

# #OccupyLanguageDesign

Give the power of language extension to the 99%!



So, here's the problem. Right now it is too difficult for the mainstream programmer to extend their language. All the power of language extension is controlled by the 1% -- the language designers! And what we want to do is give the power of language extension back the the 99%.

# Growing a Language (Guy Steele)

A photograph of Guy Steele, a man with short brown hair, wearing a dark suit jacket, a light pink shirt, and a red patterned tie. He is standing at a podium with a microphone, looking slightly to his left. On the podium, there is a white cup, a glass, and a blue folder with a white card. A purple speech bubble with a white border is overlaid on the left side of the image, containing the text "Let my language grow!".

Let my language grow!

So this idea has been around a while. Guy Steele gave a famous talk a number of years ago about growing a language. He encouraged language designers to build their languages with the goal of enabling grassroots language innovation.

Some languages  
are **extensible**...

:-)

Lisp, Scheme (Macros!)

Smalltalk, Python, Ruby  
(Pure Objects!)

...others not  
so much

:-(

JavaScript, Java

Some languages do a really good job at this. For example Lisp and Scheme have macros which allow you to do crazy things with syntax. In Smalltalk, Python, and Ruby everything is an object which make extension pretty easy. In Smalltalk you can even create your own control structures. But other languages like JavaScript and Java are not quite so extensible.

# Radical Split

Primitive Values



Objects

And the reason they are not so extensible is that these languages have a radical split between primitive values and objects. Objects allow you to extend the language but you can't add new primitives. And object don't interact well with primitives.

