Environments, Binding, and Scope

CS 350

Dr. Joseph Eremondi

Last updated: July 13, 2024

Overview

The Road to Midterm

- Today: Environments in Curly

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly
- Mon: Closures and Environments in Curly

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly
- Mon: Closures and Environments in Curly
- Tues/Wed: lectures (not included on midterm)

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly
- Mon: Closures and Environments in Curly
- Tues/Wed: lectures (not included on midterm)
- Thurs: **MIDTERM**

The Road to Midterm

- Today: Environments in Curly
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly
- Mon: Closures and Environments in Curly
- Tues/Wed: lectures (not included on midterm)
- Thurs: **MIDTERM**

Everything up to and including Closures may appear on the midterm

Environments

Functions Review

- Evaluate their argument

Functions Review

- Evaluate their argument
- Lookup the function body

Functions Review

- Evaluate their argument
- Lookup the function body
- Replace the parameter variable with the

Functions Review

- Evaluate their argument
- Lookup the function body
- Replace the parameter variable with the
- Evaluate the result

Functions Review

- Evaluate their argument
- Lookup the function body
- Replace the parameter variable with the
- Evaluate the result
- If we ever interpret a variable, raise an error

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
  - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
  - Not including the time to evaluate the function body

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
  - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
  - Not including the time to evaluate the function body
- Substitution is forgetful

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
    - Not including the time to evaluate the function body
- Substitution is forgetful
    - Just replaces function variable with expression

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
    - Not including the time to evaluate the function body
- Substitution is forgetful
    - Just replaces function variable with expression
    - Not very useful for debugging

The Solution: Environments

- Data structure for *deferred substitution*

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:
    - Instead of replacing all x with value v, keep a list of replacements you need to do

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:
    - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
    - When you interpret `x`, check the environment before raising an error

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:
    - Instead of replacing all x with value v, keep a list of replacements you need to do
    - When you interpret x, check the environment before raising an error
    - If there’s an entry for x in the environment, return that

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
  - When you interpret `x`, check the environment before raising an error
  - If there’s an entry for `x` in the environment, return that
    - Error otherwise

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
  - When you interpret `x`, check the environment before raising an error
  - If there's an entry for `x` in the environment, return that
    - Error otherwise
    - Means reference to undefined variable

The Environment Data Structure: Bindings

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

```
;; Get helper functions from the type-def
(bind-name (bind 'x 3))
(bind-val (bind 'x 3))
```

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

```
;; Get helper functions from the type-def
(bind-name (bind 'x 3))
(bind-val (bind 'x 3))
```

```
'x
3
```

The Environment Data Structure: Environments

The Environment Data Structure: Environments

```
;; Lets us write Env instead of (Listof Binding)
;; But it's not defining a new type,
;; just a new name for the same type.
(define-type-alias Env (Listof Binding))
;; Environment is either empty or extended env
(define emptyEnv : Env
  empty)
(define (extendEnv [bnd : Binding]
                   [env : Env])
        : Env
  (cons bnd env))

emptyEnv
(extendEnv (bind 'x 3) (extendEnv (bind 'y 4) empty))
```

The Environment Data Structure: Environments

```
;; Lets us write Env instead of (Listof Binding)
;; But it's not defining a new type,
;; just a new name for the same type.
(define-type-alias Env (Listof Binding))
;; Environment is either empty or extended env
(define emptyEnv : Env
  empty)
(define (extendEnv [bnd : Binding]
                   [env : Env])
        : Env
  (cons bnd env))

emptyEnv
(extendEnv (bind 'x 3) (extendEnv (bind 'y 4) empty))
```

```
'()
(list (bind 'x 3) (bind 'y 4))
```

Looking up variables

- Find the **first** binding in the environment

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we’ve seen lots already

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we've seen lots already

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we've seen lots already

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
   ;; Can't find a variable in an empty env
   [empty (error 'lookup "undefined variable")]
   ;; Cons: check if the first binding is the var
   ;; we're looking for.
   ;; Return its value  if it is, otherwise
   ;; keep looking in the rest of the list
   [(cons b rst-env) (cond
                       [(symbol=? n (bind-name b))
                        (bind-val b)]
                       [else (lookup n rst-env)])]))
```

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
    - Unlike fundefs, this will *change across recursive calls*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
    - Unlike fundefs, this will *change across recursive calls*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
  - Unlike fundefs, this will *change across recursive calls*

```
(define (interp [env : Env]
                [defs : (Listof FunDef)]
                [e : Expr] ) : Number
  (type-case Expr e
             ....))
```

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

```
;; {+ e1 e2} evaluates e1 and e2, then adds the results together
    [(Plus l r)
     (+ (interp env defs l) (interp env defs r))]
```

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable nto found error

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable nto found error

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable nto found error

```
[(Var x)
      (lookup x env)]
```

Case: Function call

- Just like before, we get the function body + variable, and value for argument

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, add variable-value pair to the environment

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, add variable-value pair to the environment
    - Called *binding* the variable to its value

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, add variable-value pair to the environment
    - Called *binding* the variable to its value

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, add variable-value pair to the environment
    - Called *binding* the variable to its value

```
[(Call funName argExpr)
     (let* ([argVal (interp env defs argExpr)]
            [def (get-fundef funName defs)]
            [argVar (mkFunDef-arg def)]
            [funBody (mkFunDef-body def)])
       (interp (extendEnv (bind argVar argVal) env) ;;<------
               defs
               funBody))]
```

Implementing Let

Curly-Let

- New language: Curly-Let

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`
        - Gives `x` the value `e1` in the expression `e2`

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`
        - Gives `x` the value `e1` in the expression `e2`
        - Called letvar so we don't confuse with plait

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`
        - Gives `x` the value `e1` in the expression `e2`
        - Called letvar so we don't confuse with plait
- We'll implement with both substitution and environments

Abstract Syntax

Abstract Syntax

```
(type-def Expr
  ....
  [(Letvar [x : Symbol]
           [xval : Expr]
           [body : Expr])]
          )
```

- Parsing and Desugaring are the same as usual

Abstract Syntax

```
(type-def Expr
  ....
  [(Letvar [x : Symbol]
           [xval : Expr]
           [body : Expr])]
          )
```

- Parsing and Desugaring are the same as usual
  - See `Curly-Let.rkt`

Interpreting: Substitution

- Want variable to have the given value in the body

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
    - Can have exponential speedup in some algorithms

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
    - Can have exponential speedup in some algorithms

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
    - Can have exponential speedup in some algorithms

```
(define (interp [defs : (Listof FunDef)] ;;NEW
                [e : Expr] ) : Number
  (type-case Expr e
        ;; ....
     [(Letvar x xexp body)
        (interp defs (subst x (interp defs xexp) body))]])
```

Interpreting: Environments

- Interpret the variable’s value in the current environment

Interpreting: Environments

- Interpret the variable’s value in the current environment
- Interpret the let body in the *extended* environment

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit x we'll look in the env

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit `x` we'll look in the env

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit x we'll look in the env

```
(define (interp [env : Env]
                [defs : (Listof FunDef)]
                [e : Expr] ) : Number
  (type-case Expr e
        ;; ....
     [(Letvar x xexp body)
      (let ([xval (interp env defs xexp)])
        (interp (extendEnv (bind x xval) env)
                defs body))])
```