Implementing Lambdas with Environments: Closures

CS 350

Dr. Joseph Eremondi

Last updated: July 19, 2024

Broad Goals

Overview

Goals

Overview

Goals

- Implement an interpreter for a Curly variant with functions

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature
  - Allow functions to be taken or returned by functions

Overview

Goals

- Implement an interpreter for a Curly variant with functions
    - Add a Lambda feature
    - Allow functions to be taken or returned by functions
    - Allow function calls on arbitrary expressions, not just symbols

Overview

Goals

- Implement an interpreter for a Curly variant with functions
    - Add a Lambda feature
    - Allow functions to be taken or returned by functions
    - Allow function calls on arbitrary expressions, not just symbols
- Use Environments to make the implementation efficient

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature
  - Allow functions to be taken or returned by functions
  - Allow function calls on arbitrary expressions, not just symbols
- Use Environments to make the implementation efficient
- Replicate the behaviour of the substitution-based interpreter

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature
  - Allow functions to be taken or returned by functions
  - Allow function calls on arbitrary expressions, not just symbols
- Use Environments to make the implementation efficient
- Replicate the behaviour of the substitution-based interpreter

Key Concepts

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature
  - Allow functions to be taken or returned by functions
  - Allow function calls on arbitrary expressions, not just symbols
- Use Environments to make the implementation efficient
- Replicate the behaviour of the substitution-based interpreter

Key Concepts

- Definition of a closure

Overview

Goals

- Implement an interpreter for a Curly variant with functions
  - Add a Lambda feature
  - Allow functions to be taken or returned by functions
  - Allow function calls on arbitrary expressions, not just symbols
- Use Environments to make the implementation efficient
- Replicate the behaviour of the substitution-based interpreter

Key Concepts

- Definition of a closure
- Static and Dynamic Scope for first-class functions

The Details

Substitution to Environmenst, Review

- Interpreter takes environment argument

Substitution to Environmenst, Review

- Interpreter takes environment argument

Substitution to Environmenst, Review

- Interpreter takes environment argument

```
  (define (interp [env : Env]
                  [expr : Expr])
          : Value ;; Was Number
    ....)
```

- Function-calls evaluate body in environment containing argument

Core and Abstract Syntax

- Exact same as Curly-Lambda substitution version

Core and Abstract Syntax

- Exact same as Curly-Lambda substitution version

Core and Abstract Syntax

- Exact same as Curly-Lambda substitution version

```
(define-type Expr
  ....
  (Fun [arg : Symbol]
       [body : Expr]))
```

- Goal is to interpret the same language, but with environments

A (Wrong) First Attempt: Values

- Define `Value` just like in substitution version

A (Wrong) First Attempt: Values

- Define `Value` just like in substitution version
    - Functions consist of their argument and the body to execute

A (Wrong) First Attempt: Values

- Define `Value` just like in substitution version
    - Functions consist of their argument and the body to execute

A (Wrong) First Attempt: Values

- Define `Value` just like in substitution version
    - Functions consist of their argument and the body to execute

```
(define-type Value
  (NumV [num : Number])
  (FunV [arg : Symbol]
        [body : Expr]))
```

A Wrong First Attempt: Functions Interp

- As a first attempt, try building functions just like in substitition version

A Wrong First Attempt: Functions Interp

- As a first attempt, try building functions just like in substitition version

A Wrong First Attempt: Functions Interp

- As a first attempt, try building functions just like in substitition version

```
(define (interp env expr)
  (type-case Expr interp
     ....
     [(Fun x body)
       (FunV x body)] ))
```

A Wrong First Attempt: Calls Interp

- Just like before

A Wrong First Attempt: Calls Interp

- Just like before
    - Evaluate argument to value

A Wrong First Attempt: Calls Interp

- Just like before
  - Evaluate argument to value
  - Evaluate function, make sure it’s actually a function, and get its parameter and body

A Wrong First Attempt: Calls Interp

- Just like before
  - Evaluate argument to value
  - Evaluate function, make sure it’s actually a function, and get its parameter and body
- Interpret function body in the environment with the parameter bound to the argument’s value

