Private Symbols, WeakMaps, and Relationships

Mark S. Miller,
with thanks to Allen Wirfs-Brock

TC39 March 2013

“Map” inverts thinking

“Map” inverts thinking

Syntax: base@field >> map.get(key)

“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

Lookup: Inherited props >> own props

“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

Lookup: Inherited props >> own props

Intuition: Relationship >> Symbol >> Map

“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

Lookup: Inherited props >> own props

Intuition: Relationship >> Symbol >> Map

Unique Symbols ok.

“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

Lookup: Inherited props >> own props

Intuition: Relationship >> Symbol >> Map

Unique Symbols ok.

(non-weak) Maps ok for container thinking

ES6 Encapsulation Mechanisms

Closures hide lexical state (ES5)

Modules hide non-exports

Direct Proxies hide handler & target

WeakMaps hide keys, do rights-amplification

Private Symbols....? Do we really need 5?

GC: base@field = value

Abstract heap maps(base, field) => value.
base *and* field reachable -> value reachable

Obvious representations:

1. impl(base)[field] => value
   better when field lives longer

2. impl(field)[base] => value
   better when base lives longer

GC by use cases

GC by use cases

When base is known to live longer.
Just use a map! (Thanks Yehuda)

GC by use cases

When base is known to live longer.
Just use a map! (Thanks Yehuda)

oo private field ***is known*** to live longer.
Just use representation #1

GC by use cases

When base is known to live longer.
Just use a map! (Thanks Yehuda)

oo private field ***is known*** to live longer.
Just use representation #1

Most remaining WeakMap use cases
would do better with rep #1 (untested claim)

GC by use cases

Only need ephemeron collection when
you guessed wrong relative longevity
you care about the memory pressure

Felix's O(N) algorithm is affordable
with inverted representation

Example: Membranes

GC by use cases

Only need ephemeron collection when
you guessed wrong relative longevity
you care about the memory pressure

Felix's O(N) algorithm is affordable
with inverted representation

Example: Membranes

Speaking of which...

Transparency vs Privacy

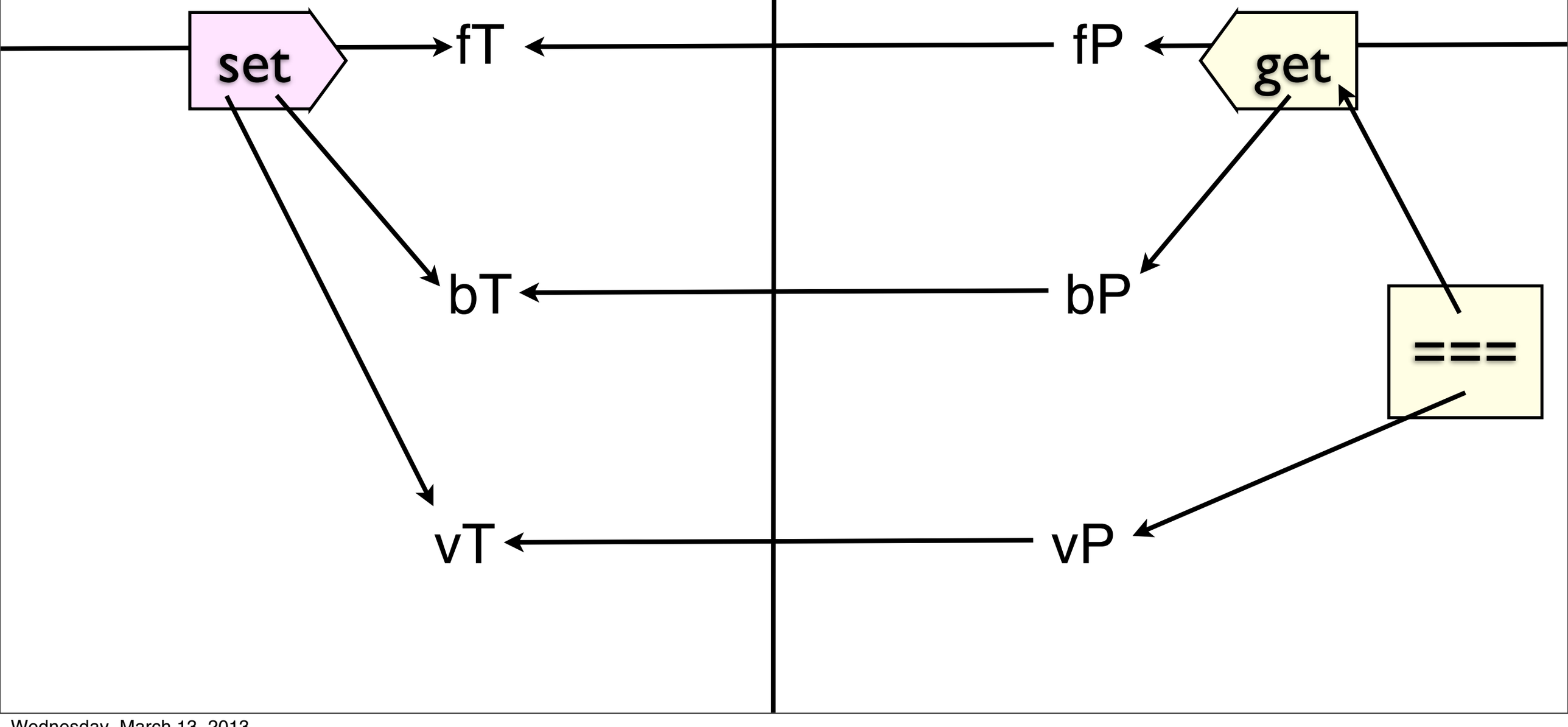
bL@fL = vL	bR@fR === vR
bT@fT = vT	bP@fP === vP
bT@fT = vP	bP@fP === vT
bT@fP = vT	bP@fT === vP
bT@fP = vP	bP@fT === vT
bP@fT = vT	bT@fP === vP
bP@fT = vP	bT@fP === vT
bP@fP = vT	bT@fT === vP
bP@fP = vP	bT@fT === vT

