Virtual Values for Language Extension

Thomas H. Austin, **Tim Disney**, Cormac Flanagan
University of California Santa Cruz

Virtual Values provide the extensibility of purely OO languages in mixed languages

extensibility: ability for user-programmers to change the language

$$x + y$$

How do we add complex numbers?

Extensibility in Python is clean

```
class Complex(object):

  def __init__(self, real, imag):
    self.r = real
    self.i = imag

  def __add__(self, other):
    return Complex(self.r + other.r,
                   self.i + other.i)
```

Everything is an object in Python

Extensibility in Python is clean

```
class Complex(object):

  def __init__(self, real, imag):
    self.r = real
    self.i = imag

  def __add__(self, other):
    return Complex(self.r + other.r,
                   self.i + other.i)
```

```
x = Complex(2, 1)
y = Complex(3, 1)

x + y
```

Everything is an object in Python

Extensibility in Python is clean

```
class Complex(object):

  def __init__(self, real, imag):
    self.r = real
    self.i = imag

  def __add__(self, other):
    return Complex(self.r + other.r,
                   self.i + other.i)
```

```
x = Complex(2, 1)
y = Complex(3, 1)

x + y
```

Everything is an object in Python

Extensibility in JavaScript is ugly

```
function Complex(real, imag) {
  this.r = real;
  this.i = imag;
}
Complex.prototype.plus(other) {
  return new Complex(this.r + other.r,
                     this.i + other.i);
}
```

JS has mixed objects and primitives so ugly

Extensibility in JavaScript is ugly

```
function Complex(real, imag) {
  this.r = real;
  this.i = imag;
}
Complex.prototype.plus(other) {
  return new Complex(this.r + other.r,
                     this.i + other.i);
}
```

```
var x = new Complex(2, 1);
var y = new Complex(3, 1);

x.plus(y);
```

JS has mixed objects and primitives so ugly

Extensibility in JavaScript is ugly

```
function Complex(real, imag) {
  this.r = real;
  this.i = imag;
}
Complex.prototype.plus(other) {
  return new Complex(this.r + other.r,
                     this.i + other.i);
}
```

```
var x = new Complex(2, 1);
var y = new Complex(3, 1);

x.plus(y);
```

JS has mixed objects and primitives so ugly

Even worse than ugly!

```
function matrixMult(a, b) { ... }

matrixMult([[-3,-8,3],
            [-2,1,4]])
```

Our complex extension doesn’t work with existing code.

Even worse than ugly!

```
function matrixMult(a, b) { ... }

matrixMult([[-3,-8,3],
            [-2,1,4]])


matrixMult([[new Complex(4,1), new Complex(2,1)],
            [new Complex(5,1), new Complex(8,1)]])
```

Our complex extension doesn't work with existing code.

Even worse than ugly!

```
function matrixMult(a, b) { ... }

matrixMult([[-3,-8,3],
            [-2,1,4]])

matrixMult([[new Complex(4,1), new Complex(2,1)],
            [new Complex(5,1), new Complex(8,1)]])
```

Our complex extension doesn't work with existing code.

x + y

vs.

```
x.plus(y)
```

How do we get nicety of purely OO in langs with primitives?
(1) change semantics to have everything be an object
    (but this would be a hard/impossible task...note that it hasn't been done in Java/JavaScript)
(2) we propose a targeted change: adding one new value

Virtual Values:

Virtualize the interface between code and data

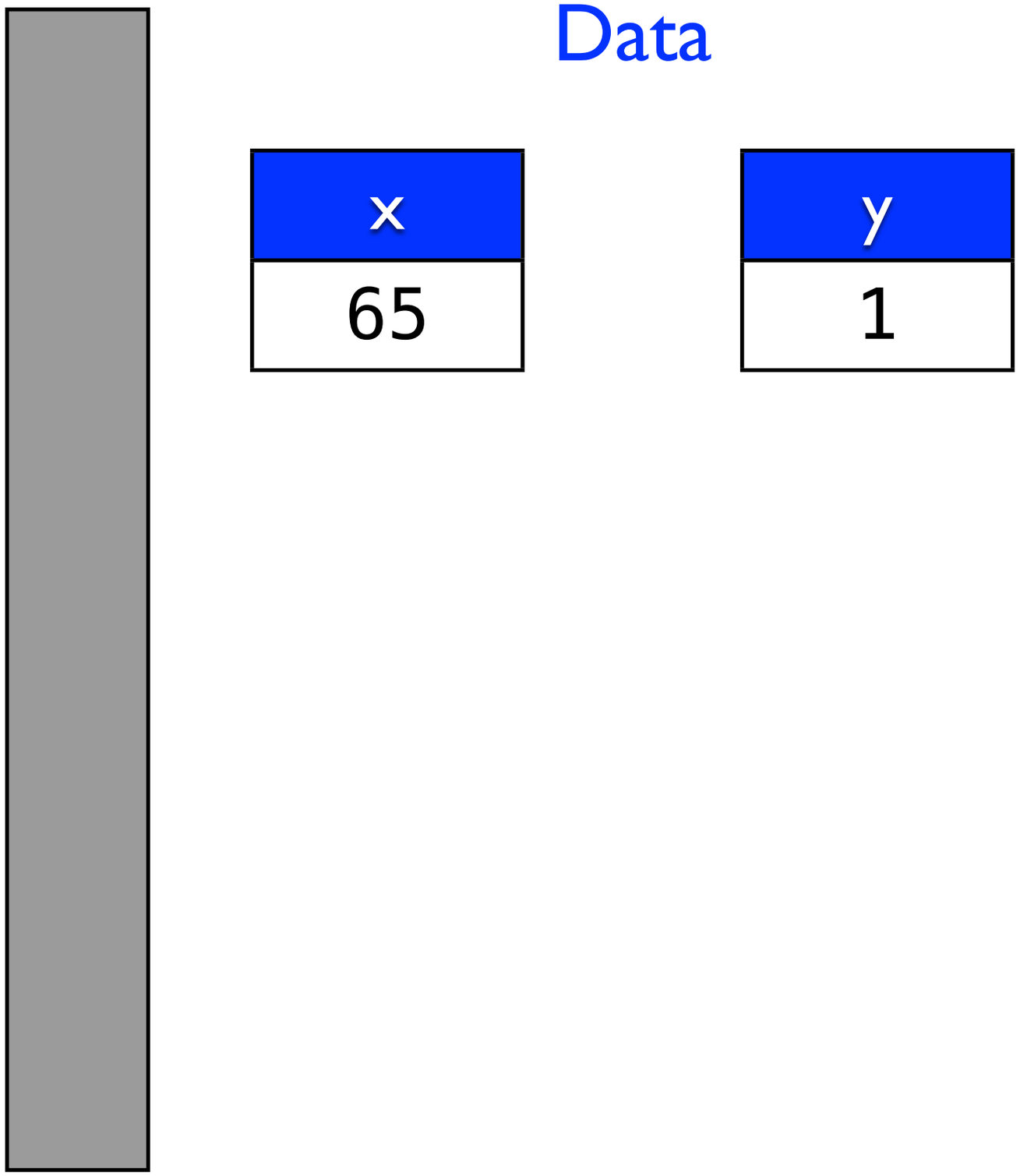
Have a targeted change...virtualize the interface with virtual values.

Standard Addition

Code

```
x = 65
y = 1
```

Data



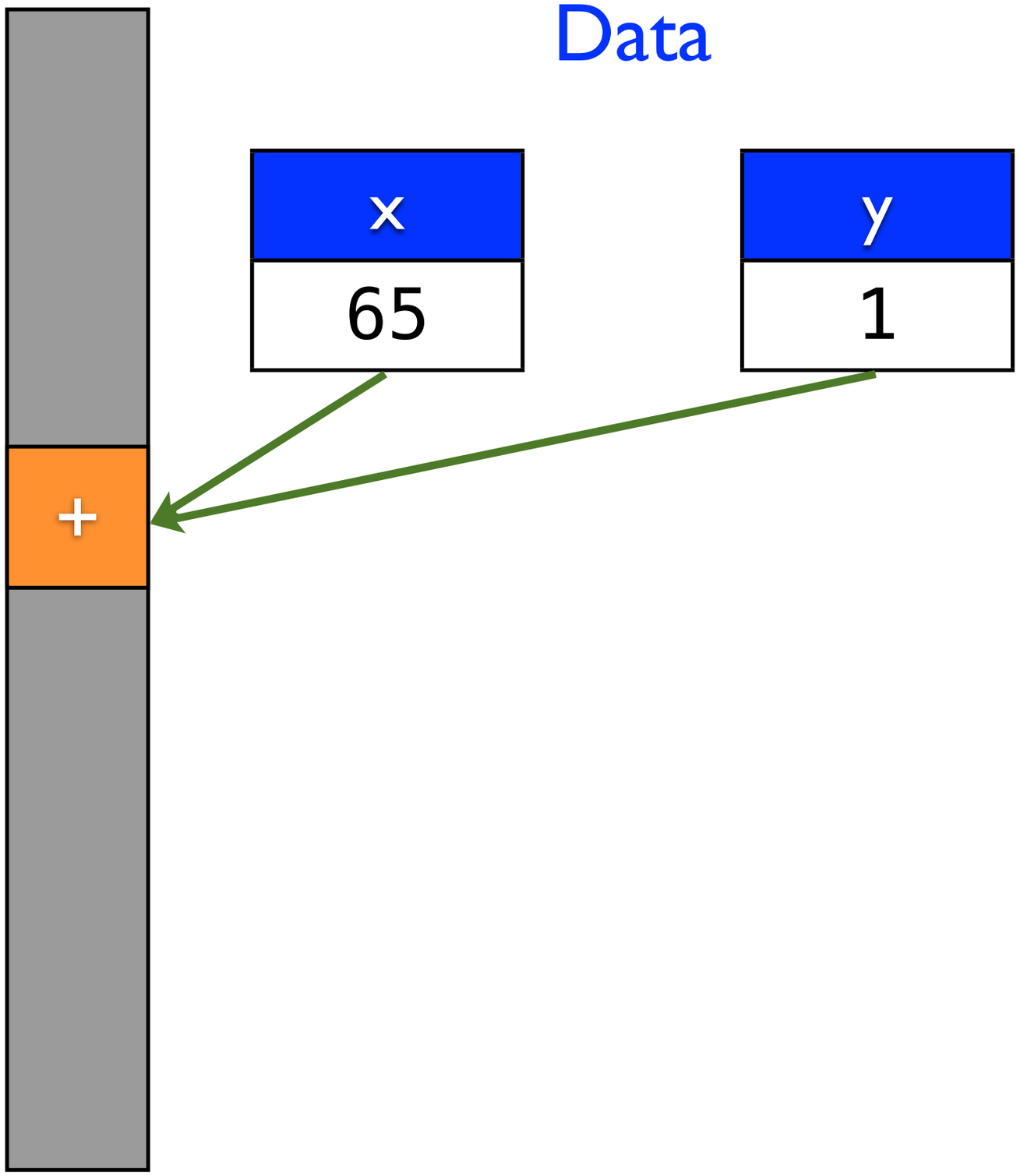
Here's how normal addition looks like in pseudo-JS

Standard Addition

Code

```
x = 65
y = 1

z = x + y
```

Data



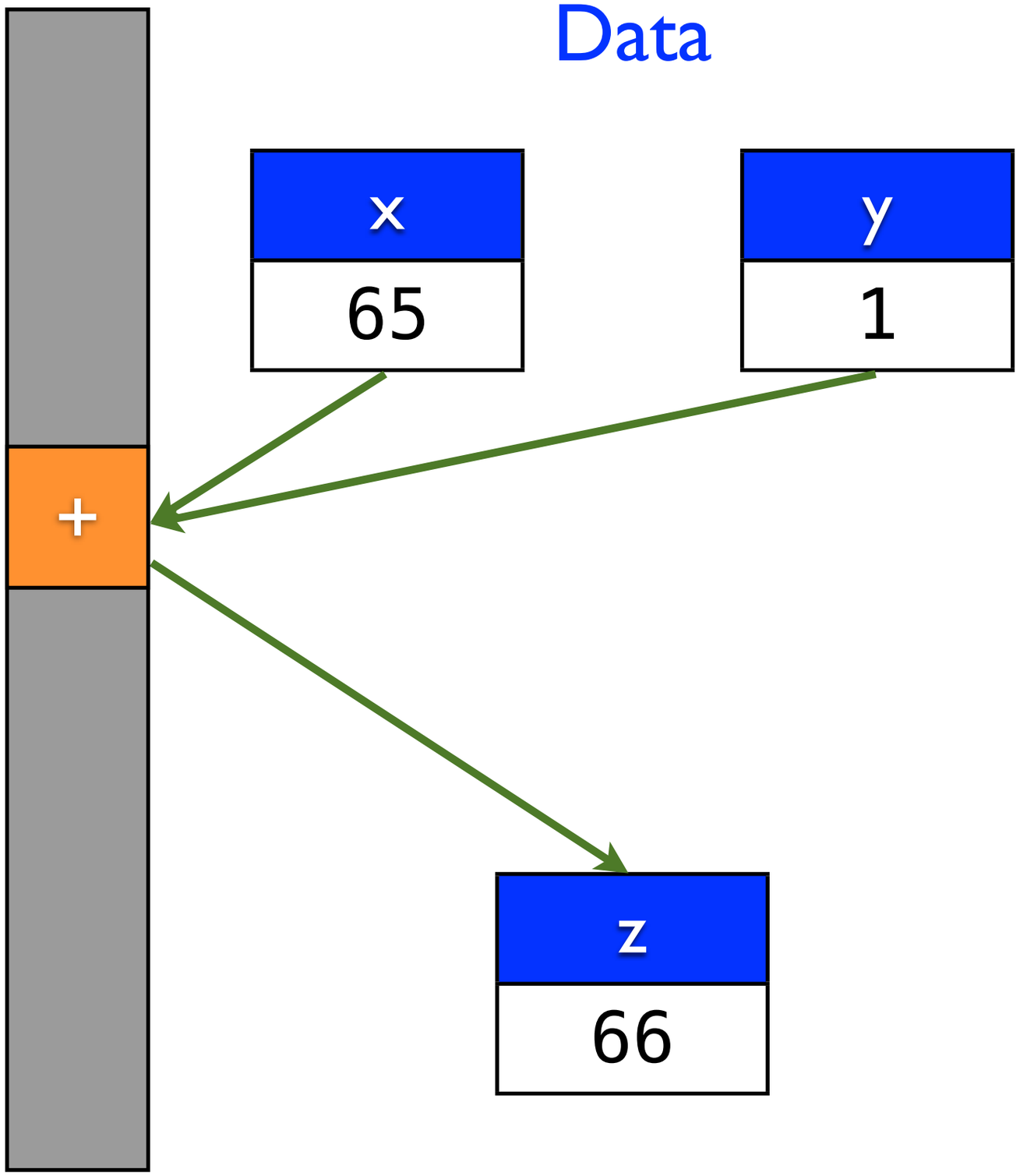
Here's how normal addition looks like in pseudo-JS

Standard Addition

Code

```
x = 65
y = 1

z = x + y
```

Data

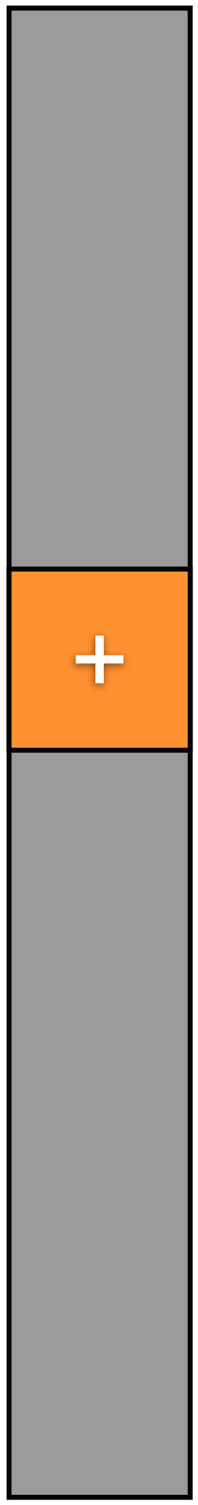


Here's how normal addition looks like in pseudo-JS

Virtualized Addition

Code

```
handler = {
    ...
    plus: λr.
      1 + r
}
```



Data

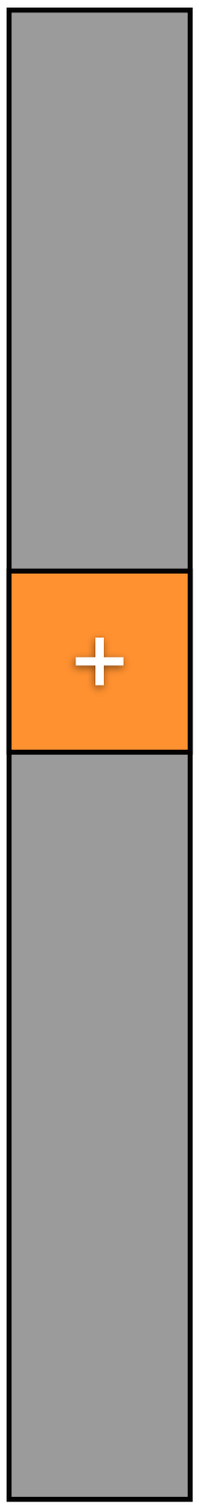
handler	
...	...
plus	λ...
...	...

