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**

Syntax and Parsing

- `{fun SYMBOL <expr>}`

Syntax and Parsing

- `{fun SYMBOL <expr>}`
  - `{fun {x} body}` is anonymous function with argument `x` and body `body`

Syntax and Parsing

- `{fun SYMBOL <expr>}`
  - `{fun {x} body}` is anonymous function with argument `x` and body `body`
  - fun instead of lambda to distinguish Curly vs Plait

Syntax and Parsing

- `{fun SYMBOL <expr>}`
  - `{fun {x} body}` is anonymous function with argument `x` and body `body`
  - fun instead of lambda to distinguish Curly vs Plait

Syntax and Parsing

- `{fun SYMBOL <expr>}`
  - `{fun {x} body}` is anonymous function with argument x and body body
  - fun instead of lambda to distinguish Curly vs Plait

```
(define (parse s-expr)
  (cond
    ....
    [(s-exp-match? `{fun {SYMBOL} ANY} s)
     (SurfFun (s-exp->symbol
               (first (s-exp->list
                       (second (s-exp->list s)))))
              (parse (third (s-exp->list s))))]))
```

Parsing Calls

- Same as before, but now we need to allow any expression in function position, not just symbols

Parsing Calls

- Same as before, but now we need to allow any expression in function position, not just symbols

Parsing Calls

- Same as before, but now we need to allow any expression in function position, not just symbols

```
[(s-exp-match? `{ANY ANY} s)
     (SurfCall (parse (first (s-exp->list s))
               (parse (second (s-exp->list s)))))]
```

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

- Similar for `SurfExpr`

Lambdas in our ASTs

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

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

Lambdas in our ASTs

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

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

Lambdas in our ASTs

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

- Similar for `SurfExpr`
- Notice that `Fun` 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
  ....
  (Fun [arg : Symbol]
       [body : Expr]))
```

- Similar for `SurfExpr`
- Notice that `Fun` 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`

Values to Expressions

- We’re going to need to substitute values into expressions

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

```
(define (value->expr [v : Value]) : Expr
  (type-case Value v
             [(NumV v) (NumLit v)]
             [(FunV x body) (Fun x body)]))
```

- `Value` is *embedded* in `Expr`

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

```
(define (value->expr [v : Value]) : Expr
  (type-case Value v
             [(NumV v) (NumLit v)]
             [(FunV x body) (Fun x body)]))
```

- `Value` is *embedded* in `Expr`
  - Want `(interp (value->expr v))` to produce `v`

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

```
(define (value->expr [v : Value]) : Expr
  (type-case Value v
             [(NumV v) (NumLit v)]
             [(FunV x body) (Fun x body)]))
```

- `Value` is *embedded* in `Expr`
  - Want `(interp (value->expr v))` to produce `v`
- Not actually doing any computation, just changing the constructors so the type checker is happy

Values to Expressions

- We're going to need to substitute values into expressions
  - But `subst` works on expressions, not values

```
(define (value->expr [v : Value]) : Expr
  (type-case Value v
             [(NumV v) (NumLit v)]
             [(FunV x body) (Fun x body)]))
```

- `Value` is *embedded* in `Expr`
  - Want `(interp (value->expr v))` to produce `v`
- Not actually doing any computation, just changing the constructors so the type checker is happy
- In a more sophisticated implementation language, we could make values a *subtype* of expressions, but that's beyond this course

Substitution for Lambda

- Fun binds its variable

Substitution for Lambda

- Fun binds its variable
  - Similar to LetVar

Substitution for Lambda

- Fun 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

- Fun 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

- Fun 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

- Fun 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
  ....
  [(Fun x body)
     ;; Don't substitute if variable is shadowed
     (if (symbol=? x toReplace)
         (Fun x body)
         (Fun 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

Evaluating Functions

Evaluating Functions

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

- Interp no longer needs a list of function definitions

Evaluating Functions

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

- Interp no longer needs a list of function definitions
  - Can use let + lambda for the same effect

Evaluating Functions

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

- Interp no longer needs a list of function definitions
  - Can use let + lambda for the same effect
- Nothing to do to turn function into a value

Evaluating Functions

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

- Interp no longer needs a list of function definitions
  - Can use let + lambda for the same effect
- Nothing to do to turn function into a value
  - Just package up the data in the Value type

Calls

Calls

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- Mostly the same as for Curly-Fundef

Calls

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- Mostly the same as for Curly-Fundef
  - Except don't have to look up the function body + param

Calls

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- Mostly the same as for Curly-Fundef
  - Except don't have to look up the function body + param
- The thing we're calling might not be a Lambda *yet*

Calls

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- Mostly the same as for Curly-Fundef
  - Except don't have to look up the function body + param
- The thing we're calling might not be a Lambda *yet*
  - So we evaluate it recursively

Calls

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- Mostly the same as for Curly-Fundef
  - Except don't have to look up the function body + param
- The thing we're calling might not be a Lambda *yet*
  - So we evaluate it recursively
  - Do a dynamic type check to make sure the result is a function, not a number

Capturing the environment

- Functions might contain *free variables*

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`
  - Interp replaces `f` with the fun

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`
  - Interp replaces `f` with the fun
  - Calling `f` with `3` replaces `x` with `3`

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`
  - Interp replaces `f` with the fun
  - Calling `f` with `3` replaces `x` with `3`
    - Result: `{fun {y} {+ 3 y}`

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`
  - Interp replaces `f` with the fun
  - Calling `f` with `3` replaces `x` with `3`
    - Result: `{fun {y} {+ 3 y}`
    - e.g. The function that adds 3 to its argument

Capturing the environment

- Functions might contain *free variables*
  - Variables that are not bound/defined by the function itself
- Subst *will* replace those variables when concrete values are given

