Objects and OOP

CS 350

Dr. Joseph Eremondi

Last updated: August 8, 2024

Concepts of Objects

Overview

- Objectives

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments
    - Locations and Stores

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments
    - Locations and Stores
- Key Concepts

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments
    - Locations and Stores
- Key Concepts
  - Members/Fields

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments
    - Locations and Stores
- Key Concepts
  - Members/Fields
  - Methods

Overview

- Objectives
  - To see how OOP and datatype-oriented programming are dual/inverses
  - To see how objects can be implemented using concepts we've already seen
    - Closures
    - Environments
    - Locations and Stores
- Key Concepts
  - Members/Fields
  - Methods
  - Encapsulation

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities
- To define a new operation taking some datatype T as input we:

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities
- To define a new operation taking some datatype T as input we:
    - Pattern matched on the value of type T

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities
- To define a new operation taking some datatype T as input we:
  - Pattern matched on the value of type T
  - Produced a result for each value

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities
- To define a new operation taking some datatype T as input we:
  - Pattern matched on the value of type T
  - Produced a result for each value
- This is a *local change*: we can add a new operation without needing to change any other code in the codebase

The Object/Datatype Duality

- Recall that a datatype is defined by specifying a list of *variants* e.g. different ways of constructing the datatype
- The definition of the type carried with it a finite list of possibilities
- To define a new operation taking some datatype `T` as input we:
  - Pattern matched on the value of type `T`
  - Produced a result for each value
- This is a *local change*: we can add a new operation without needing to change any other code in the codebase
- To add a new variant, we needed to refactor *every single* definition that uses `type-case` to have a new case for the new variant

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
    - For us, just an informal description

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
  - For us, just an informal description
  - Some languages let you specify interfaces as types

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
  - For us, just an informal description
  - Some languages let you specify interfaces as types
- Every time we define a new object with its members and methods, we define a new variant satisfying that interface

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
  - For us, just an informal description
  - Some languages let you specify interfaces as types
- Every time we define a new object with its members and methods, we define a new variant satisfying that interface
  - Local change: define the object in one place and you have a new variant

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
  - For us, just an informal description
  - Some languages let you specify interfaces as types
- Every time we define a new object with its members and methods, we define a new variant satisfying that interface
  - Local change: define the object in one place and you have a new variant
- Adding a new operation to an interface requires refactoring *every object implementing that interface*

The Opposite: OOP

- An object consists of some data, bundled with a **fixed set** of operations on that data
- The operations available (and their types, when relevant) define an *interface*
  - For us, just an informal description
  - Some languages let you specify interfaces as types
- Every time we define a new object with its members and methods, we define a new variant satisfying that interface
  - Local change: define the object in one place and you have a new variant
- Adding a new operation to an interface requires refactoring *every object implementing that interface*
  - Since you have to add the new method

OOP vs Algebraic Datatypes

	Add new variant	**Add new operation**
Datatypes	Needs global refactoring	Local additions only
Objects	Local additions only	Needs global refactoring

Curly-Obj-Immut

Immutable Objects

- We'll add **immutable objects** to Curly

Immutable Objects

- We'll add **immutable objects** to Curly
  - No way to mutate field values

Immutable Objects

- We'll add **immutable objects** to Curly
  - No way to mutate field values
  - Copying the object makes a new copy

Immutable Objects

- We'll add **immutable objects** to Curly
  - No way to mutate field values
  - Copying the object makes a new copy
- Like Tuples in Python

Object Operations

- Object creation

Object Operations

- Object creation

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object
  - `{get o field}` is like `o.field`

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object
  - `{get o field}` is like `o.field`
- Method calling `{send <expr> SYMBOL <expr> }`

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object
  - `{get o field}` is like `o.field`
- Method calling `{send <expr> SYMBOL <expr> }`
  - Call the given object's method (with the given name), with the given argument

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object
  - `{get o field}` is like `o.field`
- Method calling `{send <expr> SYMBOL <expr> }`
  - Call the given object's method (with the given name), with the given argument
  - `{send e method arg}` is like `e.method(arg)`

Object Operations

- Object creation

```
{object {{SYMBOL <expr>}*} {SYMBOL {SYMBOL} <expr>}* }
```