Here’s our targeted virtualization. One new “virtual” value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
   plus: λr.
      1 + r
}

p = proxy(handler)
```



Data

handler	
...	...
plus	λ...
...	...

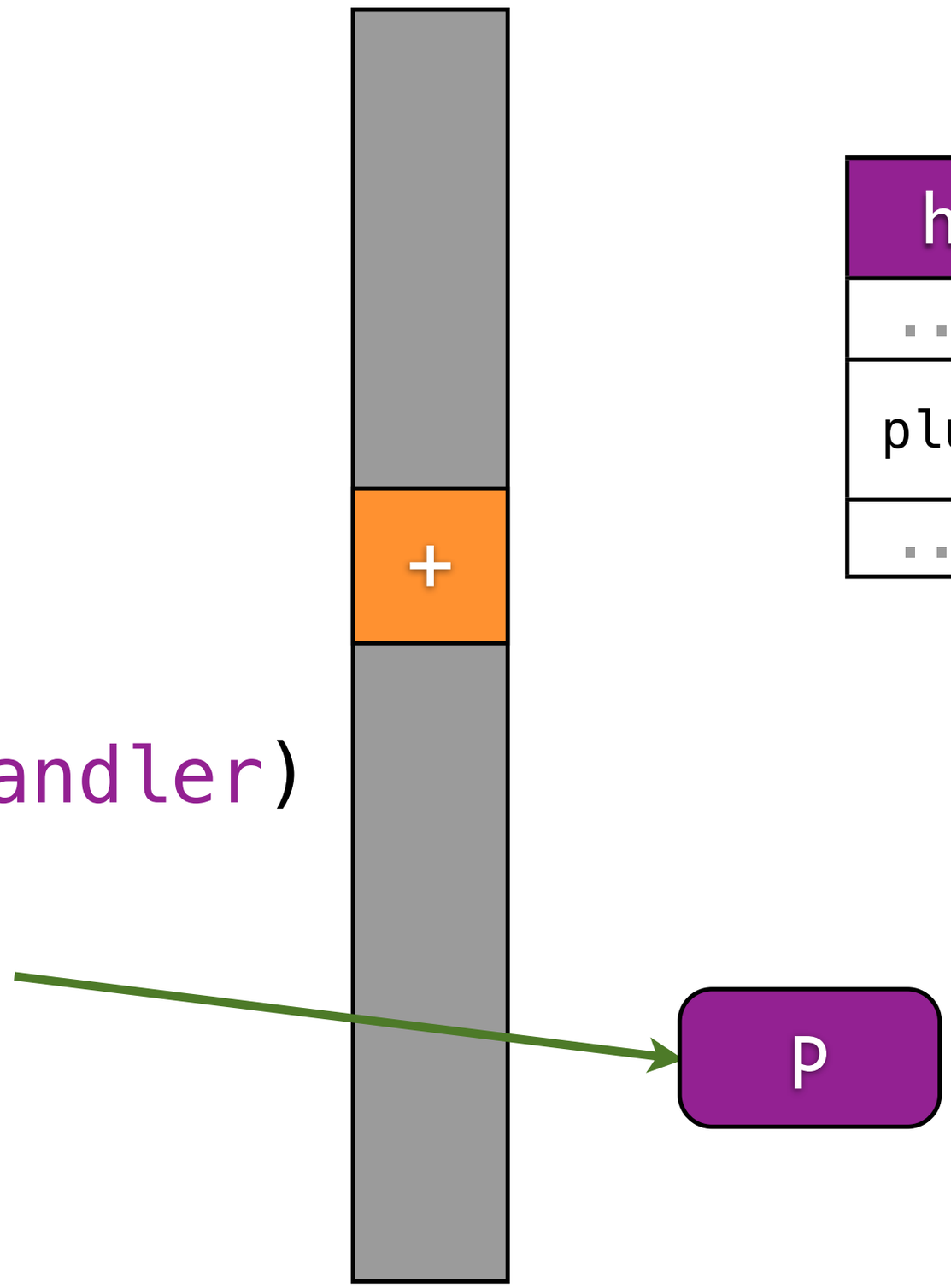
p

Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
   plus: λr.
      1 + r
}

p = proxy(handler)

z = p + 65
```



Data

handler	
...	...
plus	λ...
...	...

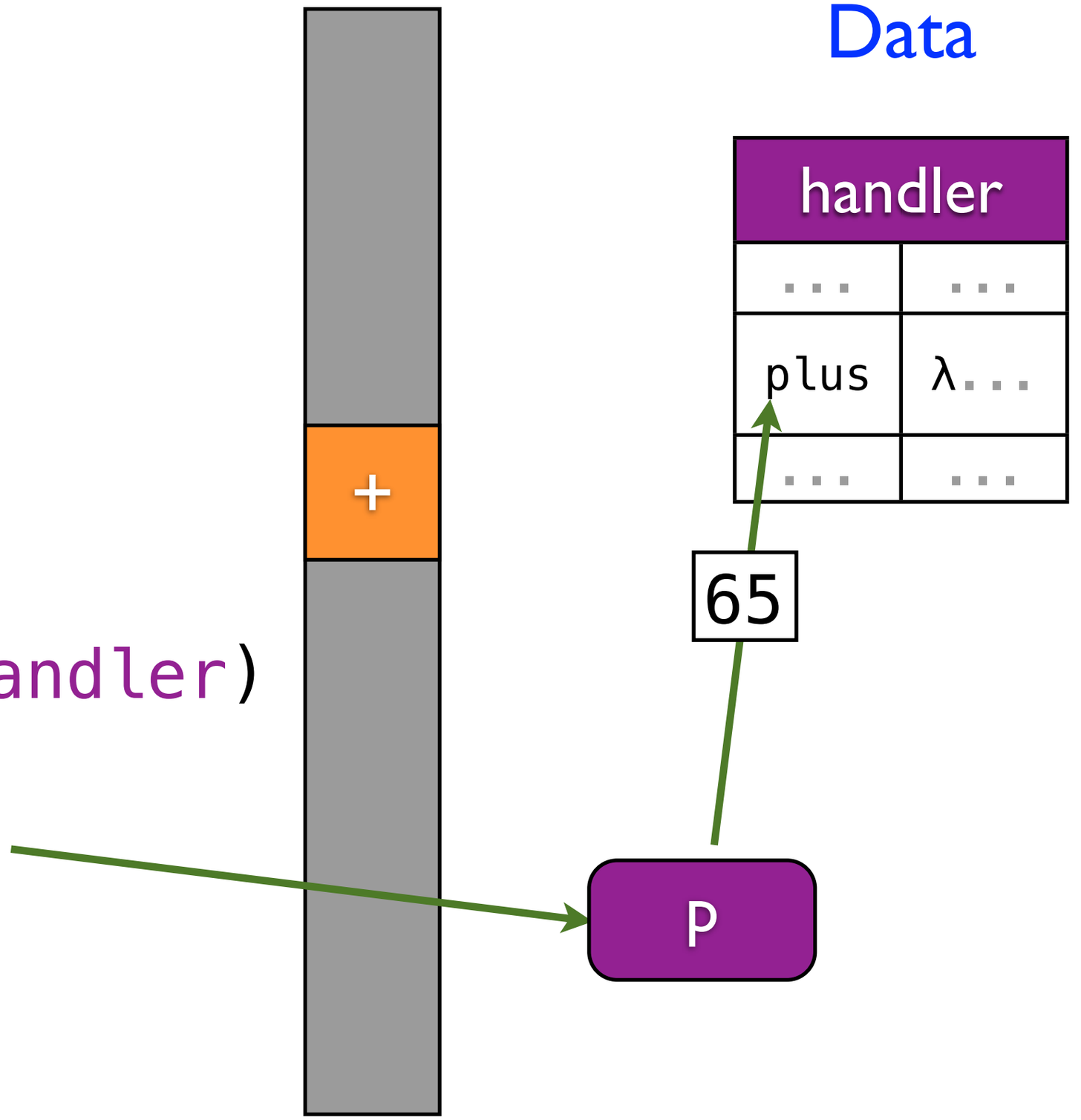
Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
   plus: λr.
      1 + r
}


p = proxy(handler)

z = p + 65
```

Data

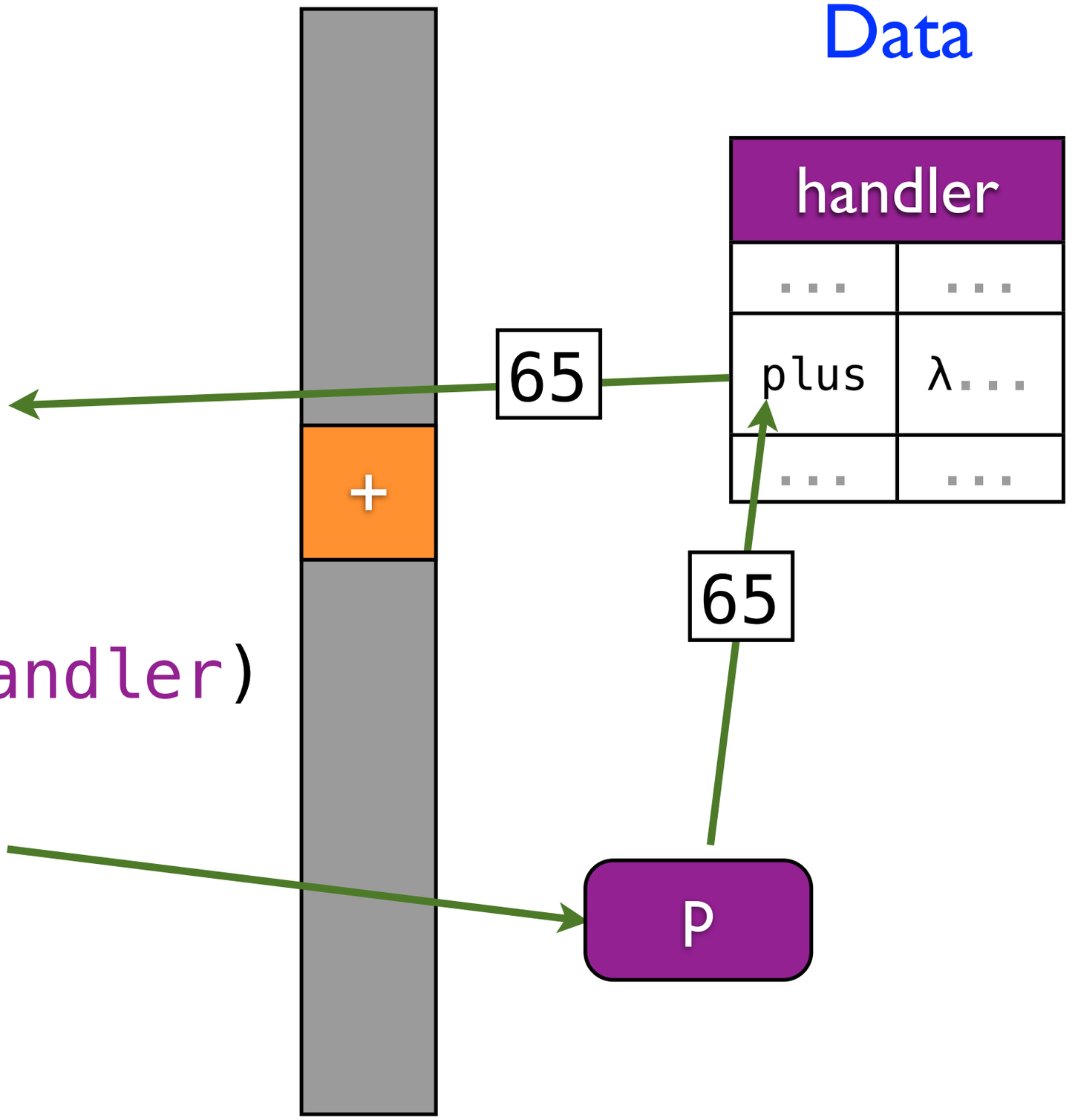


Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
   plus: λr.
      1 + r
}

p = proxy(handler)

z = p + 65
```



