An executable formal semantics for PHP

Daniele Filaretti & Sergio Maffeis

www.phpsemantics.org

Imperial College London

ECOOP 2014
Uppsala, Sweden

phpMyAdmin

CodeIgniter

SquirrelMail
WEBMAIL FOR NUTS

Drupal™

SilverStripe®

CakePHP

php

MediaWiki

moodle

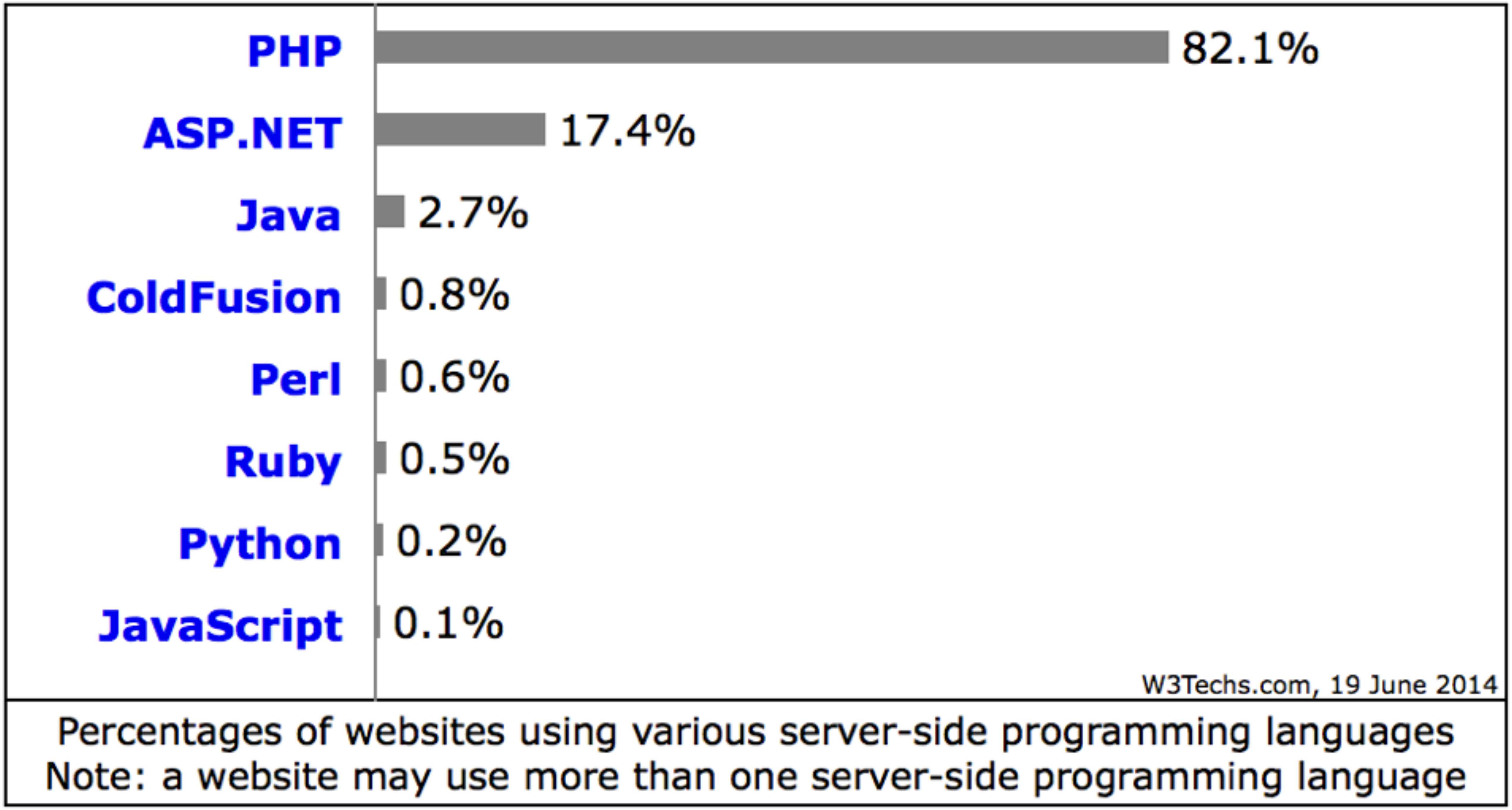
Joomla!™

phpBB™
creating communities

WordPress

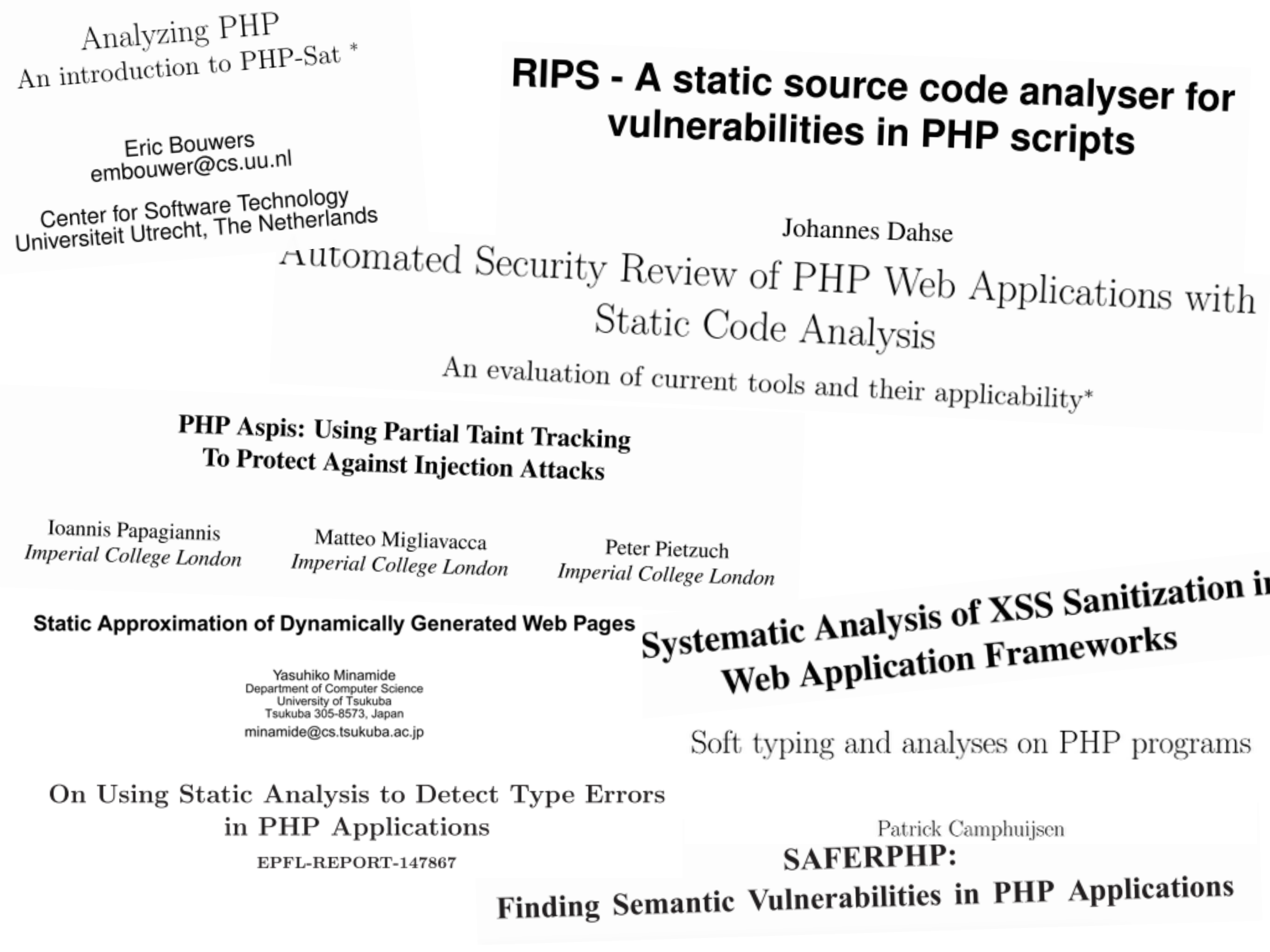
f

YAHOO!



Percentages of websites using various server-side programming languages
Note: a website may use more than one server-side programming language

http://w3techs.com/technologies/overview/programming_language/all



Analyzing PHP
An introduction to PHP-Sat *

Eric Bouwers
embouwer@cs.uu.nl

Center for Software Technology
Universiteit Utrecht, The Netherlands

RIPS - A static source code analyser for vulnerabilities in PHP scripts

Johannes Dahse

Automated Security Review of PHP Web Applications with Static Code Analysis

An evaluation of current tools and their applicability*

PHP Aspis: Using Partial Taint Tracking To Protect Against Injection Attacks

Ioannis Papagiannis
Imperial College London

Matteo Migliavacca
Imperial College London

Peter Pietzuch
Imperial College London

Static Approximation of Dynamically Generated Web Pages

Yasuhiko Minamide
Department of Computer Science
University of Tsukuba
Tsukuba 305-8573, Japan
minamide@cs.tsukuba.ac.jp

Systematic Analysis of XSS Sanitization i Web Application Frameworks

Soft typing and analyses on PHP programs

Patrick Camphuijsen

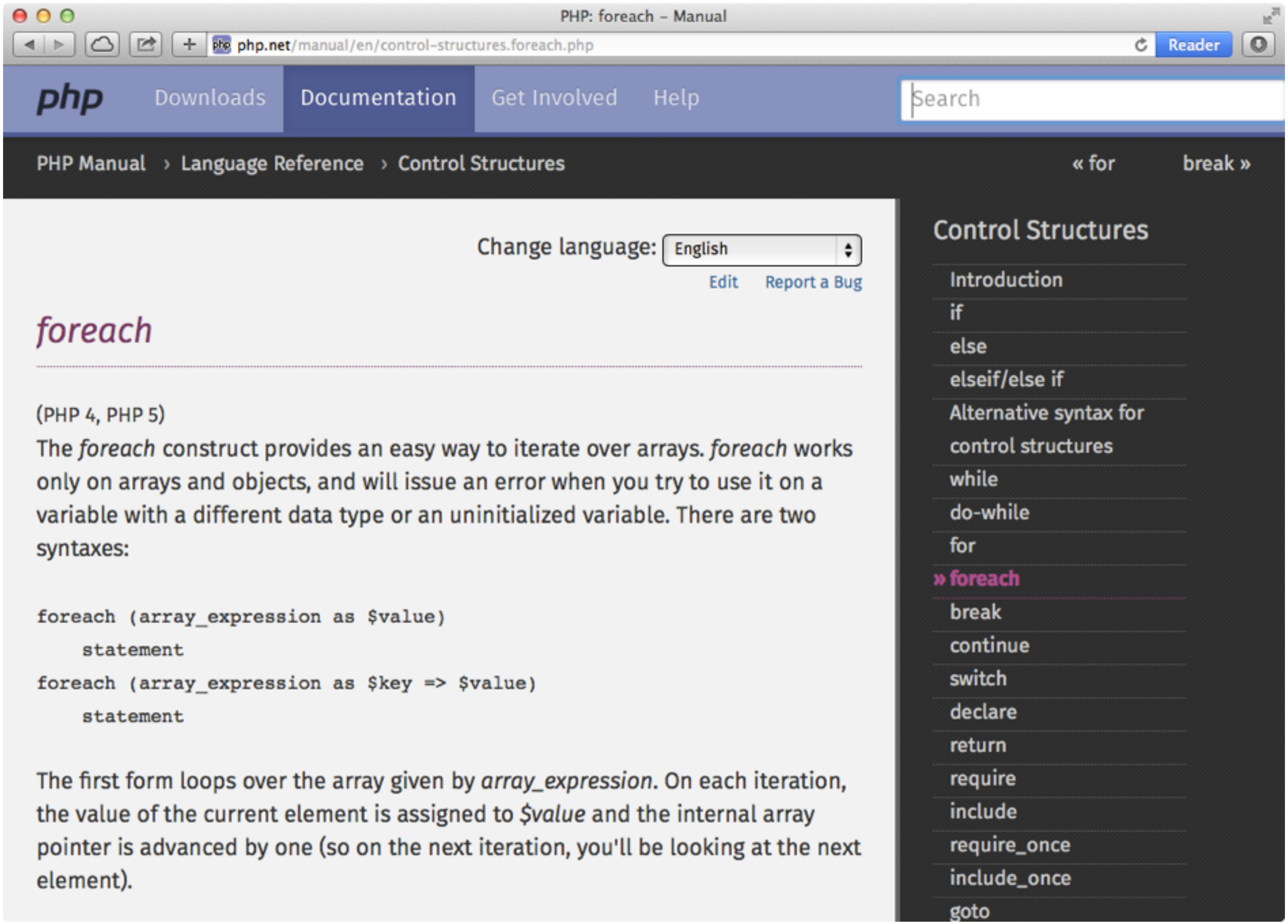
On Using Static Analysis to Detect Type Errors in PHP Applications

EPFL-REPORT-147867

SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications

Limitations

- **partial coverage** of the language - i.e. features ignored because “too hard” for analysis
- sometimes, **features modelled incorrectly**
- **no formal guarantees** of soundness



foreach

(PHP 4, PHP 5)

The *foreach* construct provides an easy way to iterate over arrays. *foreach* works only on arrays and objects, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variable. There are two syntaxes:

```
foreach (array_expression as $value)
    statement
foreach (array_expression as $key => $value)
    statement
```

The first form loops over the array given by *array_expression*. On each iteration, the value of the current element is assigned to *$value* and the internal array pointer is advanced by one (so on the next iteration, you'll be looking at the next element).

Absence of a specification

~~Absence of a specification~~

HHVM releases PHP spec on 30th July 2014

```php
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c;
```

```php
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"
```

```
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"

$a = "one";
$c = $a . ($a = "two");
echo $c;
```

```
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"

$a = "one";
$c = $a . ($a = "two");
echo $c; // "twotwo"
```

```php
$a = array("a", "b", "c");
foreach($a as &$v) {};
foreach($a as $v) {};
var_dump($a);
```

```
$a = array("a", "b", "c");
foreach($a as &$v) {};
foreach($a as $v) {};
var_dump($a);
```

```
  array(3) {
    [0]=> string(1) "a"
    [1]=> string(1) "b"
    [2]=> &string(1) "b"
  }
```

```
$x = array(1, 2, 3);
$y = $x;
$x[0] = "updated";
echo $y[0];          // prints 1
```

```php
$x = array(1, 2, 3);
$temp = &$x[1]; // aliasing!
$y = $x;// assign normally
$x[0] = “regular"; // no shared
$x[1] = "shared"; // shared
```

```
$x = array(1, 2, 3);
$temp = &$x[1]; // aliasing!
$y = $x;// assign normally
$x[0] = “regular"; // no shared
$x[1] = "shared"; // shared
```

$x

```
array (3) {
  [0]=> string(7) "regular"
  [1]=> &string(6) "shared"
  [2]=> int(3)
}
```

$y

```
array (3) {
  [0]=> int(1)
  [1]=> &string(6) "shared"
  [2]=> int(3)
}
```

PHP tricky features

- **Aliasing**
- Complex array and object **iteration**
- Automatic **type conversions**
- Complex **array copy**
- Complex instance variable **lookup**
- Variable variables

Challenge

- Dynamic scripting language
- Not specified
- Documentation often incomplete
- **Source of confusion** for developers but also security specialists, tool designers etc.

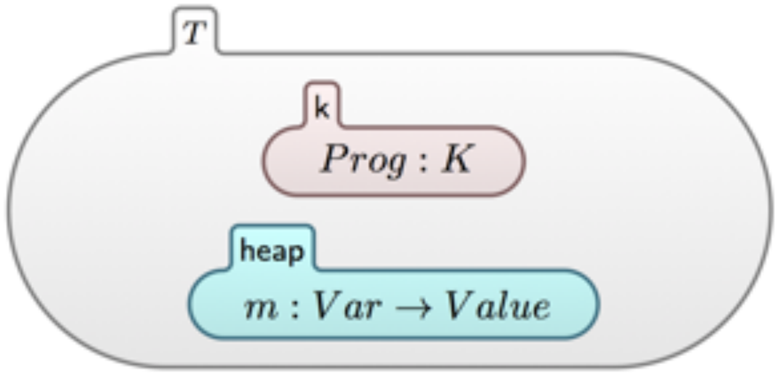
Contribution: A *Trusted* Executable Formal Semantics of PHP

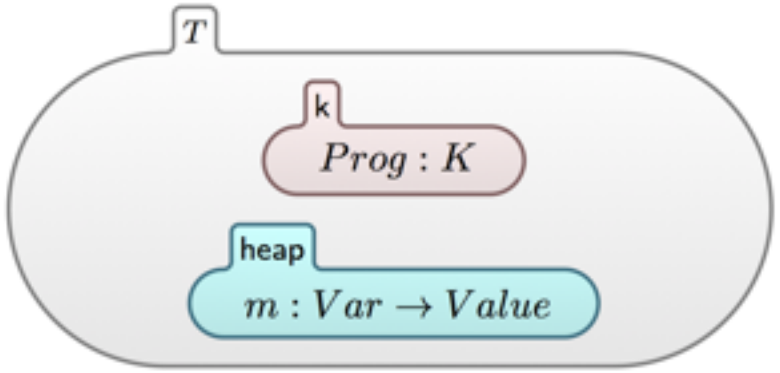
foundation for *reliable* tool development

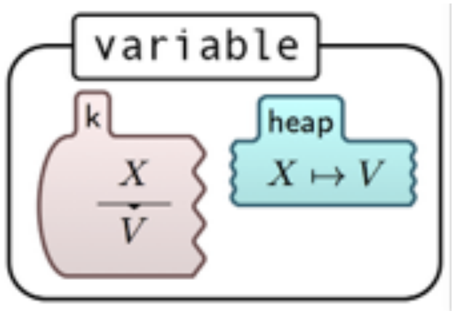
the missing *specification*

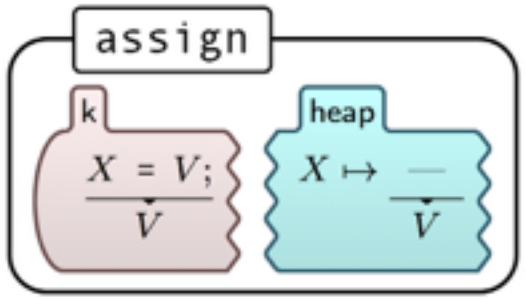
Methodology

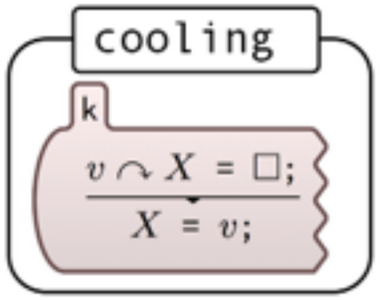
- Semantics must be based on experiments and testing against the "reference" implementation
  - Need a tight test-design loop
- We use the K Framework (UIUC)
  - amenable to formal proofs
  - executable
  - supports LTL model checking and symbolic execution

