Functional Smalltalk

Dave Mason
Toronto Metropolitan University

Value

I’m going to start with a quote from Kent Beck

Value

Software creates value 2 ways:

- *What it does today*
- *What new things we can make it do tomorrow*

Value

Software creates value 2 ways:

- *What it does today*
- *What new things we can make it do tomorrow*

Value

Software creates value 2 ways:

- *What it does today*
- *What new things we can make it do tomorrow*

Value

Smalltalk creates value 2 ways:

- *What it does today*
- *What new things we can make it do tomorrow*

Functional Smalltalk

Smalltalk already has many functional features

- extensions by syntax
- extensions by class

Functional Smalltalk

Smalltalk already has many functional features

- extensions by syntax
- extensions by class

Syntax: Functional Programming

- Smalltalk has always had blocks - needed full closures
- CompileWithCompose in Pharo-Functional repo
- leverages class-bounded alternative compiler
- just syntactic sugar - more succinct
- all are upward compatible as they are currently syntax errors

Syntax: Functional Programming

- Smalltalk has always had blocks - needed full closures
- CompileWithCompose in Pharo-Functional repo
- leverages class-bounded alternative compiler
- just syntactic sugar - more succinct
- all are upward compatible as they are currently syntax errors

Syntax: Functional Programming

- Smalltalk has always had blocks - needed full closures
- CompileWithCompose in Pharo-Functional repo
- leverages class-bounded alternative compiler
- just syntactic sugar - more succinct
- all are upward compatible as they are currently syntax errors

Syntax: Functional Programming

- Smalltalk has always had blocks - needed full closures
- CompileWithCompose in Pharo-Functional repo
- leverages class-bounded alternative compiler
- just syntactic sugar - more succinct
- all are upward compatible as they are currently syntax errors

Syntax: Functional Programming

- Smalltalk has always had blocks - needed full closures
- CompileWithCompose in Pharo-Functional repo
- leverages class-bounded alternative compiler
- just syntactic sugar - more succinct
- all are upward compatible as they are currently syntax errors

Compose/pipe/parrot operator

- very convenient to pass result of one expression to another without parentheses
- particularly convenient in PharoJS for e.g. D3

Compose/pipe/parrot operator

- very convenient to pass result of one expression to another without parentheses
- particularly convenient in PharoJS for e.g. D3

```
foo
"  self  new  foo  >>>  42  "
   ↑ 17 negated
      :> min: -53
      :> abs
      :> < 100
      :> and: [ 4 > 2 ]
      :> and: [ 5 < 10 ]
      :> ifTrue: [ 42 ] ifFalse: [ 99 ]
```

Compose/pipe/parrot operator

- very convenient to pass result of one expression to another without parentheses
- particularly convenient in PharoJS for e.g. D3

```
foo
" self new foo >>> 42 "
   ↑ 17 negated
      :> min: -53
      :> abs
      :> < 100
      :> and: [ 4 > 2 ]
      :> and: [ 5 < 10 ]
      :> ifTrue: [ 42 ] ifFalse: [ 99 ]
```

... Compose/pipe/parrot operator

The precedence is the same as cascade, so you can intermix them and could say something like:

```
x := OrderedCollection new
         add: 42;
         add: 17;
         yourself
        :> collect: #negated
        :> add: 35;
         add: 99;
      yourself
        :> with: #(1 2 3 4) collect: [:l :r| l+r ]
        :> max
```

... Compose/pipe/parrot operator

If you don’t want to use the alternate compiler (and get the `:>` syntax) PharoFunctional also provides a `chain` method on Object that supports chaining using cascades (unfortunately quite a bit slower because it requires a DNU and perform for each chained message):

```
foo
   " self new foo >>> 42 "
   ↑ 17 chain
           negated
      ; min: -53
      ; abs
      ; < 100
      ; and: [ 4 > 2 ]
      ; and: [ 5 < 10 ]
      ; ifTrue: [ 42 ] ifFalse: [ 99 ]
```

Point-free programming style

- popular style of functional programming
- composing functions to build up operations with implicit parameters
- various “combinators” that recognize patterns in these compositions
- in Smalltalk this is composing symbols and blocks
- e.g.

```
isPalindrome := #reverse <|> #= .
isPalindrome value: 'madam'
```

Point-free programming style

- popular style of functional programming
- composing functions to build up operations with implicit parameters
- various “combinators” that recognize patterns in these compositions
- in Smalltalk this is composing symbols and blocks
- e.g.

```
isPalindrome := #reverse <|> #= .
isPalindrome value: 'madam'
```

Point-free programming style

- popular style of functional programming
- composing functions to build up operations with implicit parameters
- various “combinators” that recognize patterns in these compositions
- in Smalltalk this is composing symbols and blocks
- e.g.

```
isPalindrome := #reverse <|> #= .
isPalindrome value: 'madam'
```

Point-free programming style

- popular style of functional programming
- composing functions to build up operations with implicit parameters
- various “combinators” that recognize patterns in these compositions
- in Smalltalk this is composing symbols and blocks
- e.g.

```
isPalindrome := #reverse <|> #= .
isPalindrome value: 'madam'
```