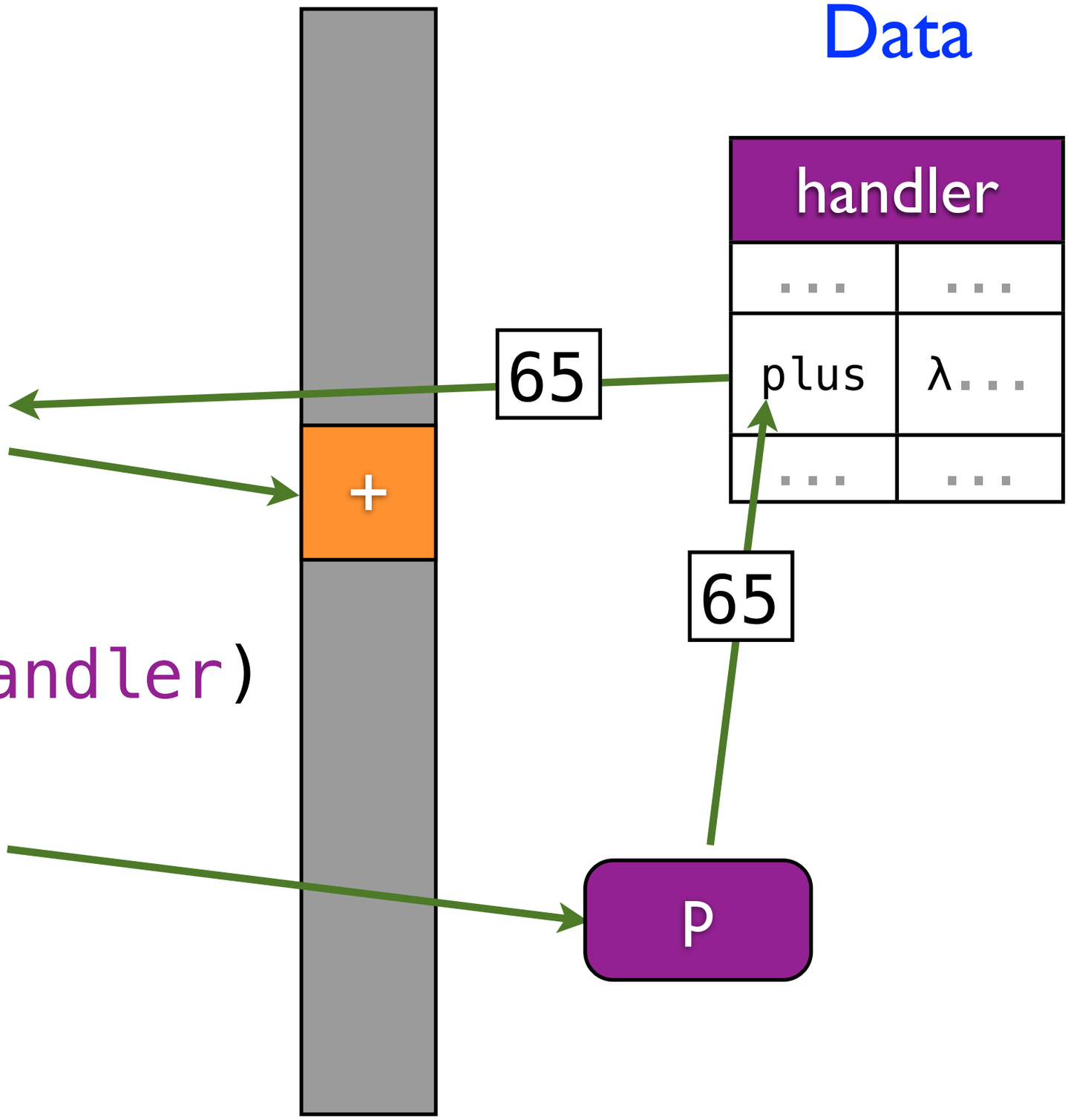
Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
   plus: λr.
      1 + r
}

p = proxy(handler)

z = p + 65
```

Data



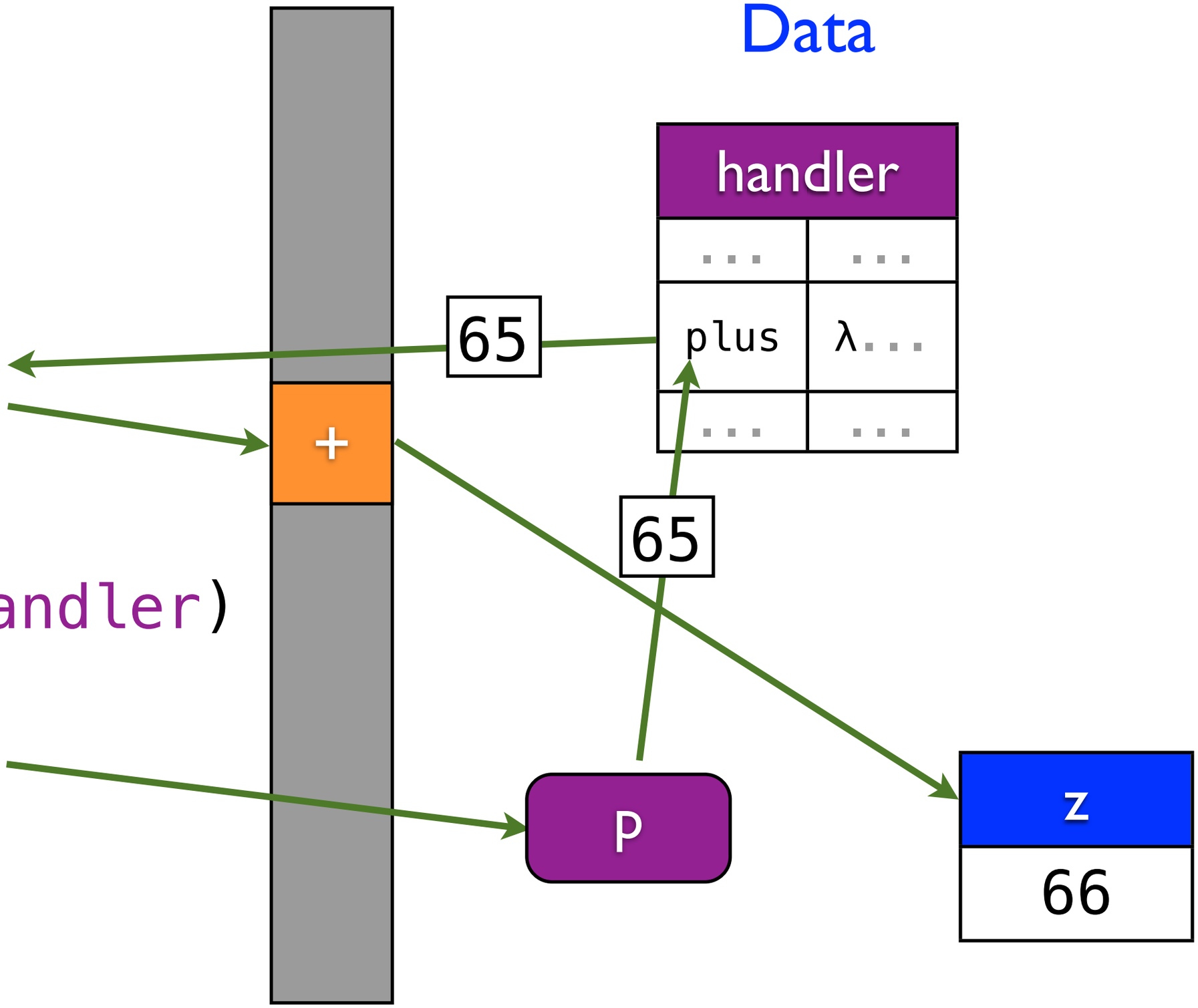
Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualized Addition

Code

```
handler = {
    ...
    plus: λr.
      1 + r
}

p = proxy(handler)

z = p + 65
```

Data



Here's our targeted virtualization. One new "virtual" value (syn with proxy) that has traps.

Virtualization is powerful!

Numeric types
Units
Contracts
Taint analysis

Revocable membranes
Lazy Evaluation
...

Once we have proxies we can do cool stuff.
Not possible in JS without virtual values.

λ_{proxy}

Idealized JavaScript-like language

idealized language with objects and primitives.

λ_{proxy}

Idealized JavaScript-like language

$\lambda x.\ e$

$e_1(e_2)$

idealized language with objects and primitives.

λ_{proxy}

Idealized JavaScript-like language

`λx. e`	`{ f : v }`
$e_1(e_2)$	`o[f]`
	`o[f] = v`

idealized language with objects and primitives.

λ_{proxy}

Idealized JavaScript-like language

`λx. e`	`{ f : v }`	`24`
$e_1(e_2)$	`o[f]`	`true`
	`o[f] = v`	`!true`
		`24 + 42`
		`if b` e_1 e_2

idealized language with objects and primitives.

λ_{proxy}

Idealized JavaScript-like language

```
proxy(handler)
```

λx. e	{ f : v }	24
$e_1(e_2)$	o[f]	true
	o[f] = v	!true
		24 + 42
		if b e_1 e_2

idealized language with objects and primitives.

```
handler = {
```

```
handler = {
   get:   λf...
```

```
handler = {
   get:   λf...
```

```
p = proxy(h)
```

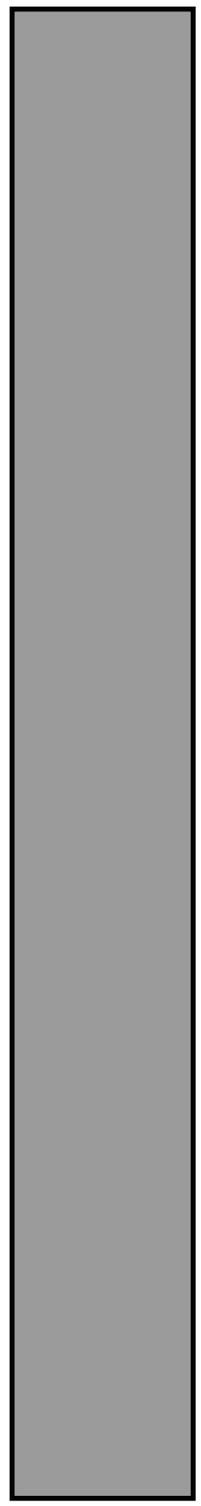
```
handler = {
   get:   λf...
```

```
p = proxy(h)
p[f]          → h.get(f)
```

Code

```
o1 = {
   “f”: 42
}
```

Data



meta

base

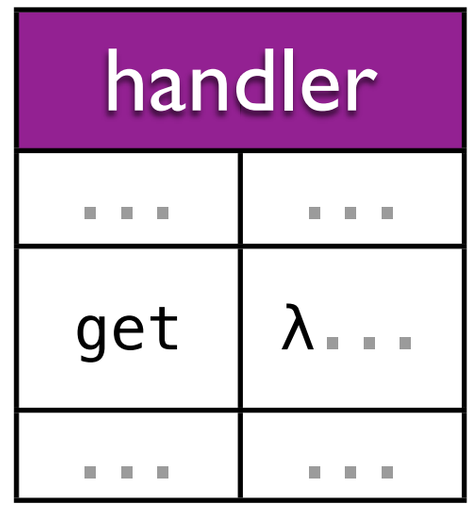
o1	
“f”	42

how the virtualization works agin but for an object get

Code

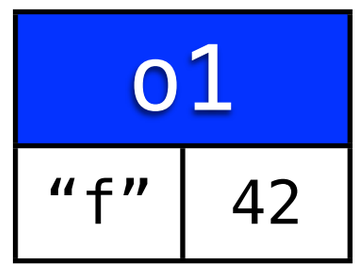
```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}
```

Data



meta

base

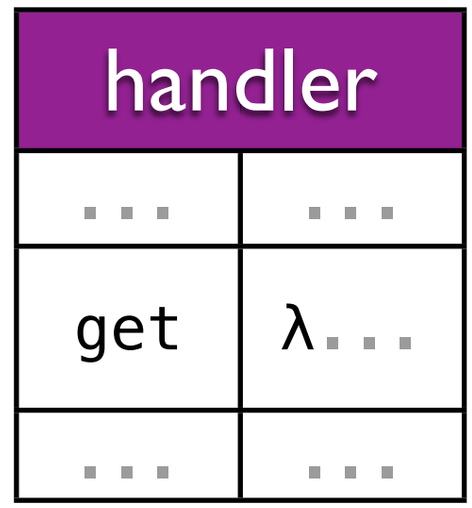


how the virtualization works agin but for an object get

Code

```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)
```

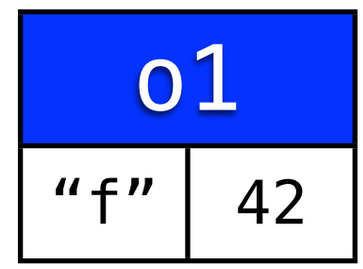
Data



meta

base

P

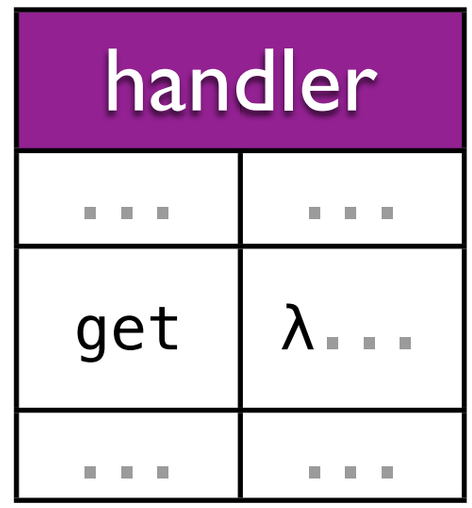


how the virtualization works agin but for an object get

Code

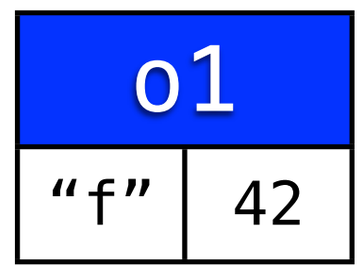
```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)

p[“f”]
```

Data



meta

base

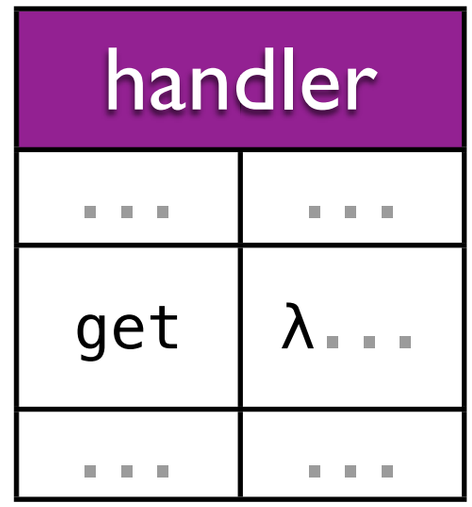


how the virtualization works agin but for an object get

Code

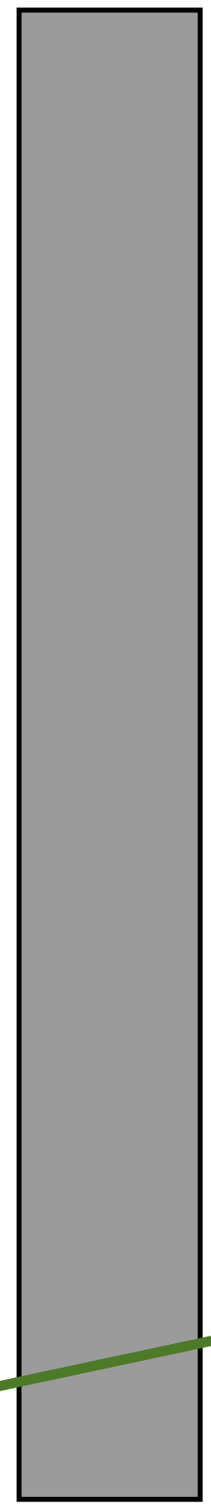
```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)