```
{letvar
   f {fun {x} {fun {y} {+ x y}}}
   {f 3}}
```

  - For any `x`, `f` produces another function that adds its argument to `x`
  - `x` is *free* in `{fun {y} {+ x y}}`
  - Interp replaces `f` with the fun
  - Calling `f` with `3` replaces `x` with `3`
    - Result: `{fun {y} {+ 3 y}`
    - e.g. The function that adds 3 to its argument
  - Substitution lets us build new functions at run time based on values to other functions

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`
  - See in-class file / Curly-Lambda for full details

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`
  - See in-class file / Curly-Lambda for full details
- Don't need to change `If0` branches or Let, since they don't do numeric operations

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`
  - See in-class file / Curly-Lambda for full details
- Don't need to change `If0` branches or Let, since they don't do numeric operations
  - Can have an if that produces a function

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`
  - See in-class file / Curly-Lambda for full details
- Don't need to change `If0` branches or Let, since they don't do numeric operations
  - Can have an if that produces a function

Fixing the rest of the interpreter

- Need to change operations to use Value instead of Number
  - Mostly just adding calls to `checkAndGetNum`
  - See in-class file / Curly-Lambda for full details
- Don't need to change If0 branches or Let, since they don't do numeric operations
  - Can have an if that produces a function

```
(define (interp expr)
  (type-case Expr expr
        [(If0 test thn els)
           (let ([testNum (checkAndGetNum (interp test))])
             (if (= 0 testNum)
                 (interp thn)
                 (interp els)))]))
```

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))))
```

- Then can write:

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))))
```

- Then can write:

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))))
```

- Then can write:

```
(define (interp expr)
  (type-case Expr expr
     [(Plus l r)
       (liftVal2 + (interp l) (interp r))]
     [(Times l r)
       (liftVal2 * (interp l) (interp r))]))
```

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

```
{letvar f {fun x {x x}}
  {f f}}
```

- Replaces `f` with `{{fun {x} {x x}} {fun x {x x}}}` calling itself

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

```
{letvar f {fun x {x x}}
  {f f}}
```

- Replaces f with `{{fun {x} {x x}} {fun x {x x}}}` calling itself
- Function call: takes body `{x x}`, replaces x with argument `{fun {x} {x x}}`

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

```
{letvar f {fun x {x x}}
  {f f}}
```

- Replaces `f` with `{{fun {x} {x x}} {fun x {x x}}}` calling itself
- Function call: takes body `{x x}`, replaces `x` with argument `{fun {x} {x x}}`
- Result is `{{fun {x} {x x}} {fun {x} {x x}}}`, exactly what we started with!

Our Language Is Now Turing Complete!

- Curly-Lambda can simulate every computer program ever written
  - Not counting syscalls, IO, networking, effects, etc.
- Just because we *can* write an equivalent Curly-Lambda program, doesn't mean it's easy/succinct
- We can now write Curly-Lambda programs that run forever

```
{letvar f {fun x {x x}}
  {f f}}
```

- Replaces `f` with `{{fun {x} {x x}} {fun x {x x}}}` calling itself
- Function call: takes body `{x x}`, replaces `x` with argument `{fun {x} {x x}}`
- Result is `{{fun {x} {x x}} {fun {x} {x x}}}`, exactly what we started with!
- `interp` runs forever

A First Taste of Laziness

- How we implement `ifo` really matters now

A First Taste of Laziness

- How we implement `ifo` really matters now
  - Code might run forever

A First Taste of Laziness

- How we implement `ifo` really matters now
  - Code might run forever
  - Don’t want to run code in the branch we don’t take

A First Taste of Laziness

- How we implement `ifo` really matters now
  - Code might run forever
  - Don’t want to run code in the branch we don’t take
- What if we did this?

A First Taste of Laziness

- How we implement `ifo` really matters now
  - Code might run forever
  - Don’t want to run code in the branch we don’t take
- What if we did this?

A First Taste of Laziness

- How we implement `ifo` really matters now
  - Code might run forever
  - Don't want to run code in the branch we don't take
- What if we did this?

```
;; BAD! Don't do this
(define (interp expr
  (type-case Expr expr
      [(Ifo test thn els)
      (let ([thenVal (interp thn)]
            [elseVal (interl els)])
        (if (= 0 (checkAndGetNum (interp test)))
            thenVal
            elseVal))])))
(run `{ifo 0
           {+ 1 1}
           {letvar f {fun x {x x}} {f f}}})
```

- Loops because it evaluates the untaken branch