A Wrong First Attempt: Calls Interp

- Just like before
  - Evaluate argument to value
  - Evaluate function, make sure it's actually a function, and get its parameter and body
- Interpret function body in the environment with the parameter bound to the argument's value

A Wrong First Attempt: Calls Interp

- Just like before
  - Evaluate argument to value
  - Evaluate function, make sure it's actually a function, and get its parameter and body
- Interpret function body in the environment with the parameter bound to the argument's value

```
(define (interp env expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (extendEnv (bind funParam argVal)
                          env) ;;<------
                funBody))] ))
```

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables
    - With substitutions, the variables get replaced in the lambda

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables
    - With substitutions, the variables get replaced in the lambda
    - With environments, the variables aren't replaced *until we interpret the body*

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables
  - With substitutions, the variables get replaced in the lambda
  - With environments, the variables aren't replaced *until we interpret the body*
- When we actually go to interpret the body, we don't have the environment that the function was created in

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables
    - With substitutions, the variables get replaced in the lambda
    - With environments, the variables aren't replaced *until we interpret the body*
- When we actually go to interpret the body, we don't have the environment that the function was created in
    - Just the environment from the time of the call

The problem

- With a lambda, there could have been many substitutions that were applied to its free variables
  - With substitutions, the variables get replaced in the lambda
  - With environments, the variables aren't replaced *until we interpret the body*
- When we actually go to interpret the body, we don't have the environment that the function was created in
  - Just the environment from the time of the call
- We've implemented **dynamic scoping** by accident!

The Solution: Closures

- A function should be *closed* over its environment at the point it's created (interpreted)

The Solution: Closures

- A function should be *closed* over its environment at the point it’s created (interpreted)
- So we add an extra piece of data to the `Value` variant for functions: the environment at the time of creation

The Solution: Closures

- A function should be *closed* over its environment at the point it's created (interpreted)
- So we add an extra piece of data to the `Value` variant for functions: the environment at the time of creation
    - The combination of a function variable+body and an environment is called a **closure**

The Solution: Closures

- A function should be *closed* over its environment at the point it's created (interpreted)
- So we add an extra piece of data to the `Value` variant for functions: the environment at the time of creation
    - The combination of a function variable+body and an environment is called a **closure**
- Closures give environment interpreters the same behavior as substitution interpreters

The New Value Type

- Value version of functions contains an environment in addition to its variable and body

The New Value Type

- Value version of functions contains an environment in addition to its variable and body

The New Value Type

- Value version of functions contains an environment in addition to its variable and body

```
(define-type Value
  (NumV [num : Number])
  ;; Like FunV but with an environment
  (ClosureV [arg : Symbol]
            [body : Expr]
            [env : Env]))
```

A New Dynamic Type Checker

- Same idea as `checkAndGetFun`, just has an extra piece of data to retrieve

A New Dynamic Type Checker

- Same idea as `checkAndGetFun`, just has an extra piece of data to retrieve

A New Dynamic Type Checker

- Same idea as `checkAndGetFun`, just has an extra piece of data to retrieve

```
(define (checkAndGetClosure [v : Value]) : ((Symbol * Expr) * Env)
  (type-case Value v
     [(ClosureV x body env)
      (pair (pair x body) env)]
     [else
     (error 'curlyTypeError
            (string-append "Expected Function, got number:"
                           (to-string v)))]))
```

Properly Interpreting Functions

- When we interpret a lambda, *package the current environment up with it*

Properly Interpreting Functions

- When we interpret a lambda, *package the current environment up with it*
    - This is what lets us dynamically create new functions

Properly Interpreting Functions

- When we interpret a lambda, *package the current environment up with it*
    - This is what lets us dynamically create new functions
    - For top-level functions, this is the empty environment

Properly Interpreting Functions

- When we interpret a lambda, *package the current environment up with it*
  - This is what lets us dynamically create new functions
  - For top-level functions, this is the empty environment