# Radical Split

Primitive Values

$x = 5$

$y = 4$

$z = x + y$



Objects

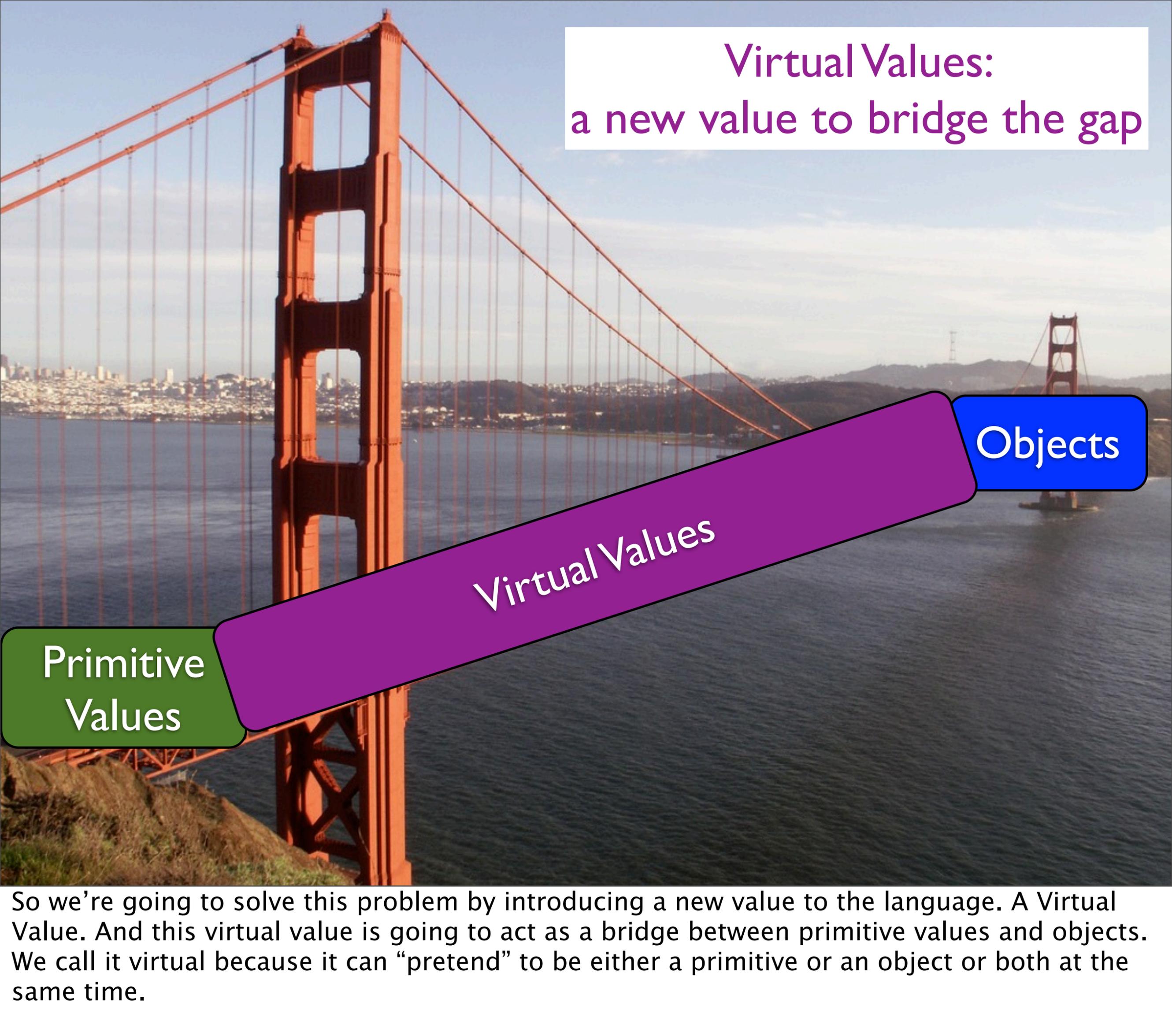
$x = \text{Complex}(5, 1)$

$y = \text{Complex}(4, 2)$

~~$z = x + y$~~

$z = x.\text{plus}(y)$

For example, consider adding a Complex number to the language. You can do this by creating a new Complex object with the appropriate logic. But the built-in “add” operator is only defined for primitive numbers so you can’t use “add” for the Complex object. So existing code that expects numbers and uses the “add” operator won’t work with the new complex object.

The background of the slide is a photograph of the Golden Gate Bridge in San Francisco, California. The bridge's iconic orange-red towers and suspension cables are visible against a clear blue sky. The water of the bay is a deep blue, and the city skyline is visible in the distance.

Virtual Values:  
a new value to bridge the gap

Primitive  
Values

Virtual Values

Objects

So we're going to solve this problem by introducing a new value to the language. A Virtual Value. And this virtual value is going to act as a bridge between primitive values and objects. We call it virtual because it can "pretend" to be either a primitive or an object or both at the same time.

# Behavioral Intercession

**add** behavior

$x + y$

**set** behavior

$x.foo = 42$

...

...

Virtual values work by using the technique of behavioral intercession. The idea of behavioral intercession is that it allows programmers to write custom logic for every behavior that occurs on a value. Behaviors are things like “add” (the plus operator) and “set” (setting a property on an object). You can probably fill in the rest of the behaviors.

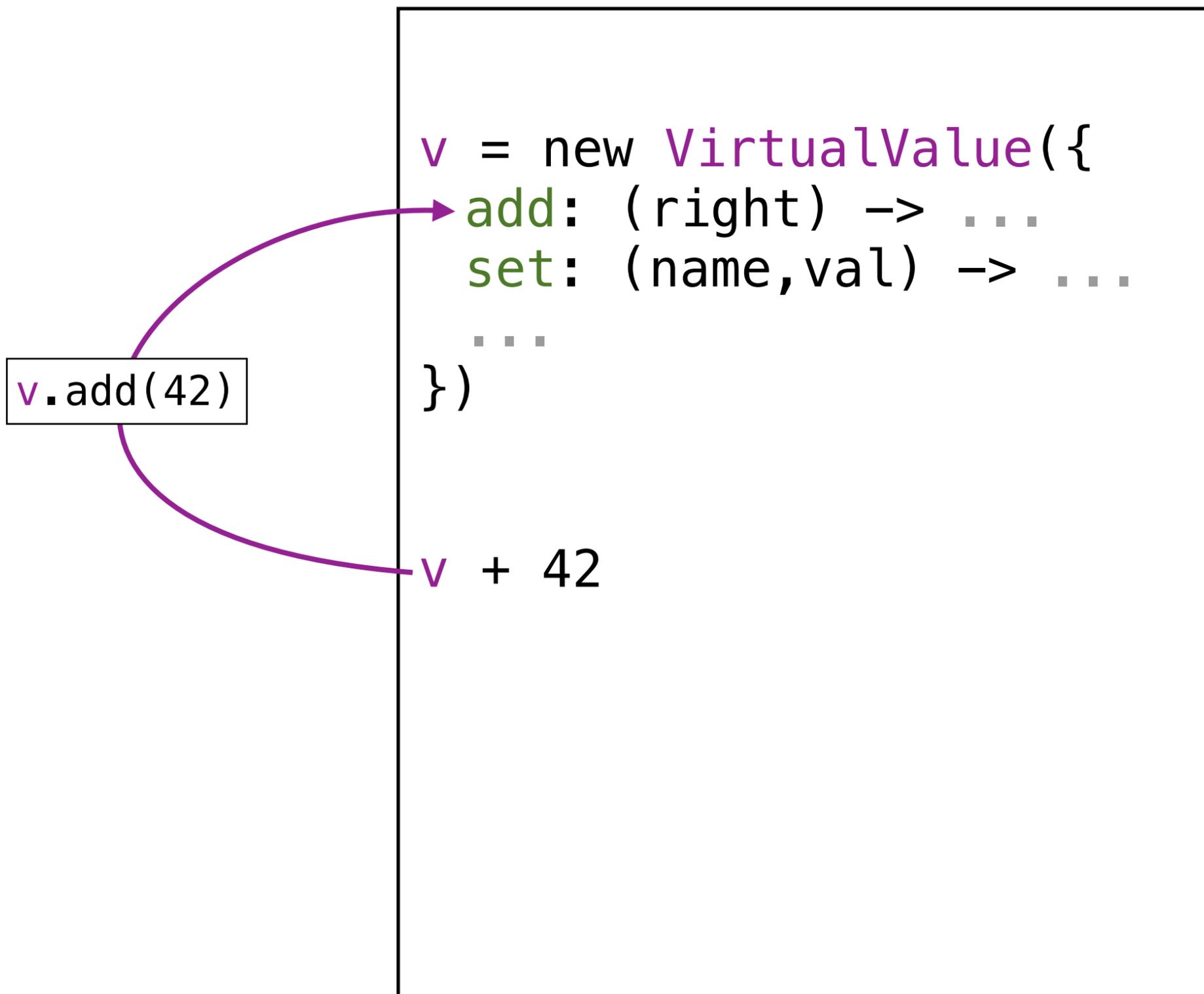
# Creating a Virtual Value

```
v = new VirtualValue({  
  add: (right) -> ...  
  set: (name, val) -> ...  
  ...  
})
```

```
...
```