p[“f”]
```

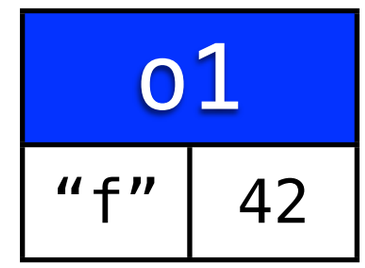
Data



meta

base



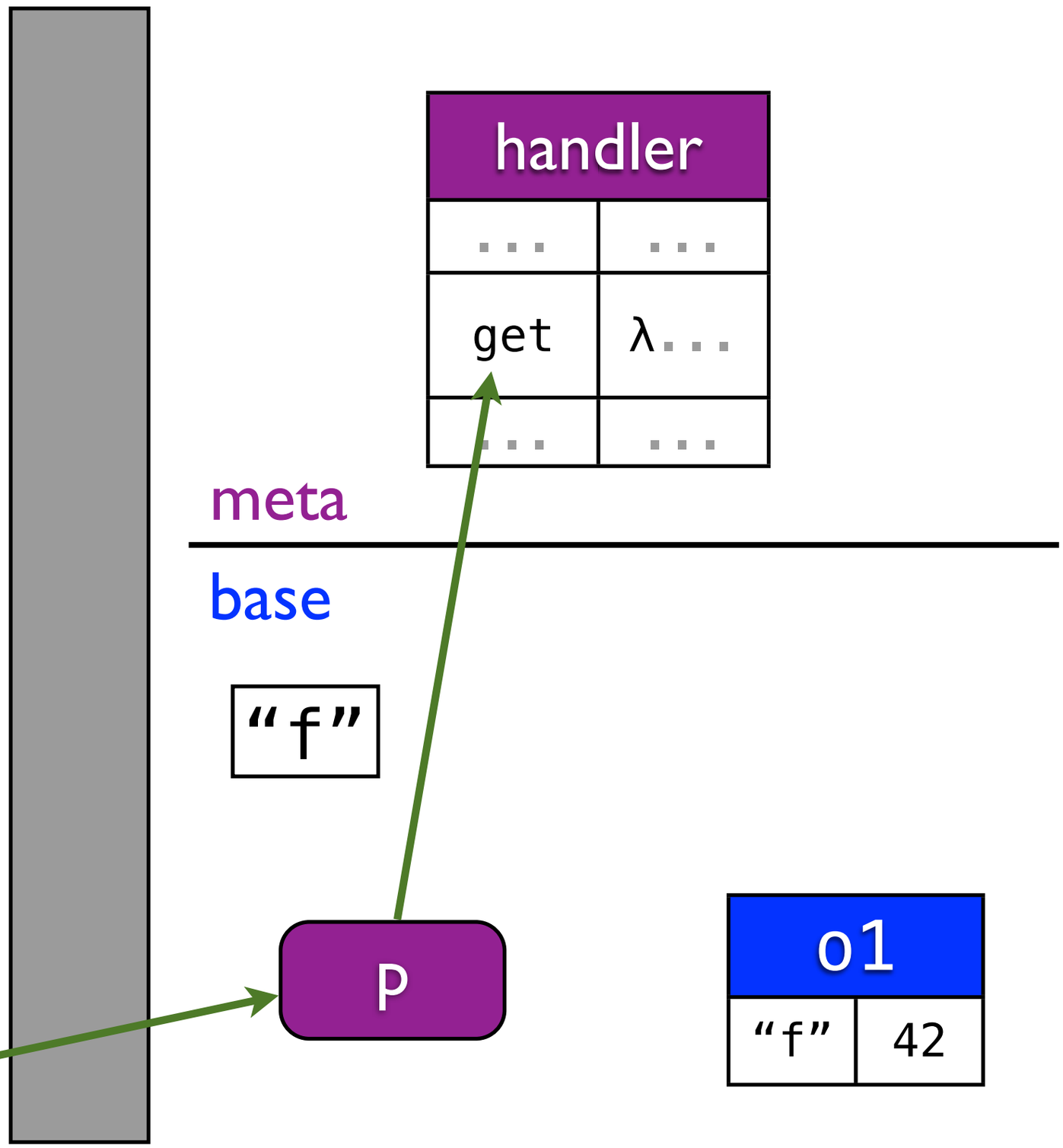


how the virtualization works agin but for an object get

Code

```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)

p[“f”]
```

Data

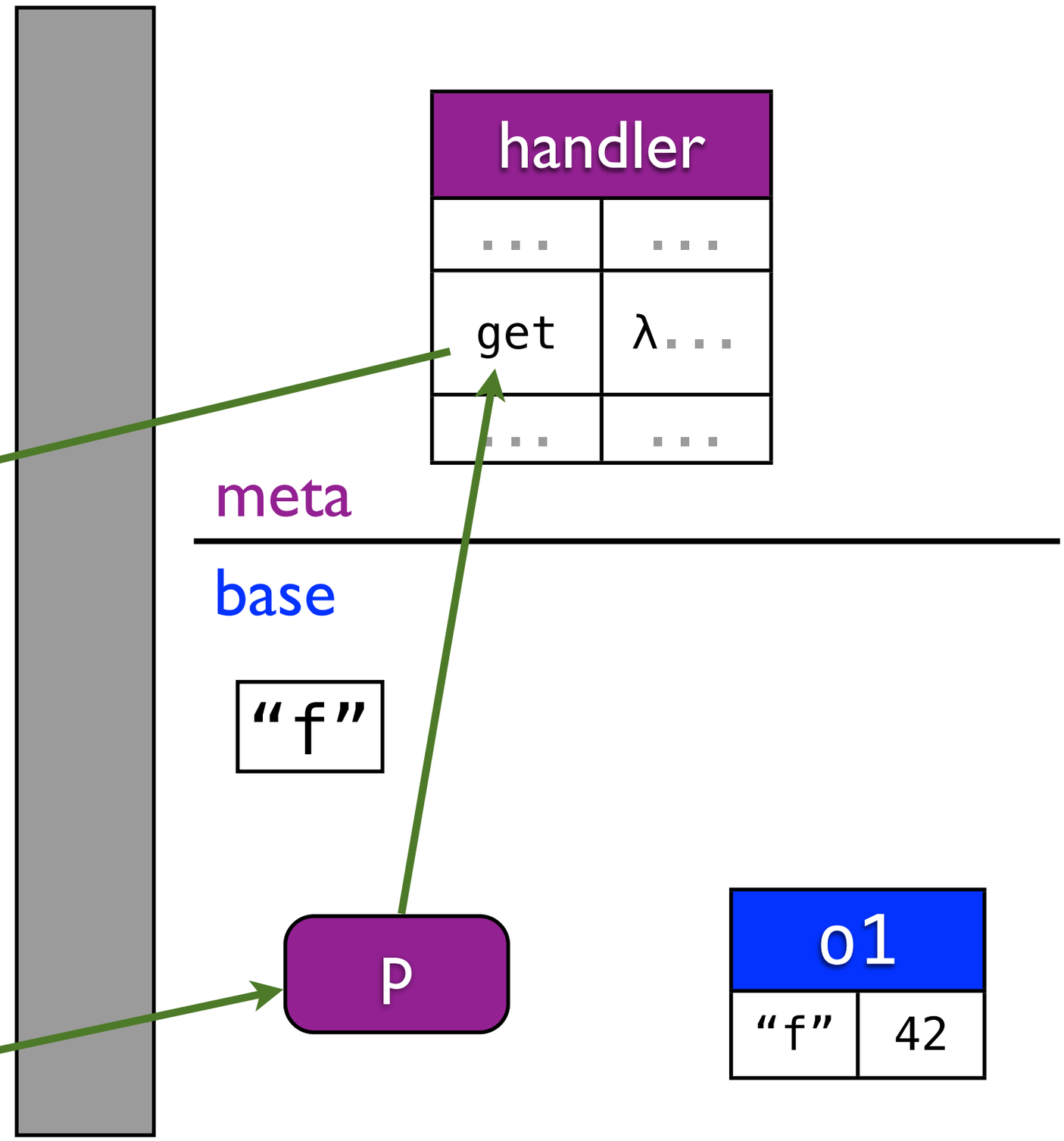


how the virtualization works agin but for an object get

Code

```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)

p[“f”]
```

Data

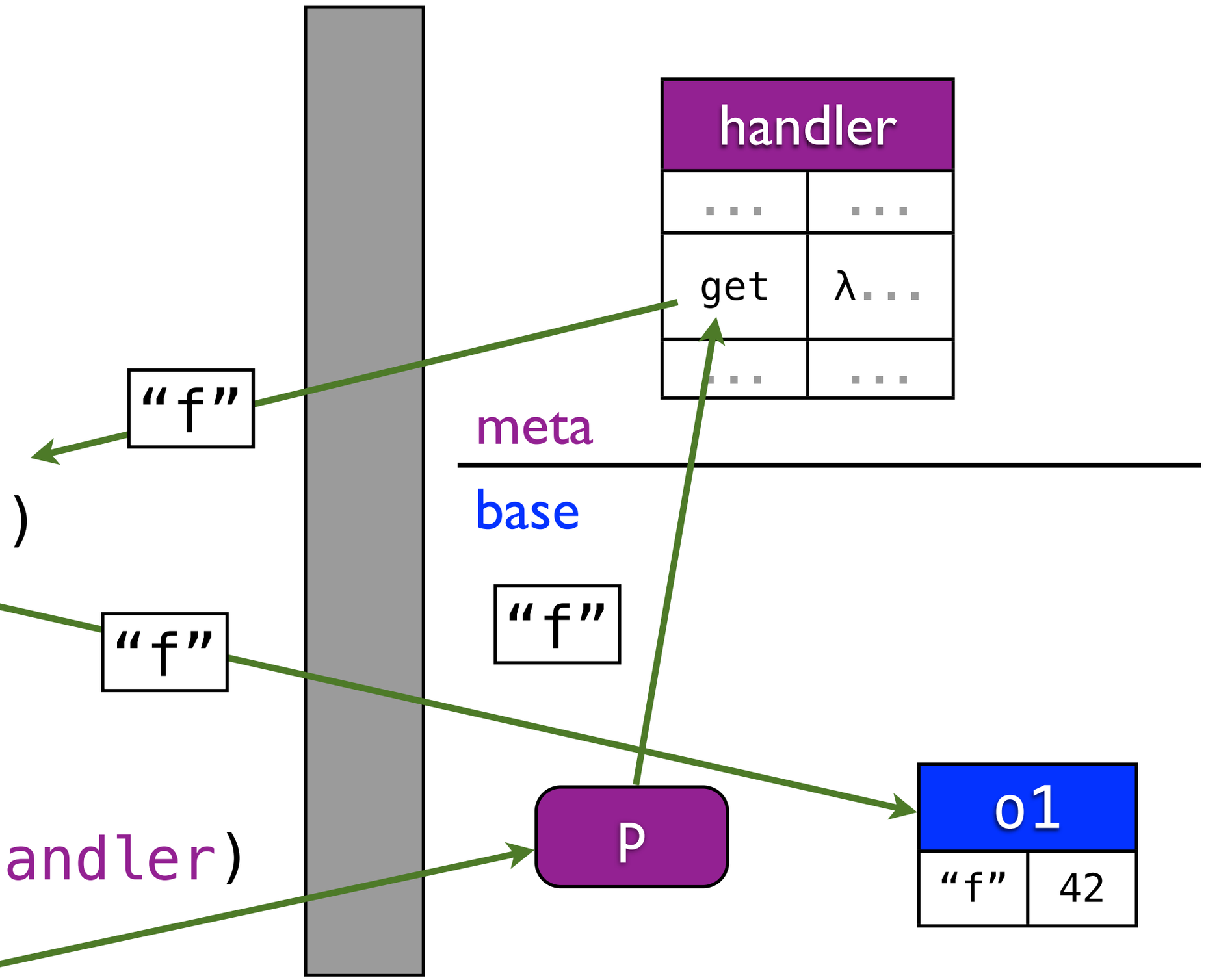


how the virtualization works agin but for an object get

Code

```
o1 = {
   “f”: 42
}

handler = {
   get: λn.
      log(...)
      o1[n]
   ...
}

p = proxy(handler)

p[“f”]
```

Data



how the virtualization works agin but for an object get

```
handler = {
   get:   λf...
```

```
p = proxy(h)
p[f]          → h.get(f)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
```

```
p = proxy(h)
p[f]       → h.get(f)
p[f] = v   → h.set(f,v)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
```

```
p = proxy(h)
p[f]        → h.get(f)
p[f] = v    → h.set(f,v)
p(v)        → h.call(v)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
   seti:  λr,v...
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
o[p] = v      → h.seti(o,v)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
   seti:  λr,v...
   unary: λo...
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
o[p] = v      → h.seti(o,v)
!p            → h.unary("!")
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
   seti:  λr,v...
   unary: λo...
   left:  λo,r...
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
o[p] = v      → h.seti(o,v)
!p            → h.unary(“!”)
p + x         → h.left(“+”,x)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
   seti:  λr,v...
   unary: λo...
   left:  λo,r...
   right: λo,l...
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
o[p] = v      → h.seti(o,v)
!p            → h.unary(“!”)
p + x         → h.left(“+”,x)
x + p         → h.right(“+”,x)
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

```
handler = {
   get:   λf...
   set:   λf,v...
   call:  λv...
   geti:  λr...
   seti:  λr,v...
   unary: λo...
   left:  λo,r...
   right: λo,l...
   test:  λ...
}
```

```
p = proxy(h)
p[f]          → h.get(f)
p[f] = v      → h.set(f,v)
p(v)          → h.call(v)
o[p]          → h.geti(o)
o[p] = v      → h.seti(o,v)
!p            → h.unary(“!”)
p + x         → h.left(“+”,x)
x + p         → h.right(“+”,x)
if p e e      → if h.test() e e
```

Virtualization often considered esoteric, interaction between meta levels.
But! Simple evaluation rules, basically as seen here.

