CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

April 15, 2020

Plan

- **Announcements**
  - HW8 is due today
  - HW9 was posted last Friday and is due Wednesday April 22th
- **Last time**
  - Moving from type schemes to types (Instantiation)
  - Moving from types to type schemes (Generalization)
- **Today**
  - Finish example where deriving type constraints
  - Objects
  - Message passing

Let Examples

THE UNIVERSITY OF ARIZONA
Computer Science

- **Questions**
  - What are types for cons and pair? cons : forall ‘a. ’a * ‘a list -> ‘a list, pair : forall ‘a, ‘b. ‘a * ‘b -> ’a * ‘b
  - Can the type for ys be of the form forall ’a . ‘a list? No. If not why? See 7
  - For the below, what are the types for s (??) and extend (??)?
  - Which of the below will correctly type check?

```
(lambda (ys)
   (let ([s (lambda (x) (cons x '()))])
      (pair (s 1) (s #t))))
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (pair (extend 1) (extend #t))))
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (extend 1)))
```

Let Examples

- **Question: What are the type constraints for the below?**

```
(lambda (ys)
   (let ([s (lambda (x) (cons x '()))])
      (pair (s 1) (s #t))))
```

```
???
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (pair (extend 1) (extend #t))))
```

```
???
```

From Type Scheme to Type

THE UNIVERSITY OF ARIZONA
Computer Science

VAR rule instantiates type schema with fresh and distinct type variables:

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots \alpha_n . \tau \qquad \alpha'_1, \ldots \alpha'_n \text{ are fresh and distinct}}{T, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \ldots \circ (\alpha_n \mapsto \alpha'_n))\tau} \quad (\textsc{Var})$$

$$\frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \sim \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \alpha} \quad (\textsc{Apply})$$

Object-Oriented Programming

- **Languages: JavaScript, Ruby, Java, C++, Python, …**
- **What is it about?**
  - Encapsulation
  - Higher-order programming
  - Dynamic dispatch
  - Inheritance
- **You can't touch things directly**
  - Every object is a black box
  - You can send it messages (What messages? Depends on protocol.)
  - Even objects of the same class can't see each other's fields
  - It's like everything is automatically generic

Example uSmalltalk code

THE UNIVERSITY OF ARIZONA
Computer Science

```
(val point-vectors (Dictionary new))
(point-vectors at:put: 'Center    (CoordPair withX:y: 0 0))
(point-vectors at:put: 'East      (CoordPair withX:y: 1 0))
(point-vectors at:put: 'Northeast (CoordPair withX:y: 1 1))
; ... six more definitions follow ...
```

- **Notes**
  - Each class is represented with an object
  - In (Dictionary new) expression, "Dictionary" is the receiver
  - "new" is a message name
- **Questions**
  - What are the receivers for the next three lines of code?
  - The message names?
  - What are parameters? Relationship to message names?

Key concepts of object-orientation

THE UNIVERSITY OF ARIZONA
Computer Science

- **Key concepts**
  - Never make a decision based on someone else's data
  - I know my own form of data (no one should ask me about it)
  - Instead ask me to tell you something (functional) or to do something (imperative)
  - I might have my parent do it (code reuse via inheritance)
- **Contrast with functional and procedural languages**
  - Functional and procedural: I will find out what form you are, and I will decide what to do and how to do it
  - Object oriented: I ask you to do something and you decide how to do it based on what form you are
- **That's why in the syntax, you'll see the receiver come first: the decider is first!**

Key mechanisms

- **Encapsulate: Private instance variables**
  – Only object knows its instance variables and can see them
  – This is the information hiding
- **Higher order: Code attached to objects and classes**
  – Construction code attached to the class object
  – Other code attached to the object instances
- **Dynamic dispatch (NEW for this semester)**
  – Caller doesn't know what function will be invoked; called a "method"
  – The caller is not in charge; the object is in charge

```
(val point-vectors (Dictionary new))
(point-vectors at:put: 'Center    (CoordPair withX:y: 0 0))
(point-vectors at:put: 'East      (CoordPair withX:y: 1 0))
(point-vectors at:put: 'Northeast (CoordPair withX:y: 1 1))
; ... six more definitions follow ...
```

Examples

- **Suppose I want to print every element of a list**
  - Functional program starts with, are you nil or cons?
  - Object-oriented program starts with, do something on every element

- **Arithmetic, say, multiplication of natural numbers**
  - Function program starts, are you zero or nonzero?
  - Object-oriented program says, answer a number that is 10 times yourself.

```
;; Example: list filter

-> (val ns (List withAll: '(1 2 3 4 5)))
List( 1 2 3 4 5 )
-> (ns filter: [block (n) ((n mod: 2) = 0)])
List( 2 4 )
```

Object-oriented iterations: messages

- **No interogation about form!**
- **Design process still works**
  - 1. Each method defined on a class
  - 2. Class determines
    - How object is formed (class method)
    - From what parts (instance variables)
    - How object responds to messages (instance method)
- **Each form of data gets its own methods!**