bT@fT = vT

fT.set(bT, vT)

bP@fP === vP

fP.get(bP) === vP

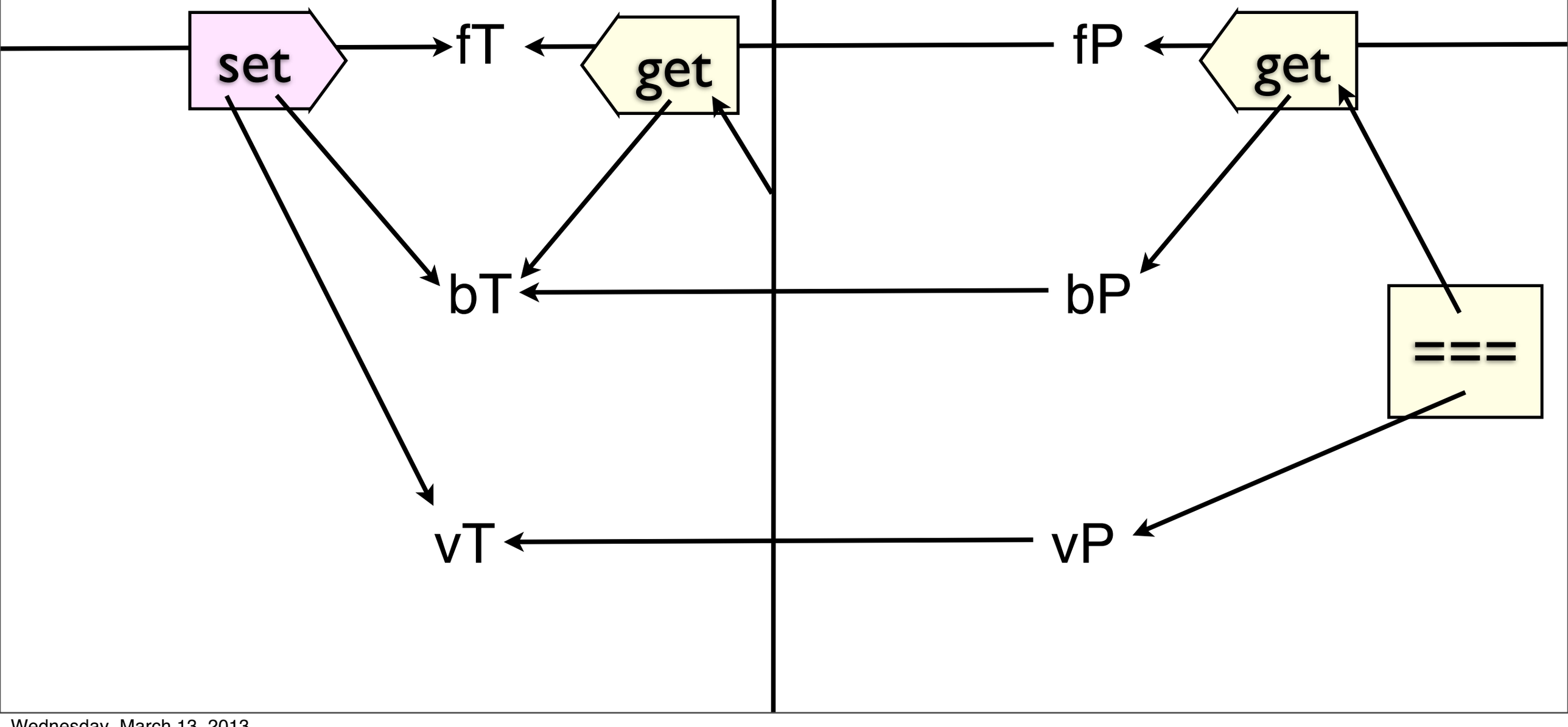


bT@fT = vT