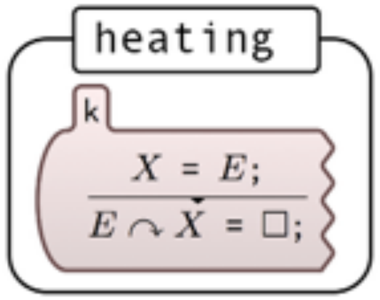

T
k
$Prog : K$
heap
$m : Var \rightarrow Value$

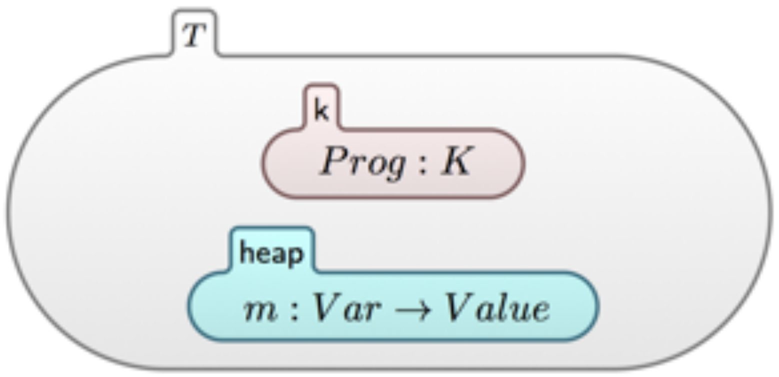

T
k
Prog : K
heap
m : Var → Value

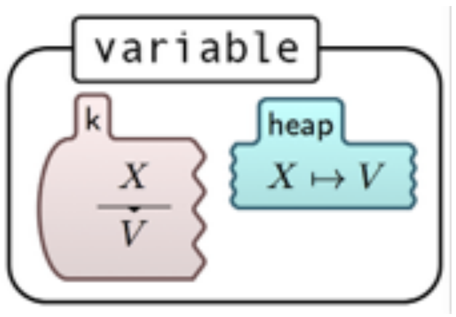

variable
k
X
V
heap
X ↦ V

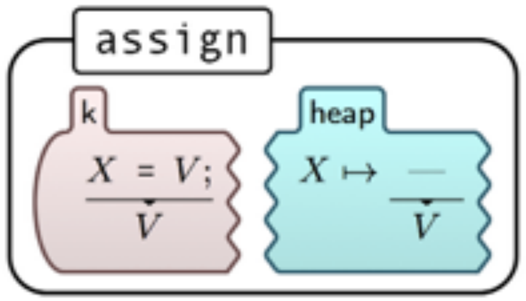

assign
k
X = V;
V
heap
X ↦ —
V

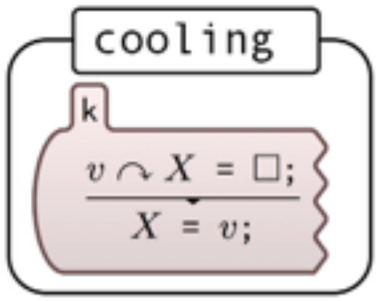

cooling
k
v ↷ X = □;
X = v;

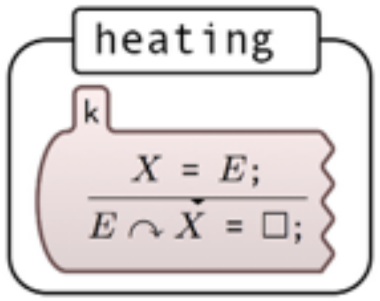

heating
k
X = E;
E ↷ X = □;


T
k
Prog : K
heap
m : Var → Value

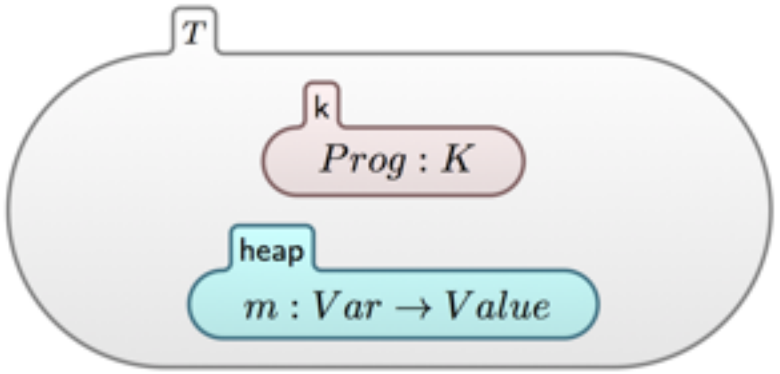

variable
k
X
V
heap
X ↦ V

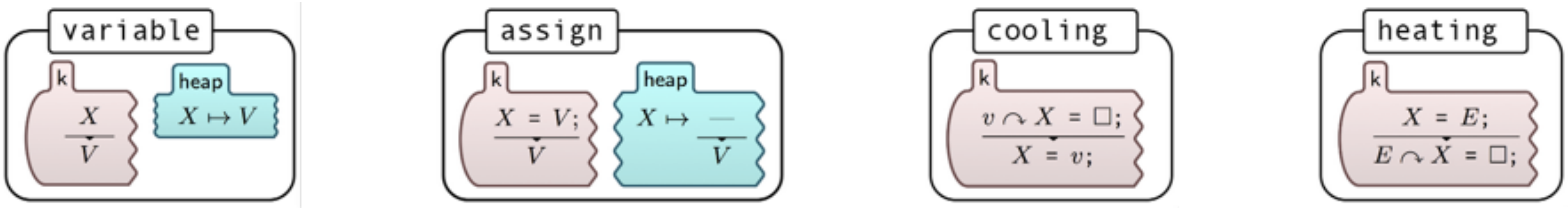

assign
k
X = V;
V
heap
X ↦ —
V


cooling
k
v ↷ X = □;
X = v;

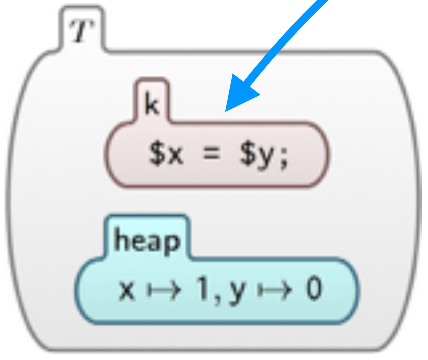

heating
k
X = E;
E ↷ X = □;

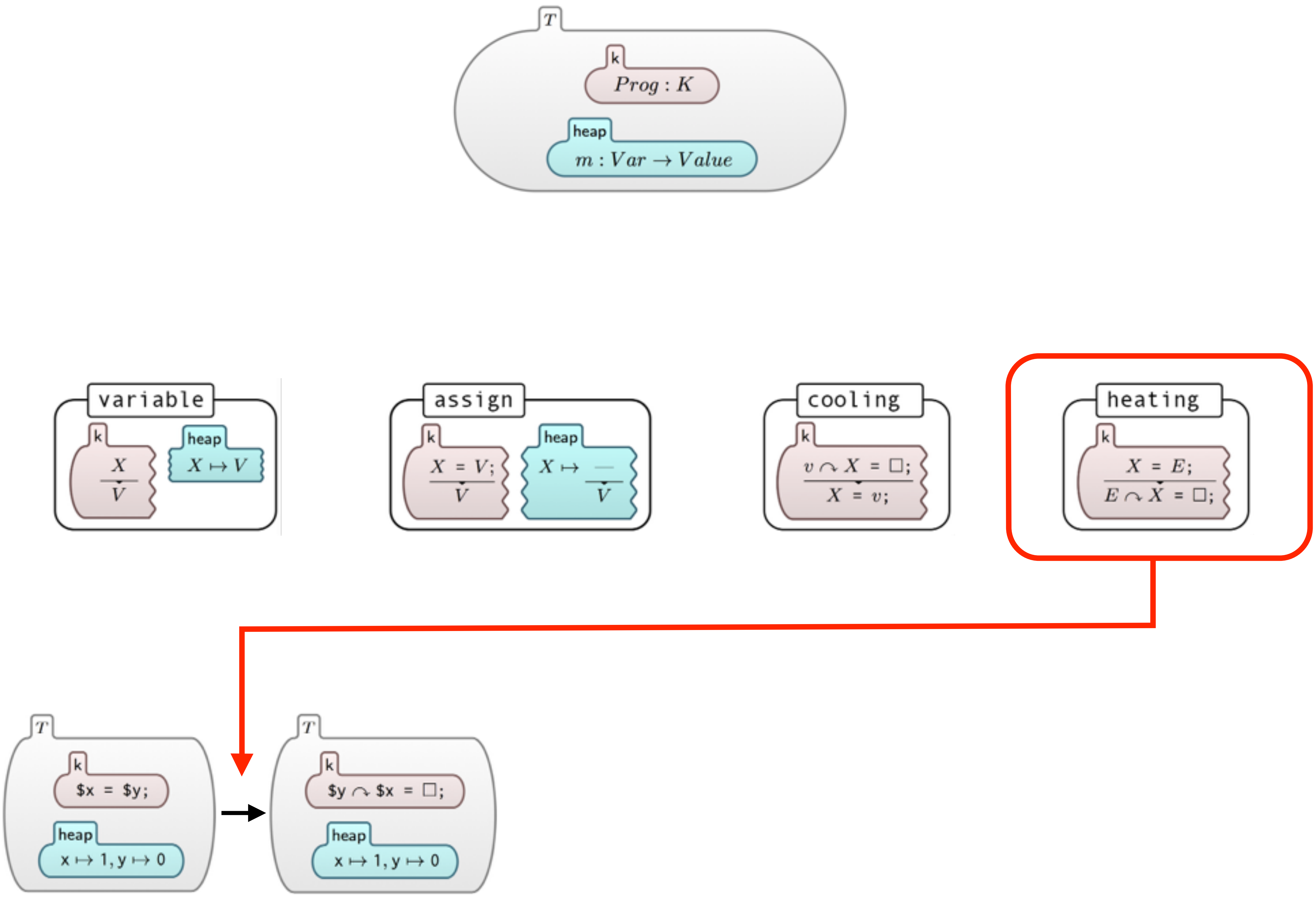
```
$x = $y
```

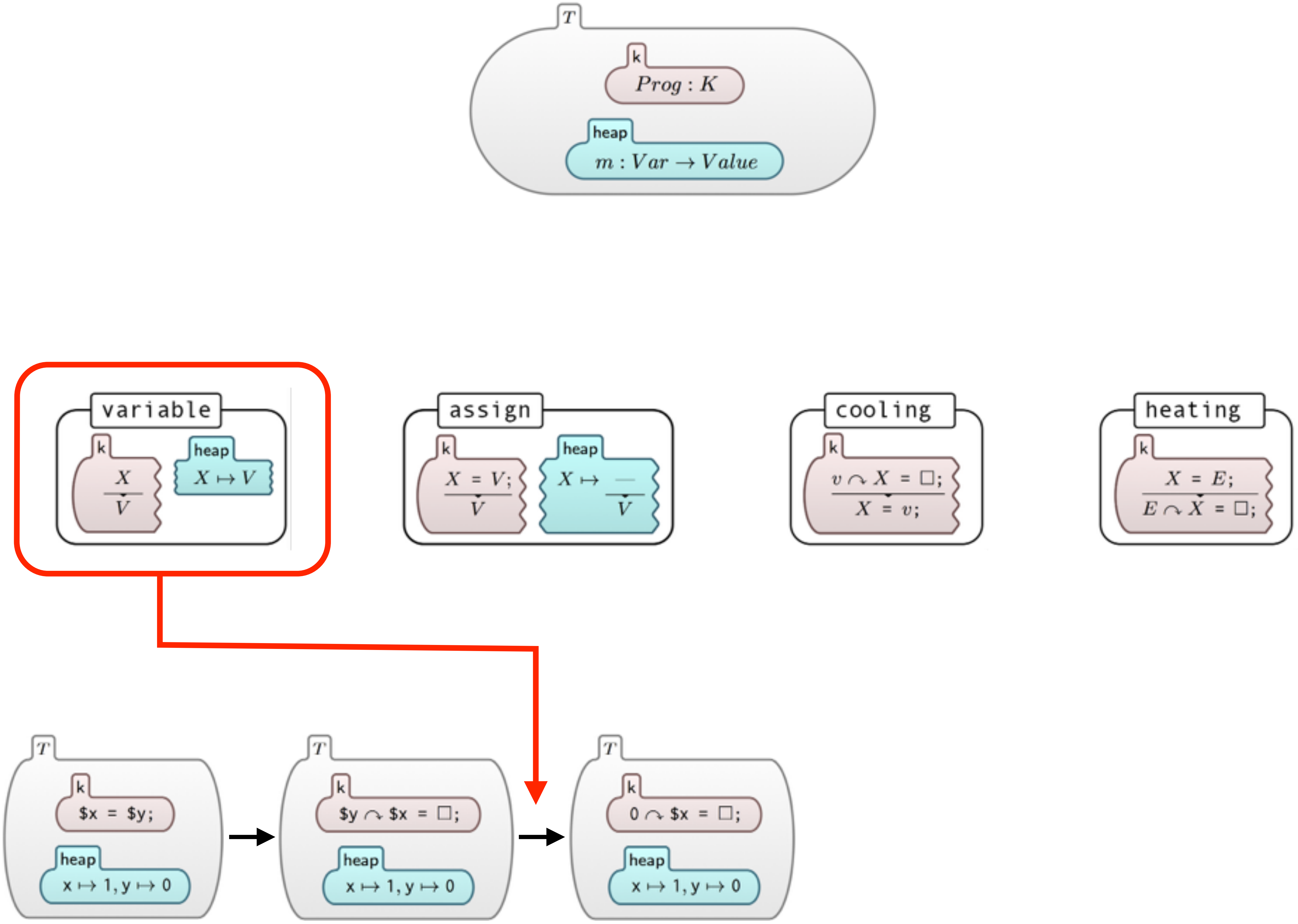

T
k
Prog : K
heap
m : Var → Value

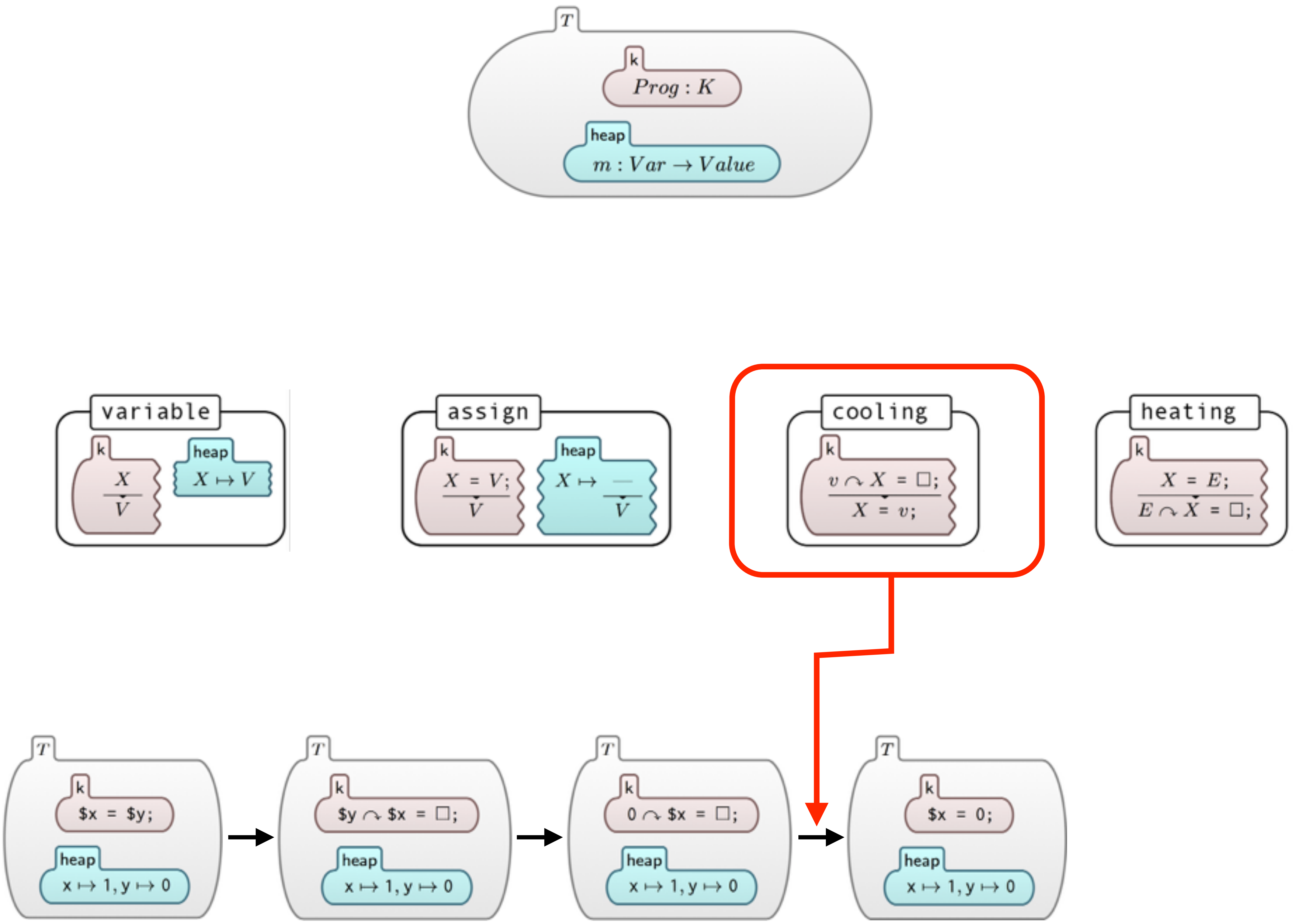

variable
k
X / V
heap
X ↦ V
assign
k
X = V; / V
heap
X ↦ — / V
cooling
k
v ↷ X = □; / X = v;
heating
k
X = E; / E ↷ X = □;