Properly Interpreting Functions

- When we interpret a lambda, *package the current environment up with it*
  - This is what lets us dynamically create new functions
  - For top-level functions, this is the empty environment

```
(define (interp env expr)
  (type-case Expr interp
     ....
     [(Fun x body)
       (ClosureV x body env) ;;<------
       ] ))
```

Properly Interpreting Calls

- When we call a function, extend *the environment that was packaged up with it*

Properly Interpreting Calls

- When we call a function, extend *the environment that was packaged up with it*
  - Any free variables in the body get values from the place the closure was constructed

Properly Interpreting Calls

- When we call a function, extend *the environment that was packaged up with it*
  - Any free variables in the body get values from the place the closure was constructed

Properly Interpreting Calls

- When we call a function, extend *the environment that was packaged up with it*
  - Any free variables in the body get values from the place the closure was constructed

```
(define (interp env expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst (fst funVal))]
             [funBody (snd (fst funVal))]
             [funEnv (snd funVal)]))
        (interp (extendEnv (bind funParam argVal)
                           funEnv);;<------
                funBody))] )
```

Summary: Dynamic vs. Static Scope

- Free variables in a function body are variables that are not defined/bound in that function body

Summary: Dynamic vs. Static Scope

- Free variables in a function body are variables that are not defined/bound in that function body
- Static scope gives free variables values from the environment when the function was *constructed*

Summary: Dynamic vs. Static Scope

- Free variables in a function body are variables that are not defined/bound in that function body
- Static scope gives free variables values from the environment when the function was *constructed*
- Dynamic scope gives variables values from the environment when the function was *called*

Example: Static Scope

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
    - Functions and variables are in *the same namespace* in Curly-Lambda

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
  - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
    - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
    - Body is `{double {double x}}`

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
    - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
    - Body is `{double {double x}}`
    - Env is `double := {fun {x} {* x 2}}`

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
  - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
  - Body is `{double {double x}}`
  - Env is `double := {fun {x} {* x 2}}`
- Env at call:

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
  - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
  - Body is `{double {double x}}`
  - Env is `double := {fun {x} {* x 2}}`
- Env at call:
  - `double := 2`

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
  - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
  - Body is `{double {double x}}`
  - Env is `double := {fun {x} {* x 2}}`
- Env at call:
  - `double := 2`
  - `quadruple := {fun {y} {double {double x}}}`

Example: Static Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` has one free variable, `double`
  - Functions and variables are in *the same namespace* in Curly-Lambda
- `{fun {y} {double {double x}}}` evaluates to closure
  - Body is `{double {double x}}`
  - Env is `double := {fun {x} {* x 2}}`
- Env at call:
  - `double := 2`
  - `quadruple := {fun {y} {double {double x}}}`
  - `double := {fun {x} {* x 2}}`

Example: Static Scope ctd

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure
- Finally evaluates `{double {double x}}`

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure
- Finally evaluates `{double {double x}}`
  - In extended closure environment:

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure
- Finally evaluates `{double {double x}}`
    - In extended closure environment:
        - `x := 3`

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure
- Finally evaluates `{double {double x}}`
    - In extended closure environment:
        - `x := 3`
        - `double := {fun {x} {* x 2}}`

Example: Static Scope ctd

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- Call evaluates `quadruple` to closure
- Finally evaluates `{double {double x}}`
    - In extended closure environment:
        - `x := 3`
        - `double := {fun {x} {* x 2}}`
- Result is 12

Example: Dynamic Scope

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn’t save environment

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
    - Doesn't save environment
- Call evaluates `{double {double x}}`
    - In extended *call site* environment:

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`
    - `double := 2`

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`
    - `double := 2`
    - `quadruple := {fun {y} {double {double x}}}`

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`
    - `double := 2`
    - `quadruple := {fun {y} {double {double x}}}`
    - `double := {fun {x} {* x 2}}`

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`
    - `double := 2`
    - `quadruple := {fun {y} {double {double x}}}`
    - `double := {fun {x} {* x 2}}`
- Dynamic type error

