)
}

function g() {
    let x = 2;
    f();
}

g();
```

- What happens when we follow the control link?
Congrats, you did it!

You invented dynamic scoping!
How do we “fix” this?

• We need more bookkeeping!
  ➤ access link: reference to activation record of closest enclosing lexical scope

• Modify our lookup algorithm:
  ➤ Find the current environment
  ➤ Check to see if variable being reference is in env
  ➤ If not, follow the access link and repeat
Consider activation records when calling `g`:

```javascript
let x = 1;
function f() {
    console.log(x)
}

function g() {
    let x = 2;
    f();
}

g();
```
Consider activation records when calling `g`:

```javascript
let x = 1;
function f() {
  console.log(x)
}

function g() {
  let x = 2;
  f();
}

g();
```
Consider activation records when calling \texttt{g}:

```javascript
let x = 1;
function f() {
  console.log(x)
}

function g() {
  let x = 2;
  f();
}

g();
```
Wait, there is some magic here

- How do we know how to wire up the access links?

```javascript
let x = 1;
function f() {
    console.log(x)
}

function g() {
    let x = 2;
    f();
}

g();
```
Functions are data!

The act of defining a function should include the act of recording the access link associated with the function.
Treating functions as data

• Let’s look at the example again, with minor rewrite

```javascript
let x = 1;
let f = () => {
  console.log(x)
}

let g = () => {
  let x = 2;
  f();
}

g();
```

• Function as data = closures = (current env ptr, code pointer)
Treating functions as data

• Let’s look at the example again, with minor rewrite

```javascript
let x = 1;
let f = () => {
    console.log(x)
}

let g = () => {
    let x = 2;
    f();
}

g();
```

• Function as data = closures = (current env ptr, code pointer)
Treating functions as data

• Let’s look at the example again, with minor rewrite

```javascript
let x = 1;
let f = () => {
    console.log(x)
}

let g = () => {
    let x = 2;
    f();
}

g();
```

• Function as data = closures = (current env ptr, code pointer)

```
<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
</tbody>
</table>
```

environment ptr
Treating functions as data

- Let’s look at the example again, with minor rewrite

```javascript
let x = 1;
let f = () => {
  console.log(x)
}

let g = () => {
  let x = 2;
  f();
}

g();
```

- Function as data = closures = (current env ptr, code pointer)
Treating functions as data

• When we evaluate function, the access link is set to the pointer in the closure

```javascript
let x = 1;
let f = () => {
    console.log(x)
}

let g = () => {
    let x = 2;
    f();
}

g();
```
Treating functions as data

• When we evaluate function, the access link is set to the pointer in the closure

```javascript
let x = 1;
let f = () => {
    console.log(x)
}

let g = () => {
    let x = 2;
    f();
}

g();
```
Treating functions as data

• When we evaluate function, the access link is set to the pointer in the closure

```javascript
let x = 1;
let f = () => {
    console.log(x)
}

let g = () => {
    let x = 2;
    f();
}

f();
g();
```
Treating functions as data

- When we evaluate function, the access link is set to the pointer in the closure

```javascript
let x = 1;
let f = () => {
    console.log(x) // 1
}

let g = () => {
    let x = 2;
    f();
}

g();
```
Closures

environment → code
The environment model (by example)

• Anatomy of a scope ✓
• First-order functions ✓
• Free variables ✓
• High-order functions (bonus)
Higher-order functions

- Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
  return () => {
    return c++;
  };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```
Higher-order functions

Consider the use of high-order \texttt{mkCounter} function

```javascript
function mkCounter(c) {
    return () => {
        return c++;
    };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```

- env
- x
- y
- control
- access
- c 0

```
Higher-order functions

• Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
  return () => {
    return c++;
  };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```

```
<table>
<thead>
<tr>
<th>env</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>control</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```
Higher-order functions

- Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
    return () => {
        return c++;
    };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```
Higher-order functions

• Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
    return () => {
        return c++;
    };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```
Higher-order functions

- Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
    return () => {
        return c++;
    };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```

```
<table>
<thead>
<tr>
<th>control</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c 0</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>control</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c 2</td>
</tr>
</tbody>
</table>

```
Higher-order functions

• Consider the use of high-order `mkCounter` function

```javascript
function mkCounter(c) {
    return () => {
        return c++;
    };
}

let x = mkCounter(0);
let y = mkCounter(2);
console.log(x());
```

Diagram:
- `function mkCounter(c)`
- `let x = mkCounter(0);`
- `let y = mkCounter(2);`
- `console.log(x());`
The environment model (by example)

• Anatomy of a scope ✓
• First-order functions ✓
• Free variables ✓
• High-order functions (bonus) ✓