So programmers create virtual values with a handler. The handler is a collection of traps and each trap is a function that corresponds to a particular behavior. Each of the traps have the custom logic for handling the appropriate behavior.

# Using a Virtual Value



So what happens is the runtime system converts behaviors on virtual values to calls to a trap. For example the “add” operator here is converted to a call to the “add” trap. This is the key idea: virtual values trap on behaviors and allow the programmer to add custom logic.

# Virtual Values are powerful!

Numeric types  
Units of measure  
Contracts  
Taint analysis

Revocable membranes  
Lazy Evaluation  
FRP  
Partial Evaluation  
...

Once we've added virtual values to our language we can now do all kinds of extensions. Adding new numeric types like Complex numbers, units of measure, contracts, and so on. None of this was possible before in a language like JavaScript until we added virtual values.

# Related Work

handler = {

get:	...	JavaScript Proxies
set:	...	contracts      nonProxy
call:	...	membranes

geti:	...	
seti:	...	Virtual Values
unary:	...	complex      taint tracking
left:	...	units          lazy evaluation
right:	...	
test:	...	

}

T.V. Cutsem and M. S. Miller. *Proxies: Design principles for robust object-oriented intercession APIs*

Virtual values are based on some prior work on JavaScript proxies (this is work by Tom van Cutsem and Mark Miller). JS proxies do behavior intercession but only for objects and functions. And we show how we can extend that to include primitive values. Doing this enables a bunch of extensions that weren't possible with just JS proxies.

