Example: list filter

-> (val ns (new List))
List( )
-> (addAll: ns '(1 2 3 4 5 6))
List( 1 2 3 4 5 6 )
-> (select: ns [block (n) (= 0 (mod: n 2))])
List( 2 4 6 )
select: dispatches to class Collection

Classic imperative paradigm:

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

Name self receives message
Iteration in Scheme:

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

Functional code: forms of data
Object-oriented code: dynamic dispatch

Instead of \((\text{app } f \; x s)\), we have

\((\text{do: } x s \; f\text{-block})\)

What happens if we send "do f" to the empty list?

What happens if we send "do f" to a cons cell?
Dynamic dispatch revealed

Sending do: to the empty list:

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

Sending do: to a cons cell:

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

What's missing? if!
“Collection hierarchy”

Collection

Set

KeyedCollection

Dictionary

SequenceableCollection

List

Array
select: **dispatches to class** Collection

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

<table>
<thead>
<tr>
<th>Message</th>
<th>Protocol</th>
<th>Dispatched to</th>
</tr>
</thead>
<tbody>
<tr>
<td>species</td>
<td>Collection</td>
<td>List</td>
</tr>
<tr>
<td>new</td>
<td>class</td>
<td>List, others</td>
</tr>
<tr>
<td>do:</td>
<td>Collection</td>
<td>List, Cons (delegated)</td>
</tr>
<tr>
<td>ifTrue:</td>
<td>Boolean</td>
<td>Boolean, del. True, False</td>
</tr>
<tr>
<td>value</td>
<td>block</td>
<td>primitive</td>
</tr>
<tr>
<td>add:</td>
<td>Collection</td>
<td>List (then addLast:, insertAfter:)</td>
</tr>
</tbody>
</table>
Church encoding with blocks

Blocks are closures

- [block (x) ...]
- Instead of [block ( ) ...], just {...}

Passed as continuations to Booleans

They are objects
Block Examples

-> (val twice [block (n) (+ n n)])
<Block>
-> (value twice 3)
6
-> (val delayed {(println 'hello) 42})
<Block>
<Block>
-> delayed
<Block>
-> (value delayed)
hello
42
Boolean example: minimum

\[\rightarrow (\text{val} \ x \ 10)\]
\[\rightarrow (\text{val} \ y \ 20)\]
\[\rightarrow (\text{ifTrue:ifFalse:} \ (\leq x \ y) \ \{x\} \ \{y\})\]
\[10\]
### Protocol for Booleans

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifTrue:ifFalse:</code> trueBlock falseBlock</td>
<td>Full conditional</td>
</tr>
<tr>
<td><code>ifTrue:</code> trueBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td><code>ifFalse:</code> falseBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td><code>&amp; aBoolean</code></td>
<td>Conjunction</td>
</tr>
<tr>
<td>`</td>
<td>aBoolean`</td>
</tr>
<tr>
<td><code>not</code></td>
<td>Negation</td>
</tr>
<tr>
<td><code>eqv: aBoolean</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>xor: aBoolean</code></td>
<td>Difference</td>
</tr>
<tr>
<td><code>and: altBlock</code></td>
<td>Short-circuit conjunction</td>
</tr>
<tr>
<td><code>or: altBlock</code></td>
<td>Short-circuit disjunction</td>
</tr>
</tbody>
</table>
Classes True and False

(class True Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (value trueBlock))
)
(class False Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (value falseBlock))
)

What happens if ifTrue: is sent to true?
Protocol for Booleans

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifTrue:ifFalse:</code></td>
<td>Full conditional</td>
</tr>
<tr>
<td><code>trueBlock</code></td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td><code>falseBlock</code></td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td><code>&amp; aBoolean</code></td>
<td>Conjunction</td>
</tr>
<tr>
<td>`</td>
<td>aBoolean`</td>
</tr>
<tr>
<td><code>not</code></td>
<td>Negation</td>
</tr>
<tr>
<td><code>eqv: aBoolean</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>xor: aBoolean</code></td>
<td>Difference</td>
</tr>
<tr>
<td><code>and: altBlock</code></td>
<td>Short-circuit conjunction</td>
</tr>
<tr>
<td><code>or: altBlock</code></td>
<td>Short-circuit disjunction</td>
</tr>
</tbody>
</table>
**ifTrue:** message dispatched to class **Boolean**

(class Boolean Object ()
   (method ifTrue:ifFalse: (trueBlock falseBlock)
      (subclassResponsibility self))
   (method ifTrue: (trueBlock)
      (ifTrue:ifFalse: self trueBlock {}))
   ...
)

*Message sent to self starts over (with class of receiver)*
Dispatching to True

(class True Boolean ()
    (method ifTrue:ifFalse: (trueBlock falseBlock)
        (value trueBlock))
; all other methods are inherited
)
Your turn: not

What should not look like?
  • Implemented on what class?
  • With what method definition?
Implementing `not`

```plaintext
(class Boolean Object ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (subclassResponsibility self))
  (method ifTrue: (trueBlock)
    (ifTrue:ifFalse: self trueBlock {}))
  (method not ()
    (ifTrue:ifFalse: self {false} {true})))
...
)
```
Inheritance for Booleans

Boolean is abstract class
  • Instances of True and False only

Method ifTrue:ifFalse: defined on True and False

All others defined on Boolean
Each class has one of two roles

Abstract class
• Meant to be inherited from
• Some (> 0) subclass Responsibility methods
• Examples: Boolean, Shape, Collection

Regular (“concrete”) class
• Meant to be instantiated
• No subclass Responsibility methods
• Examples: True, Triangle, List
Syntax comparison: Impcore

\[
\text{Exp} = \text{LITERAL of value} \\
| \text{VAR} \quad \text{of name} \\
| \text{SET} \quad \text{of name * exp} \\
| \text{IF} \quad \text{of exp * exp * exp} \\
| \text{WHILE} \quad \text{of exp * exp} \\
| \text{BEGIN} \quad \text{of exp list} \\
| \text{APPLY} \quad \text{of name * exp list}
\]
Syntax comparison: Smalltalk

\[ \text{Exp} = \text{LITERAL} \text{ of rep} \]
\[ \quad | \quad \text{VAR} \text{ of name} \]
\[ \quad | \quad \text{SET} \text{ of name} \ast \text{exp} \]
\[ \quad | \quad \text{IF} \text{ of exp} \ast \text{exp} \ast \text{exp} \]
\[ \quad | \quad \text{WHILE} \text{ of exp} \ast \text{exp} \]
\[ \quad | \quad \text{BEGIN} \text{ of exp list} \]
\[ \quad | \quad \text{APPLY} \text{ of name} \ast \text{exp list} \]
\[ \quad | \quad \text{SEND} \text{ of name} \ast \text{exp} \ast \text{exp list} \]
\[ \quad | \quad \text{BLOCK} \text{ of name list} \ast \text{exp list} \]
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 name * exp * exp list
   | BLOCK of name list * exp list