Implementing Lambdas with Substitution and Dynamic Typing

CS 350

Dr. Joseph Eremondi

Last updated: July 18, 2024

Broad Strategy

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions
  - Functions as a value: function carries a parameter name and a variable body

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions
  - Functions as a value: function carries a parameter name and a variable body
- Non-trivial parts:

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions
  - Functions as a value: function carries a parameter name and a variable body
- Non-trivial parts:
  - Now we have *two* possible kinds of values

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions
  - Functions as a value: function carries a parameter name and a variable body
- Non-trivial parts:
  - Now we have *two* possible kinds of values
  - Functions are not numbers

Implementation Strategy

- In some ways, it's *easier* to implement first-class functions with substitution than normal substitutions
  - We can get rid of the whole list-of-definitions
  - Functions as a value: function carries a parameter name and a variable body
- Non-trivial parts:
  - Now we have *two* possible kinds of values
  - Functions are not numbers
  - Need to implement **dynamic typing**

The Value Type

The Value Type

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

- The result of interpretation is called a *value*

The Value Type

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

- The result of interpretation is called a *value*
  - Number, or a function

The Value Type

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

- The result of interpretation is called a *value*
  - Number, or a function
  - Function stores the info we need to call it

Lambdas in our ASTs

Lambdas in our ASTs

```
(define-type Expr
  ....
  ;; Capital L, so we don't clash with the plait keyword
  (Lambda [arg : Symbol]
          [body : Expr]))
```

- Similar for `SurfExpr`

Lambdas in our ASTs

```
(define-type Expr
  ....
  ;; Capital L, so we don't clash with the plait keyword
  (Lambda [arg : Symbol]
          [body : Expr]))
```

- Similar for `SurfExpr`
- Notice that `Lambda` and `FunV` have the *exact* same fields

Lambdas in our ASTs

```
(define-type Expr
  ....
  ;; Capital L, so we don't clash with the plait keyword
  (Lambda [arg : Symbol]
          [body : Expr]))
```

- Similar for `SurfExpr`
- Notice that `Lambda` and `FunV` have the *exact* same fields
  - Functions *are* values

Lambdas in our ASTs

```
(define-type Expr
  ....
  ;; Capital L, so we don't clash with the plait keyword
  (Lambda [arg : Symbol]
          [body : Expr]))
```

- Similar for `SurfExpr`
- Notice that `Lambda` and `FunV` have the *exact* same fields
  - Functions *are* values
  - A function is saying “here’s a computation to do later”, so once we’ve got a Lambda, there’s no more evaluation to do

Lambdas in our ASTs

```
(define-type Expr
  ....
  ;; Capital L, so we don't clash with the plait keyword
  (Lambda [arg : Symbol]
          [body : Expr]))
```

- Similar for `SurfExpr`
- Notice that `Lambda` and `FunV` have the *exact* same fields
  - Functions *are* values
  - A function is saying "here's a computation to do later", so once we've got a Lambda, there's no more evaluation to do
  - We'll see more of this for `interp`

Substitution for Lambda

- Lambda binds its variable

Substitution for Lambda

- Lambda binds its variable
  - Similar to LetVar

Substitution for Lambda

- Lambda binds its variable
  - Similar to LetVar
- Don’t substitute in a Lambda body if the variable we’re replacing matches the function variable

Substitution for Lambda

- Lambda binds its variable
  - Similar to LetVar
- Don’t substitute in a Lambda body if the variable we’re replacing matches the function variable
  - Ensures that the function variable shadows any previous declarations

Substitution for Lambda

- Lambda binds its variable
  - Similar to LetVar
- Don’t substitute in a Lambda body if the variable we’re replacing matches the function variable
  - Ensures that the function variable shadows any previous declarations

Substitution for Lambda

- Lambda binds its variable
  - Similar to LetVar
- Don't substitute in a Lambda body if the variable we're replacing matches the function variable
  - Ensures that the function variable shadows any previous declarations

```
(define (subst [toReplace : Symbol]
               [replacedBy : Expr]
               [replaceIn : Expr]) : Expr
  (type-case Expr replaceIn
  ....
  [(Lambda x body)
     (if (symbol=? x toReplace)
         (Lambda x body) ;; Don't substitute if the variable is shado
         (Lambda x (subst toReplace replacedBy body)))]
  ))
```

The Problem: Interpretation

- We want `interp` to produce a `Value`

The Problem: Interpretation

- We want `interp` to produce a `Value`
  - So that we can produce functions as the result of expressions