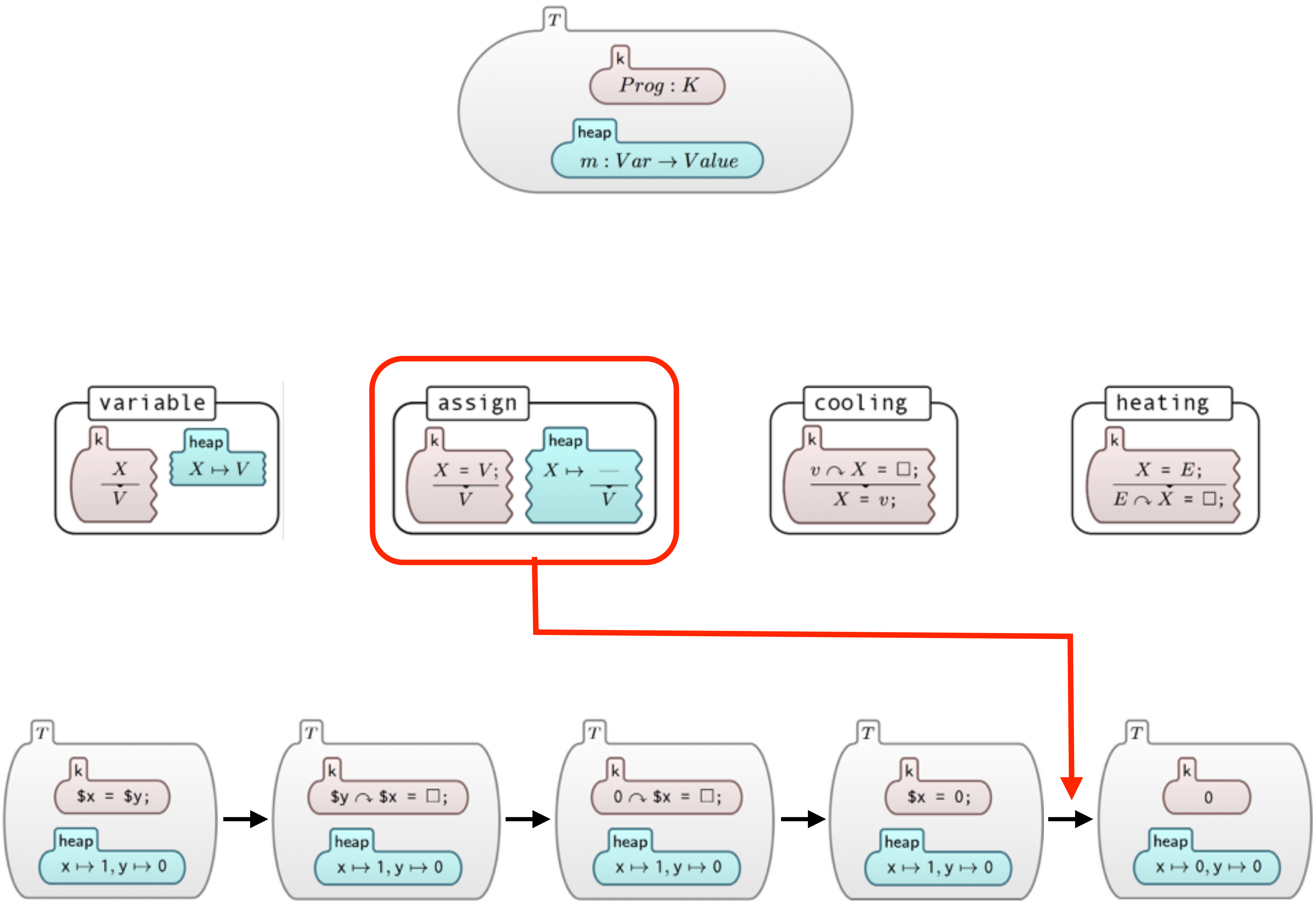
$x = $y


T
k
$x = $y;
heap
x ↦ 1, y ↦ 0


T
k
Prog : K
heap
m : Var → Value
variable
k
X / V
heap
X ↦ V
assign
k
X = V; / V
heap
X ↦ — / V
cooling
k
v ↷ X = □; / X = v;
heating
k
X = E; / E ↷ X = □;
T
k
$x = $y;
heap
x ↦ 1, y ↦ 0
T
k
$y ↷ $x = □;
heap
x ↦ 1, y ↦ 0


T
k
Prog : K
heap
m : Var → Value
variable
k
X / V
heap
X ↦ V
assign
k
X = V; / V
heap
X ↦ — / V
cooling
k
v ↷ X = □; / X = v;
heating
k
X = E; / E ↷ X = □;
T
k
$x = $y;
heap
x ↦ 1, y ↦ 0
T
k
$y ↷ $x = □;
heap
x ↦ 1, y ↦ 0
T
k
0 ↷ $x = □;
heap
x ↦ 1, y ↦ 0


T
k
$Prog : K$
heap
$m : Var \to Value$
variable
k
X / V
heap
$X \mapsto V$
assign
k
X = V; / V
heap
$X \mapsto$ — / V
cooling
k
$v \curvearrowright X$ = □; / X = v;
heating
k
X = E; / $E \curvearrowright X$ = □;
T
k
$x = $y;
heap
x ↦ 1, y ↦ 0
T
k
$y ↷ $x = □;
heap
x ↦ 1, y ↦ 0
T
k
0 ↷ $x = □;
heap
x ↦ 1, y ↦ 0
T
k
$x = 0;
heap
x ↦ 1, y ↦ 0

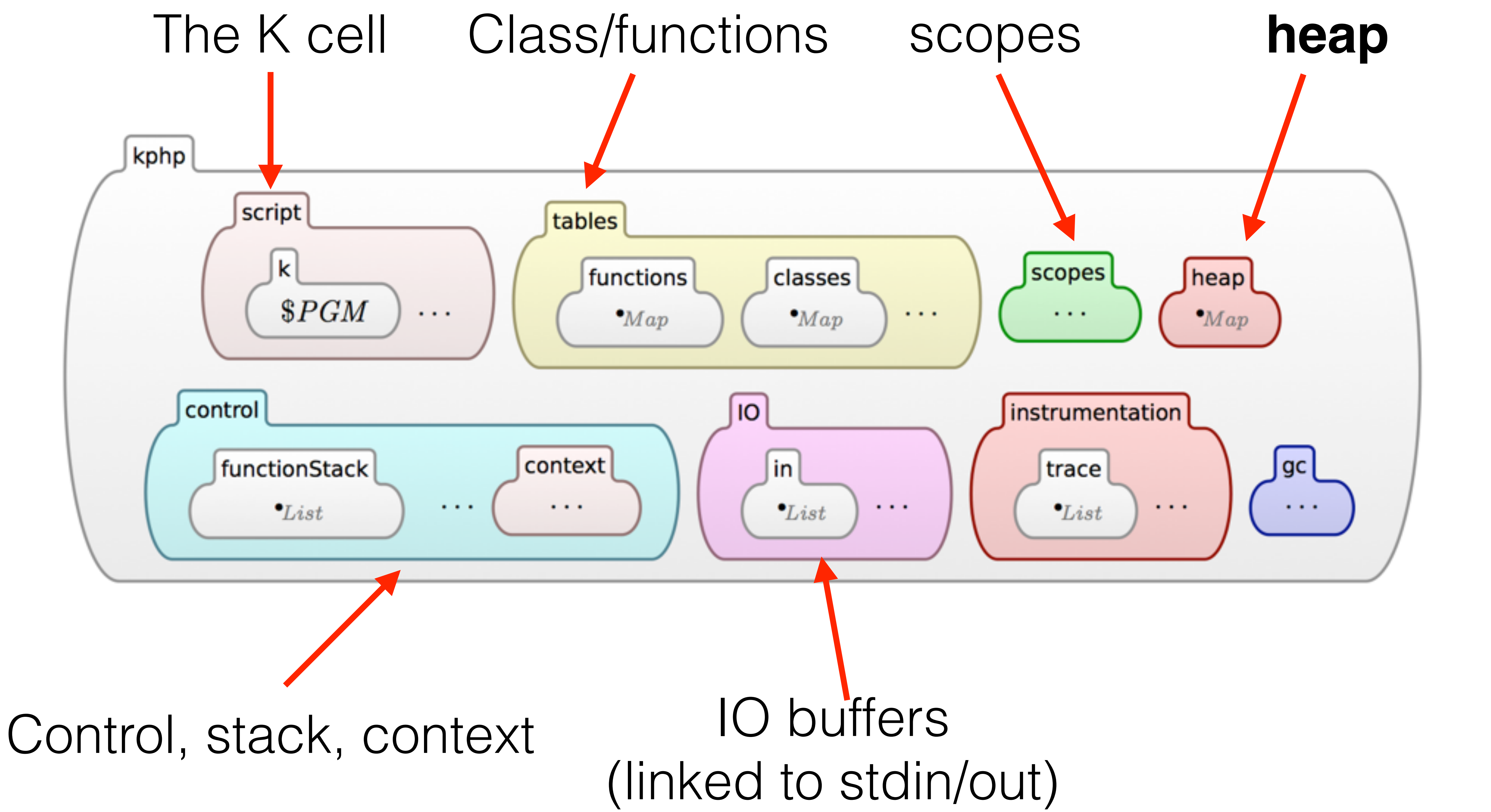

T
k
Prog : K
heap
m : Var → Value
variable
k
X
V
heap
X ↦ V
assign
k
X = V;
V
heap
X ↦
V
cooling
k
v ↷ X = □;
X = v;
heating
k
X = E;
E ↷ X = □;
T
k
$x = $y;
heap
x ↦ 1, y ↦ 0
T
k
$y ↷ $x = □;
heap
x ↦ 1, y ↦ 0
T
k
0 ↷ $x = □;
heap
x ↦ 1, y ↦ 0
T
k
$x = 0;
heap
x ↦ 1, y ↦ 0
T
k
0
heap
x ↦ 0, y ↦ 0

KPHP
(Formalising PHP in K)

www.phpsemantics.org

Artifact * Consistent * Complete * Well Documented * Easy to Reuse * Evaluated * ECOOP * AEC *

Configuration (~30 cells)



Language Values

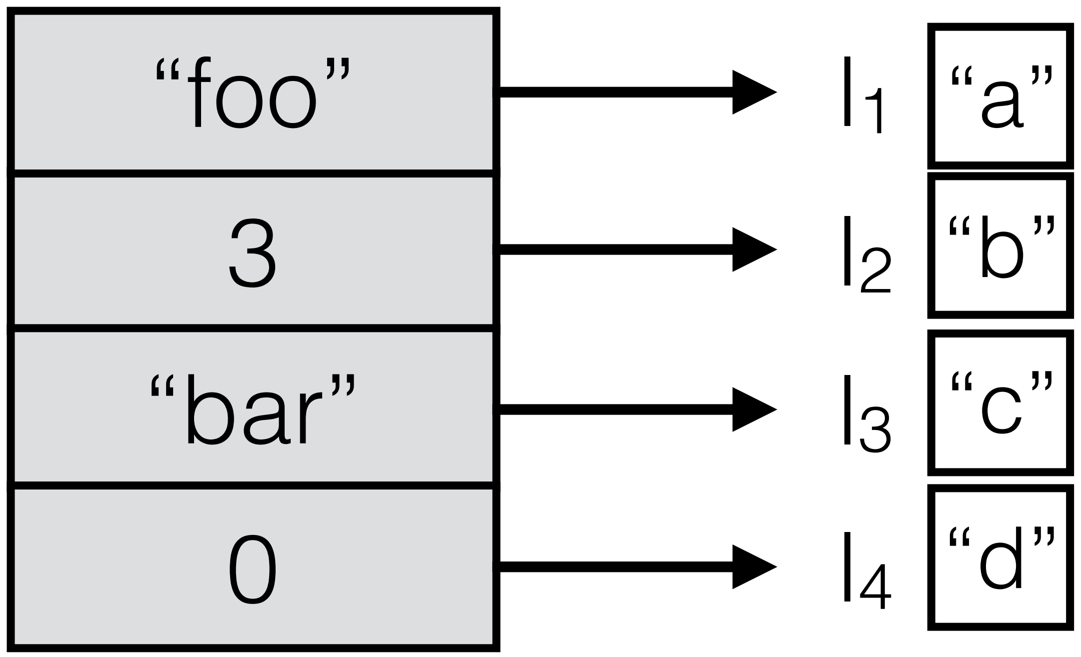
Scalar	Compound	Special
`boolean`	`array`	`resource`
`integer`	`object`	`NULL`
`float`		
`string`		

Language Values

Scalar	Compound	Special
boolean	array	~~resource~~
integer	object	NULL
float		
string		

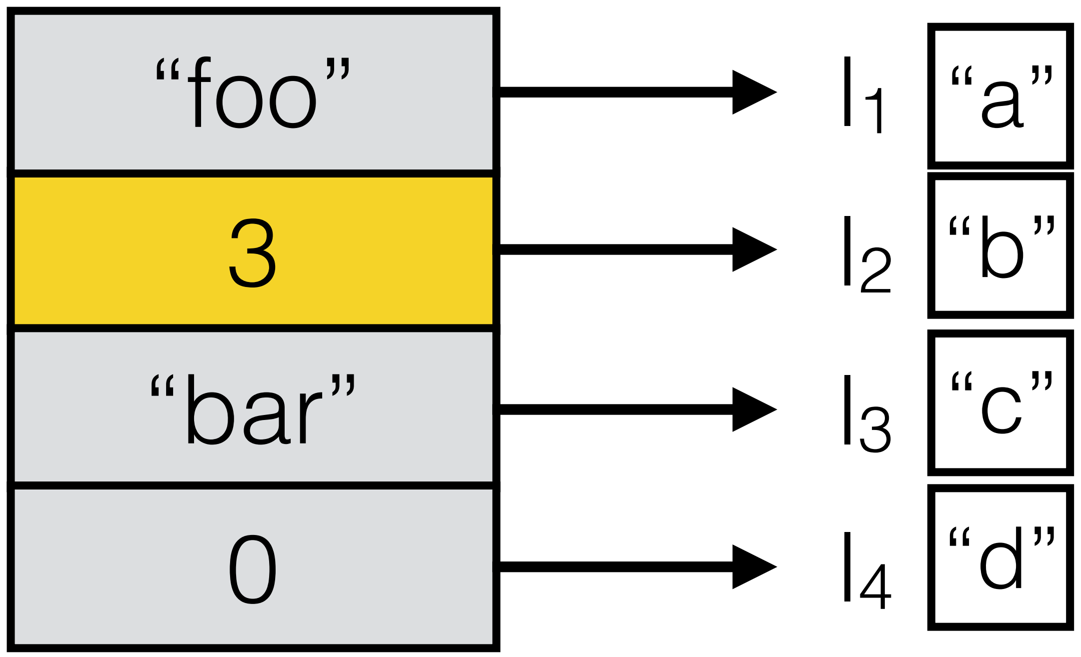
Arrays

Int U String —> Locations



Arrays

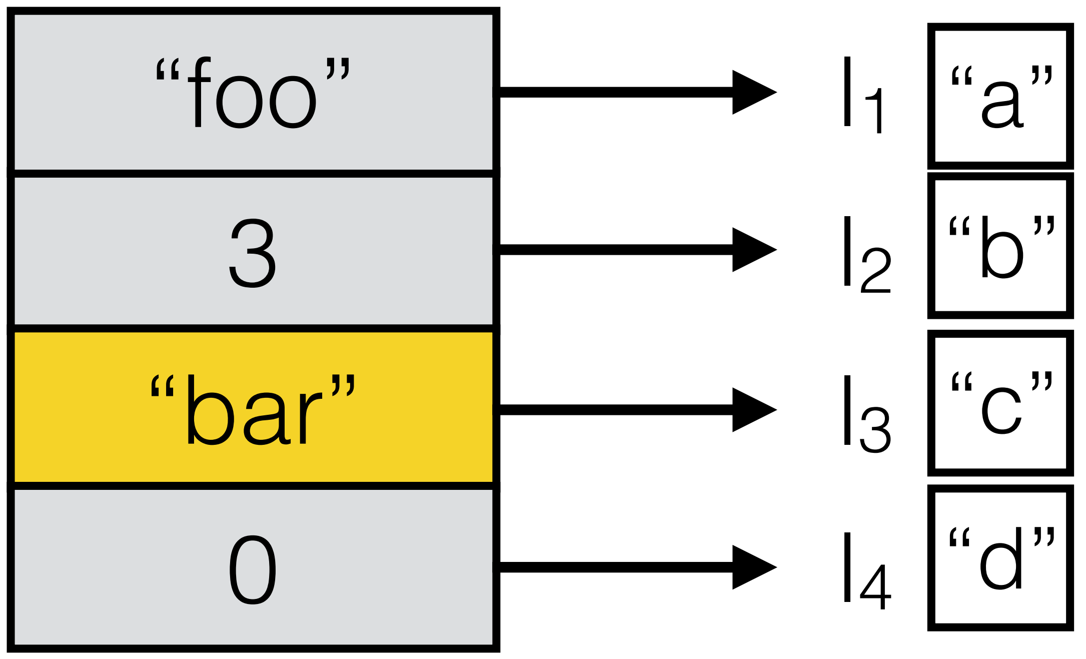
Int U String —> Locations



```
echo current($x) // "b"
next($x);
echo current($) // "bar"
```

