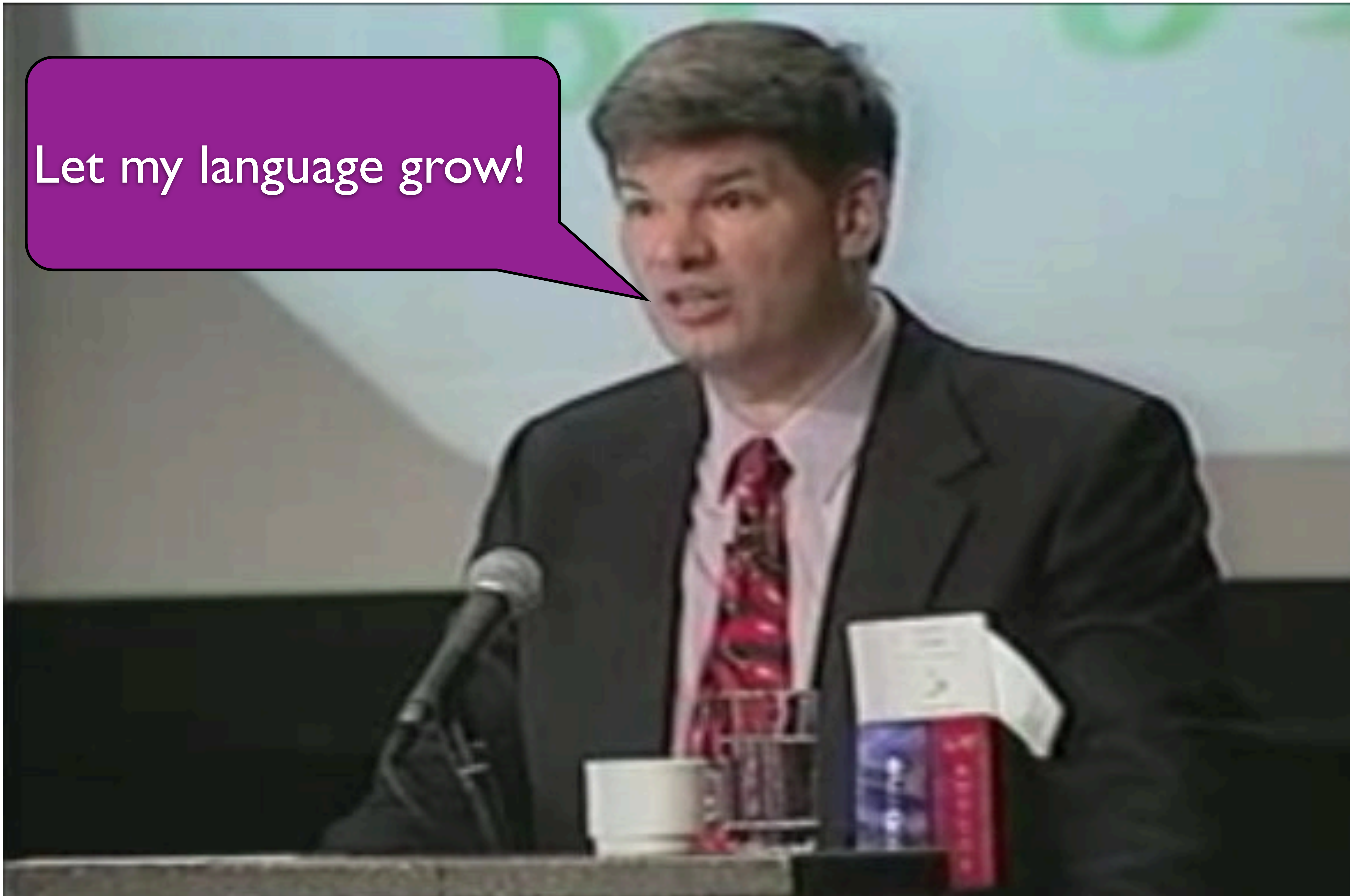
#OccupyLanguageDesign

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)



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...

...others not so much

:-)

Lisp, Scheme (Macros!)