Extensions

Modules that provide proxy creating functions that enable language extension

Now here's what we can build with proxies

```
private secret = {}

private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
    let h = unProxy secret n
    if (i = 0)                          // drop zero-ary unit
        n
    else if (h && h.unit = u)           // same unit, avoid duplicates
        makeQuantity u (h.index + i) h.value
    else if (h && h.unit > u)           // keep proxies ordered
        makeQuantity h.unit h.index (makeQuantity u i h.value)
    else                                // add this unit to proxy chain
        proxy secret {
            unit : u                    // record the unit, index, and underlying value in the handler
            index: i
            value: n
            // no call, getr, geti, setr, seti traps
            unary: λo.   unitUnaryOps[o] u i n
            left : λo,r. unitLeftOps [o] u i n r
            right: λo,l. unitRightOps[o] u i n l
            test : λ.    n // ignore units in test
        }

private unitUnaryOps :: UnaryOp ⇒ String → Int → Quantity → Any = {
    "-"      : λu,i,n. makeQuantity u i (-n)
    tostring : λu,i,n. (tostring n) + " " + u + "^" + i
    ...
}
private unitLeftOps :: BinaryOp ⇒ String → Int → Quantity → Any → Any = {
    "+": λu,i,n,r. makeQuantity u i (n + (dropUnit u i r))
    "*": λu,i,n,r. makeQuantity u i (n * r)
    "/": λu,i,n,r. makeQuantity u i (n / r)
    "=": λu,i,n,r. n = (dropUnit u i r)
    ...
}
private unitRightOps :: BinaryOp ⇒ String → Int → Quantity → Any → Any = {
    // left arg never a proxy
    "+": λu,i,n,l. assert false // unit mismatch
    "*": λu,i,n,l. makeQuantity u i    (l * n)
    "/": λu,i,n,l. makeQuantity u (-i) (l / n)
    "=": λu,i,n,l. false        // unit mismatch
    ...
}

private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
     let h = unProxy secret n
     assert h != false && h.unit = u && h.index = i
     h.value

makeUnit :: String → Quantity = λu. makeQuantity u 1 1
Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units…now just write this code…it exports “makeUnit” and you’re done.

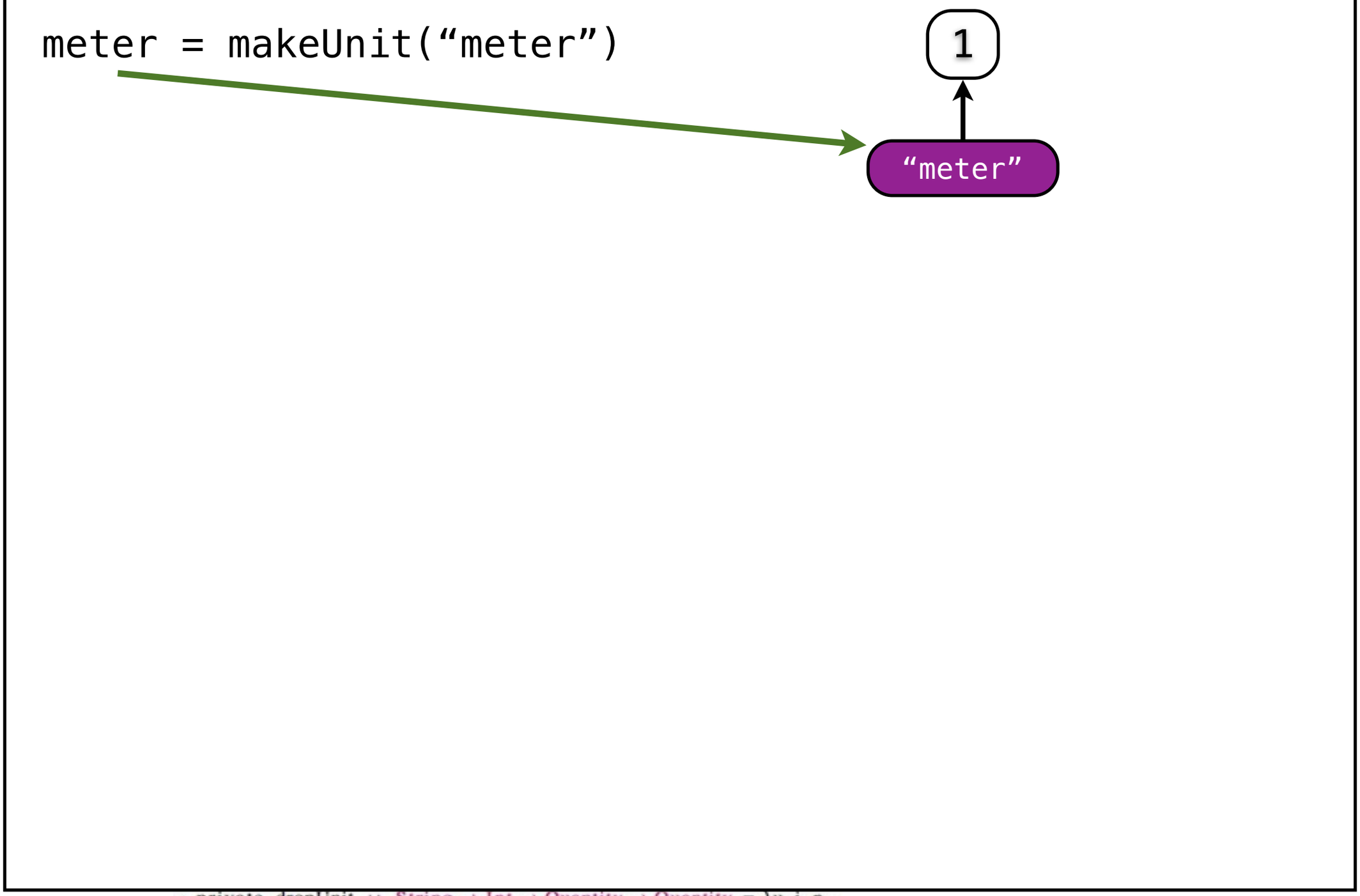
```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

meter = makeUnit(“meter”)

```
44 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports “makeUnit” and you’re done.

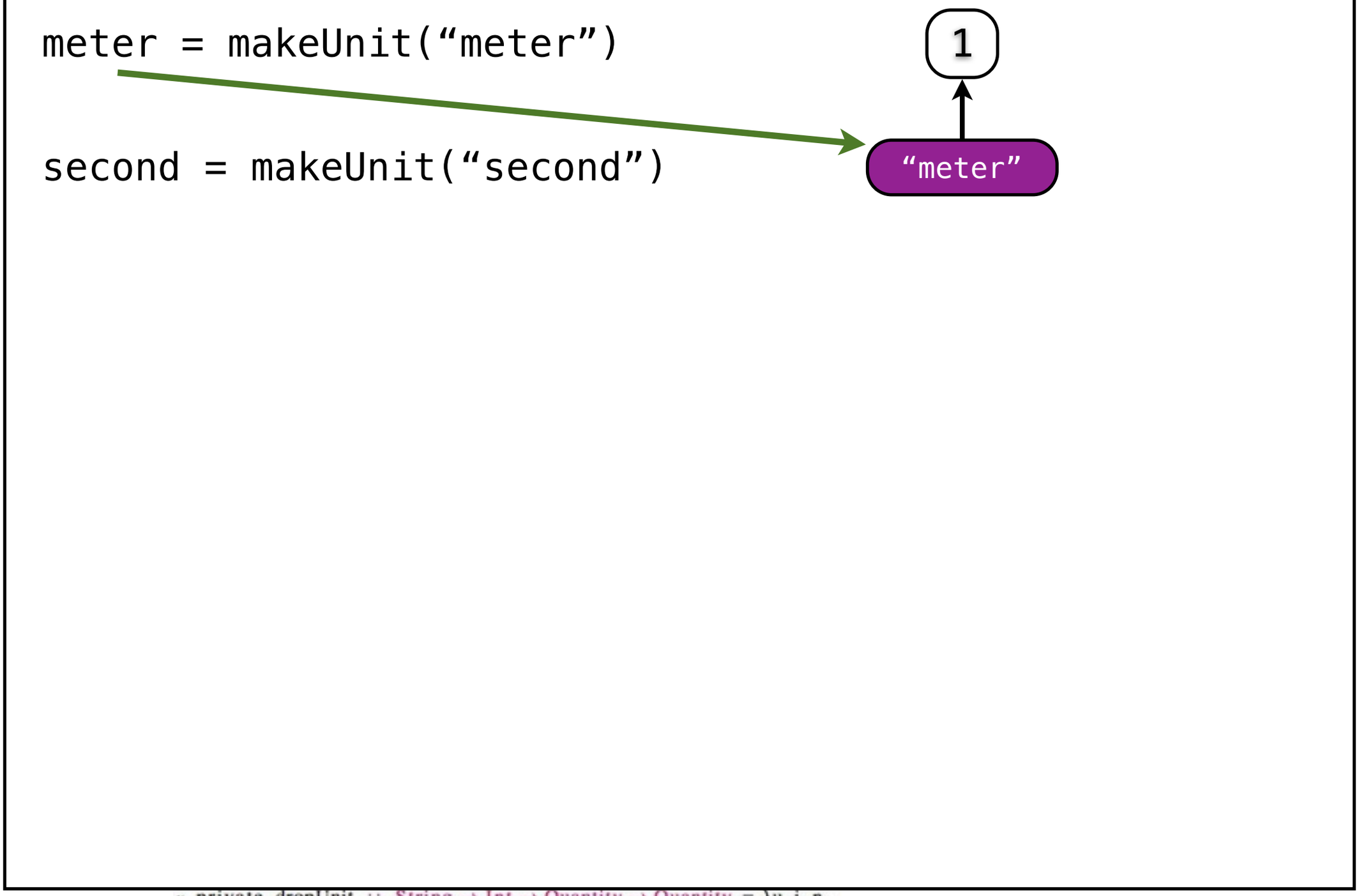
```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```




```
46 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports “makeUnit” and you’re done.

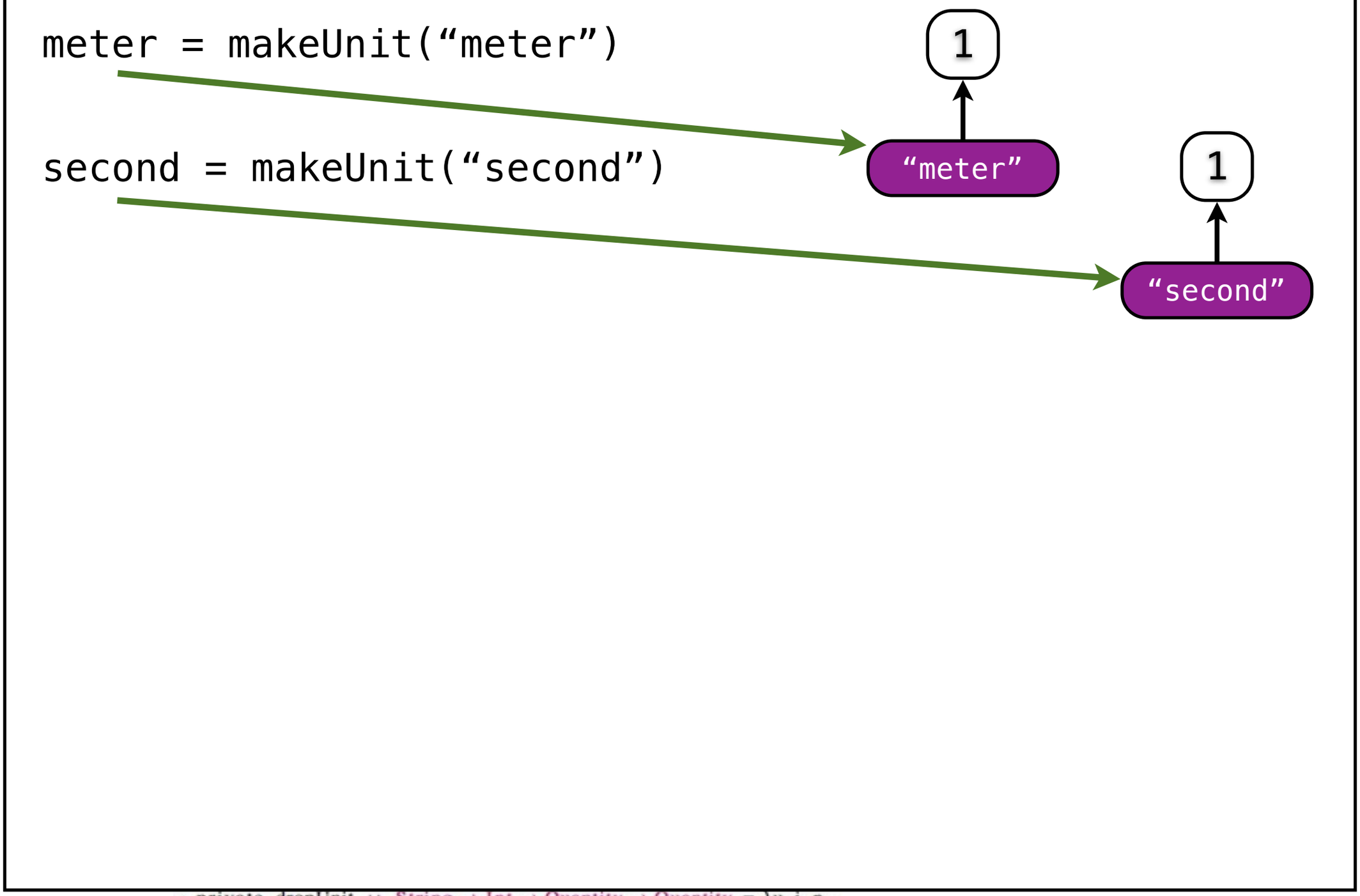
```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```




```
44 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports “makeUnit” and you’re done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

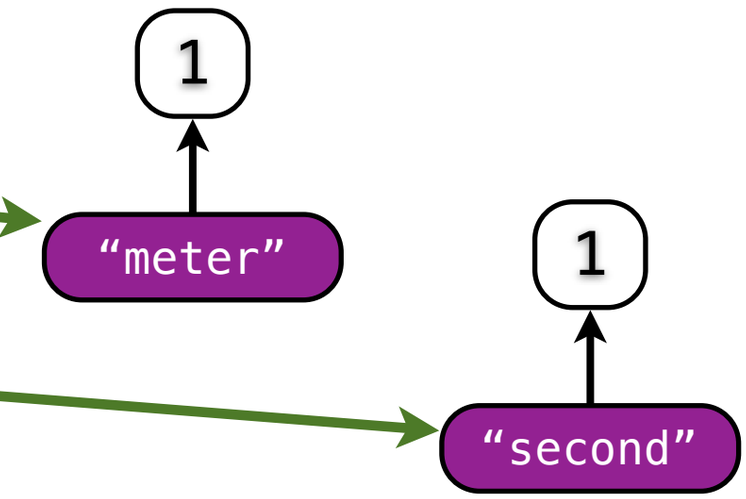



```
46 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports “makeUnit” and you’re done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second
```

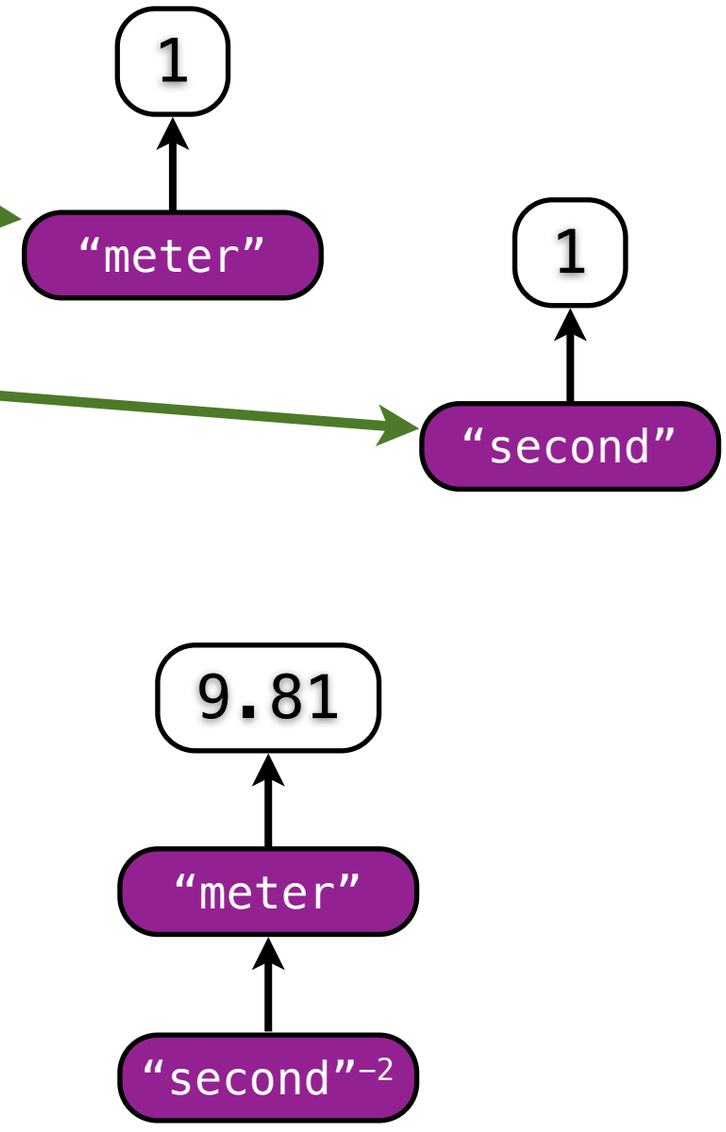