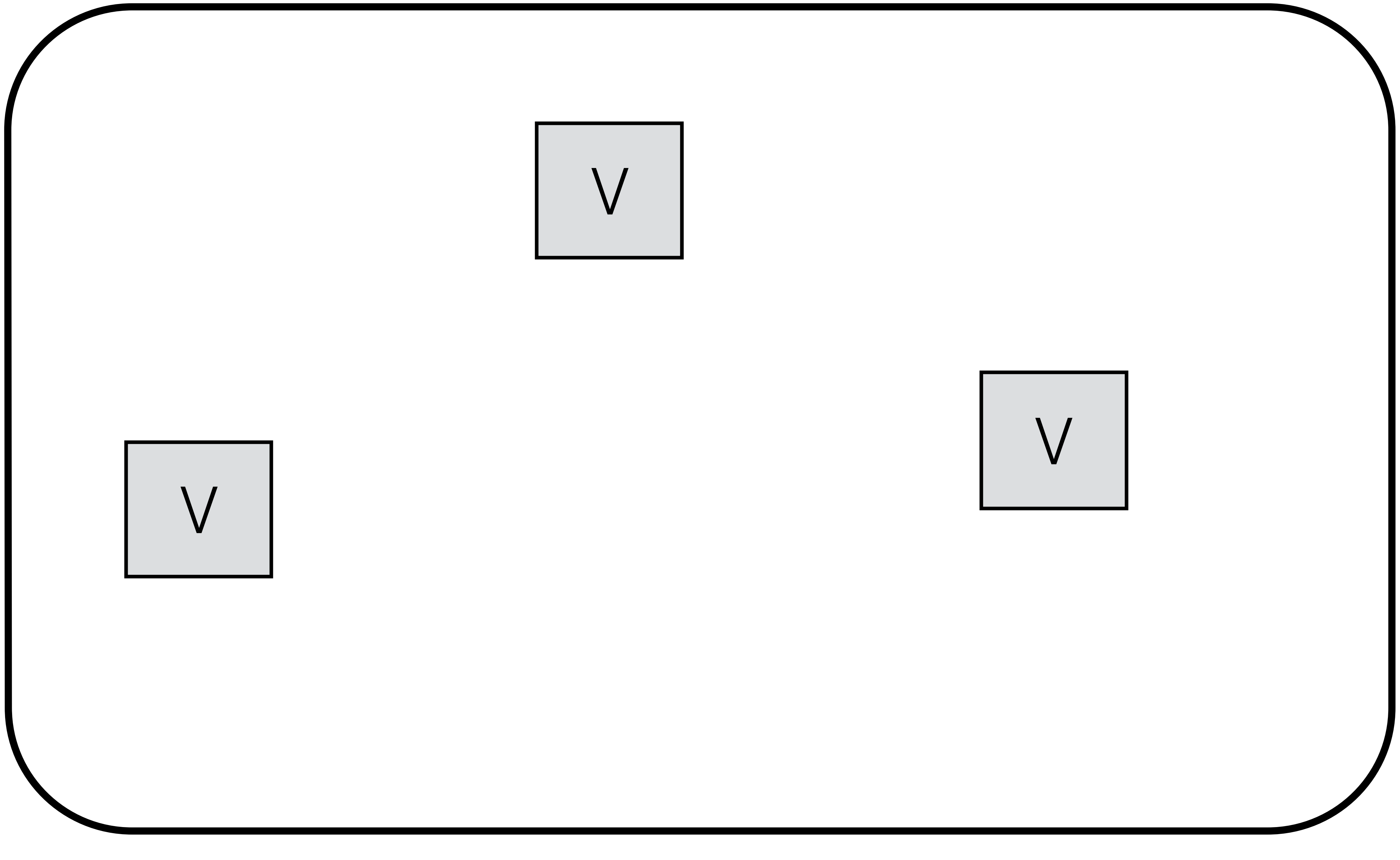
Arrays

Int U String —> Locations

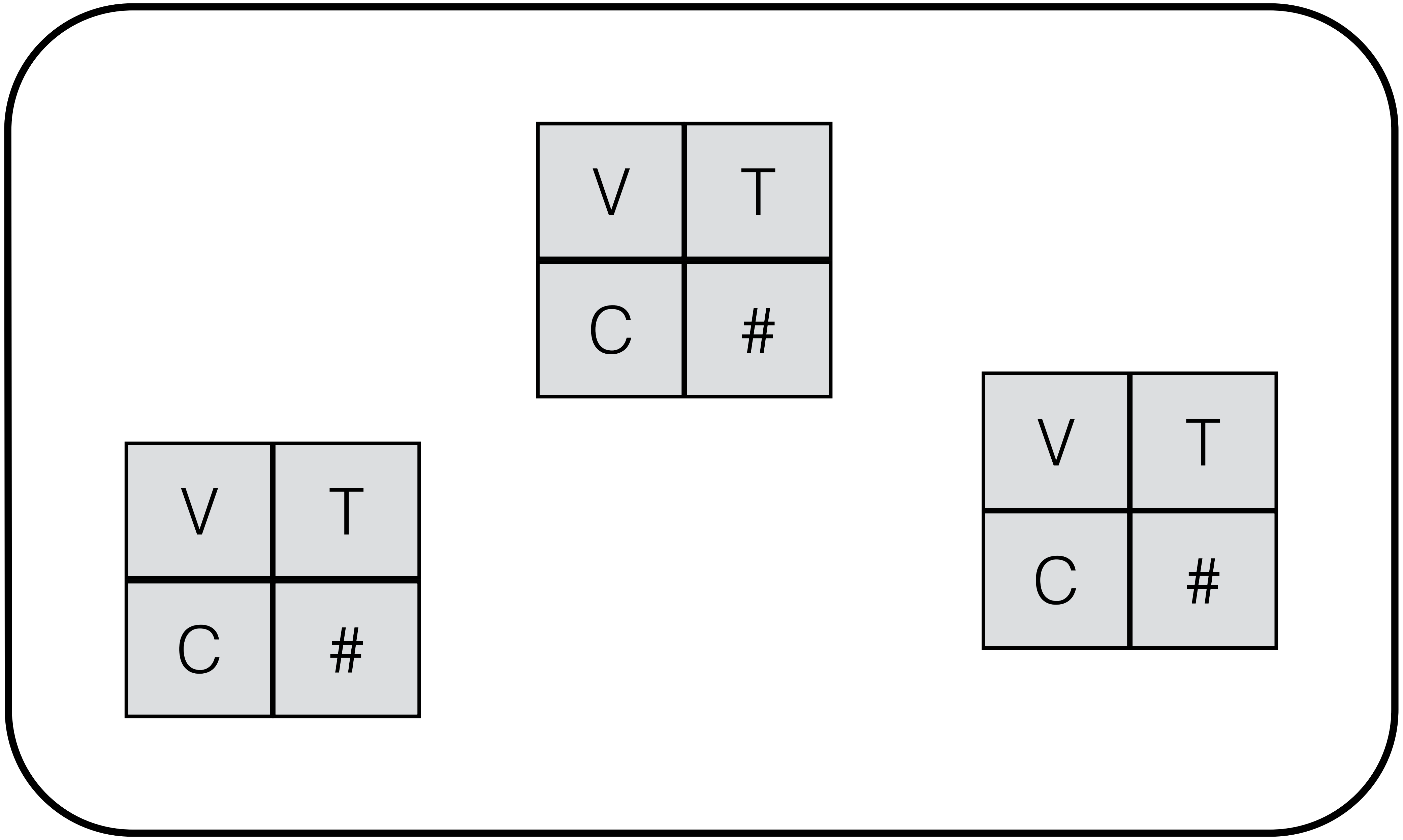


```
echo current($x) // "b"
next($x);
echo current($x) // "c"
```

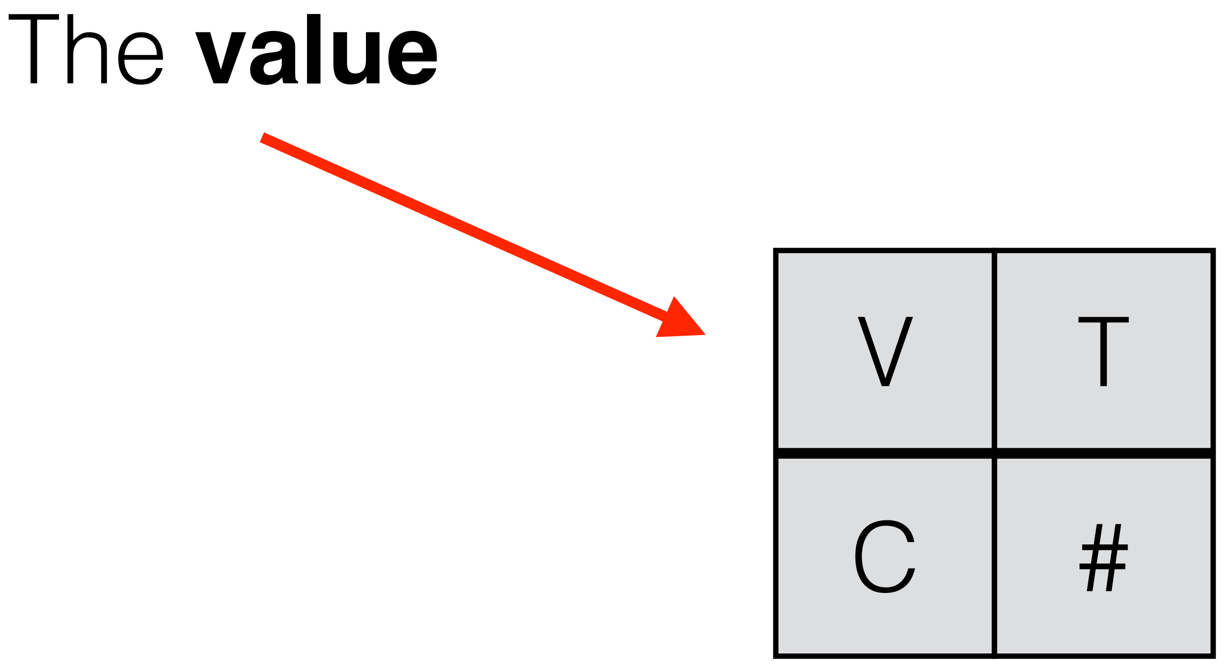
Values and Z-Values



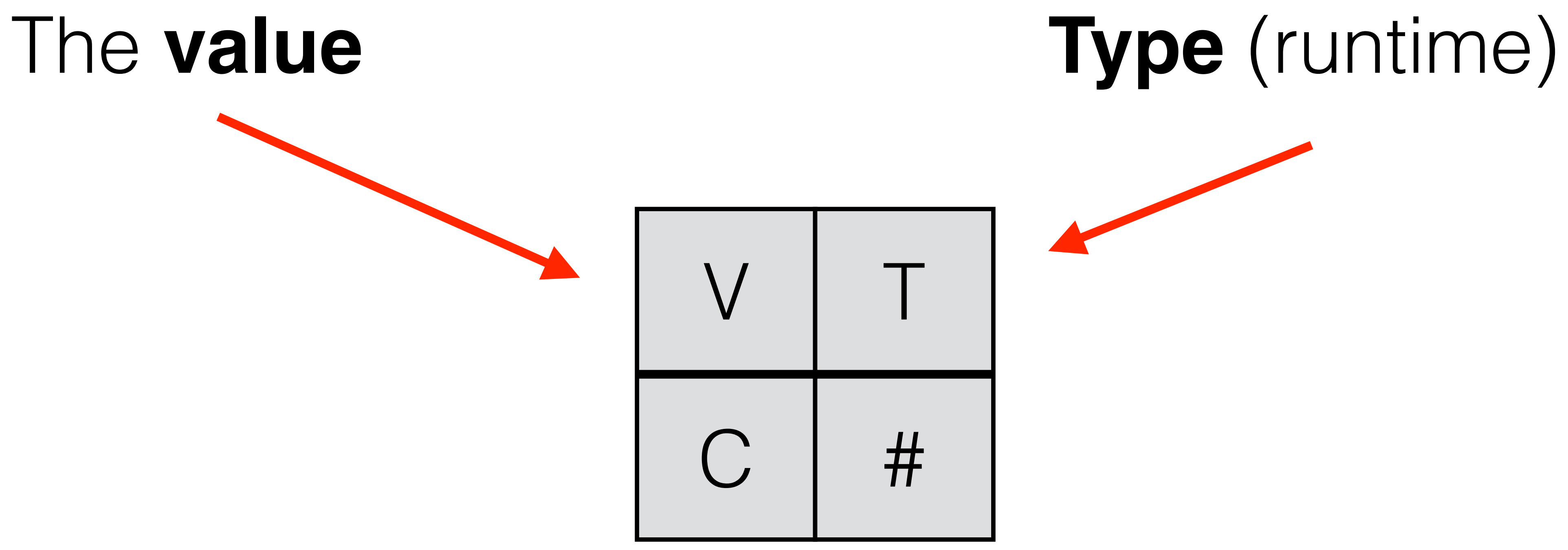
Values and **Z-Values**



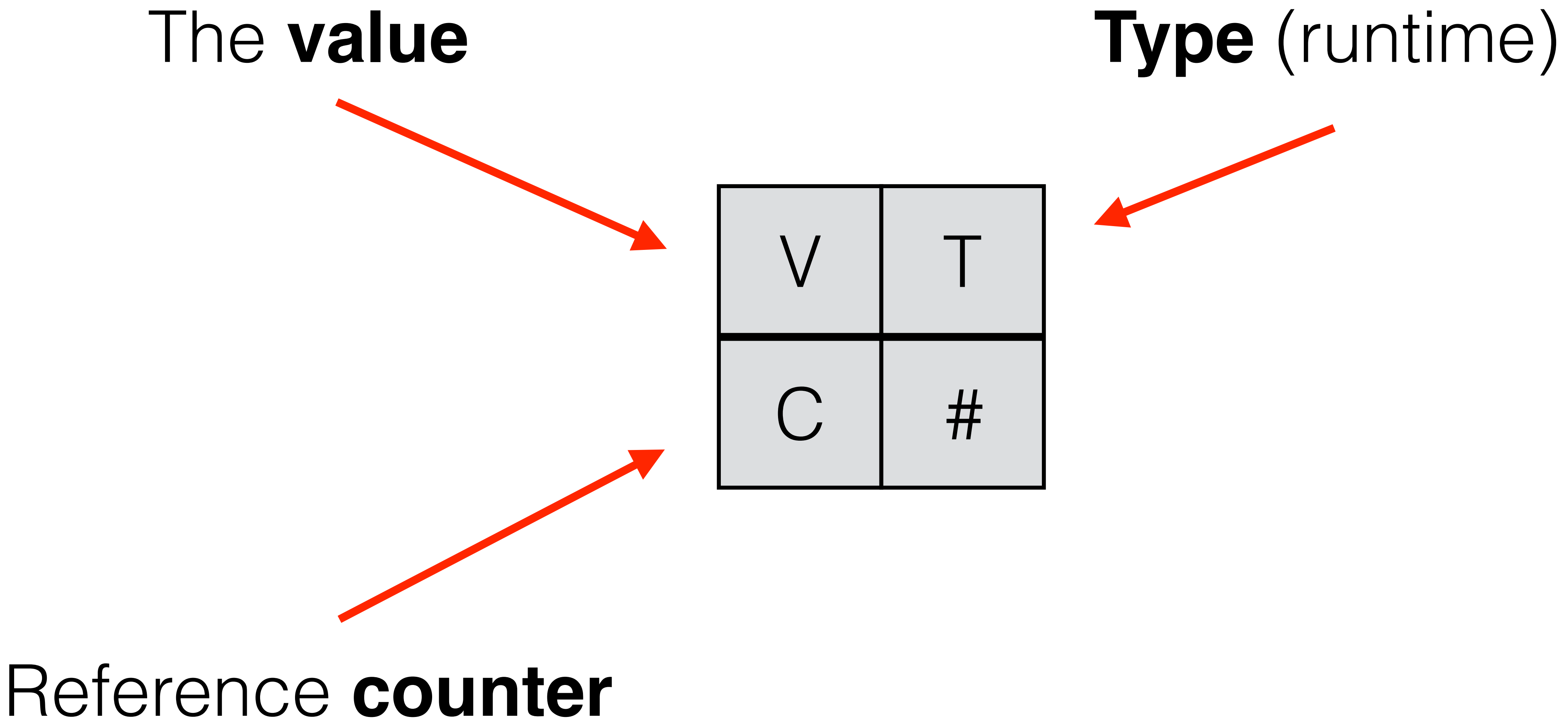
Z-Values



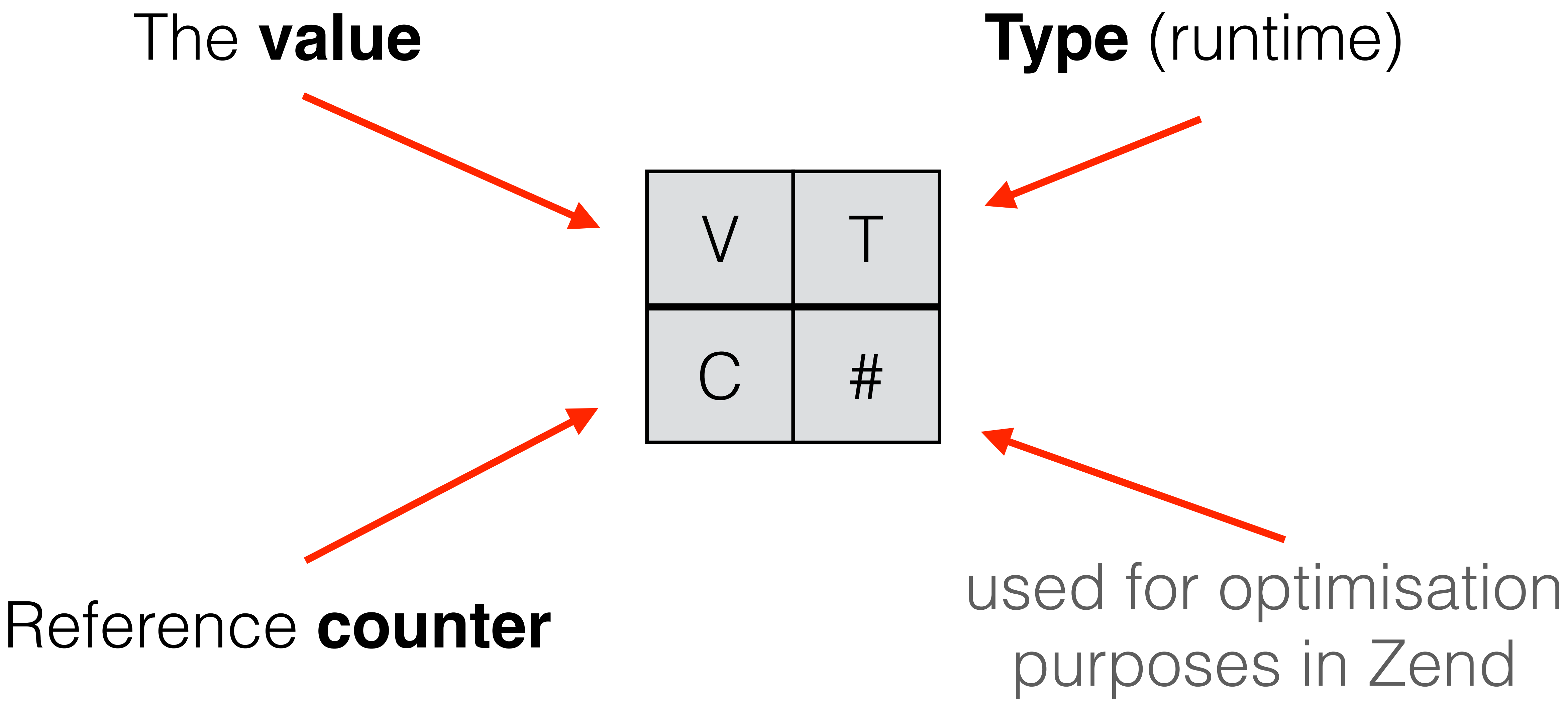
Z-Values



Z-Values

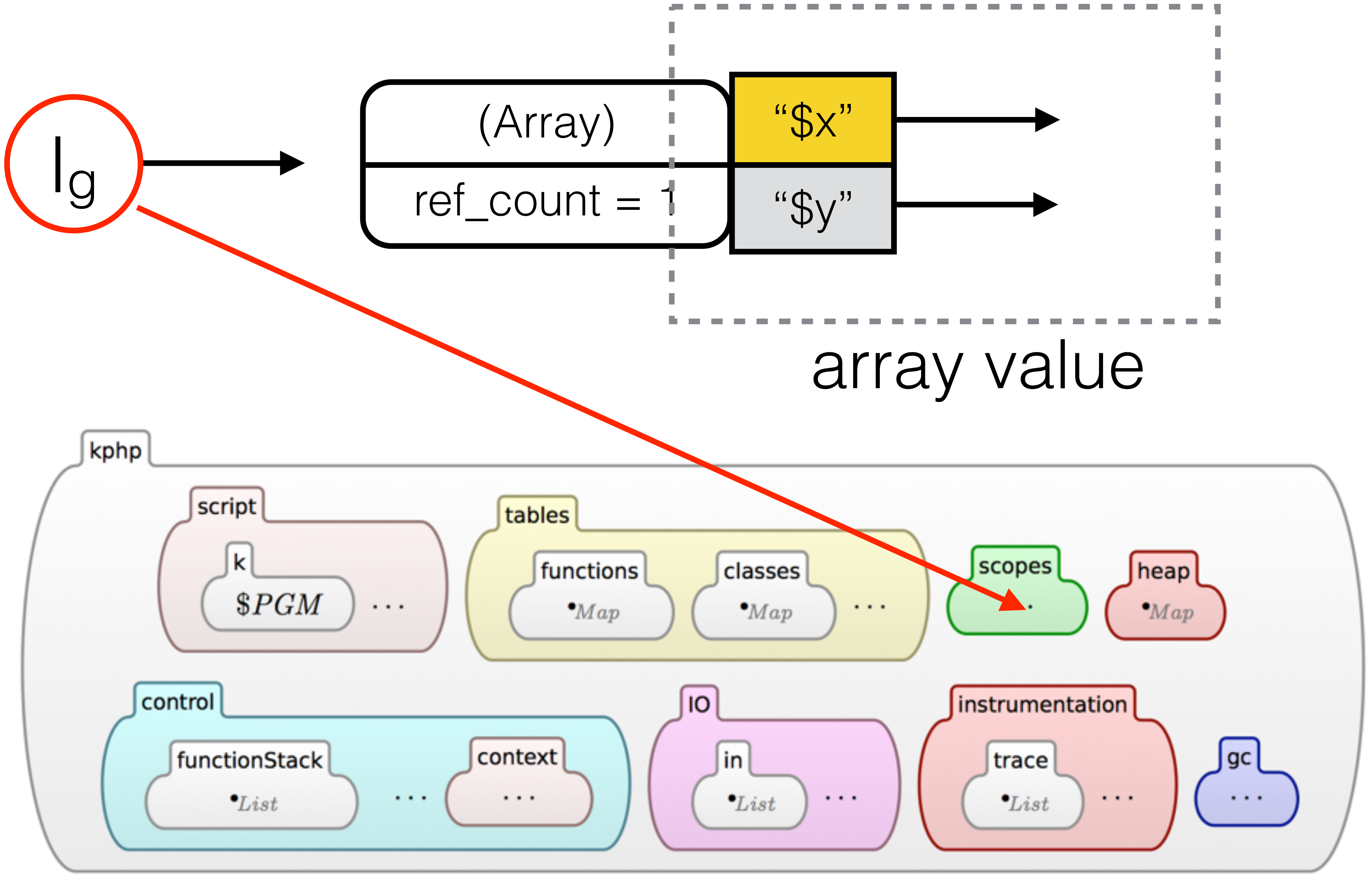


Z-Values


The **value**
Type (runtime)
V
T
C
#
Reference **counter**
used for optimisation purposes in Zend