The Problem: Interpretation

- We want `interp` to produce a `Value`
  - So that we can produce functions as the result of expressions
- Our previous interpreter assumed that `interp` always returned a number

The Problem: Interpretation

- We want `interp` to produce a `Value`
  - So that we can produce functions as the result of expressions
- Our previous interpreter assumed that `interp` always returned a number
- We need to introduce **dynamic type checking** in Curly-Fun

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error
- Repeat until there are no type errors:

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error
- Repeat until there are no type errors:
  - Go to the first type error

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error
- Repeat until there are no type errors:
  - Go to the first type error
  - Change the code to have the right type

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error
- Repeat until there are no type errors:
  - Go to the first type error
  - Change the code to have the right type
    - Wrap numeric results in `NumV`

Aside: Type-Based Refactoring

- We can use the plait type inference to help us write our implementation
- Change `interp` to produce `Value` instead of `Number`
- The type-checker sees an error
- Repeat until there are no type errors:
  - Go to the first type error
  - Change the code to have the right type
    - Wrap numeric results in `NumV`
    - Perform dynamic type checks to extract fields

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation
    - Make sure that Plus and Times are only given numbers

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation
  - Make sure that Plus and Times are only given numbers
  - Make sure that the condition of `ifo` is a number

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation
  - Make sure that Plus and Times are only given numbers
  - Make sure that the condition of `ifo` is a number
  - Make sure the thing in a Call is actually a function

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation
  - Make sure that Plus and Times are only given numbers
  - Make sure that the condition of `ifo` is a number
  - Make sure the thing in a Call is actually a function
- *Dynamic* because we check *while the program is running*

Dynamic Typing

- Curly now has two different types of things, `FunV` and `NumV`
- It's possible
- *Dynamic type checking* checks that the inputs to an operation are valid before running that operation
  - Make sure that Plus and Times are only given numbers
  - Make sure that the condition of `ifo` is a number
  - Make sure the thing in a Call is actually a function
- *Dynamic* because we check *while the program is running*
  - If we checked before it ran, it would be *static type checking*

Type Safety

- Racket is pretty safe

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory
- In other languages like C++, type errors (e.g. improper casts) can lead to safety issues, security bugs, etc.

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory
- In other languages like C++, type errors (e.g. improper casts) can lead to safety issues, security bugs, etc.
- Curly is safe because Racket is, but we’ll define our own notion of “type safety”:

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory
- In other languages like C++, type errors (e.g. improper casts) can lead to safety issues, security bugs, etc.
- Curly is safe because Racket is, but we’ll define our own notion of “type safety”:
  - Want to raise an error with an informative message when a Curly program performs a type-unsafe operation, instead of a generic Racket error message

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory
- In other languages like C++, type errors (e.g. improper casts) can lead to safety issues, security bugs, etc.
- Curly is safe because Racket is, but we’ll define our own notion of “type safety”:
  - Want to raise an error with an informative message when a Curly program performs a type-unsafe operation, instead of a generic Racket error message
  - e.g. Dynamic type checks make sure that our interpreter, rather than Racket’s built in functions, discover the error

Type Safety

- Racket is pretty safe
  - Can’t write to arbitrary memory
- In other languages like C++, type errors (e.g. improper casts) can lead to safety issues, security bugs, etc.
- Curly is safe because Racket is, but we’ll define our own notion of “type safety”:
  - Want to raise an error with an informative message when a Curly program performs a type-unsafe operation, instead of a generic Racket error message
  - e.g. Dynamic type checks make sure that our interpreter, rather than Racket’s built in functions, discover the error
  - Good practice for programming in less safe languages

Defining some helper functions

Defining some helper functions

```
(define (checkAndGetNum [v : Value]) : Number
  (type-case Value V
             [(NumV n) n]
             [else
              (error 'curlyTypeError
                     (string-append "Expected Number, got function:"
                                    (to-string v)))]))

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

- Lets us turn a Value into Number/Function

Defining some helper functions

```
(define (checkAndGetNum [v : Value]) : Number
  (type-case Value V
             [(NumV n) n]
             [else
              (error 'curlyTypeError
                     (string-append "Expected Number, got function:"
                                    (to-string v)))]))

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

- Lets us turn a Value into Number/Function
  - Better error-message than e.g. `NumV-num` gives

A Helpful Higher Order Function

A Helpful Higher Order Function

```
(define (liftVal2 [f : (Number Number -> Number)]
                  [x : Value]
                  [y : Value]) : Value
  (let ([nx (checkAndGetNum x)]
        [ny (checkAndGetNum y)])
    (NumV (f nx ny))))
```