```
41 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
42     let h = unProxy secret n
43     assert h != false && h.unit = u && h.index = i
44     h.value
45
46 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
47 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second
```

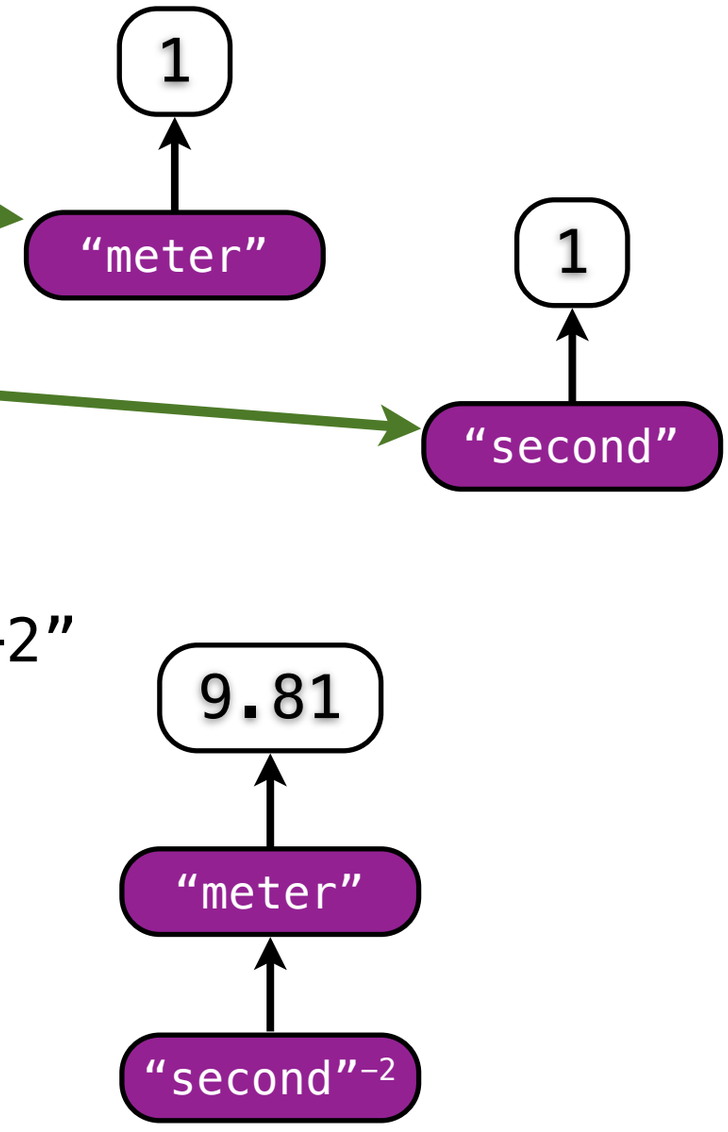



```
46 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units...now just write this code...it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second

print(g)  // "9.81 meters seconds^-2"
```

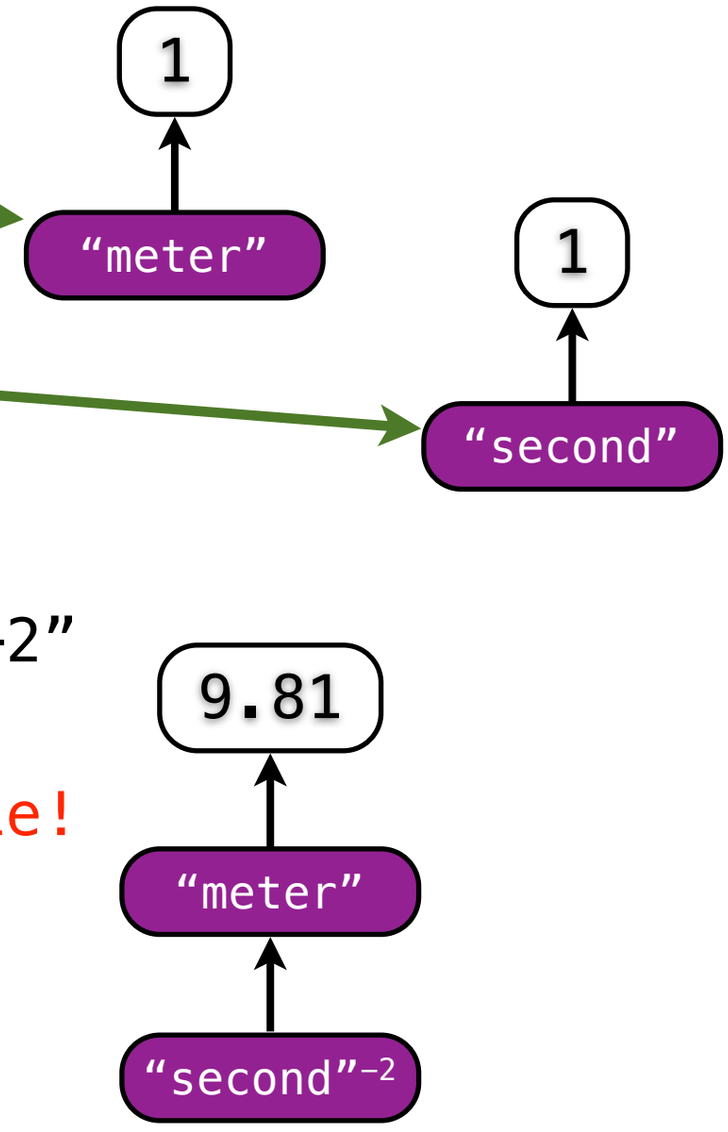



```
private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
    let h = unProxy secret n
    assert h != false && h.unit = u && h.index = i
    h.value

makeUnit :: String → Quantity = λu. makeQuantity u 1 1
Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units…now just write this code…it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second

print(g)  // "9.81 meters seconds^-2"

g + 1 // Error: Units not compatible!
```

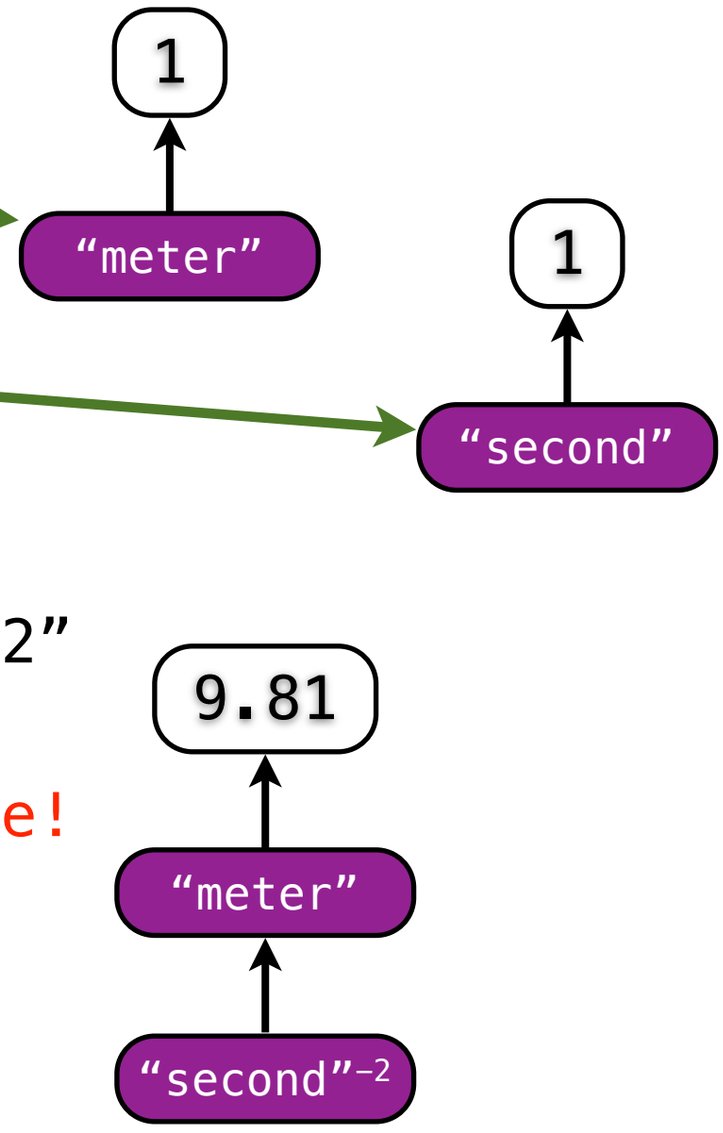



```
46 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units…now just write this code…it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second

print(g)  // "9.81 meters seconds^-2"

g + 1 // Error: Units not compatible!

g + 1 * meter / second / second
```

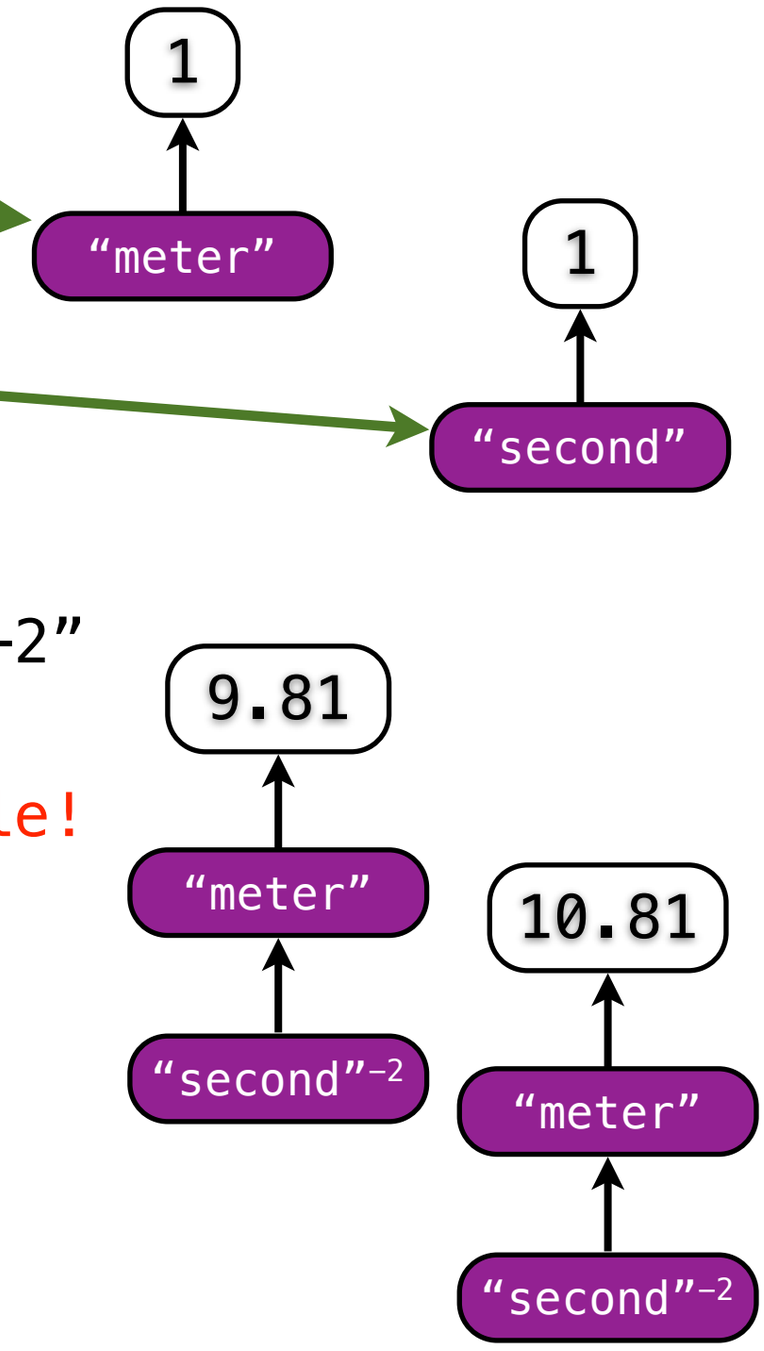



```
44 private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
45     let h = unProxy secret n
46     assert h != false && h.unit = u && h.index = i
47     h.value
48
49 makeUnit :: String → Quantity = λu. makeQuantity u 1 1
50 Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units…now just write this code…it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
4      let h = unProxy secret n
5      if (i = 0)                          // drop zero-ary unit
```

```
meter = makeUnit("meter")

second = makeUnit("second")

g = 9.81 * meter / second / second

print(g)  // "9.81 meters seconds^-2"

g + 1 // Error: Units not compatible!

g + 1 * meter / second / second
```




```
private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
    let h = unProxy secret n
    assert h != false && h.unit = u && h.index = i
    h.value

makeUnit :: String → Quantity = λu. makeQuantity u 1 1
Quantity = Flat_e (λx. if (isNum x || unProxy secret x) true false)
```

research languages to track units…now just write this code…it exports "makeUnit" and you're done.