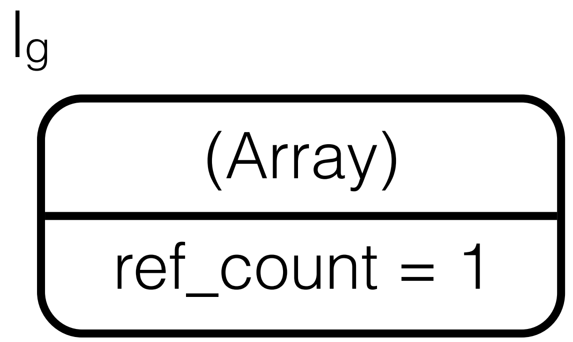
Scopes via arrays

heap: Loc -> Z-Value



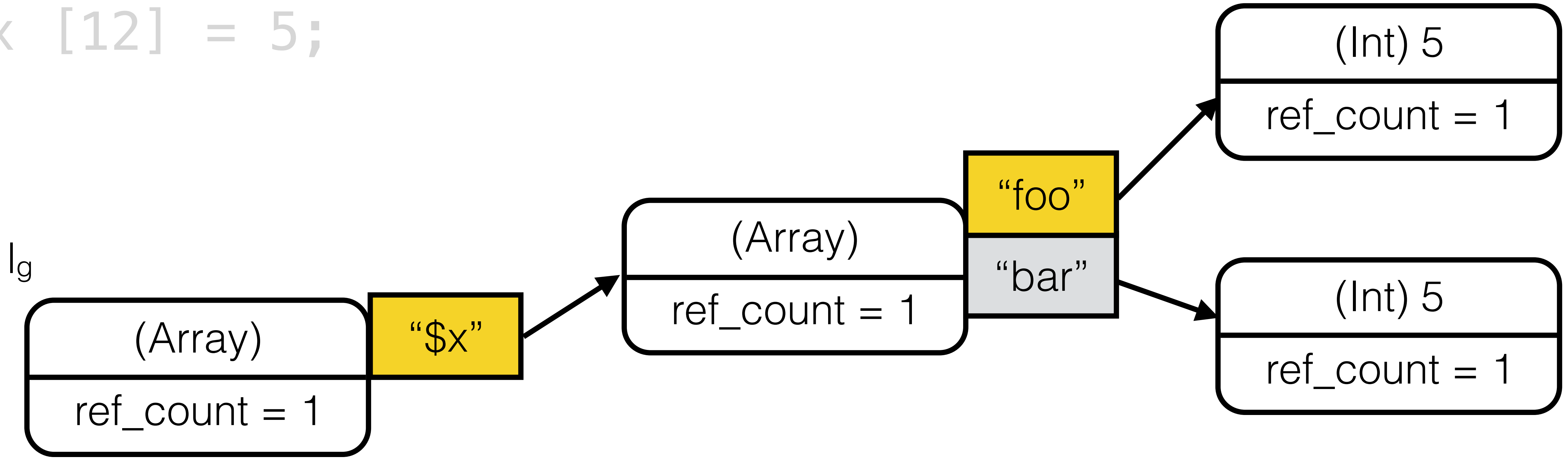
Memory - example

```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



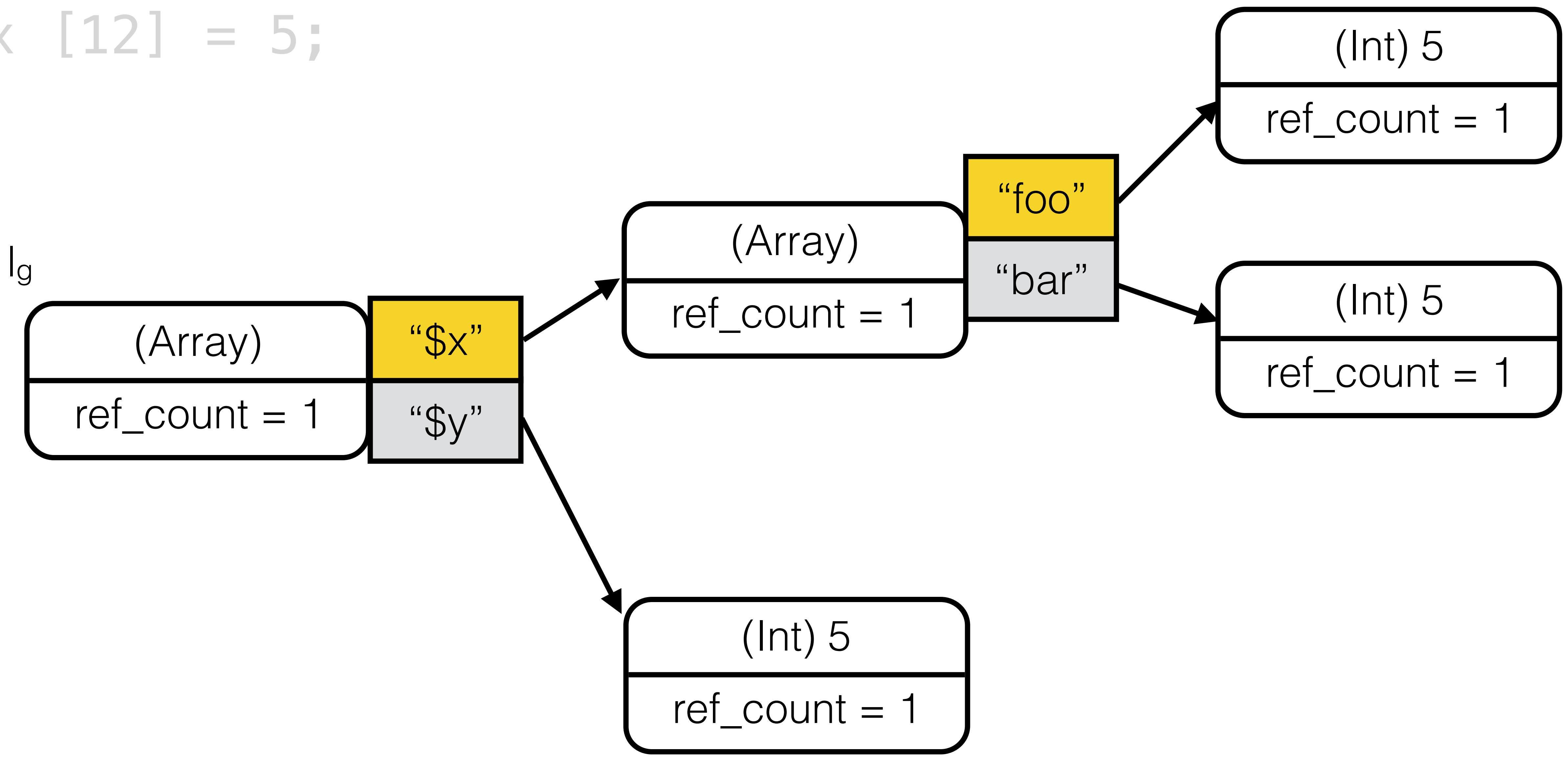
Memory - example

```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



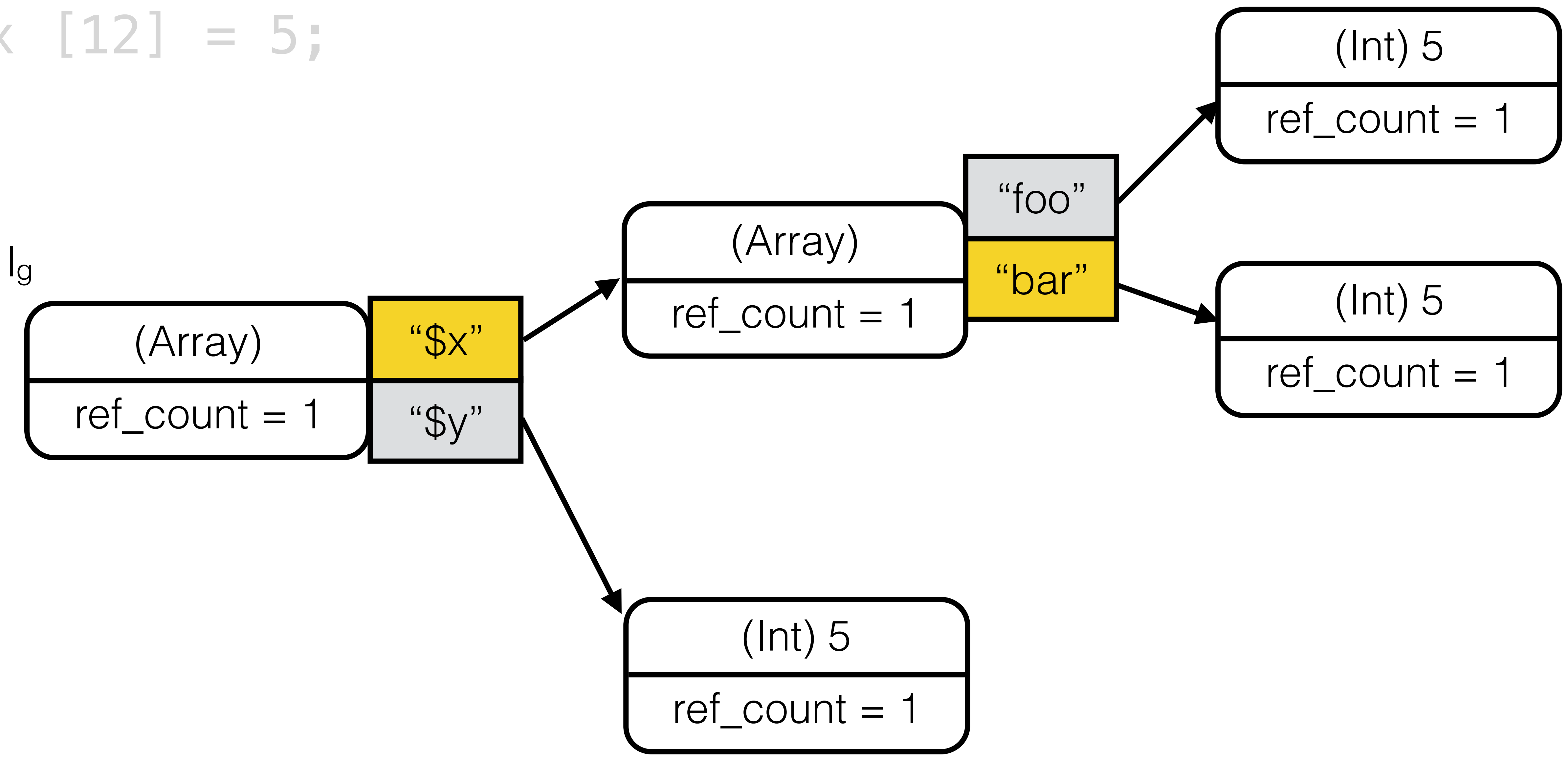
Memory - example

```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



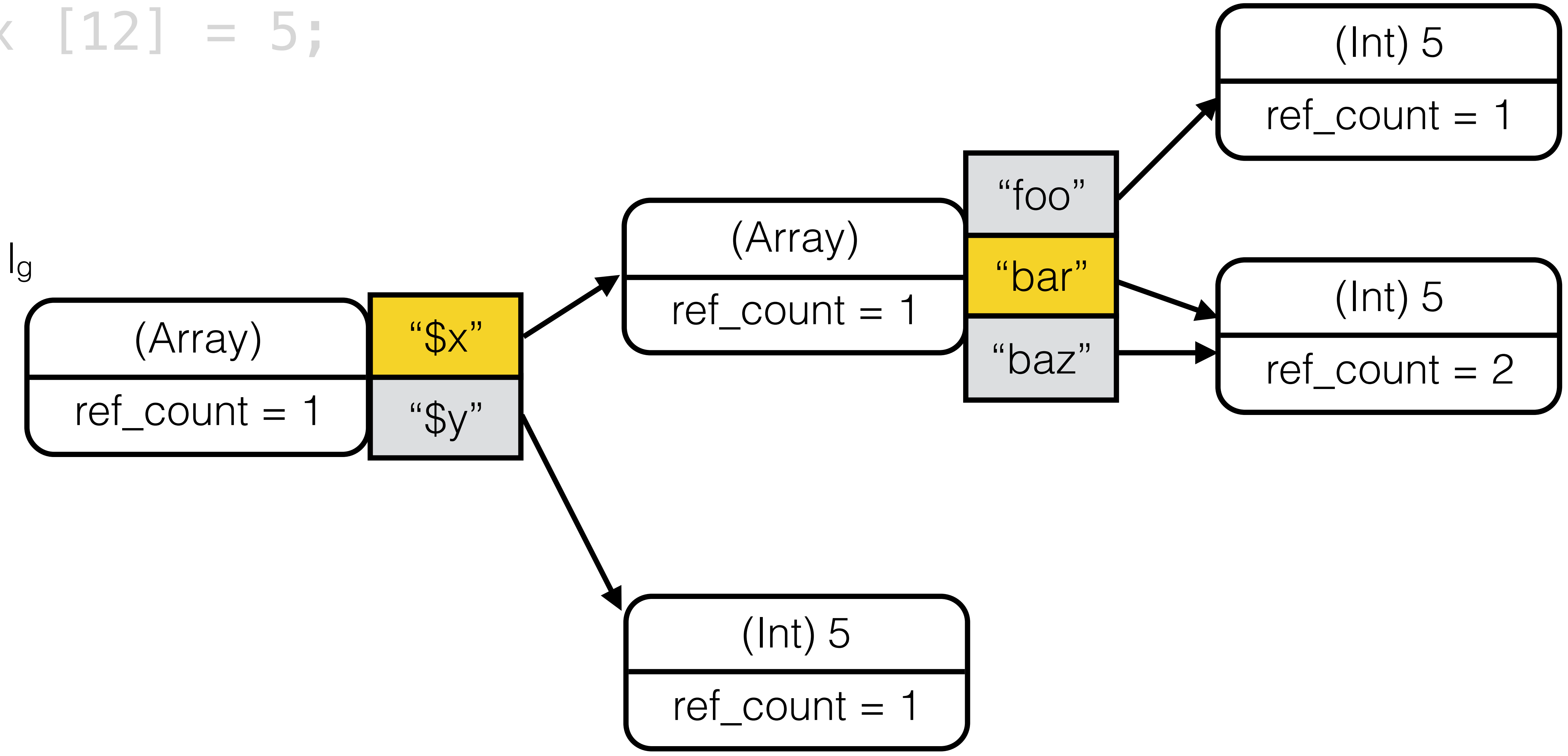
Memory - example