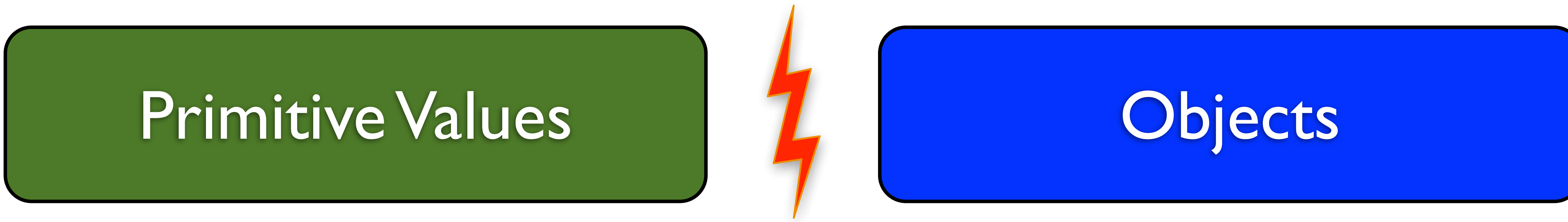
Smalltalk, Python, Ruby (Pure Objects!)

:-(

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



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 | **Objects**

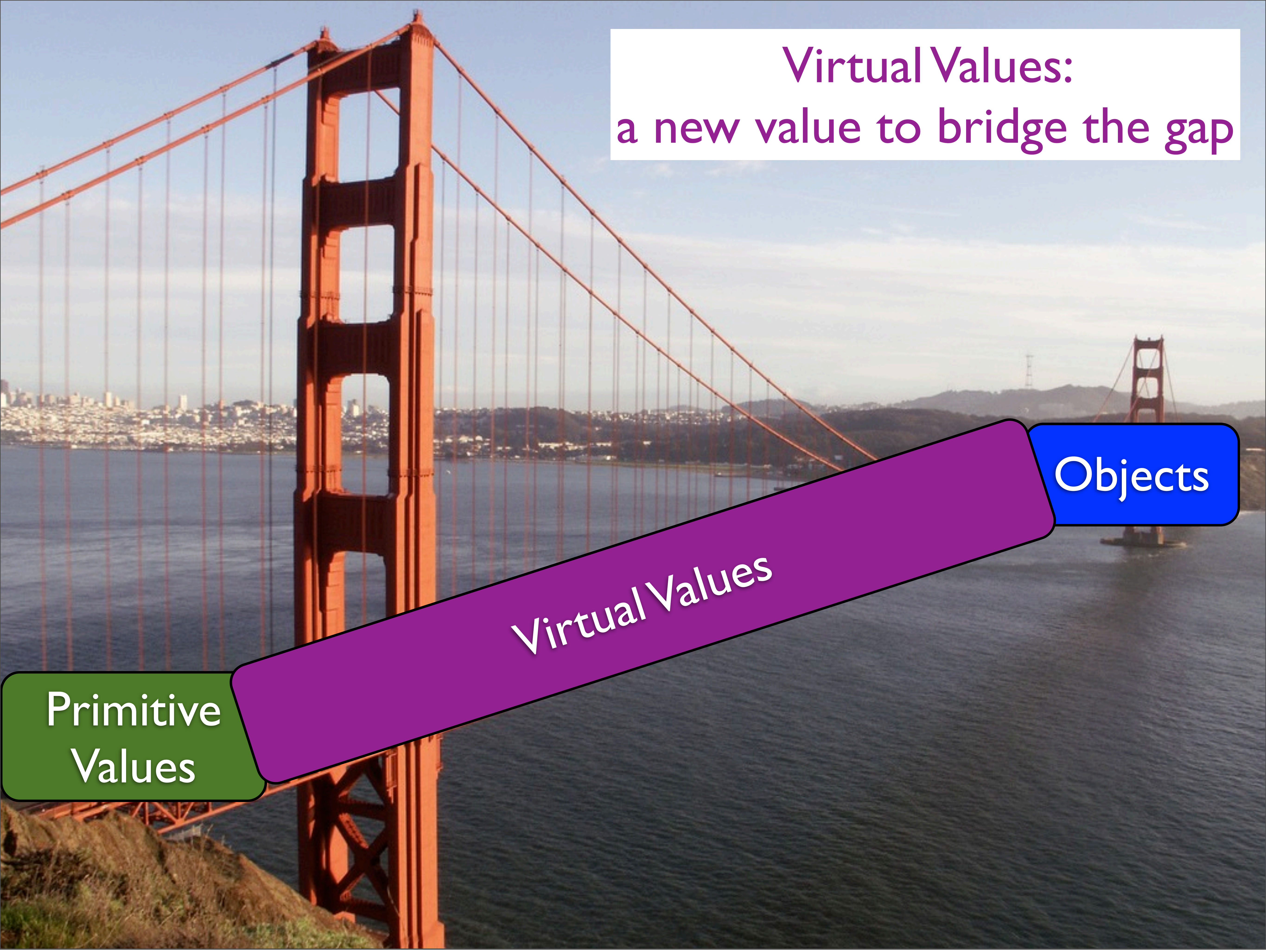
x = 5
y = 4

z = x + y

x = Complex(5,1)
y = Complex(4, 2)

~~z = x + y~~
z = x.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.



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 the 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

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

```
v.add(42)
```

```
v + 42
```

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. The is the key idea: virtual values trap on behaviors and allow the programmer to add custom logic.

Virtual Values are powerful!

Numeric types	Revocable membranes
Units of measure	Lazy Evaluation
Contracts	FRP
Taint analysis	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:    ...
  set:    ...
  call:   ...