Filter implementation uses classes

- **Class determines how object responds: method defined on the class**

- **Key classes in lists**
  - Instance of class Cons: a cons cell
  - Instance of class ListSentinel: end of list

```
(method filter: (_) self) ;; on ListSentinel

(method filter: (aBlock)  ;; on Cons
   ([aBlock value: car] ifFalse:ifTrue:
        {(cdr filter: aBlock)}
        {(((Cons new) car: car)
                      cdr: (cdr filter: aBlock))}))
```

List filtering via iteration

- Use the imperative way
- Functional iteration: forms of data
- Iteration in Scheme: ask value about form

```
(define app (f xs)
  (if (null? xs)
      'do-nothing
      (begin
         (f (car xs))
         (app f (cdr xs)))))
```

Object-oriented iteration: dynamic

- **Instead of (app f xs) we have**

```
(xs do: f-block)
```

- **Which means for each element x in xs send**

- **Example: iteration**

```
-> (val ms (ns filter: [block (n) ((n mod: 2) = 0)]))
List( 2 4 )

-> (ms do: [block (m) ('element print) (space print)
                      ('is print) (space print)
                      (m println)])
element is 2
element is 4
nil
->
```

Implementing iteration

- **What happens if we send “do: f” to an empty list?**

```
(method do: (aBlock) nil)   ; nil is a global object
```

- **What happens if we send ”do: f” to a cons cell?**

```
(method do: (aBlock)
    ; car and cdr are "instance variables"
    (aBlock value: car)
    (cdr do: aBlock))
```

List selection by iteration

- **Example: method**

- **Like filter, but works with more collections like arrays and sets**

```
-> (val ns (List withAll: '(1 2 3 4 5)))
List( 1 2 3 4 5 )

-> (ns select: [block (n) (0 = (n mod: 2))])
List( 2 4 )

->
```

`select:` dispatches to class `Collection`

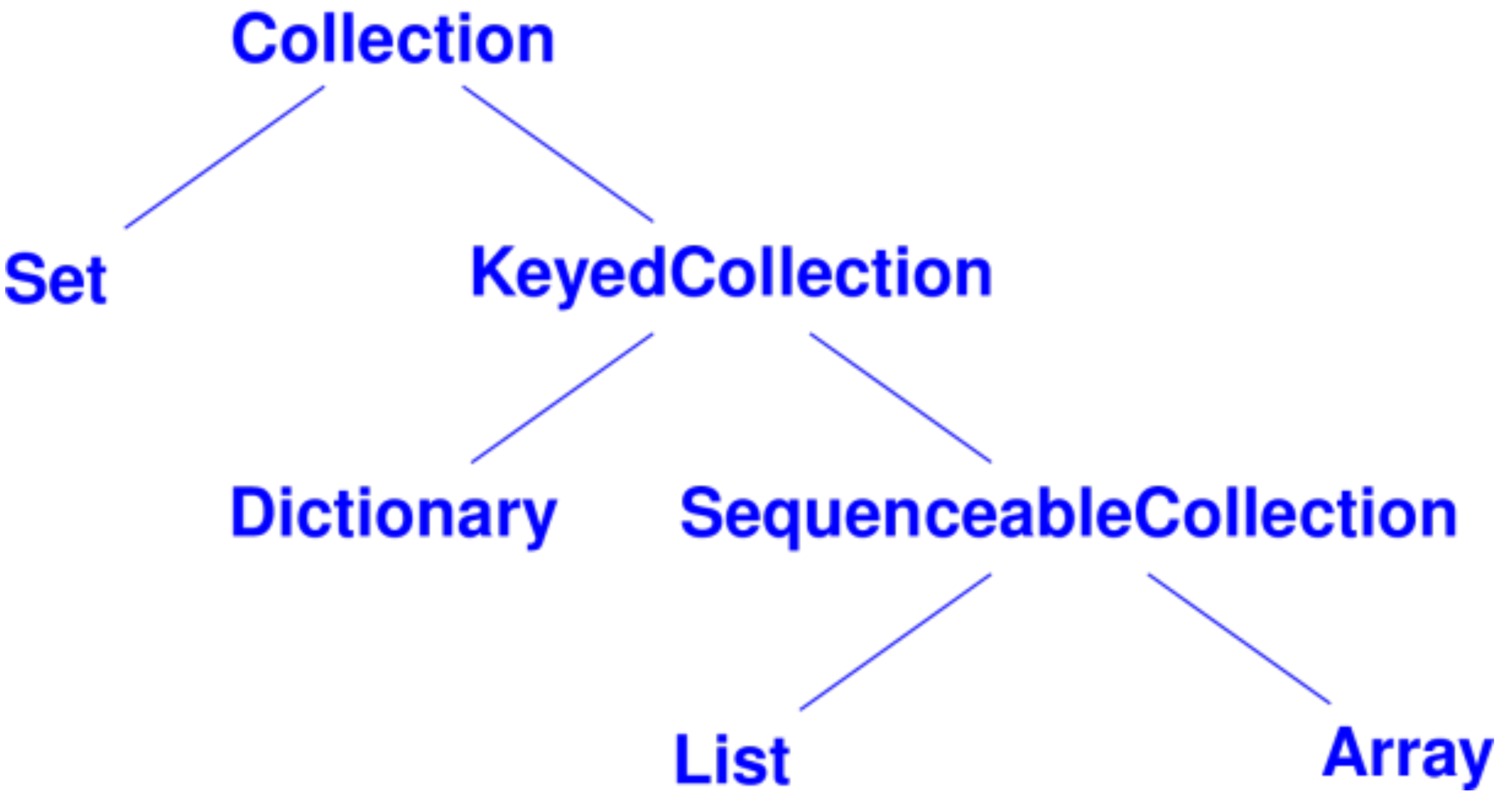
`List` says, "ask my parent to do it"

