An executable formal semantics for PHP

Daniele Filaretti & Sergio Maffeis

www.phpsemantics.org

Summary

- Background (PHP)
- Semantics of PHP in K
- Validation via testing
- Limitations
- Applications
- Future work

Background

PHP

- A **scripting** language
  - type juggling, aliasing, array/object iteration etc.
- A **server side** language
  - Web security - XSS, SQLi, etc.
- No formal (or even semi-formal) spec
  - tight design-test loop, K good for this

Hello World

http://example.com/hello.php?name=xyz

```
<?php
  echo “<HTML><BODY>”;
  echo “Hello ” . $_GET[“name”] . “!”;
  echo </BODY></HTML>”;
<?
```

Example: evaluation order

Undefined in C. Left-to-right on most languages. In PHP?

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

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

Example: evaluation order

Undefined in C. Left-to-right on most languages. In PHP?

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

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

Similar example...

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

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

Goals

- Developing tools based on semantics
  - current tools have no formal guarantees, partial support of the language etc.

Goals

- Developing tools based on semantics
  - current tools have no formal guarantees, partial support of the language etc.
- Our K semantics as specification?
  - similar to ECMA for JS

PHP in K

Semantics, statistics

- ~30 cells
- ~1000 rules
- 29 .k files
- ~9000 LOC
- ~2.5 years work

- Wrong memory model 3 times
- models almost all core language...
- ... and a few library functions (where needed)

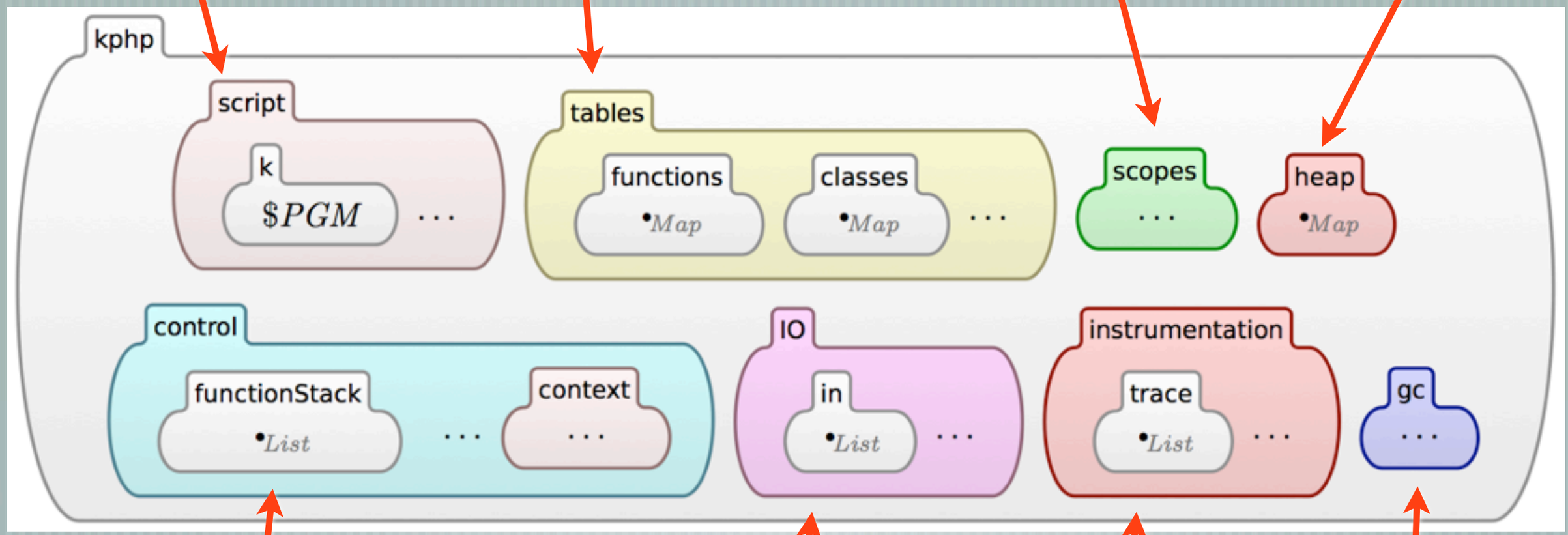
Configuration

Input program (commands + declarations)

Classes and functions

Locations of global and current scopes

Heap



Function stack, exception stack, current context etc.

Usual IO

Housekeeping stuff

Simple garbage collection

Values

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

Using K builtins

Our own modeling (challenging!)