```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



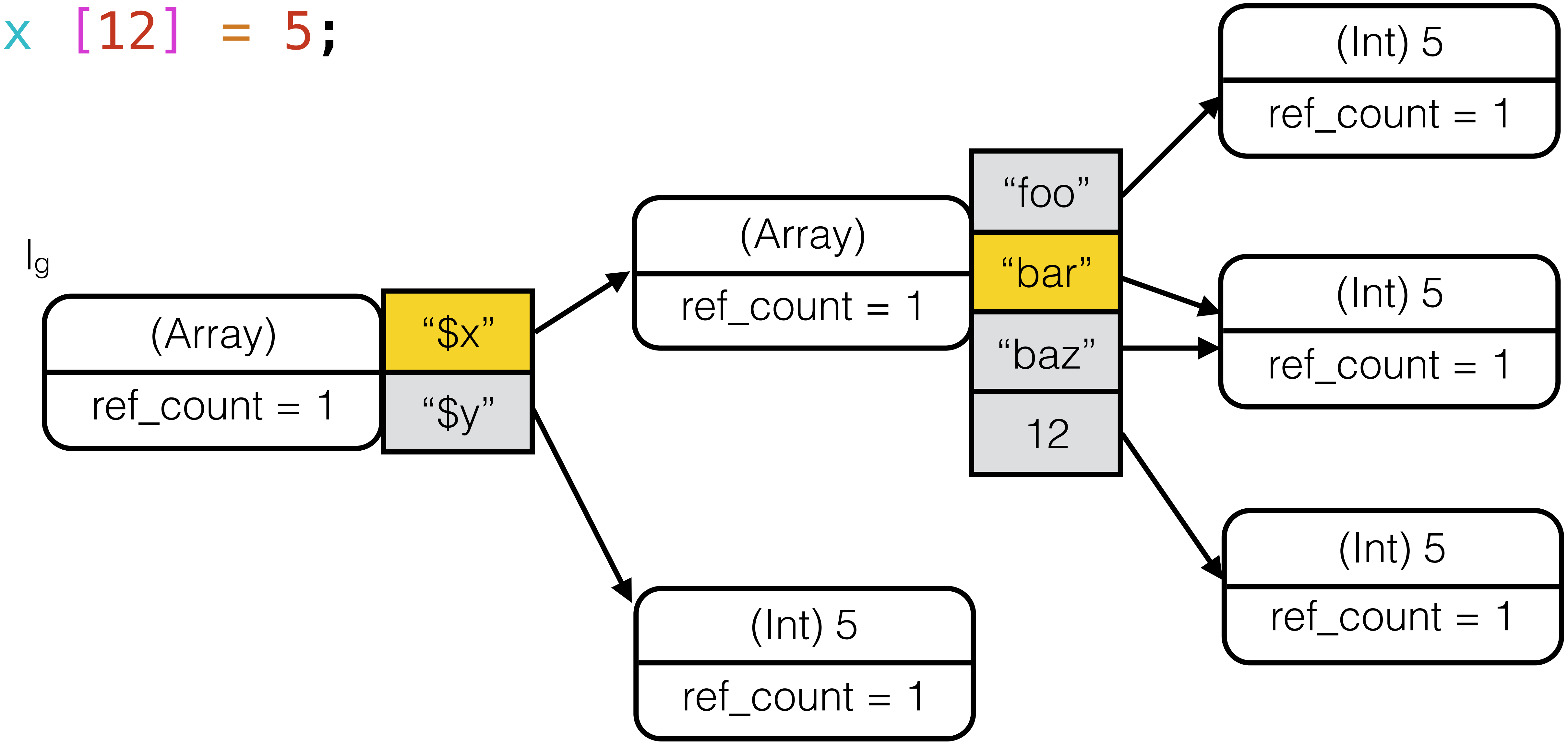
Memory - example

```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



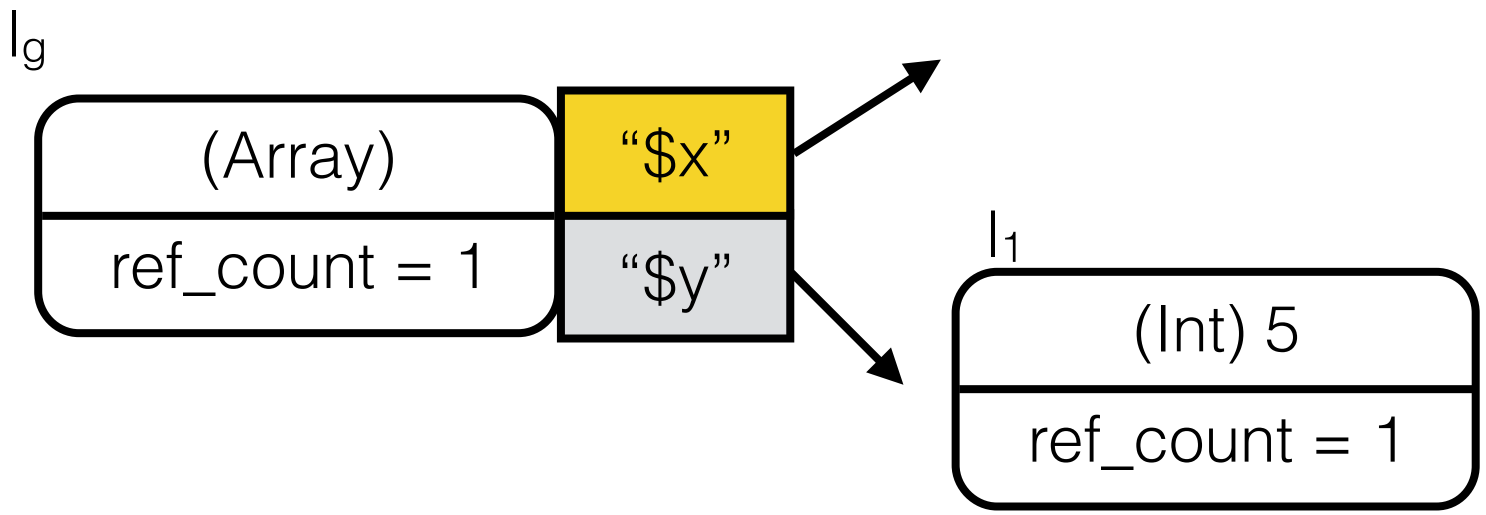
Memory - example

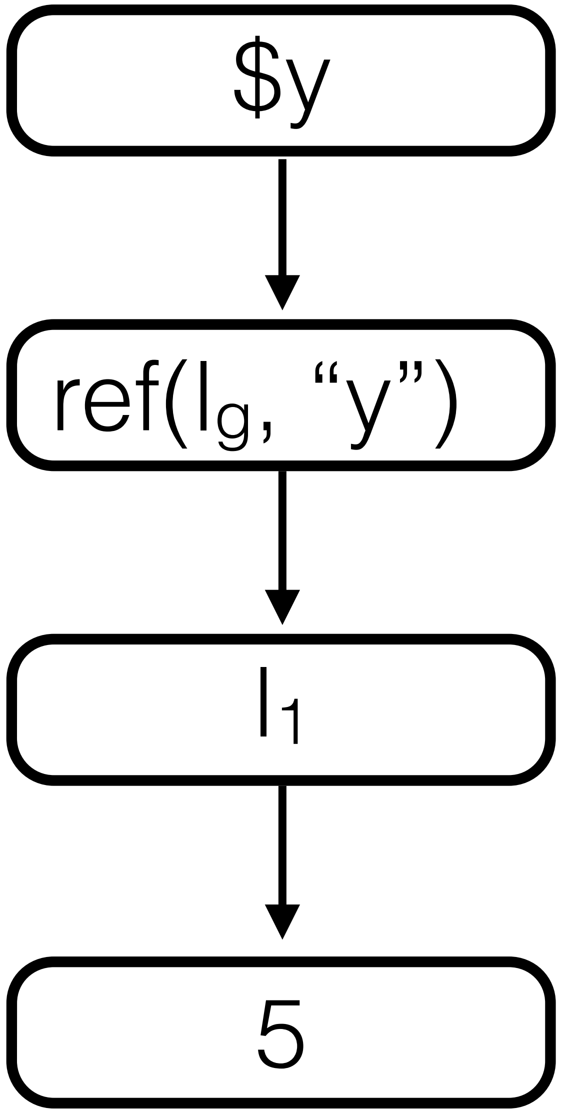
```
$x = array("foo" => 5, "bar" => 5); $y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x [12] = 5;
```



Internal values

- Locations: **l_1, l_2, l_3**....
- References: **ref(l, "x")**





Semantic rules: numbers

- Covering most of **the core language** (except interfaces, abstract classes, some minor features)
- ~ 800 rules
- ~ 8000 LOC
- 29 *.k files

Layers

Low-level rules
(copy values, inc. ref. counter, update scope etc.)

Layers

Language **features**
(e.g.: assignment, function call)

Low-level rules
(copy values, inc. ref. counter, update scope etc.)

Layers

Derived Construct
(e.g. x++ —> x = x + 1)

Language **features**
(e.g.: assignment, function call)

Low-level rules
(copy values, inc. ref. counter, update scope etc.)

Example: assignment

(A) $\texttt{CONTEXT}\quad \texttt{'Assign}(\square,\_)$

(B) $\texttt{CONTEXT}\quad \texttt{'Assign}(\_:\texttt{KResult},\square)$

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}},\_\right)\quad \texttt{[intermediate]}$

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)}\curvearrowright\texttt{V}}\quad \texttt{[step]}$

(E) $\texttt{'Assign}\left(\_:\texttt{KResult},\dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

$\texttt{when}\ \neg\ \texttt{isLiteral(V)}\quad \texttt{[intermediate]}$

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)}\curvearrowright\texttt{'Assign(L, L1)}}$

$\texttt{when currentOverflow(L1)}\quad \texttt{[intermediate]}$

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

$\texttt{when}\ \neg\ \texttt{currentOverflow(L1)}\quad \texttt{[intermediate]}$

Example: assignment

(A) CONTEXT 'Assign(□,_)

(B) CONTEXT 'Assign(_:KResult,□)

(C) $$\texttt{'Assign}\left(\frac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right) \quad \texttt{[intermediate]}$$

(D) $$\frac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}} \quad \texttt{[step]}$$

(E) $$\texttt{'Assign}\left(\frac{\_}{} : \texttt{KResult}, \frac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$$

when $\neg$ isLiteral(V) [intermediate]

(F) $$\frac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$$

when currentOverflow(L1) [intermediate]

(G) $$\frac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$$

when $\neg$ currentOverflow(L1) [intermediate]

Example: assignment

(A) `CONTEXT 'Assign(□,_)`

(B) `CONTEXT 'Assign(_:KResult,□)`

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}\texttt{, \_}\right)$ `[intermediate]`

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ `[step]`

(E) $\texttt{'Assign}\left(\_ \texttt{ : KResult, } \dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

when $\neg$ `isLiteral(V)` `[intermediate]`

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$

when `currentOverflow(L1)` `[intermediate]`

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

when $\neg$ `currentOverflow(L1)` `[intermediate]`

Example: assignment

(A) `CONTEXT 'Assign(□,_)`

(B) `CONTEXT 'Assign(_:KResult,□)`

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right)$ `[intermediate]`

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ `[step]`

(E) $\texttt{'Assign}\left(\_ : \texttt{KResult}, \dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

`when ¬ isLiteral(V)` `[intermediate]`

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$

`when currentOverflow(L1)` `[intermediate]`

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

`when ¬ currentOverflow(L1)` `[intermediate]`

Example: assignment

(A) CONTEXT   'Assign(□,_)

(B) CONTEXT   'Assign(_:KResult,□)

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right)$ [intermediate]

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ [step]

(E) $\texttt{'Assign}\left(\_ : \texttt{KResult}, \dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

when $\neg$ isLiteral(V)   [intermediate]

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$

when currentOverflow(L1)   [intermediate]

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

when $\neg$ currentOverflow(L1)   [intermediate]

Example: assignment

(A) CONTEXT 'Assign(□,_)

(B) CONTEXT 'Assign(_:KResult,□)

(C) $$\texttt{'Assign}\left(\frac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right) \quad \texttt{[intermediate]}$$

(D) $$\frac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}} \quad \texttt{[step]}$$

(E) $$\texttt{'Assign}\left(\_ : \texttt{KResult}, \frac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$$

$$\texttt{when} \ \neg \ \texttt{isLiteral(V)} \quad \texttt{[intermediate]}$$

(F) $$\frac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$$

$$\texttt{when currentOverflow(L1)} \quad \texttt{[intermediate]}$$

(G) $$\frac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$$

$$\texttt{when} \ \neg \ \texttt{currentOverflow(L1)} \quad \texttt{[intermediate]}$$

Example: assignment

(A) CONTEXT 'Assign(□,_)

(B) CONTEXT 'Assign(_:KResult,□)

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right)$ [intermediate]

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ [step]

(E) $\texttt{'Assign}\left(\_ : \texttt{KResult}, \dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

when $\neg$ isLiteral(V) [intermediate]

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$

when currentOverflow(L1) [intermediate]

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

when $\neg$ currentOverflow(L1) [intermediate]

Example: assignment

(A) CONTEXT 'Assign(□,_)

(B) CONTEXT 'Assign(_:KResult,□)

(C) $\texttt{'Assign}\left(\dfrac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right)$ [intermediate]

(D) $\dfrac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ [step]

(E) $\texttt{'Assign}\left(\_ : \texttt{KResult}, \dfrac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$

when $\neg$ isLiteral(V) [intermediate]

(F) $\dfrac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L1)} \curvearrowright \texttt{'Assign(L, L1)}}$

when currentOverflow(L1) [intermediate]

(G) $\dfrac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$

when $\neg$ currentOverflow(L1) [intermediate]

Example: function call

$$\left\langle \frac{\texttt{runFunction(FN:String, f(FP:K, FB:K, RT:RetType, LS:Loc), Args:K)} \curvearrowright \texttt{K}}{\left(\begin{array}{r} \texttt{processFunArgs(FP, Args)} \curvearrowright \\ \texttt{pushStackFrame(FN, K, L, CurrentClass, CurrentObj, RT, D)} \curvearrowright \\ \texttt{ArrayCreateEmpty(L1)} \curvearrowright \texttt{setCrntScope(L1)} \curvearrowright \texttt{incRefCount(L1)} \curvearrowright \\ \texttt{copyFunArgs} \curvearrowright \texttt{FB} \curvearrowright \texttt{'Return(NULL)} \end{array}\right)} \right\rangle_{\texttt{k}}$$

$$\langle \texttt{L:Loc} \rangle_{\texttt{currentScope}} \qquad \langle \texttt{CurrentClass:Id} \rangle_{\texttt{class}} \qquad \langle \texttt{CurrentObj:Loc} \rangle_{\texttt{object}}$$

$$\left\langle \frac{\texttt{D:K}}{\cdot} \right\rangle_{\texttt{functionArgumentDeclaration}}$$

`when fresh(L1)     [internal]`

Example - revisited

```
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"
```

```
$a = "one";
$c = $a . ($a = "two");
echo $c; // "twotwo"
```

Evaluation order: LR or RL?

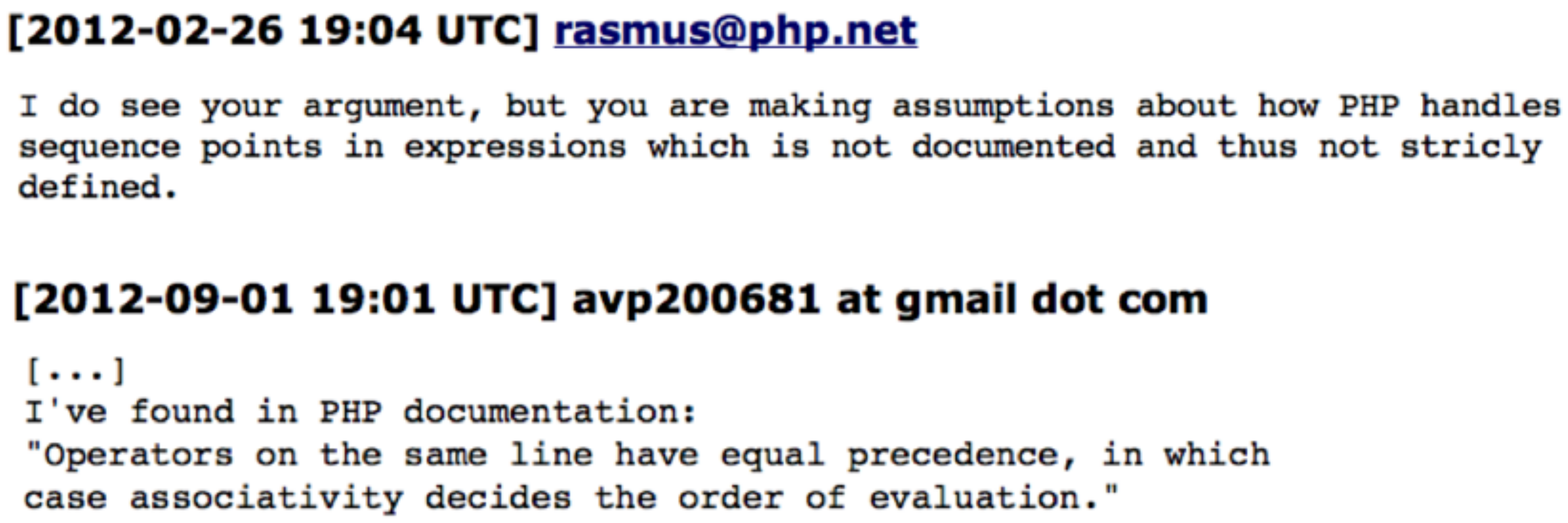
Example - revisited

```
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"
```

```
$a = "one";
$c = $a . ($a = "two");
echo $c; // "twotwo"
```

PHP bug 61188

Evaluation order: LR or RL?



[2012-02-26 19:04 UTC] rasmus@php.net

```
I do see your argument, but you are making assumptions about how PHP handles
sequence points in expressions which is not documented and thus not stricly
defined.
```

[2012-09-01 19:01 UTC] avp200681 at gmail dot com

```
[...]
I've found in PHP documentation:
"Operators on the same line have equal precedence, in which
case associativity decides the order of evaluation."
```

Example - explained

```
$a = array("one");
$c = $a[0] . ($a[0] = "two");
echo $c; // “onetwo"
```

```
$a = "one";
$c = $a . ($a = "two");
echo $c; // "twotwo"
```

- evaluation order **is left-to-right**
- array access evaluates to values
- variables evaluate to references
- references are resolved lazily

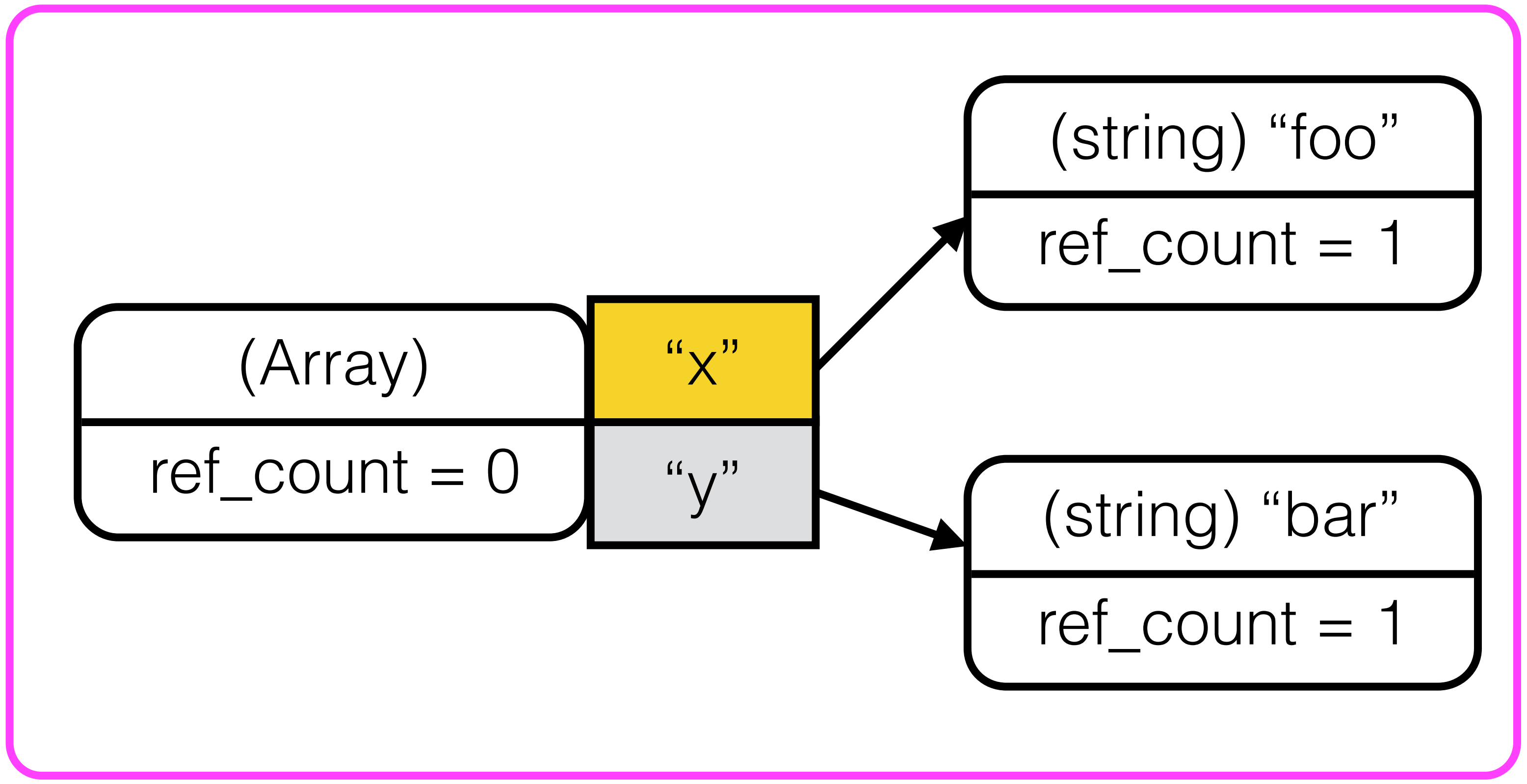
What about objects?