Example: Dynamic Scope

```
{let double {fun {x} {* x 2}}
  {let quadruple {fun {y} {double {double x}}}
    {let double 2
      {quadruple 3}}}}
```

- `{fun {y} {double {double x}}}` evaluates to closure
  - Doesn't save environment
- Call evaluates `{double {double x}}`
  - In extended *call site* environment:
    - `x := 3`
    - `double := 2`
    - `quadruple := {fun {y} {double {double x}}}`
    - `double := {fun {x} {* x 2}}`
- Dynamic type error
  - Can't call 2 as a function

But Professor, When Will I Ever Use This?

Static Scoping in the Wild

Python:

Static Scoping in the Wild

Python:

Static Scoping in the Wild

Python:

```
timesTwo = lambda x : 2 * x
quadruple = lambda y : timesTwo(timesTwo(y))
def mainFun(x):
    timesTwo = 2.0
    return quadruple(x)
return mainFun(3)
```

12

Result:

12

Static Scoping in the Wild

JavaScript:

Static Scoping in the Wild

JavaScript:

Static Scoping in the Wild

JavaScript:

```
var timesTwo = function (x) { return x * 2 };
var quadruple =
    function (x) {return timesTwo(timesTwo(x)) };
function mainFun(x){
    var timesTwo = 2.0;
    return quadruple(x)}
return mainFun(3)
```

12

Result:

12

Async in JavaScript

- From the w3schools async tutorial

Async in JavaScript

- From the w3schools async tutorial

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function
    - Takes in another function as an argument

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function
    - Takes in another function as an argument
- They call the function argument the *callback*

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function
    - Takes in another function as an argument
- They call the function argument the *callback*
- `function(value)` is just the Javascript syntax for lambda

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function
  - Takes in another function as an argument
- They call the function argument the *callback*
- `function(value)` is just the Javascript syntax for lambda
  - Dynamically creates the function that is run when `myFunction` actually runs

Async in JavaScript

- From the w3schools async tutorial

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

- `myFunction.then` is a higher order function
  - Takes in another function as an argument
- They call the function argument the *callback*
- `function(value)` is just the Javascript syntax for lambda
  - Dynamically creates the function that is run when `myFunction` actually runs
- Concurrency in JS is mostly just syntactic sugar for lambda/higher-order functions

And More

- Swift “Closures” are just lambdas

And More

- Swift “Closures” are just lambdas

And More

- Swift "Closures" are just lambdas

```
names.sorted(by:
  { (s1: String, s2: String) -> Bool
      in return s1 > s2
  } )
```

- C++11 added anonymous functions

And More

- Swift "Closures" are just lambdas

```
names.sorted(by:
  { (s1: String, s2: String) -> Bool
      in return s1 > s2
  } )
```

- C++11 added anonymous functions

And More

- Swift "Closures" are just lambdas

```
names.sorted(by:
  { (s1: String, s2: String) -> Bool
      in return s1 > s2
  } )
```

- C++11 added anonymous functions

```
sort(V.begin(), V.end(), [](auto& a, auto& b)
{
   return a > b;
});
```

- Java 8 added anonymous functions

And More

- Swift “Closures” are just lambdas

```
names.sorted(by:
  { (s1: String, s2: String) -> Bool
      in return s1 > s2
  } )
```

- C++11 added anonymous functions

```
sort(V.begin(), V.end(), [](auto& a, auto& b)
{
   return a > b;
});
```

- Java 8 added anonymous functions

And More

- Swift "Closures" are just lambdas

```
names.sorted(by:
  { (s1: String, s2: String) -> Bool
        in return s1 > s2
  } )
```

- C++11 added anonymous functions

```
sort(V.begin(), V.end(), [](auto& a, auto& b)
{
   return a > b;
});
```

- Java 8 added anonymous functions

```
Arrays.sort(arr,
  (String a, String b) ->
            a.length() - b.length());
```

- This is all just lambda with different syntax