  geti:   ...
  seti:   ...
  unary:  ...
  left:   ...
  right:  ...
  test:   ...
}
```

JavaScript Proxies

contracts
membranes
nonProxy

Virtual Values

complex
units
taint tracking
lazy evaluation

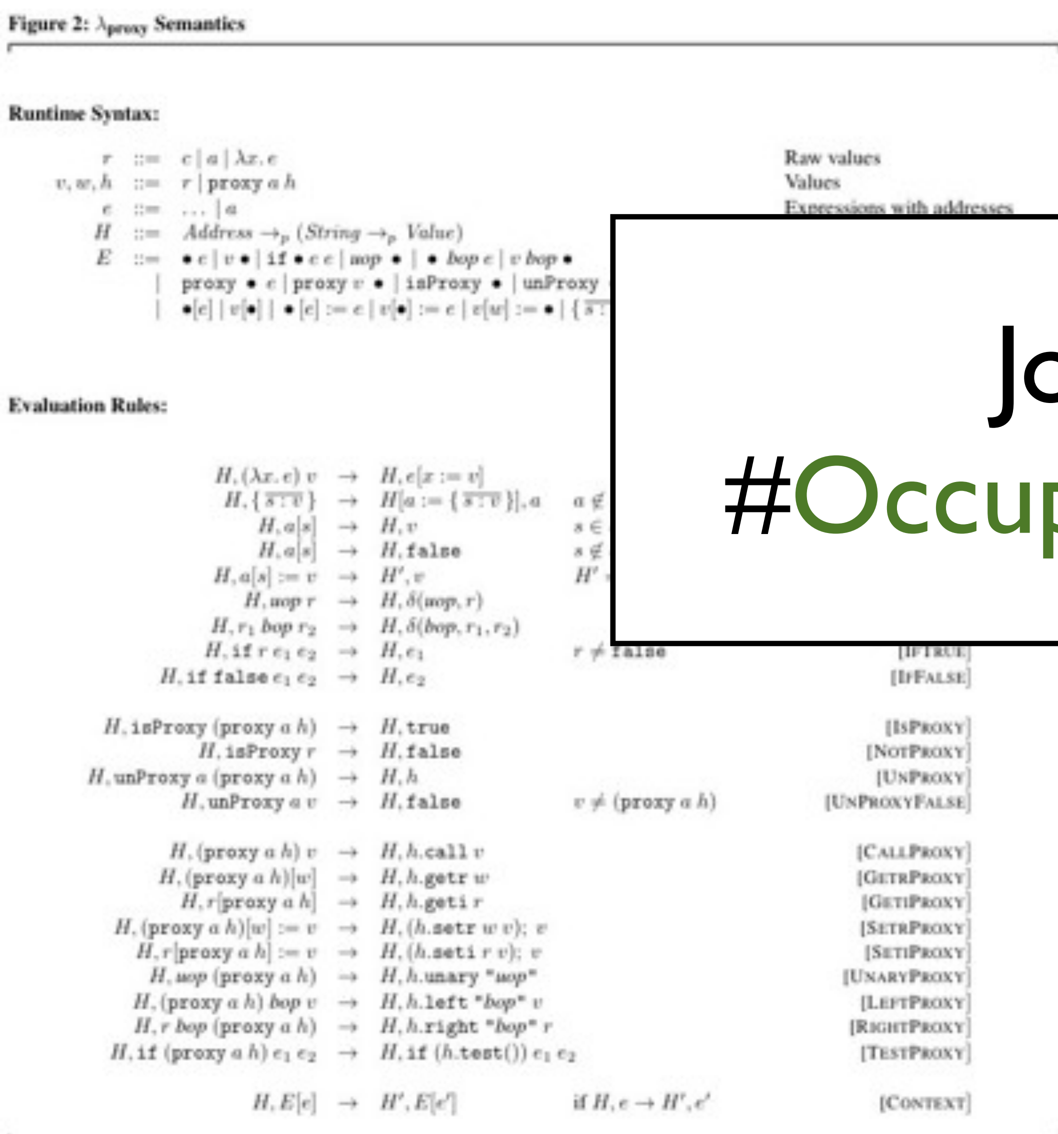
T.V. Cutsem and M. S. Miller. *Proxies: Design principles for robust object-oriented intercession APIs*

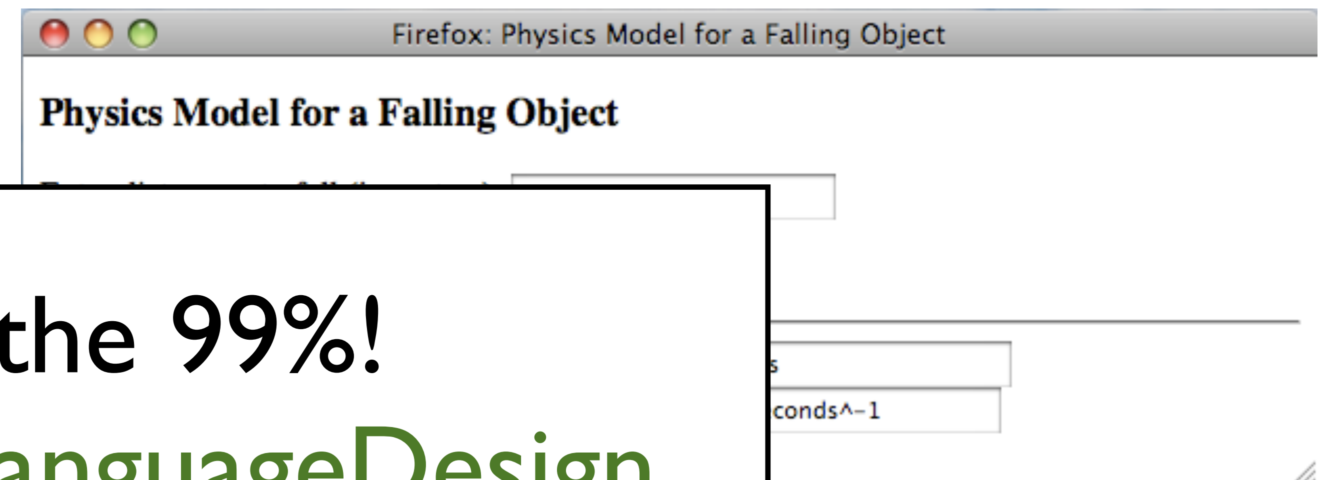
Virtual values are based on some prior work on JavaScript proxies (this is work by Tom van Cutsem and Mark Miller). JS proxies do behavior intercession but only for objects and functions. And we show how we can extend that to include primitive values. Doing this enables a bunch of extensions that weren't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

OOPSLA paper: *Virtual Values for Language Extension*

Thomas H. Austin, Tim Disney, Cormac Flanagan

semantics + example extensions + implementation





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

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!