  - Create an object with 0 or more *fields*, whose names are given by the given symbols, and 0 or more *methods*, with the given name, parameter name, and body
  - Simplest version of an (OOP-style) constructor, just takes values for all the fiels and methods
    - Not to be confused with constructor for a datatype
- Field getting `{get <expr> SYMBOL}`
  - Gets the value of the field with the given name from the given object
  - `{get o field}` is like `o.field`
- Method calling `{send <expr> SYMBOL <expr> }`
  - Call the given object's method (with the given name), with the given argument
  - `{send e method arg}` is like `e.method(arg)`
- Name comes from Smalltalk, history of OOP

Object operations ctd

- Self-reference: `this`

Object operations ctd

- Self-reference: `this`
  - Inside an object's methods, there is a special variable called `this`

Object operations ctd

- Self-reference: `this`
  - Inside an object's methods, there is a special variable called `this`
  - Refers to the object that the method is being called on, so we can get e.g. field values from methods

Example

Example

```
{letvar mkCircle
        {fun {r}
             {object {{radius r}}
                     ;; All methods take one parameter,
                     ;; so we just ignore the parameter
                     {getArea {x} {* {get this radius}
                                     {* {get this radius} 3.14}}}}}
        {letvar unitCircle {mkCircle 1}
                {send unitCircle getArea 0}}}
```

- Produces `3.14`

Example

Example

```
{letvar mkSquare
        {fun {w}
             {object {{width w}}
                     ;; All methods take one parameter,
                     ;; so we just ignore the parameter
                     {getArea {x} {* {get this width}
                                     {get this width} }}}}
        {letvar unitSquare {mkSquare 2}
                {send unitSquare getArea 0}}}
```

- Produces 4

Example

```
{letvar mkSquare
        {fun {w}
             {object {{width w}}
                     ;; All methods take one parameter,
                     ;; so we just ignore the parameter
                     {getArea {x} {* {get this width}
                                     {get this width} }}}}
        {letvar unitSquare {mkSquare 2}
                {send unitSquare getArea 0}}}
```

- Produces 4
- Same interface as circle, can use interchangably if only call methods

Implementing Objects

General Strategy

- For the most part, nothing complicated

General Strategy

- For the most part, nothing complicated
  - Just make new `Expr` and `Value` variants with data for object-fields and methods

General Strategy

- For the most part, nothing complicated
  - Just make new `Expr` and `Value` variants with data for object-fields and methods
  - Store fields as values

General Strategy

- For the most part, nothing complicated
  - Just make new `Expr` and `Value` variants with data for object-fields and methods
  - Store fields as values
  - Store methods as closures

General Strategy

- For the most part, nothing complicated
  - Just make new `Expr` and `Value` variants with data for object-fields and methods
  - Store fields as values
  - Store methods as closures
- Tricky bit: making sure there's a value for `this` in scope for method calls

General Strategy

- For the most part, nothing complicated
  - Just make new `Expr` and `Value` variants with data for object-fields and methods
  - Store fields as values
  - Store methods as closures
- Tricky bit: making sure there's a value for `this` in scope for method calls
  - Making sure it's the *right* value

New Expression Variants

New Expression Variants

```
(define-type Expr
  ...
  (Object [fields : (Listof (Symbol * Expr))]
          [methods : (Listof (Symbol * (Symbol * Expr)))])
  (Get [obj-expr : Expr]
       [field-name : Symbol])
  (Send [obj-expr : Expr]
        [method-name : Symbol]
        [arg-expr : Expr]))
```

- Nothing fancy, just the literal tree representation of the previous syntax

New Expression Variants

```
(define-type Expr
  ...
  (Object [fields : (Listof (Symbol * Expr))]
          [methods : (Listof (Symbol * (Symbol * Expr)))])
  (Get [obj-expr : Expr]
       [field-name : Symbol])
  (Send [obj-expr : Expr]
        [method-name : Symbol]
        [arg-expr : Expr]))
```

- Nothing fancy, just the literal tree representation of the previous syntax
  - Object is list of field-name field-expression pairs, and list of method-name, method-param, method-body triples

New Expression Variants

```
(define-type Expr
  ...
  (Object [fields : (Listof (Symbol * Expr))]
          [methods : (Listof (Symbol * (Symbol * Expr)))])
  (Get [obj-expr : Expr]
       [field-name : Symbol])
  (Send [obj-expr : Expr]
        [method-name : Symbol]
        [arg-expr : Expr]))
```