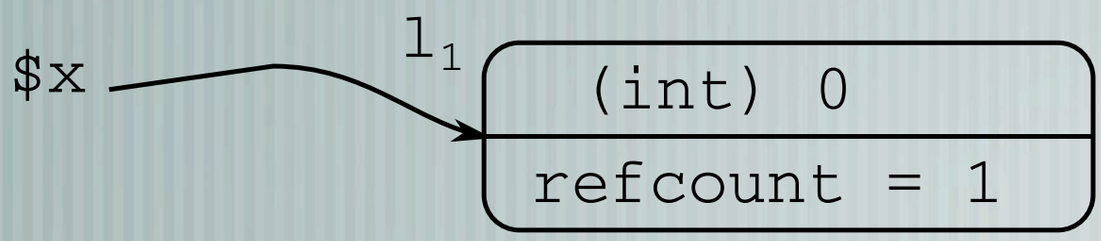
Arrays

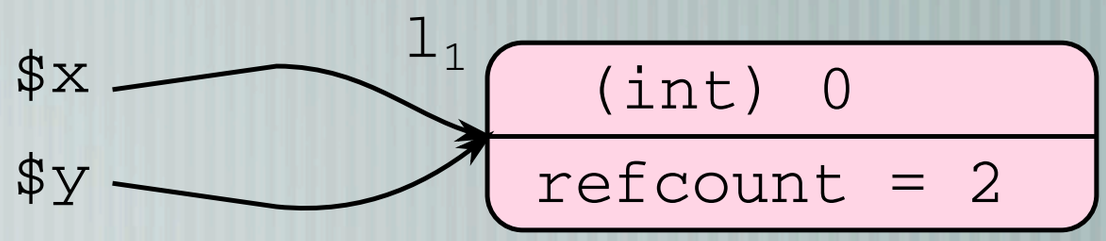
- Maps {int, string} **keys** to **locations**
- Have a 'current' element
  - **current($x)** : returns $x's current element
  - **next($x)** : advances $x's current element

Values and ZValues

- In PHP, values are internally "wrapped" into "**ZValues**"
- ZVal = (value, type, reference counter, is_ref)

True iff aliasing





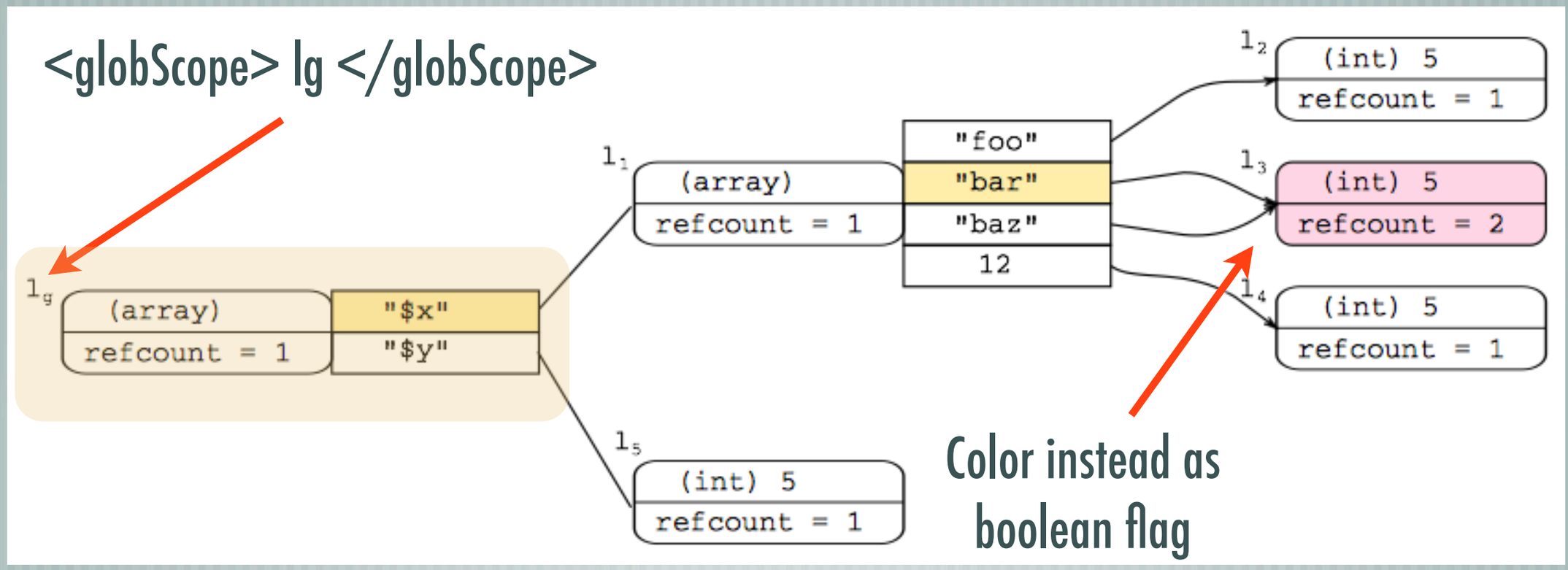
The <heap>

- **Heap** maps locations (l1, l2, ...) to ZVals
- Environments/scopes are arrays in the heap
  - locations of global and current scope are known at runtime (<scopes> cell)