Parent implements classic imperative code:

```
(method select: (aBlock) [locals temp]
   (set temp ((self class) new))
   (self do: [block (x) ((aBlock value: x)
                          ifTrue: {(temp add: x)})])
   temp)
```

Name `self` receives message

"Collection hierarchy"



`select:` dispatches to class `Collection`

```
(method select: (aBlock) [locals temp]
    (set temp ((self class) new))
    (self do: [block (x) ((aBlock value: x)
                             ifTrue: {(temp add: x)})])
   temp)
```

Message	*Protocol*	*Dispatched to*
`class`	`Object`	`Object`
`new`	*Class*	`List`, others
`do:`	`Collection`	`List`, `Cons` (delegated)
`ifTrue:`	`Boolean`	`True` or `False`
`value`	`Block`	`Block`
`add:`	`Collection`	`List` (then `addLast:`, `insertAfter:`)

Mechanisms review

- Message send replaces function application
- Receiver appears first: it's in charge
- Respond to message by evaluating method
- Which method determined by an object's class

Six questions about Smalltalk

- **1. Values are objects**
  - Even true, 3, and "hello"
  - Even classes are objects
  - There are no function values, only methods on objects
- **2. Syntax**
  - Mutable variables
  - Message send
  - Sequential composition of mutations and message sends (side effects)
  - "Blocks" (really closures, objects and closures in one, used as continuations)
  - No `if` or `while`, These are implemented by passing continuations to Boolean objects.

Syntax comparison: Impcore

```
Exp = LITERAL of value
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
```

Syntax comparison: Smalltalk

```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of exp * name * exp list
    | BLOCK   of name list * exp list
```

Syntax comparison: Smalltalk

```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of exp * name * exp list
    | BLOCK   of name list * exp list
```

Message passing

- **Look at SEND**
  - Message identified by name (messaged are not values)
  - Always sent to a receiver
  - Optional arguments must match arity of message name (no other static checking)

- **Note: BLOCK and LITERAL are special objects**

Six questions about Smalltalk

- **3. Environments**
  - Name stands for a mutable cell containing an object:
    - Global variables
    - "Instance variables" (new idea, not yet defined)

- **4. Types**
  - There is no compile-time type system
  - At runtime, Smalltalk uses behavioral subtyping, known also as "duck typing"

Six questions about Smalltalk

- **5. Dynamic semantics**
  - Main rule is **method dispatch** (complicated)
  - The rest is familiar
- **4. The initial basis is enormous**
  - Why? To demonstrate the benefits of reuse, you need something big enough to reuse.

Summary of Key Ideas

- **Protocol determines behavioral subtyping**
  - The protocol of an object is the set of messages it understands.
  - Object A is a behavioral subtype of Object B if A understand all of the messages that B does in a compatible way.
  - Intuition: If A is a behavorial subtype of B, then A can be used in any context where B can be used.

- **Class-based object-orientation**
  - Object implementations determined by its class definition
  - So, each class implicitly defines the protocol for its objects and dynamic dispatch is determined by object's class
  - Code reuse by sending messages around like crazy

Summary of Key Ideas cont…

- **What's hard**
  - Encapsulation: abstraction function and invariant
  - Higher-order programming: everything is higer order
  - Dynamic dispatch: every call is to an unknown function (trust the contract)
  - Inheritance: big vocabulary, hard to work on one function in isolation
  - Net effect: algorithms ”smeared out” over many methods
- **What's great**
  - Each method is super simple
  - Cooperating-objects model
  - Reuse, reuse, reuse

OOP Demo

THE UNIVERSITY OF ARIZONA
Computer Science

- **Demo: Circle, Square, Triangle, with these methods:**

```
create: coordinate
moveTo: coordinate
draw
```

- **Instructions to student volunteers**
  - You have one instance variable, which represents the coordinate position at the "center" of the object (mutable state is back!)

```
(val o1 (Circle create: (CoordPair withX:y: 0 0)))
(val o2 (Square create: (CoordPair withX:y: 0 1)))
(val o3 (Triangle create: (CoordPair withX:y: 1 0)))
```

OOP Demo cont…

- **Messages**

```
(o1 moveTo: (CoordPair withX:y: -1 -1))
```

- **Instructions to student volunteers**
  - You have one instance variable, which represents the coordinate position at the "center" of the object (mutable state is back!)

```
(val o1 (Circle create: (CoordPair withX:y: 0 0)))
(val o2 (Square create: (CoordPair withX:y: 0 1)))
(val o3 (Triangle create: (CoordPair withX:y: 1 0)))
```