- Nothing fancy, just the literal tree representation of the previous syntax
  - Object is list of field-name field-expression pairs, and list of method-name, method-param, method-body triples
  - Get has expression for the object whose field we're getting, and a symbol for the field name

New Expression Variants

```
(define-type Expr
  ...
  (Object [fields : (Listof (Symbol * Expr))]
          [methods : (Listof (Symbol * (Symbol * Expr)))])
  (Get [obj-expr : Expr]
       [field-name : Symbol])
  (Send [obj-expr : Expr]
        [method-name : Symbol]
        [arg-expr : Expr]))
```

- Nothing fancy, just the literal tree representation of the previous syntax
  - Object is list of field-name field-expression pairs, and list of method-name, method-param, method-body triples
  - Get has expression for the object whose field we're getting, and a symbol for the field name
  - Send has expression for the object whose method we're calling, the name of the method, and an expression for the argument

New Value Variants

New Value Variants

```
(define-type Value
  ....
  (ObjV [fields : (Listof (Symbol * Value))]
        [methods : (Listof (Symbol * Value))]))
```

- Value version of object

New Value Variants

```
(define-type Value
  ....
  (ObjV [fields : (Listof (Symbol * Value))]
        [methods : (Listof (Symbol * Value))]))
```

- Value version of object
  - Fields: just like in `Expr`, except each name has a value, not an expression

New Value Variants

```
(define-type Value
  ....
  (ObjV [fields : (Listof (Symbol * Value))]
        [methods : (Listof (Symbol * Value))]))
```

- Value version of object
  - Fields: just like in `Expr`, except each name has a value, not an expression
  - Methods: list of name-value pairs, where each value is assumed to be a closure

Starting Language

- We'll add objects onto Curly-Lambda

Starting Language

- We'll add objects onto Curly-Lambda
  - No stores, just environments

Starting Language

- We'll add objects onto Curly-Lambda
  - No stores, just environments
- Assignment 6 will be integrating Objects and Stores

Starting Language

- We'll add objects onto Curly-Lambda
  - No stores, just environments
- Assignment 6 will be integrating Objects and Stores
  - Building a small language like Python or JavaScript

Interpreting Object Creation

Interpreting Object Creation

```
(define (interp [env : Env]
                [e : Expr] : Value
  (type-case Expr e
    [(Object fields methods)
       ;; Each named field expression gets turned into a name-value pa
       (ObjV (map (lambda ([pr : (Symbol * Expr)])
                    (pair (fst pr) (interp env (snd pr))))
                  fields)
                  ;; Each method gets turned into a closure
                 (map (lambda ([pr : (Symbol * (Symbol * Expr))])
                      (pair (fst pr) (ClosureV (fst (snd pr))
                                               (snd (snd pr))
                                               env)))
                    methods))]))
```

A Helper Function for List of Pairs

A Helper Function for List of Pairs

```
(define (find [l : (Listof (Symbol * 'a))] [name : Symbol]) : 'a
  (type-case (Listof (Symbol * 'a)) l
    [empty
     (error 'find (string-append "not found: " (symbol->string name)))]
    [(cons p rst-l)
     (if (symbol=? (fst p) name)
         (snd p)
         (find rst-l name))]))
```

- The same `lookup~/~fetch` code we've written a bunch

A Helper Function for List of Pairs

```
(define (find [l : (Listof (Symbol * 'a))] [name : Symbol]) : 'a
  (type-case (Listof (Symbol * 'a)) l
    [empty
     (error 'find (string-append "not found: " (symbol->string name)))]
    [(cons p rst-l)
     (if (symbol=? (fst p) name)
         (snd p)
         (find rst-l name))]))
```

- The same `lookup~/~fetch` code we've written a bunch
  - Works for pairs, not custom datatype

A Helper Function for List of Pairs

```
(define (find [l : (Listof (Symbol * 'a))] [name : Symbol]) : 'a
  (type-case (Listof (Symbol * 'a)) l
    [empty
     (error 'find (string-append "not found: " (symbol->string name)))]
    [(cons p rst-l)
     (if (symbol=? (fst p) name)
         (snd p)
         (find rst-l name))]))
```

- The same `lookup~/~fetch` code we've written a bunch
  - Works for pairs, not custom datatype
  - Polymorphic in type of second thing in pair