<heap>, example

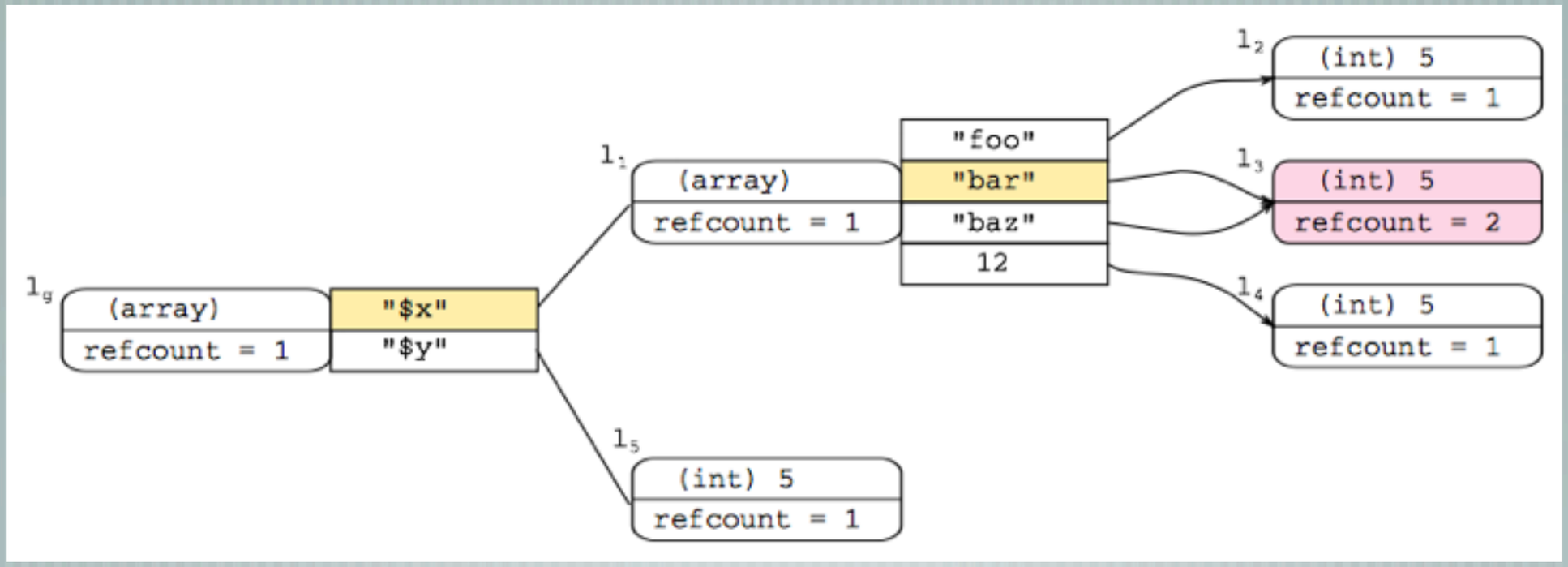
```php
$x = array("foo" => 5, "bar" => 5);
$y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x[12] = 5;
```

Tool extracting GraphViz from XML-K configuration



References and Locations #1

- Cannot directly evaluate variables into values or lvalues
- References: Internal values (hidden to the programmer)




```
unset($y) -> unset(ref(lg,'$y'))
```

References and Locations #2

- Variables evaluate to references
- When needed, further reduction apply

References and Locations #2

- Variables evaluate to references
- When needed, further reduction apply
- Different processing for LHS or RHS references
- More complex cases!