# OOPSLA paper: *Virtual Values* for Language Extension

Thomas H. Austin, Tim Disney, Cormac Flanagan

semantics + example extensions + implementation

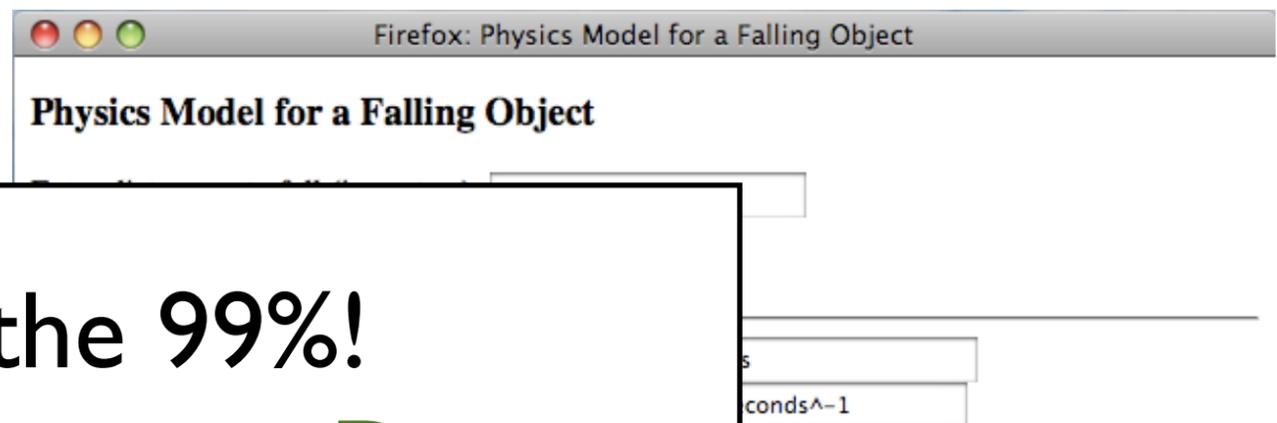
Figure 2:  $\lambda_{\text{proxy}}$  Semantics

Runtime Syntax:

```
r ::= c | a |  $\lambda x. e$ 
v, w, h ::= r | proxy a h
e ::= ... | a
H ::= Address  $\rightarrow_p$  (String  $\rightarrow_p$  Value)
E ::= • e | v • | if • e e | uop • | • bop e | v bop •
      | proxy • e | proxy v • | isProxy • | unProxy
      | •[c] | v[•] | •[e] := e | v[•] := e | v[u] := • | {  $\bar{x}$  }
```

Evaluation Rules:

$H, (\lambda x. e) v \rightarrow H, e[x := v]$	
$H, \{\bar{x} \bar{v}\} \rightarrow H[a := \{\bar{x} \bar{v}\}], a$	$a \notin \bar{x}$
$H, a[s] \rightarrow H, v$	$s \in a$
$H, a[s] \rightarrow H, \text{false}$	$s \notin a$
$H, a[s] := v \rightarrow H', v$	$H' \neq H$
$H, uop r \rightarrow H, \delta(uop, r)$	
$H, r_1 bop r_2 \rightarrow H, \delta(bop, r_1, r_2)$	
$H, \text{if } r e_1 e_2 \rightarrow H, e_1$	$r \neq \text{false}$
$H, \text{if } \text{false } e_1 e_2 \rightarrow H, e_2$	
$H, \text{isProxy}(\text{proxy } a h) \rightarrow H, \text{true}$	[ISPROXY]
$H, \text{isProxy } r \rightarrow H, \text{false}$	[NOTPROXY]
$H, \text{unProxy } a(\text{proxy } a h) \rightarrow H, h$	[UNPROXY]
$H, \text{unProxy } a v \rightarrow H, \text{false}$	$v \neq (\text{proxy } a h)$ [UNPROXYFALSE]
$H, (\text{proxy } a h) v \rightarrow H, h.\text{call } v$	[CALLPROXY]
$H, (\text{proxy } a h)[w] \rightarrow H, h.\text{getr } w$	[GETRPROXY]
$H, r[\text{proxy } a h] \rightarrow H, h.\text{geti } r$	[GETIPROXY]
$H, (\text{proxy } a h)[w] := v \rightarrow H, (h.\text{setr } w v); v$	[SETRPROXY]
$H, r[\text{proxy } a h] := v \rightarrow H, (h.\text{seti } r v); v$	[SETIPROXY]
$H, uop(\text{proxy } a h) \rightarrow H, h.\text{unary "uop"}$	[UNARYPROXY]
$H, (\text{proxy } a h) bop v \rightarrow H, h.\text{left "bop" } v$	[LEFTPROXY]
$H, r bop(\text{proxy } a h) \rightarrow H, h.\text{right "bop" } r$	[RIGHTPROXY]
$H, \text{if } (\text{proxy } a h) e_1 e_2 \rightarrow H, \text{if } (h.\text{test}()) e_1 e_2$	[TESTPROXY]
$H, E[e] \rightarrow H', E'[e']$	$\text{if } H, e \rightarrow H', e'$ [CONTEXT]



Join the 99%!  
#OccupyLanguageDesign

Photo credits:

- <http://www.flickr.com/photos/kapkap/6189122974/>
- <http://www.flickr.com/photos/breun/2149454365/>

Tim Disney | @disnet  
[tim.disney@gmail.com](mailto:tim.disney@gmail.com)

So if you're interested in this check out a paper that was in this years OOPSLA. We have full semantics, some example extensions, and an implementation for JavaScript.

And I leave you with this: Join the 99% -- Occupy Language Design!