A Helper Function for List of Pairs

```
(define (find [l : (Listof (Symbol * 'a))] [name : Symbol]) : 'a
  (type-case (Listof (Symbol * 'a)) l
    [empty
     (error 'find (string-append "not found: " (symbol->string name)))]
    [(cons p rst-l)
     (if (symbol=? (fst p) name)
         (snd p)
         (find rst-l name))]))
```

- The same `lookup~/~fetch` code we've written a bunch
  - Works for pairs, not custom datatype
  - Polymorphic in type of second thing in pair
    - Works for fields and methods

Interpreting Field Lookup

Interpreting Field Lookup

```
[(Get obj-expr field-name)
 ;; Dynamic type check
  (type-case Value (interp env obj-expr)
      [(ObjV fields methods)
        (find fields field-name)]
        [else (error 'interp "not an object")])]
```

Interpreting Method Calls

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                             (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                             (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values
- Lookup the method by name in the object

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                     (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values
- Lookup the method by name in the object
  - Gives a symbol-expr pair for parameter and body

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                             (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values
- Lookup the method by name in the object
  - Gives a symbol-expr pair for parameter and body
- Interpret the body in an environment extended with

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                             (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values
- Lookup the method by name in the object
  - Gives a symbol-expr pair for parameter and body
- Interpret the body in an environment extended with
  - The argument bound to the parameter

Interpreting Method Calls

```
(Send obj-expr method-name arg-expr)
      (let [(obj-val (interp env obj-expr))
            (arg-val (interp env arg-expr))]
        ;; dynamic type check
        (type-case Value obj-val
          [(ObjV fields methods)
           (let ([param-closure (find methods method-name)])
             (interp (extendEnv (bind 'this obj-val)
                                (extendEnv (bind (ClosureV-arg param-closure
                                                 arg-val)
                                           ClosureV-env))
                             (ClosureV-body param-closure)))]
```

- Interpret the object and argument to values
- Lookup the method by name in the object
  - Gives a symbol-expr pair for parameter and body
- Interpret the body in an environment extended with
  - The argument bound to the parameter
  - The entire object's value bound to `'this`

Passing the self reference

- Object can refer to itself in the method

Passing the self reference

- Object can refer to itself in the method
  - Access its fields

Passing the self reference

- Object can refer to itself in the method
  - Access its fields
  - Call other methods

Passing the self reference

- Object can refer to itself in the method
  - Access its fields
  - Call other methods
  - Pass itself as an argument

Passing the self reference

- Object can refer to itself in the method
  - Access its fields
  - Call other methods
  - Pass itself as an argument
- Like `self` in Python, `this` in Java/C++/JS

Example: OOP Lists

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information
  - Its length (e.g. whether empty or not)

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information
  - Its length (e.g. whether empty or not)
  - Its head (errors if empty)

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information
  - Its length (e.g. whether empty or not)
  - Its head (errors if empty)
  - Its tail (empty if empty)

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information
  - Its length (e.g. whether empty or not)
  - Its head (errors if empty)
  - Its tail (empty if empty)
- Can do `{get someList length}` or `{send someList tail 0}` on empty or cons, and will work in either case

Example: OOP Lists

```
{letvar empty {object {{length 0}}
                      {{head {x} errorNoEmptyListHead}
                      {tail {x} this}}} ;; Note the self reference
{letvar cons {fun {h t}
                  {object {length {+ 1 {get t length}}}
                          {{head {x} h}
                          {tail {x} t}}}}
}}
```

- Instead of having a type with two variants, each list carries its own information
  - Its length (e.g. whether empty or not)
  - Its head (errors if empty)
  - Its tail (empty if empty)
- Can do `{get someList length}` or `{send someList tail 0}` on empty or cons, and will work in either case
  - Return of methods carried around with the list

Example: Functions as Objects

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method
  - Equivalent to a first-class function

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method
  - Equivalent to a first-class function

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method
  - Equivalent to a first-class function

```
{fun {x} body}
```

- Can even do recursion

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method
  - Equivalent to a first-class function

```
{fun {x} body}
```

- Can even do recursion

Example: Functions as Objects

```
{object {} {call {x} body}}
```

- Object with a single “call” method
  - Equivalent to a first-class function

```
{fun {x} body}
```

- Can even do recursion

```
;; factorial with objects
{letvar fact
  {object {}
          {call {x}
            {if0 x
                 1
                 {* x {send this call {- x 1}}}}}}}
```