fT.set(bT, vT)

bP@fP === vP

fP.get(bP) === vP

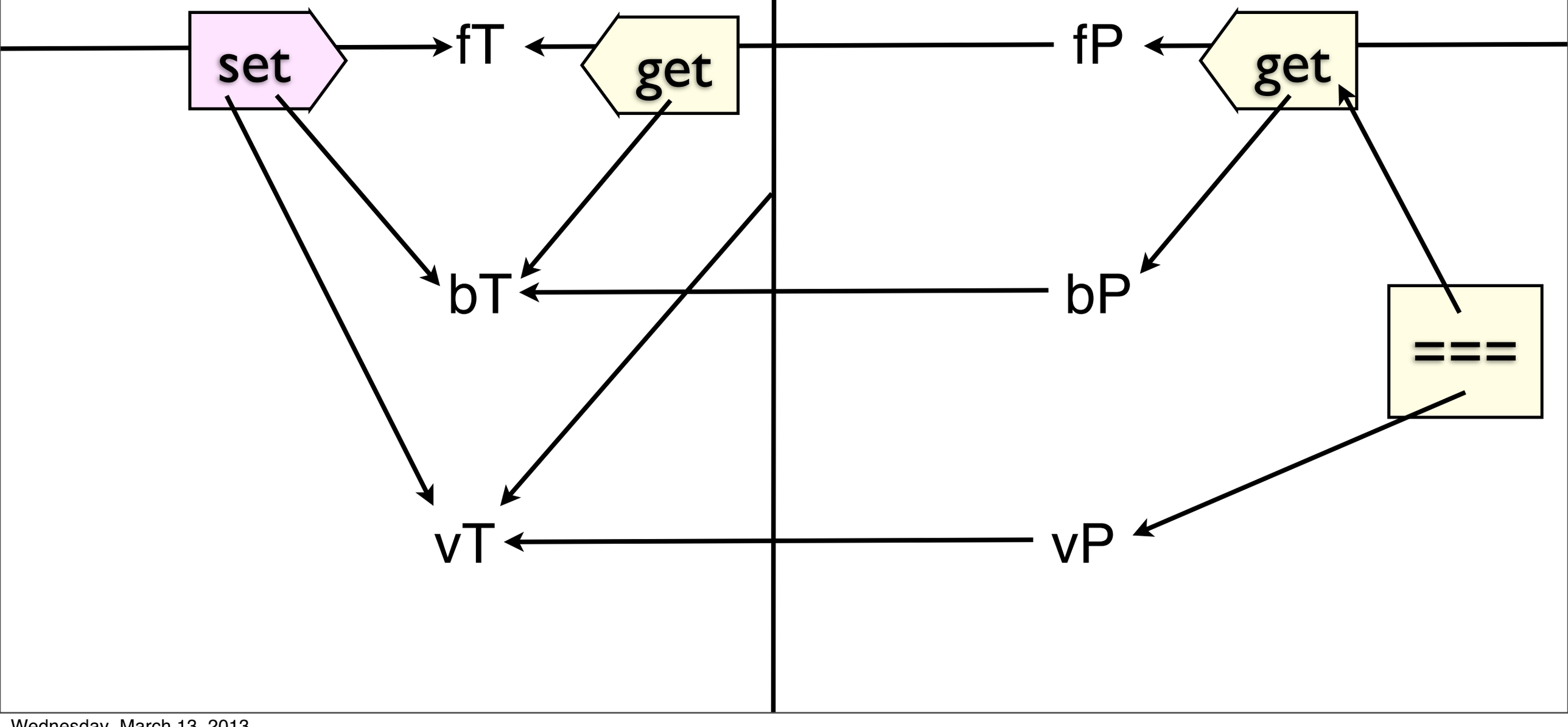


bT@fT = vT

fT.set(bT, vT)