```
1  private secret = {}
2
3  private complexUnaryOps :: UnaryOp ⇒ Num → Num → Any = {
4      "-"       : λr,i. makeComplex (−r) (−i)
5      tostring : λr,i. (tostring r) + "+" + (tostring i) + "i"
6      ...
7  }
8
9  private complexBinOps :: BinaryOp ⇒ Num → Num → Num → Num → Any = {
10     "+" : λr1,i1,r2,i2. makeComplex (r1+r2) (i1+i2)
11     "=" : λr1,i1,r2,i2. (r1=r2) && (i1=i2)
12     ...
13 }
14
15 makeComplex :: Num → Num → Complex = λr,i.
16     proxy secret {
17         real : r
18         img  : i
19         unary: λo.    complexUnaryOps[o] r i
20         left : λo,y. let h = unProxy secret y
21                      if (h)
22                         complexBinOps[o] r i h.real h.img
23                      else
24                         complexBinOps[o] r i y 0
25         right: λo,y. complexBinOps[o] y 0 r i
26         test : λ.    true // all Complex are non-false
27     }
28
29 isComplex :: Any → Bool = λx. if (unProxy secret x) true false
30
31 i :: Complex = makeComplex 0 1
32
33 Complex = Flat_c isComplex
```

can add new numeric types...complex keeps track of both the “real” and “imaginary” part

```
private secret = {}
```

```
x = 4.0 + (1.0 * i)
```

```
                    else
                        complexBinOps[o] r i y 0
        right: λo,y. complexBinOps[o] y 0 r i
        test : λ.    true // all Complex are non-false
    }

isComplex :: Any → Bool = λx. if (unProxy secret x) true false

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

can add new numeric types...complex keeps track of both the “real” and “imaginary” part

```
private secret = {}
```

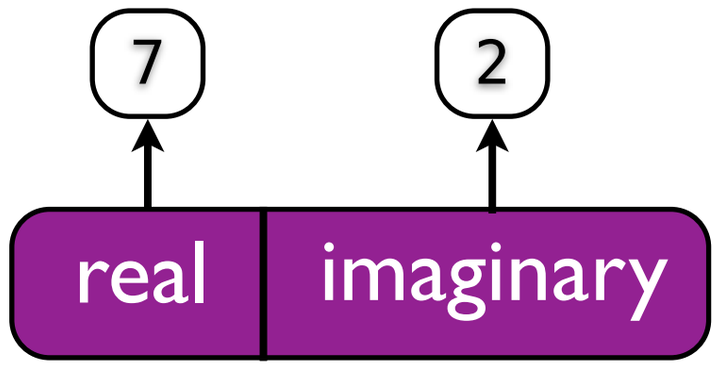
```
x = 4.0 + (1.0 * i)

y = 3.0 + (1.0 * i)
```

```
                    else
                        complexBinOps[o] r i y 0
        right: λo,y. complexBinOps[o] y 0 r i
        test : λ.    true // all Complex are non-false
    }

isComplex :: Any → Bool = λx. if (unProxy secret x) true false

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

can add new numeric types...complex keeps track of both the “real” and “imaginary” part

```
private secret = {}
```

```
x = 4.0 + (1.0 * i)

y = 3.0 + (1.0 * i)

x + y
```




```
                    else
                        complexBinOps[o] r i y 0
        right: λo,y. complexBinOps[o] y 0 r i
        test : λ.    true // all Complex are non-false
    }

isComplex :: Any → Bool = λx. if (unProxy secret x) true false

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

can add new numeric types...complex keeps track of both the “real” and “imaginary” part

How do we enable extensions to recognize the proxies they create?

during design ran into this problem
extension modules can generate proxies...how can they recognize them later

for example...

Tainting Extension

```
taint = λx.
   ...

isTainted = λx.
   ...
```

the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

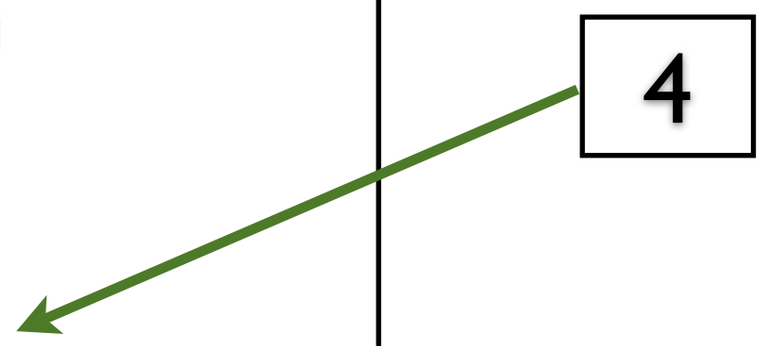
```
taint = λx.
   ...

isTainted = λx.
   ...
```

the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
taint = λx.
   ...

isTainted = λx.
   ...
```

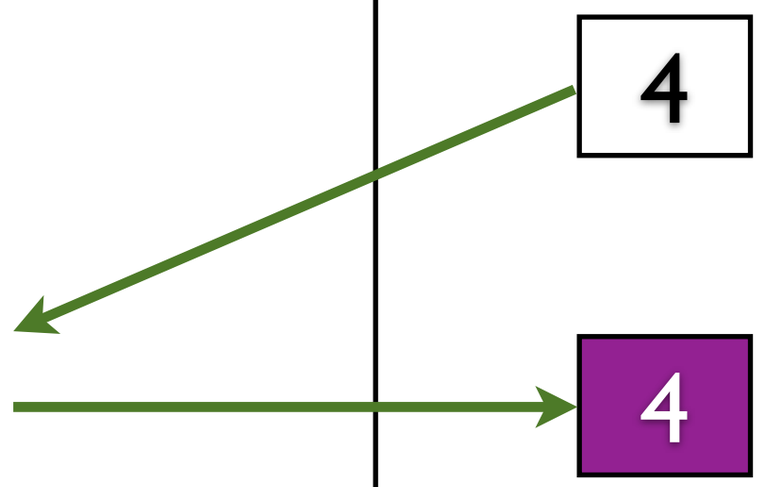


the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
taint = λx.
   ...

isTainted = λx.
   ...
```

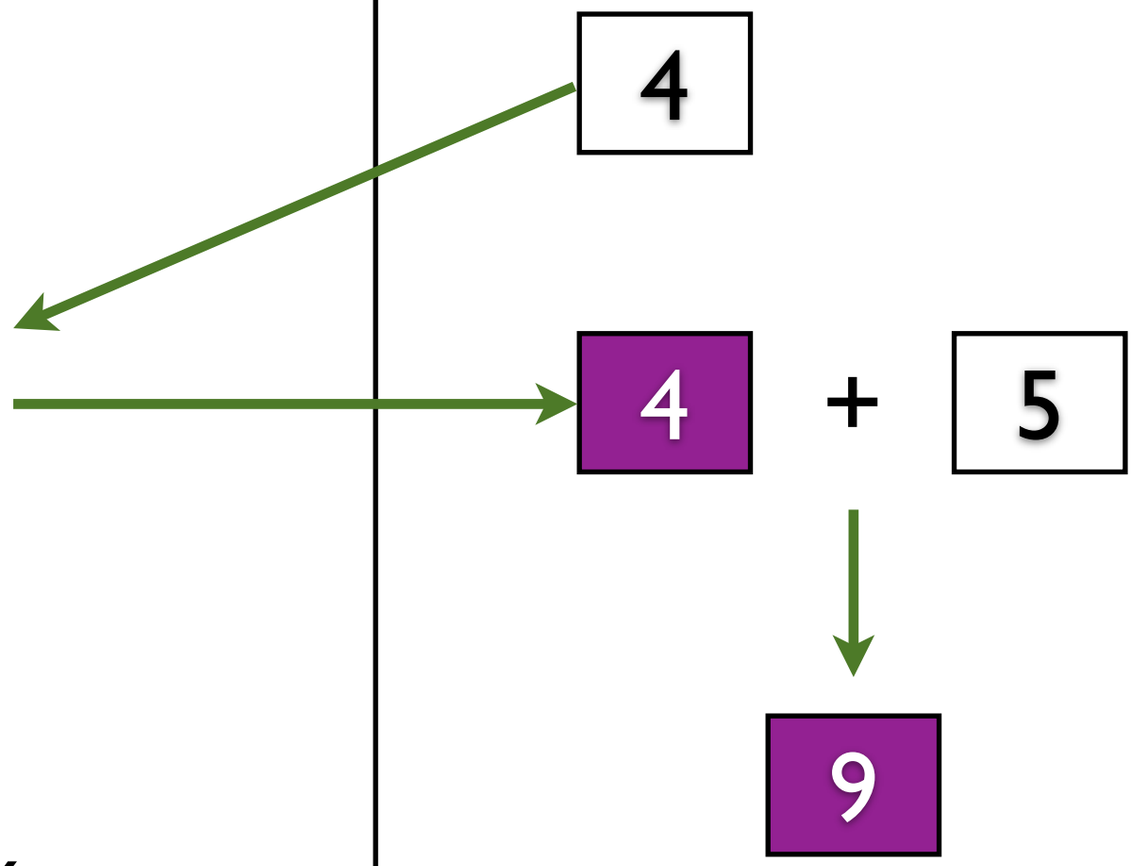


the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
taint = λx.
   ...

isTainted = λx.
   ...
```

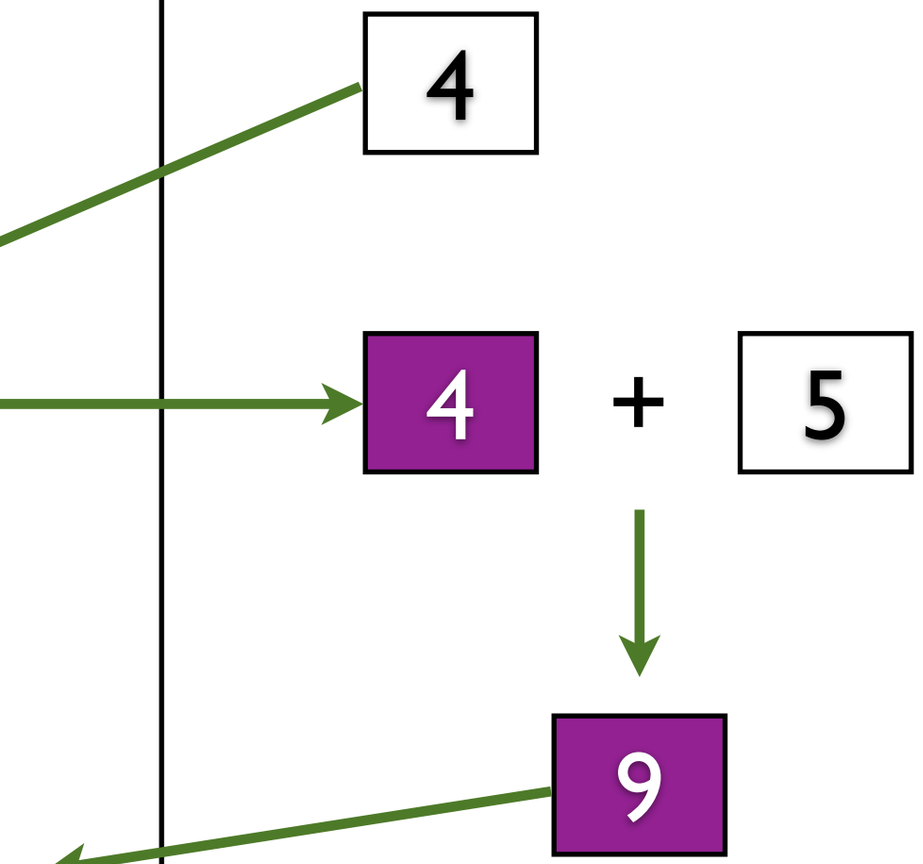


the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
taint = λx.
  ...

isTainted = λx.
  ...
```

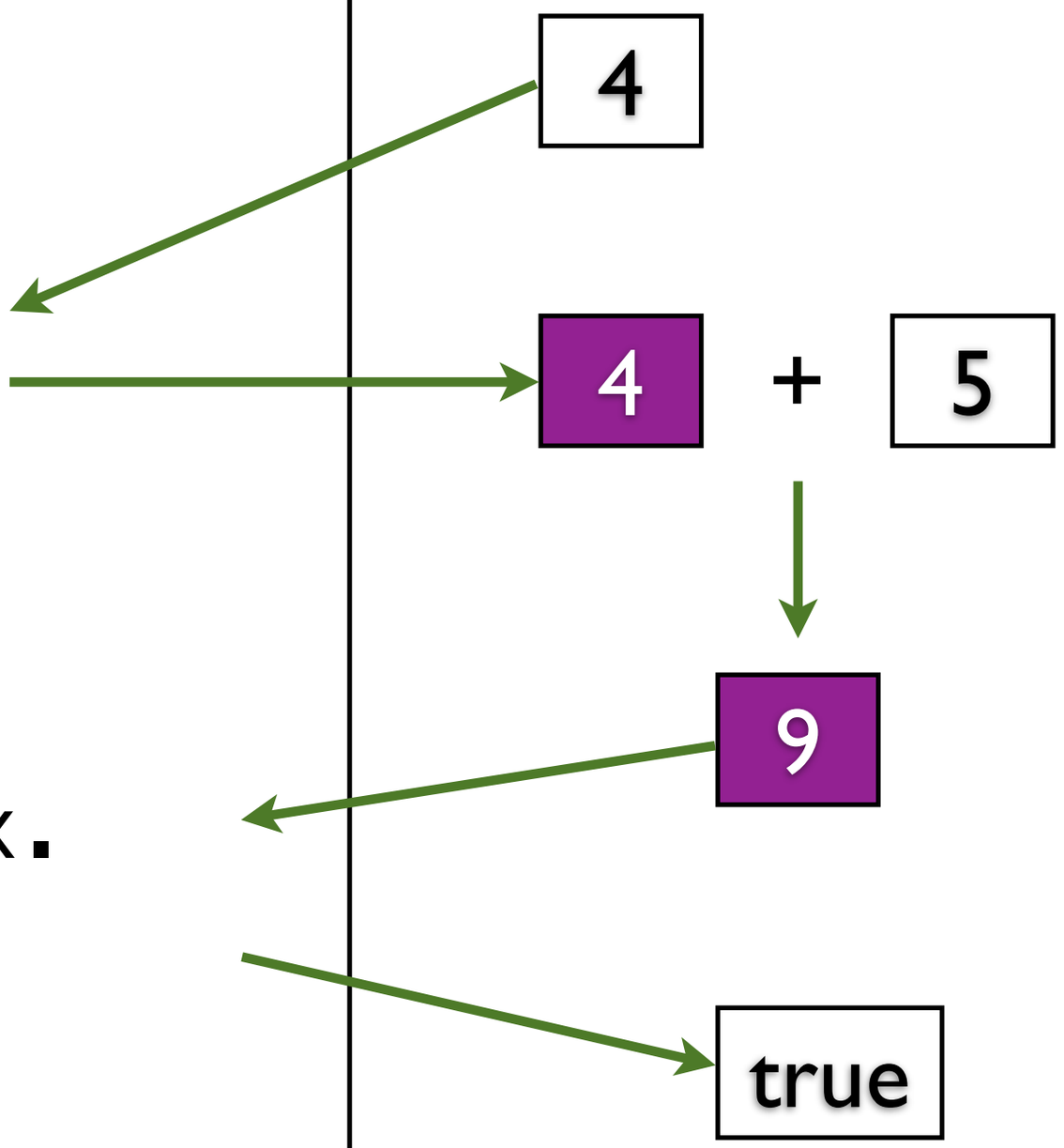


the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
taint = λx.
  ...

isTainted = λx.
  ...
```



the idea of tainting

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  unproxy[p] = x
  p

isTainted = λx.
  if unproxy[x]
    then true
    else false
```

taint and isTainted needs to collude

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  unproxy[p] = x
  p

isTainted = λx.
  if unproxy[x]
   then true
   else false
```

unproxy	
p_1	v_1
p_2	v_2
p_3	v_3
...	...

taint and isTainted needs to collude

```
isTainted(taint(4) + 5) == true
```

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  unproxy[p] = x
  p

isTainted = λx.
  if unproxy[x]
    then true
    else false
```

unproxy	
p_1	v_1
p_2	v_2
p_3	v_3
...	...

taint and isTainted needs to collude

Problem: geti/seti!

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  unproxy[p] = x
  p

isTainted = λx.
  if unproxy[x]
   then true
   else false
```

runs geti/seti traps

Problem: geti/seti!

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  unproxy[p] = x
  p

isTainted = λx.
  if unproxy[x]
    then true
    else false
```

runs geti/seti traps

Problem: geti/seti!