Point-free programming style

- popular style of functional programming
- composing functions to build up operations with implicit parameters
- various “combinators” that recognize patterns in these compositions
- in Smalltalk this is composing symbols and blocks
- e.g.

```
isPalindrome := #reverse <|> #= .
isPalindrome value: 'madam'
```

Expressions as unary or binary messages

To use point-free style, it is very convenient to have a more succinct syntax for applying them

```
x (...)
x (...) + y
x (...): y
x (#sort <|> #=)
```

Converts to.

```
([...] value: x)
([...] value: x) + y
([...] value: x value: y)
((#sort <|> #=) value: x)
```

Blocks as unary or binary messages

You can do the same with unary or binary blocks. Because we know the arity of blocks the trailing : isn't used for block operators

```
x [:w|...]
x [:w:z|...] y
```

becomes

```
([:w|...] value: x)
([:w:z|...] value: x value: y)
```

Initializing local variables at point of declaration

Even in functional languages where mutation is possible, it is rarely used. Instead programming is by a sequence of definitions, which always have a value. I personally very much miss this in Smalltalk.

```
| w x := 42. y = x+5. z a |
```

is legal, but

```
| x := 42. y = x+5. z = 17 |
```

isn't.

Collection literals

Arrays have a literal syntax `{1 . 2 . 3}`, but other collections don't. This extension recognizes `:className` immediately after the `{` and translates, e.g.

```
{:Set 3 . 4 . 5 . 3}
{:Dictionary #a->1 . #b->2}
{:Set 1 . 2 . 3 . 4 . 5 . 6 . 7}
```

to

```
Set with: 3 with: 4 with: 5 with: 3
Dictionary with: #a->1 with: #b->2
Set withAll: {1 . 2 . 3 . 4 . 5 . 6 . 7}
```

Destructuring collections

There isn't a convenient way to return multiple values from a method, or even to extract multiple values from a collection. For example:

```
:| a b c | := some-collection
```

destructures the 3 elements of a SequenceableCollection or would extract the value of keys `#a #b` etc. if it was a Dictionary, with anything else being a runtime error. This is conveniently done by converting that to:

```
([:temp|
   a := temp firstNamed: #a.
   b := temp secondNamed: #b.
   c := temp thirdNamed: #c.
   temp] value: some-collection)
```

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Classes: Functional Programming

PharoFunctional adds several new classes and a variety of extension methods to facilitate functional programming.

- `curry:` and `@@`
- `value:`, `value:value:` and `cull`, etc. for Symbol
- `map:`, `map:map:` for BlockClosure and Symbol
- `<*>` and other combinators for BlockClosure and Symbol
- `nilOr:`, `emptyOrNilOr:`
- Slice, Pair and Tuple, ZippedCollection
- `zip:`, `>===<`
- `iota`
- many algorithms on collections: `rotate:`, `slide:`, `product`, `allEqual`, `unique`, `isUnique`, `groupByRunsEqual:`, `groupByRunsTrue:`

Demo

Using CompileWithCompose

```
Metacello new
    baseline: ’PharoFunctional’;
    repository: ’github://dvmason/Pharo-Functional:m
    load: #compiler
```

Then for any class heirarchy, add a trait:

```
RBScannerTest subclass: #ComposeExampleTest
   uses: ComposeSyntax
   instanceVariableNames: ’’
   classVariableNames: ’’
   package: ’CompileWithCompose-Tests’
```

Or, on the class-side define the following method:

```
compilerClass
   " Answer a compiler class appropriate for source
   ↑ ComposeCompiler
```

You can use this second approach if you want to add it to the entire image (including in playgrounds), by defining this in Object class.

Conclusions

- Smalltalk already has the fundamentals for functional programming
- some simple syntactic suger can make it a lot more pleasant
- I would love it if some of these became mainstream (with no backward compatibility issues)
- in the meantime, anyone can add this to their Pharo
- the compiler tweaks are not hard for other Smalltalks to implement

Conclusions

- Smalltalk already has the fundamentals for functional programming
- some simple syntactic suger can make it a lot more pleasant
- I would love it if some of these became mainstream (with no backward compatibility issues)
- in the meantime, anyone can add this to their Pharo
- the compiler tweaks are not hard for other Smalltalks to implement

Conclusions

- Smalltalk already has the fundamentals for functional programming
- some simple syntactic suger can make it a lot more pleasant
- I would love it if some of these became mainstream (with no backward compatibility issues)
- in the meantime, anyone can add this to their Pharo
- the compiler tweaks are not hard for other Smalltalks to implement

Conclusions

- Smalltalk already has the fundamentals for functional programming
- some simple syntactic suger can make it a lot more pleasant
- I would love it if some of these became mainstream (with no backward compatibility issues)
- in the meantime, anyone can add this to their Pharo
- the compiler tweaks are not hard for other Smalltalks to implement

Conclusions

- Smalltalk already has the fundamentals for functional programming
- some simple syntactic suger can make it a lot more pleasant
- I would love it if some of these became mainstream (with no backward compatibility issues)
- in the meantime, anyone can add this to their Pharo
- the compiler tweaks are not hard for other Smalltalks to implement

Questions?

@DrDaveMason dmason@ryerson.ca

https://github.com/dvmason/Pharo-Functional