bP@fP === vP

fP.get(bP) === vP

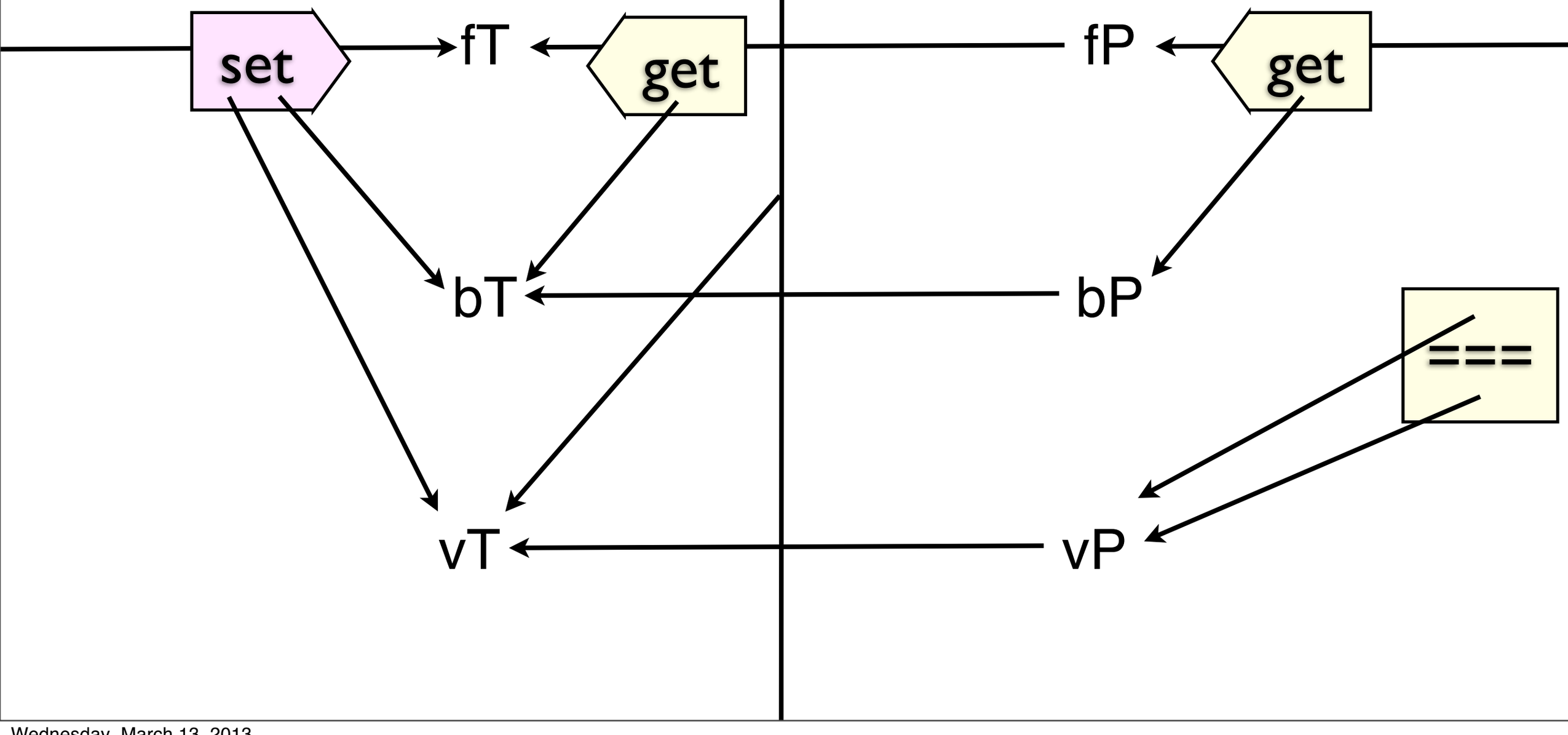


bT@fT = vT

fT.set(bT, vT)