```
Class A {public $x = “one”; private $y = “two” }
$x = new A();
```

What about objects?

```
Class A {public $x = “one”; private $y = “two” }
$x = new A();
```

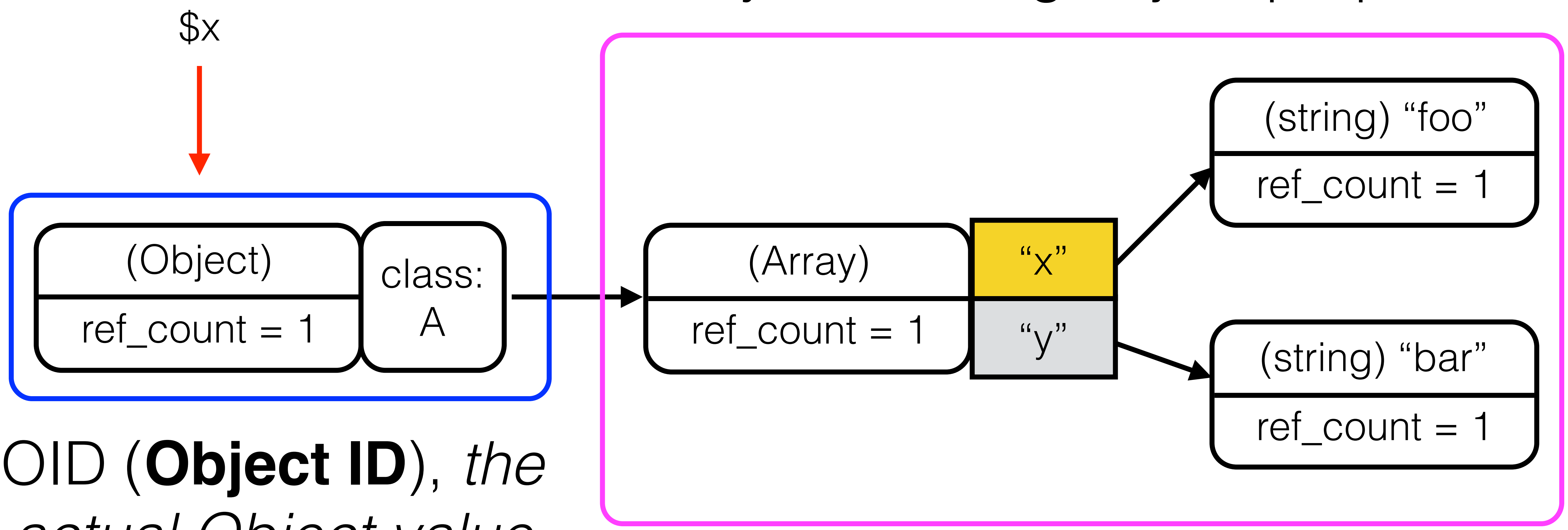
Array containing object properties



What about objects?

```
Class A {public $x = "one"; private $y = "two" }
$x = new A();
```

Array containing object properties



OID (**Object ID**), *the actual Object value*

Objects as arrays

- **Object properties have visibilities** (public, protected, private)
- We attach **visibility attributes to all arrays**
  - “Normal” arrays are always accessible, so all their elements are public
  - Arrays associated to objects may have protected or private visibility
  - Objects as “guarded arrays”
- Generalisation in semantic rules

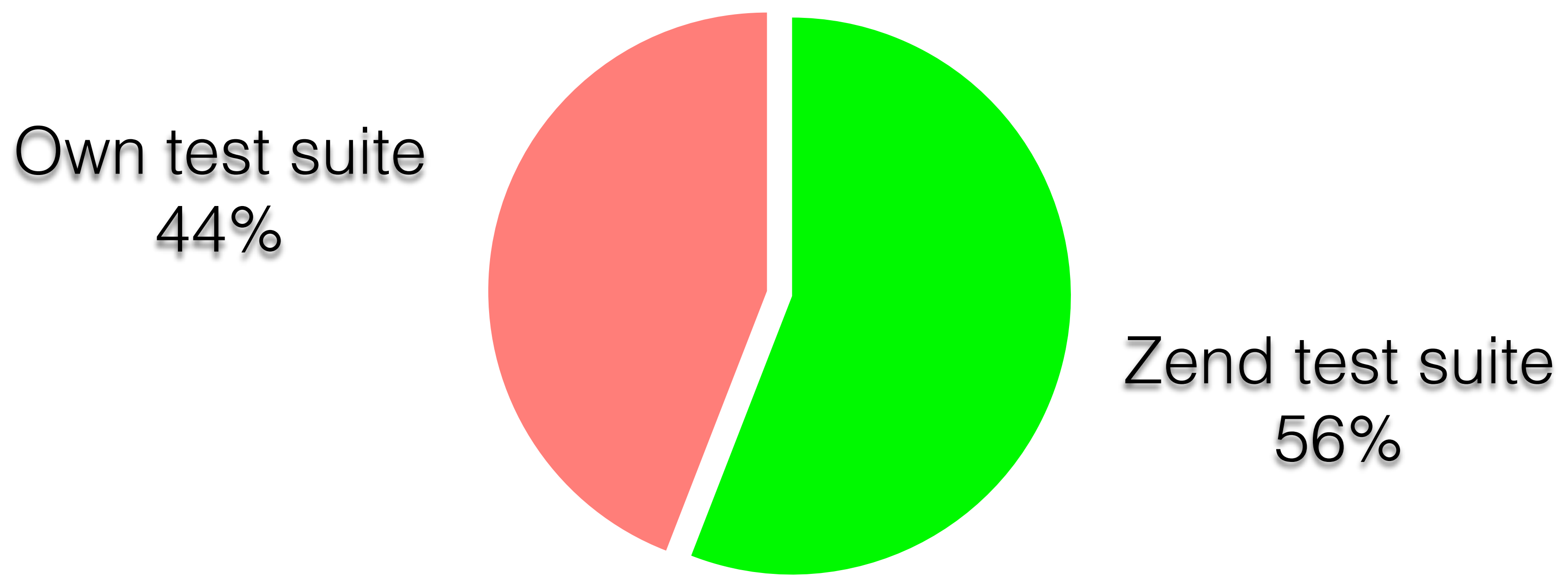
Validation

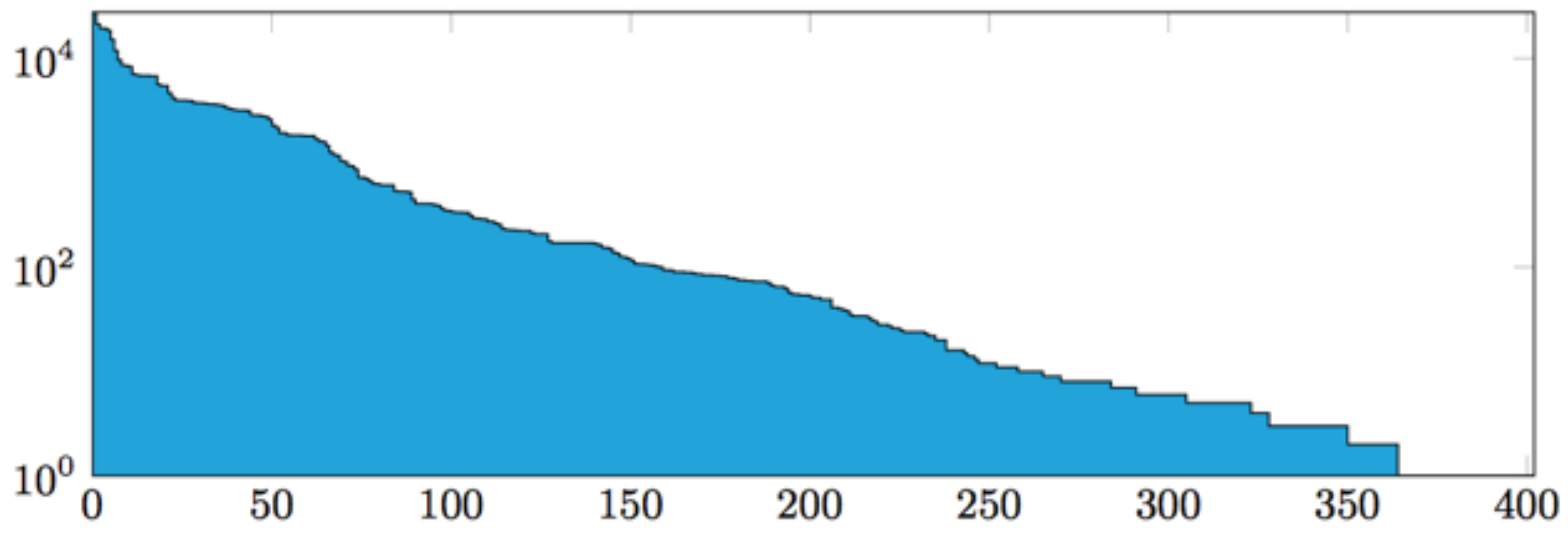
- Testing against the **Zend test suite.**
- focusing on **core language** section of test suite
- passing all tests supported by our semantics

```
--TEST--
Child public element should not
   override parent private element
   in parent methods
--FILE--
<?php
class par {
    private $id = "foo";

    function displayMe()
    {
        print $this->id;
    }
};

class chld extends par {
    public $id = "bar";
    function displayHim()
    {
        parent::displayMe();
    }
};


$obj = new chld();
$obj->displayHim();
?>
--EXPECT--
foo
```

Coverage

"How many times each rule is used by the test suite?"





Application: temporal verification of PHP programs

- Using K’s builtin support for LTL model checking and symbolic execution
- Extension of LTL with predicates over KPHP configurations
- Real-world case studies: input validation (PHPMyAdmin) and hashing function (PHP library)

Case study: hashing

```
function pbkdf2($algorithm, $password, $salt, $count, $key_length, $raw_output = false)
{
    $algorithm = strtolower($algorithm);
    if(!in_array($algorithm, hash_algos(), true))
        die('PBKDF2 ERROR: Invalid hash algorithm.');
    if($count <= 0 || $key_length <= 0)
        die('PBKDF2 ERROR: Invalid parameters.');

    $hash_length = strlen(hash($algorithm, "", true));
    $block_count = ceil($key_length / (float) $hash_length);

    echo "key len: $key_length\n";
    echo "hash len: $hash_length\n";
    echo "block count: $block_count\n";

    $output = ""; //"";
    for($i = 1; $i <= $block_count; $i++) {
        // $i encoded as 4 bytes, big endian.
        $last = $salt . pack("N", $i);
        // first iteration
        $last = $xorsum = hash_hmac($algorithm, $last, $password, true);
        // perform the other $count - 1 iterations
        for ($j = 1; $j < $count; $j++) {
            $xorsum ^= ($last = hash_hmac($algorithm, $last, $password, true));
        }
        $output .= $xorsum;
    }