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  h.seti(unproxy,x)
  p

isTainted = λx.
  if h.geti(unproxy)
    then true
    else false
```

the unproxy table has no proxies

Problem: geti/seti!

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  h.seti(unproxy,x)
  p

isTainted = λx.
  if h.geti(unproxy)
    then true
    else false
```

unproxy	
v_1	v_1
v_2	v_2
v_3	v_3
...	...

the unproxy table has no proxies

Solution

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  h.seti(unproxy,x)
  p

isTainted = λx.
  if h.geti(unproxy)
    then true
    else false
```

```
proxy(key,handler)

unProxy(key,p)
```

add function unProxy.

Solution

Tainting Extension

```
unproxy = {}

taint = λx.
  h = {...}
  p = proxy(h)
  h.seti(unproxy,x)
  p

isTainted = λx.
  if h.geti(unproxy)
    then true
    else false
```

```
proxy(key,handler)
      must match
unProxy(key,p)
```

add function unProxy.

Solution

Tainting Extension

```
key = {}

taint = λx.
  ...
  proxy(key, h)

isTainted = λx.
  if unProxy(key, x)
    then true
    else false
```

```
proxy(key,handler)
      must match
unProxy(key,p)
```

collude with shared access to key

Security

Extensibility: wants to extend behavior of library extensions

Security: wants to restrict behavior of adversaries

desires are at odds

this works focuses on extnesibility but...

Security

```
isProxy(x)
```

...brute force security mechanism.

always tells the truth, proxies can’t trap

Security

...brute force security mechanism.

always tells the truth, proxies can’t trap

Stop proxies...

```
critical = λx.
   if isProxy(x)
    then err()
    else ...
```

so if we have some critical code we can use it like so

...not quite

```
critical = λx.
   if isProxy(x)
    then err()
    else
     y = x()
     ...
```

Transitive proxy!

but! used naively it won't work since a normal value could still produce a proxy

Another proxy!

```
1  private secret = {}
2
3  swap :: Any → Any = λx.
4    if (isProxy x)
5        // error if not our proxy
6        (unProxy secret x).value
7    else if (isNum x || isBool x || isString x)
8        x
9    else
10       proxy secret {
11         value: x
12         call : λy.   swap (x (swap y))
13         getr : λn.   swap (x[swap n])
14         geti : λr.   swap ((swap r)[x])
15         setr : λn,y. x[swap n]   := swap y
16         seti : λr,y. (swap r)[x] := swap y
17         unary: λo.   swap (unaryOps[o] x)
18         left : λo,r. swap (binOps[o] x (swap r))
19         right: λo,l. swap (binOps[o] (swap l) x)
20         test : λ.    if (x) true false
21       }
```

Solution: use another proxy! Create a nonProxy extension that wraps all values in a proxy that behaves like identity (passing all operations on to the original value) but checks that each value is not a proxy. Also all values coming out of it must be wrapped in the nonProxy.

Another proxy!

```
1  private secret = {}
2
```

```
critical = λx.
   x = nonProxy(x)
   y = x()
      . . .
```

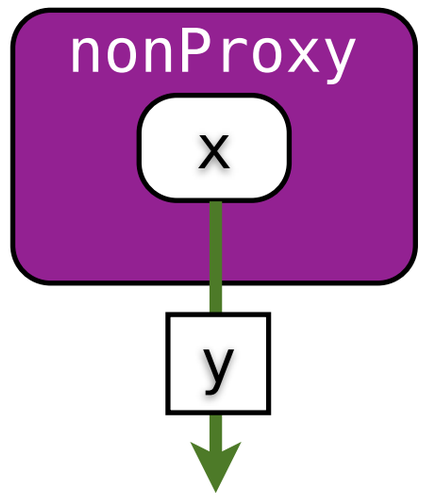
```
17        unary: λo.    swap (unaryOps[o] x)
18        left : λo,r.  swap (binOps[o] x (swap r))
19        right: λo,l.  swap (binOps[o] (swap l) x)
20        test : λ.     if (x) true false
21      }
```

Solution: use another proxy! Create a nonProxy extension that wraps all values in a proxy that behaves like identity (passing all operations on to the original value) but checks that each value is not a proxy. Also all values coming out of it must be wrapped in the nonProxy.

Another proxy!

```
1  private secret = {}
2
```

```
critical = λx.
   x = nonProxy(x)
   y = x()
      . . .
```




```
17        unary: λo.    swap (unaryOps[o] x)
18        left : λo,r.  swap (binOps[o] x (swap r))
19        right: λo,l.  swap (binOps[o] (swap l) x)
20        test : λ.     if (x) true false
21      }
```

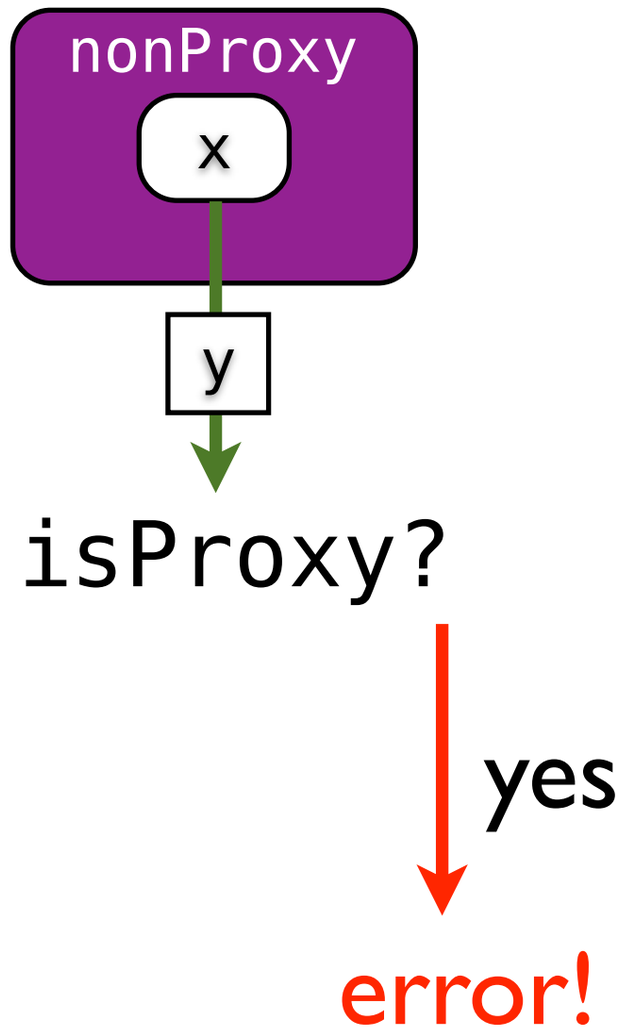
Solution: use another proxy! Create a nonProxy extension that wraps all values in a proxy that behaves like identity (passing all operations on to the original value) but checks that each value is not a proxy. Also all values coming out of it must be wrapped in the nonProxy.

Another proxy!

```
1  private secret = {}
2
```

```
critical = λx.
   x = nonProxy(x)
   y = x()
      . . .
```

```
17        unary: λo.    swap (unaryOps[o] x)
18        left : λo,r.  swap (binOps[o] x (swap r))
19        right: λo,l.  swap (binOps[o] (swap l) x)
20        test : λ.     if (x) true false
21      }
```



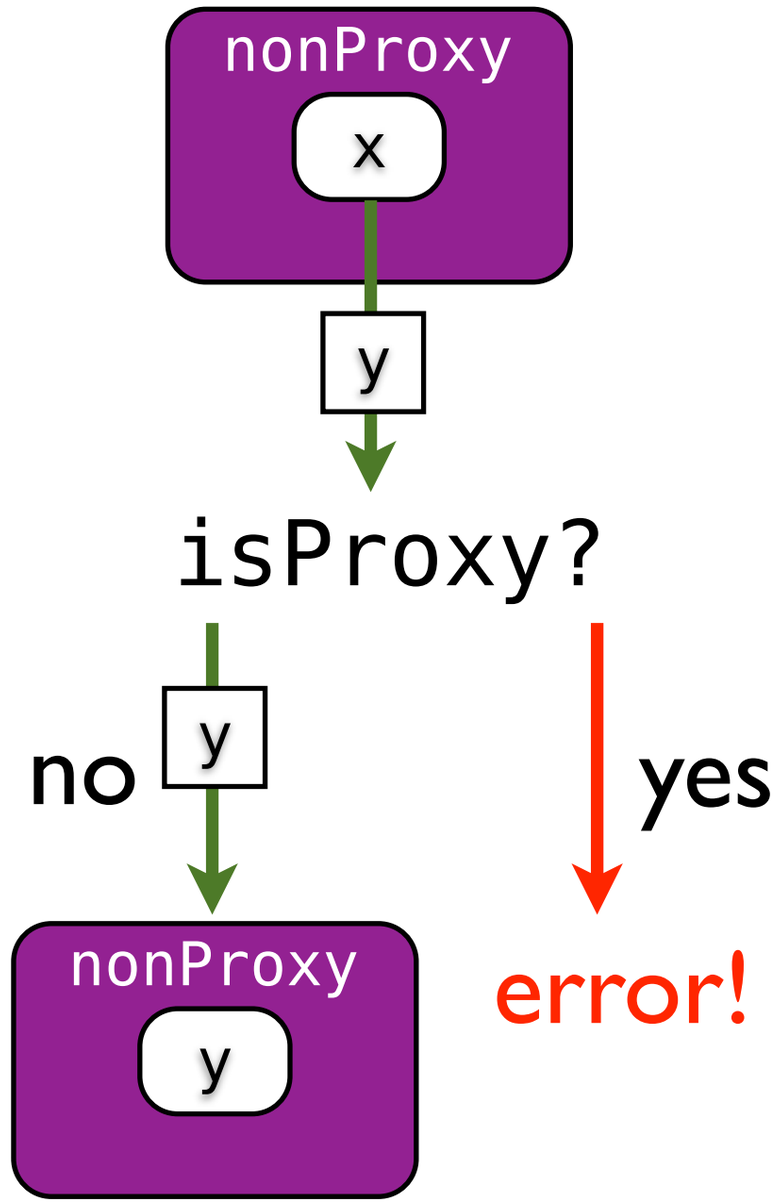
Solution: use another proxy! Create a nonProxy extension that wraps all values in a proxy that behaves like identity (passing all operations on to the original value) but checks that each value is not a proxy. Also all values coming out of it must be wrapped in the nonProxy.

Another proxy!

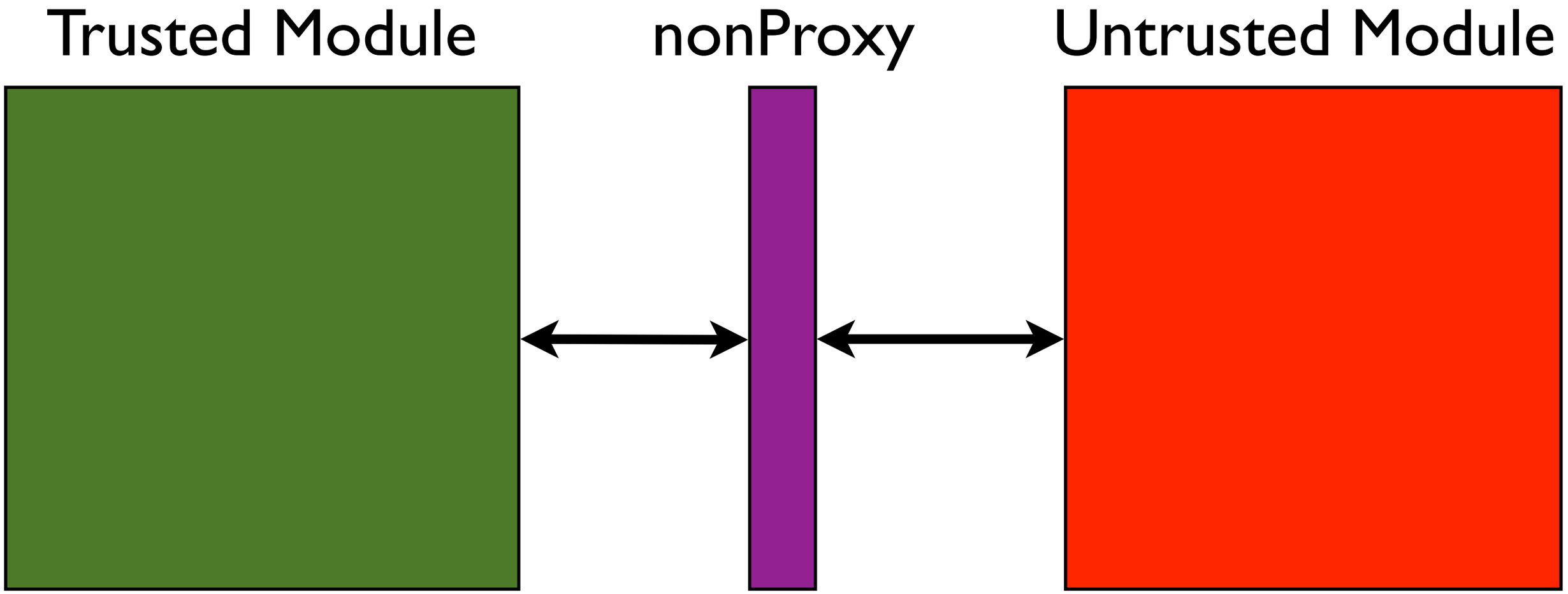
```
1  private secret = {}
2
```

```
critical = λx.
   x = nonProxy(x)
   y = x()
      . . .
```

```
17        unary: λo.    swap (unaryOps[o] x)
18        left : λo,r.  swap (binOps[o] x (swap r))
19        right: λo,l.  swap (binOps[o] (swap l) x)
20        test : λ.     if (x) true false
21     }
```



Solution: use another proxy! Create a nonProxy extension that wraps all values in a proxy that behaves like identity (passing all operations on to the original value) but checks that each value is not a proxy. Also all values coming out of it must be wrapped in the nonProxy.



So it acts a lot like a membrane.

```
handler = {
    get:    ...
    set:    ...
    call:   ...

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

JavaScript Proxies

contracts nonProxy membranes

Virtual Values

complex taint tracking units lazy evaluation

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

Quick word on related work. JS proxies are powerful can do some things
Used to build contracts.js/cs
But! Needs full set of traps to provide uniform trapping behavior.

More in paper

- Full operational semantics for λ_{proxy}
- Code for all extensions
  - all under 50 lines
- Implementation in JavaScript

check it all out

Virtual Values allow you to extend the extensibility benefits of purely OO languages

Built with JS Proxies

contracts.js

```
id = guard(
  fun(Num, Num),
  function(x) {
    return x;
});
```

contracts.coffee

```
id :: (Num) -> Num
id = (x) -> x
```

contracts in JS and CS (a JS like language with some syntax cleanup)

```
1  private secret = {}
2
3  swap :: Any → Any = λx.
4    if (isProxy x)
5        // error if not our proxy
6        (unProxy secret x).value
7    else if (isNum x || isBool x || isString x)
8        x
9    else
10       proxy secret {
11         value: x
12         call : λy.    swap (x (swap y))
13         getr : λn.    swap (x[swap n])
14         geti : λr.    swap ((swap r)[x])
15         setr : λn,y. x[swap n]    := swap y
16         seti : λr,y. (swap r)[x] := swap y
17         unary: λo.    swap (unaryOps[o] x)
18         left : λo,r. swap (binOps[o] x (swap r))
19         right: λo,l. swap (binOps[o] (swap l) x)
20         test : λ.     if (x) true false
21       }
```

Implementation in Firefox