bP@fP === vP

fP.get(bP) === vP

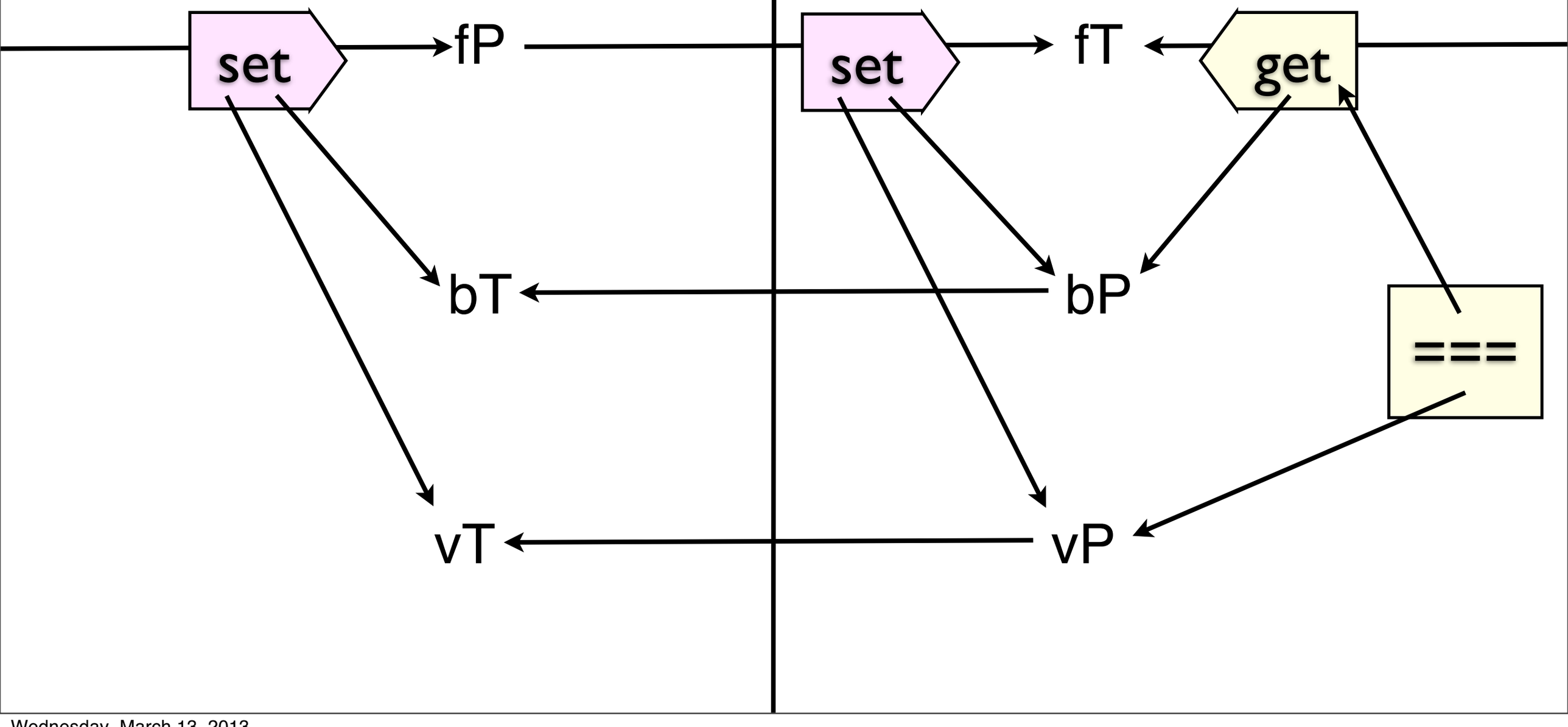


bT@fP = vT

fP.set(bT, vT)