    if($raw_output)
        return substr($output, 0, $key_length);
    else
        return bin2hex(substr($output, 0, $key_length));
}
```

Case study: hashing

Lemma: For all \$password, \$salt and for given \$algo, \$count and \$key_len:

(i) The result is a string: $\Diamond\texttt{has\_type(gv(var('result')),string)}$

(ii) The length of the output is as requested

$$\Diamond\texttt{eqTo(gv(var('key\_len')),len(gv(var('result'))))}$$

(iii) The length of the string stored in \$output grows and eventually becomes greater then the expected output length

$$\begin{aligned}
\Box\big(\ & (\texttt{inFun('pbkdf2')} \land \neg\texttt{inFun('top')} \land \Diamond\texttt{inFun('top')}) \implies \\
& (\Diamond(\texttt{geq(len(fv('pbkdf2',var('output'))), fv('pbkdf2', var('key\_len'))})) \\
& \mathcal{U}\ \texttt{inFun('top')})\ \big)
\end{aligned}$$

Improving language support

Fix bugs

Future Work

Deductive verification (**Reachability Logic**)

Static Analysis **(Abstract Interpretation)**

Conclusions

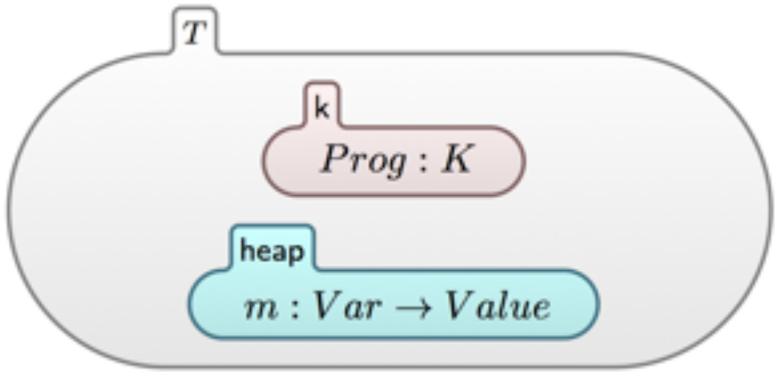
- The first formal semantics for PHP
- Semantics is directly executable
- Validated by passing all supported tests from the Zend test suite
- Full coverage of rules by adding our own tests
- Proof of concept infrastructure for verification of PHP programs
- **A first step toward defining semantics based static analysis tools for PHP**

Thank you!

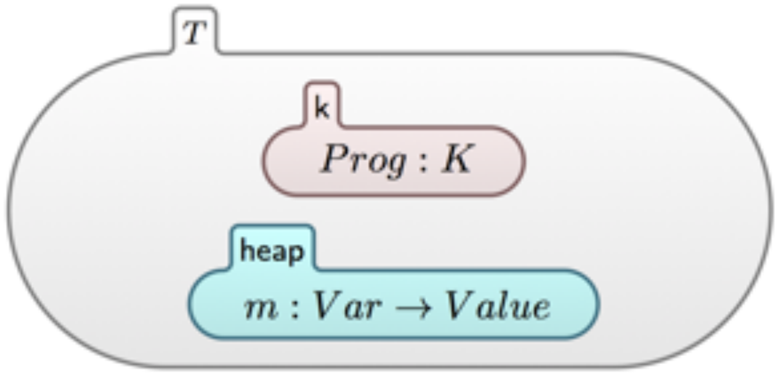
www.phpsemantics.org

paper, sources, web interface

Appendix/misc/old

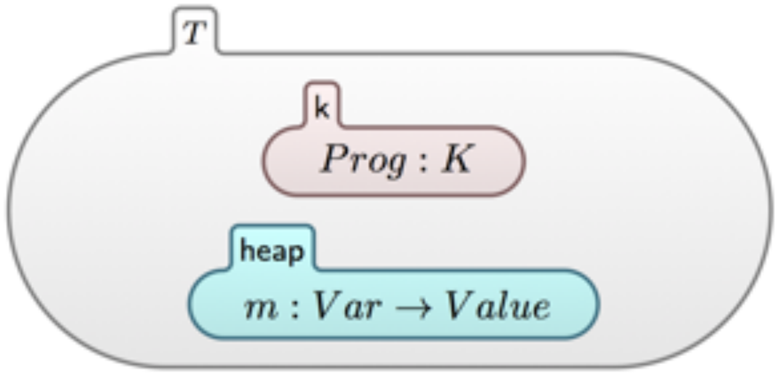


Komputations



Komputations

(i) the K cell holds **a list of computations separated by** $\curvearrowright$

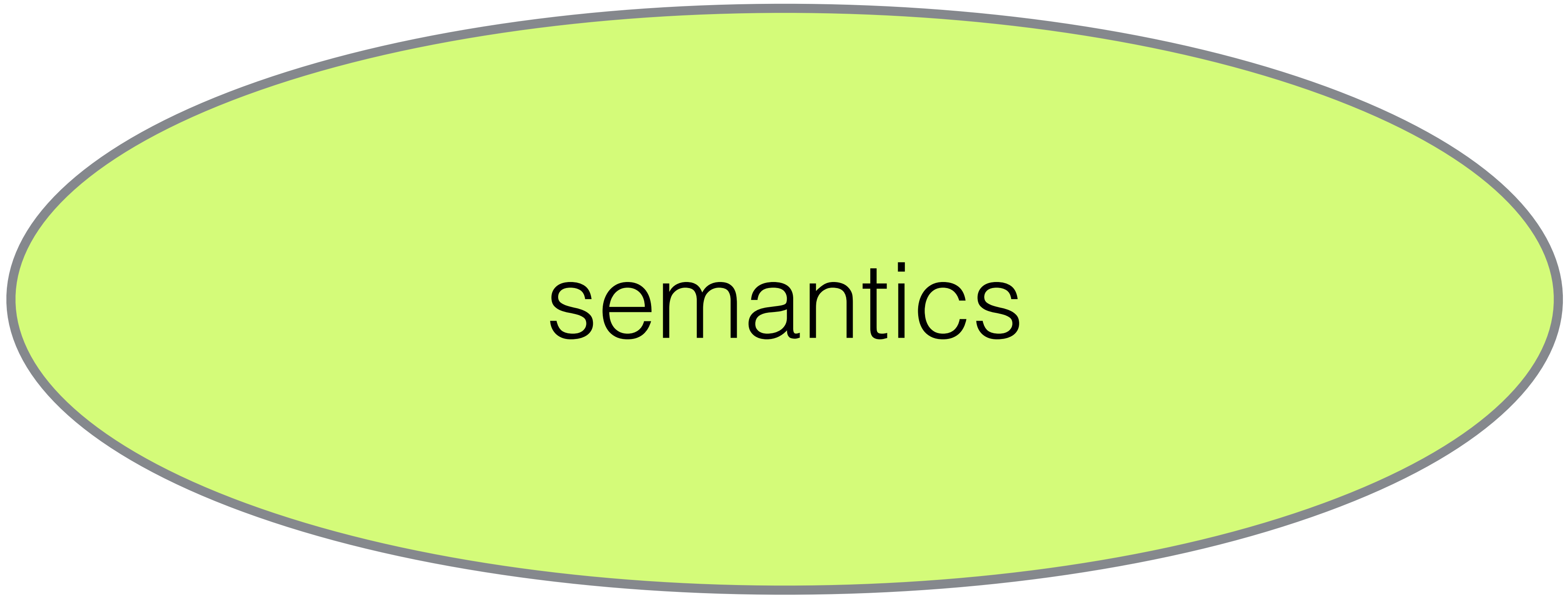


Komputations

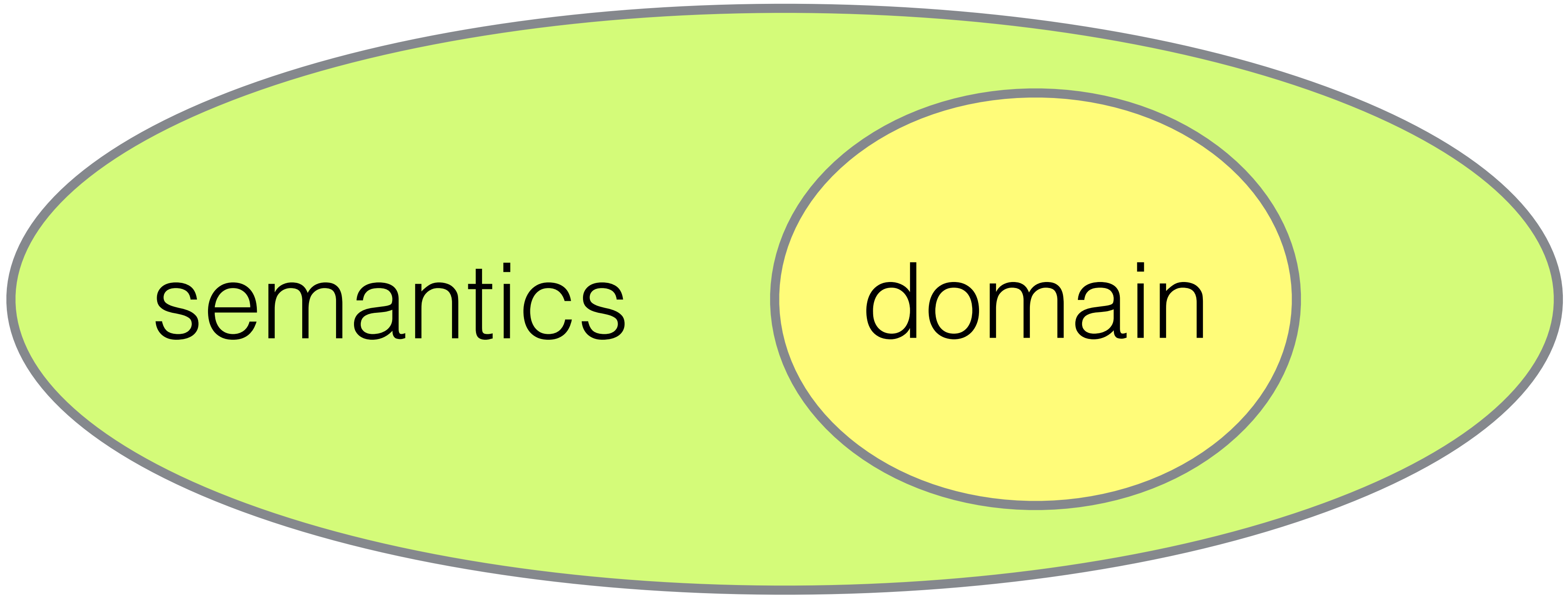
(i) the K cell holds **a list of computations separated by** $\curvearrowright$

(ii) the input program goes into the k cell

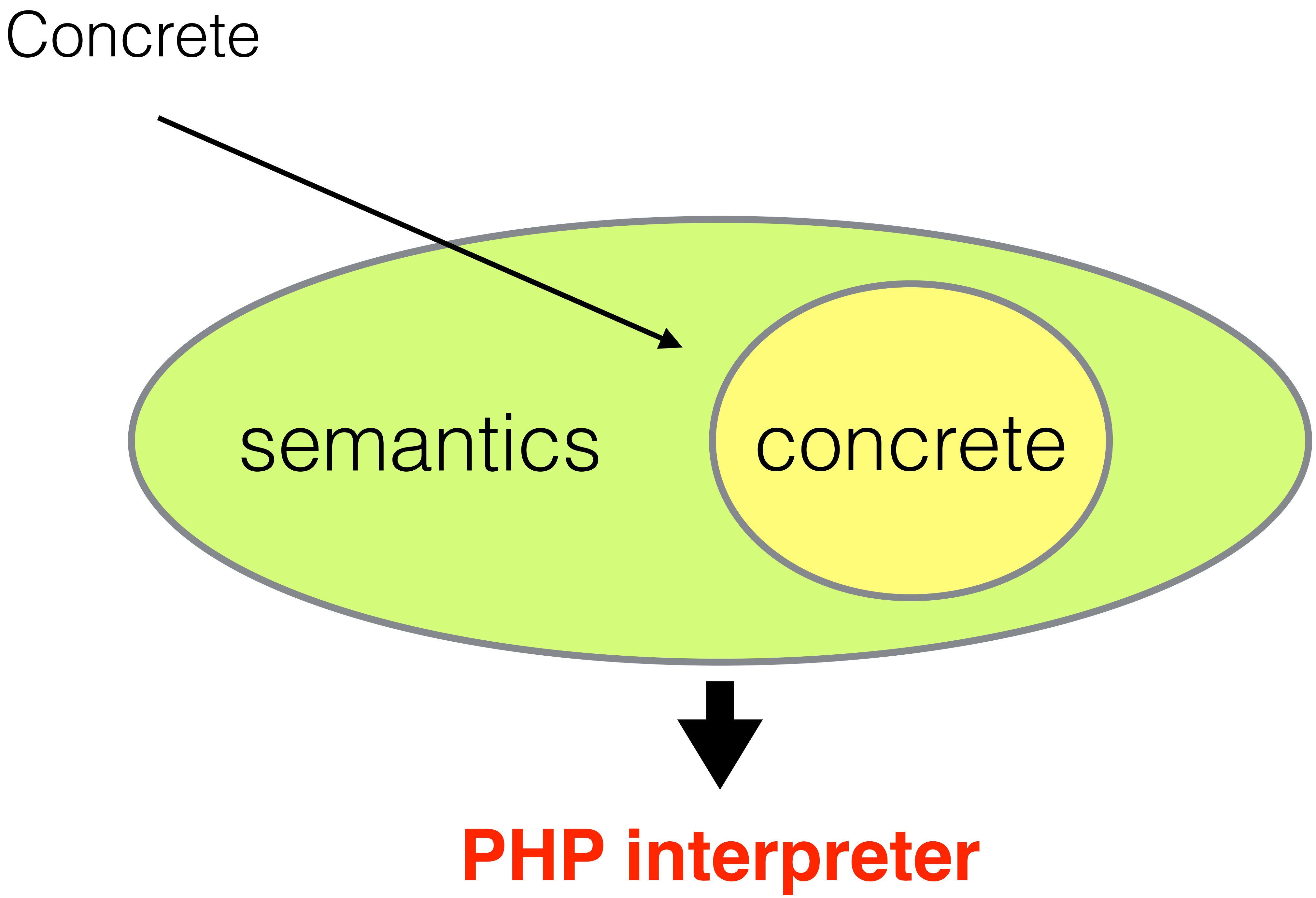
Abstract Interpretation



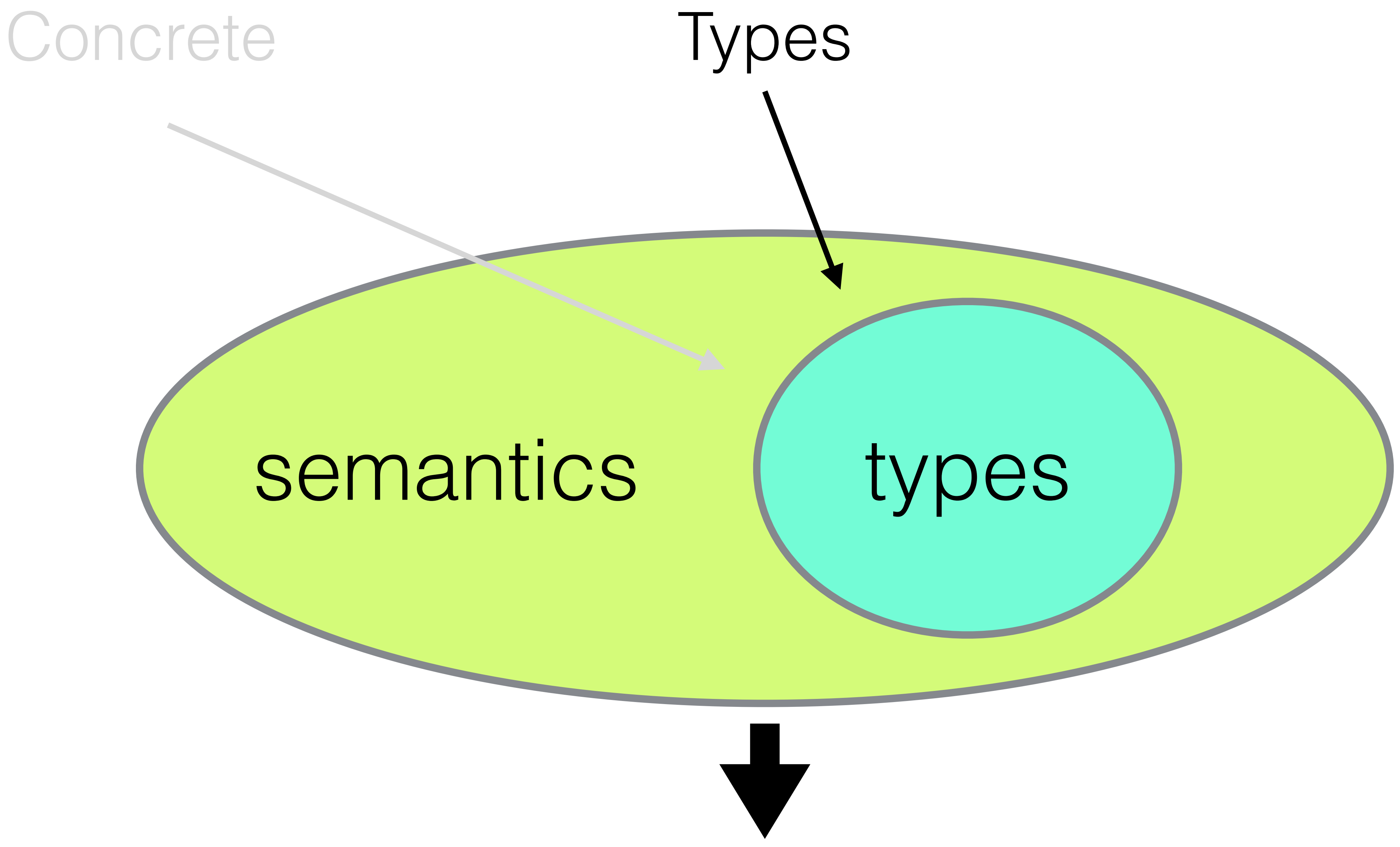
Abstract Interpretation



Abstract Interpretation

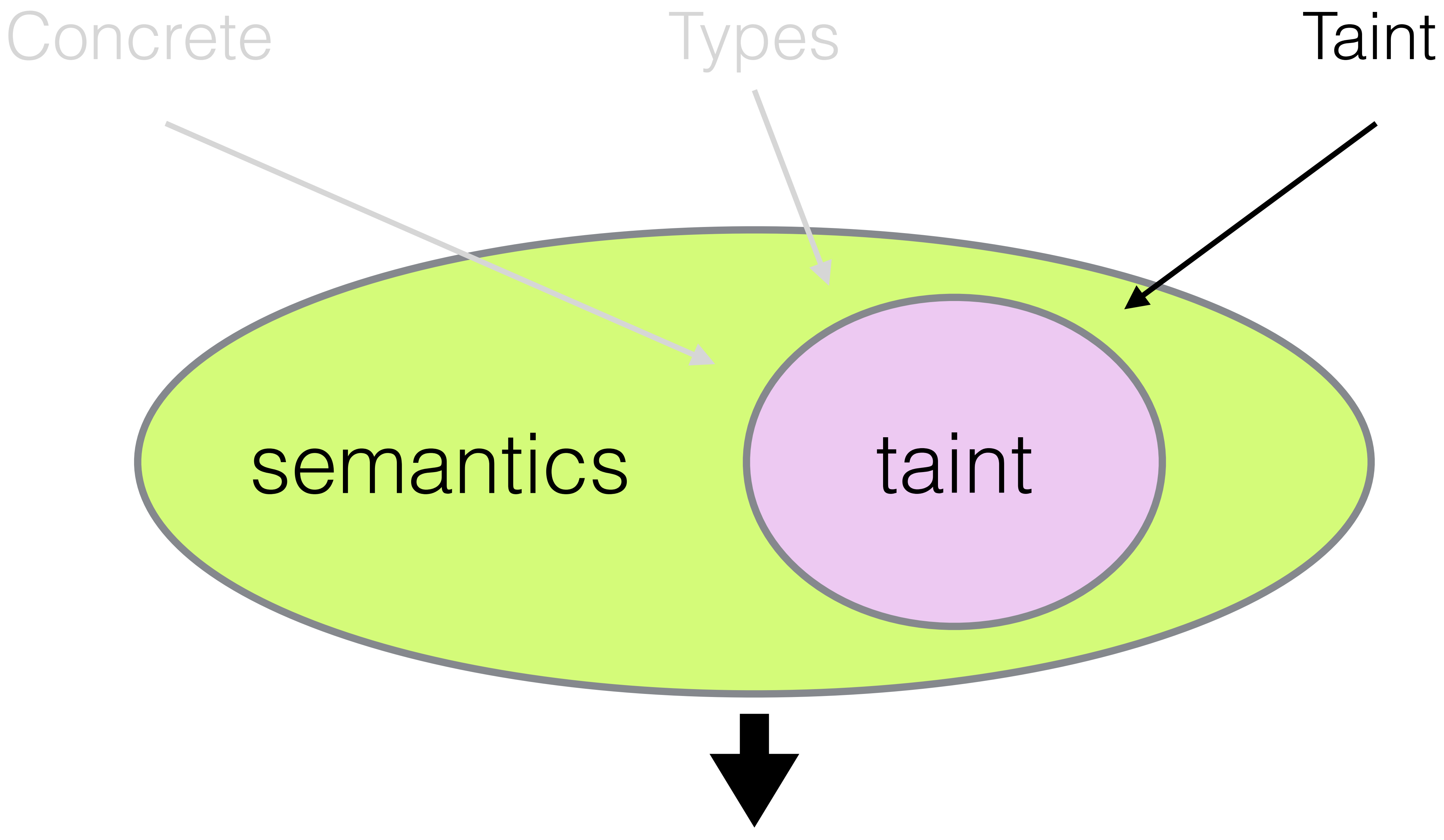


Abstract Interpretation



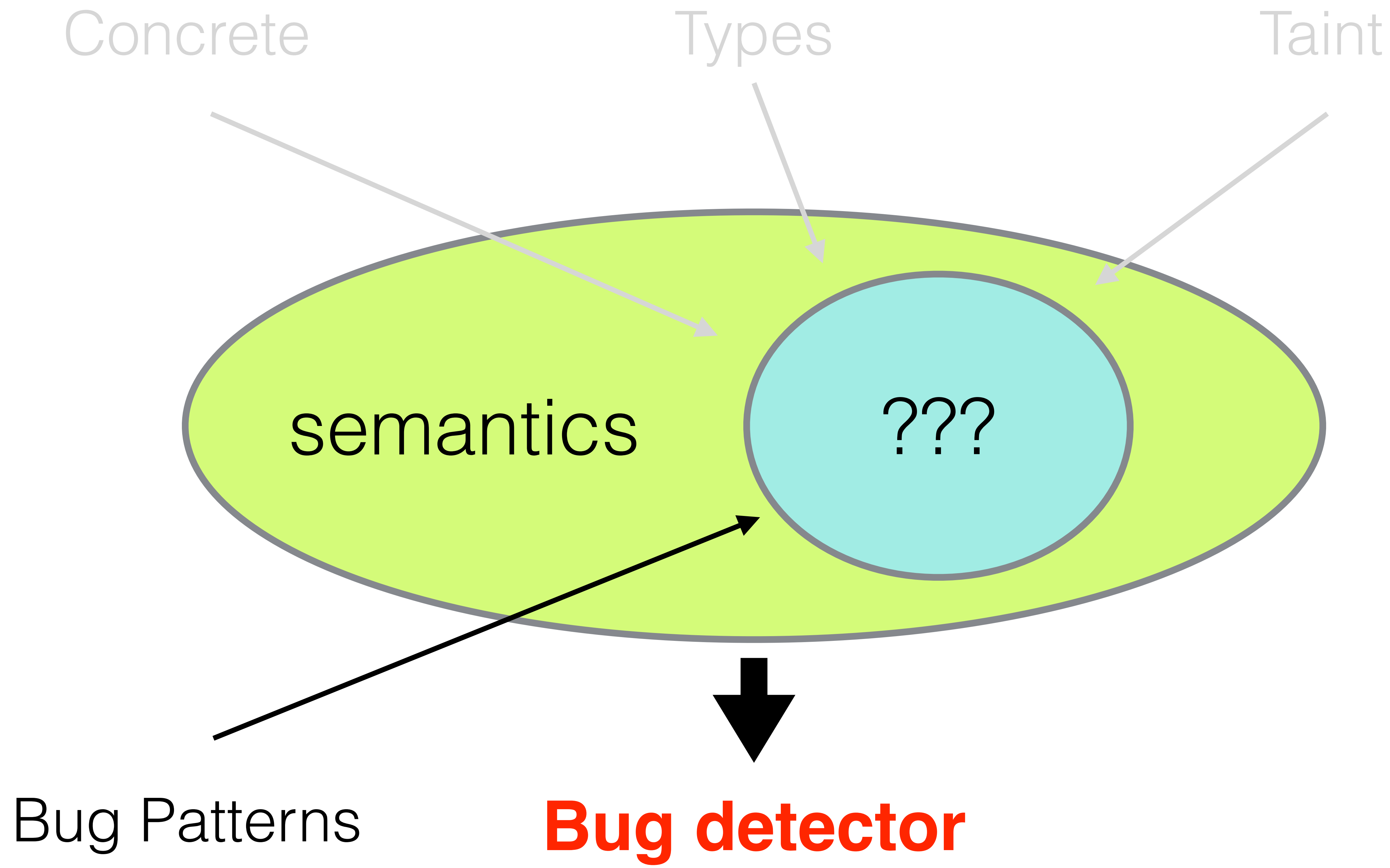
Type Analysis

Abstract Interpretation



Taint Analysis

Abstract Interpretation



zend/lang/036.phpt

```
--TEST--
Child public element should not
   override parent private element
   in parent methods
--FILE--
<?php
class par {
    private $id = "foo";

    function displayMe()
    {
        print $this->id;
    }
};

class chld extends par {
    public $id = "bar";
    function displayHim()
    {
        parent::displayMe();
    }
};


$obj = new chld();
$obj->displayHim();
?>
--EXPECT--
foo
```

- chld extends par
- both classes has member $id - private for parent, public for child
- instance of child calls parent's displayMe method, which outputs "foo"
- if par's $id was public, then output would be "bar"
- Implication: members indexed by (key, visibility) pair!

LTL example

```
function foo() {
    global $y;
    $x = &$y;
}
```

LTL example

```
function foo() {
    global $y;
    $x = &$y;
}
$y = #symbolic_input();
```

LTL example

```
function foo() {
    global $y;
    $x = &$y;
}
$y = #symbolic_input();
foo();
```

LTL example

```
function foo() {
    global $y;
    $x = &$y;
}
$y = #symbolic_input();
foo();
```

```
◊alias(fv('foo',var('x')),gv(var('y')))
```

zend/lang/036.phpt

```
--TEST--
Child public element should not override parent private element in parent methods
--FILE--
<?php
class par {
    private $id = "foo";

    function displayMe()
    {
        print $this->id;
    }
};

class chld extends par {
    public $id = "bar";
    function displayHim()
    {
        parent::displayMe();
    }
};


$obj = new chld();
$obj->displayHim();
?>
--EXPECT--
foo
```

- chld extends par
- both classes has member $id - private for parent, public for child
- instance of child calls parent’s displayMe method, which outputs “foo”
- if par’s $id was public, then output would be “bar”
- Implication: members indexed by (key, visibility) pair!