bP@fT === vP

fT.get(bP) === vP

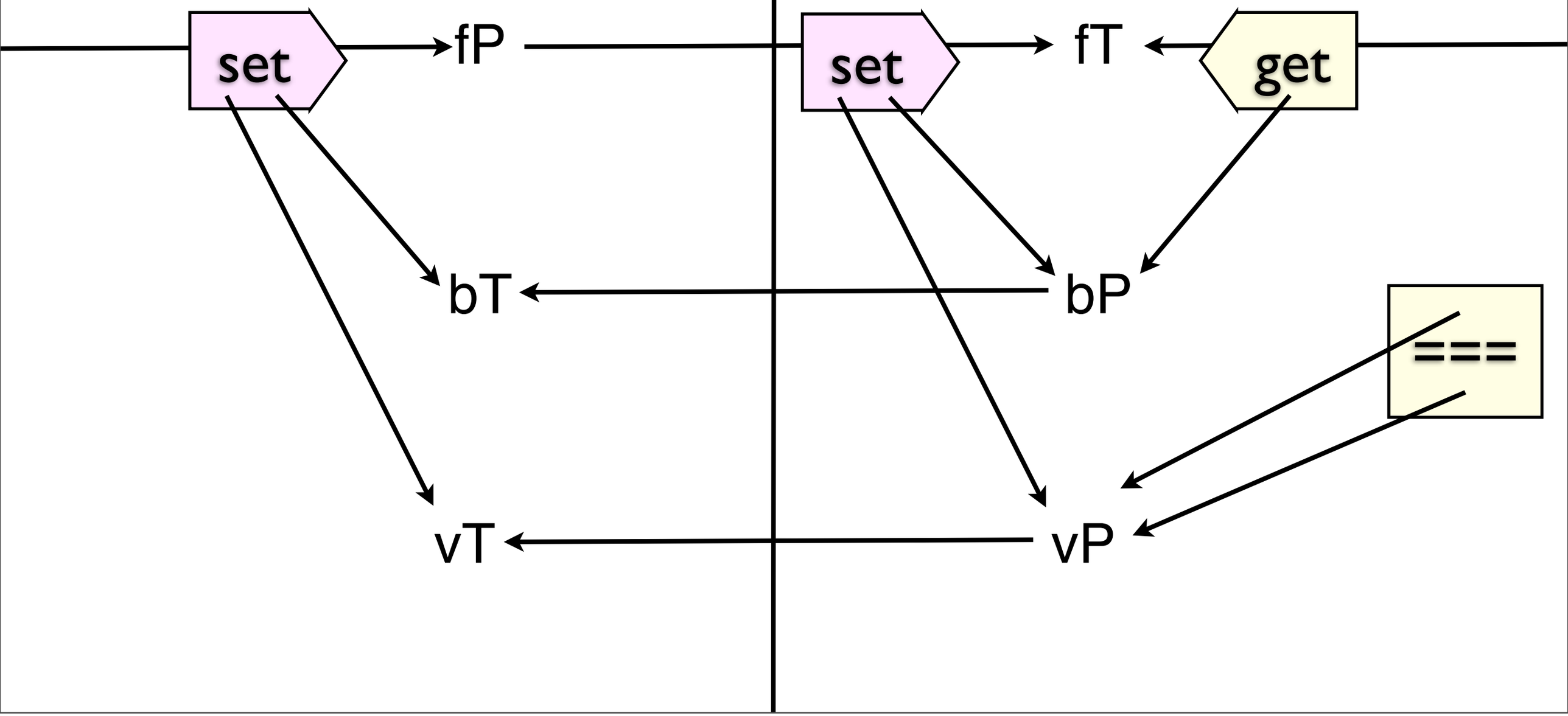


bT@fP = vT

fP.set(bT, vT)

bP@fT === vP

fT.get(bP) === vP



Desugaring private relationships

base@field
field.get(base)

base@field = value
field.set(base, value)

What about symbols and strings?

base@field
field.get(base)

base@field = value
field.set(base, value)

What about symbols and strings?

base@field

~~field.get(base)~~

field[@geti](base)

base@field = value

~~field.set(base, value)~~

field[@seti](base, value)

```
String.prototype[@geti] =
  function(base) {
    return base[this];
  };
```

```
String.prototype[@seti] =
  function(base, value) {
    base[this] = value;
  };
```

Private Instance Vars

```
class Point {
  constructor(private x, private y) {}

  toString() { return `<${this@x}, ${this@y}>`; }
  add(p) {
    return Point(this@x + p@x, this@y + p@y);
  }
}
```

```
let Point = (function(){
  const x = Rel(); const y = Rel();
  function Point(x1, y1) { this@x = x1; this@y = y1; }

  Point.prototype = {
    toString() { return `<${this@x}, ${this@y}>`; }
    add(p) {
      return Point(this@x + p@x, this@y + p@y);
    }
  };
  return Point;
})();
```