References and Locations #2

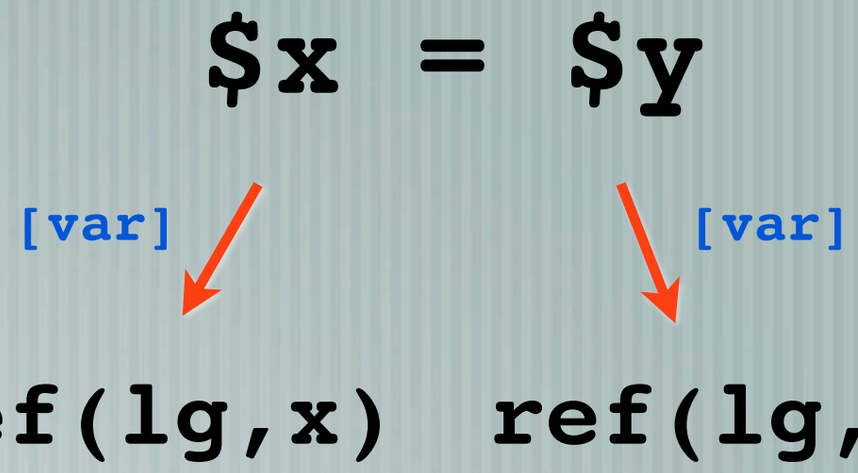
- Variables evaluate to references

```
$x = $y
```

- When needed, further reduction apply
- Different processing for LHS or RHS references
- More complex cases!

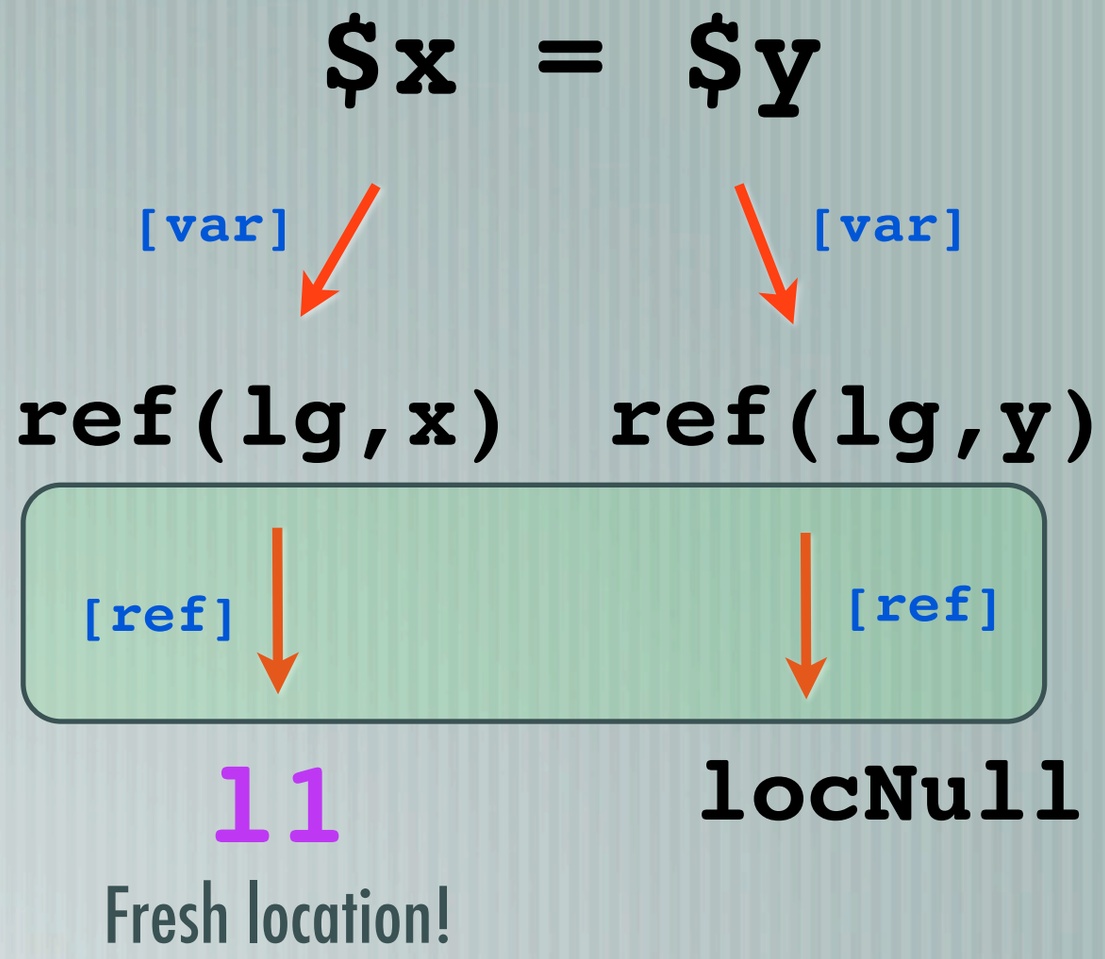
References and Locations #2

- Variables evaluate to references
- When needed, further reduction apply
- Different processing for LHS or RHS references
- More complex cases!



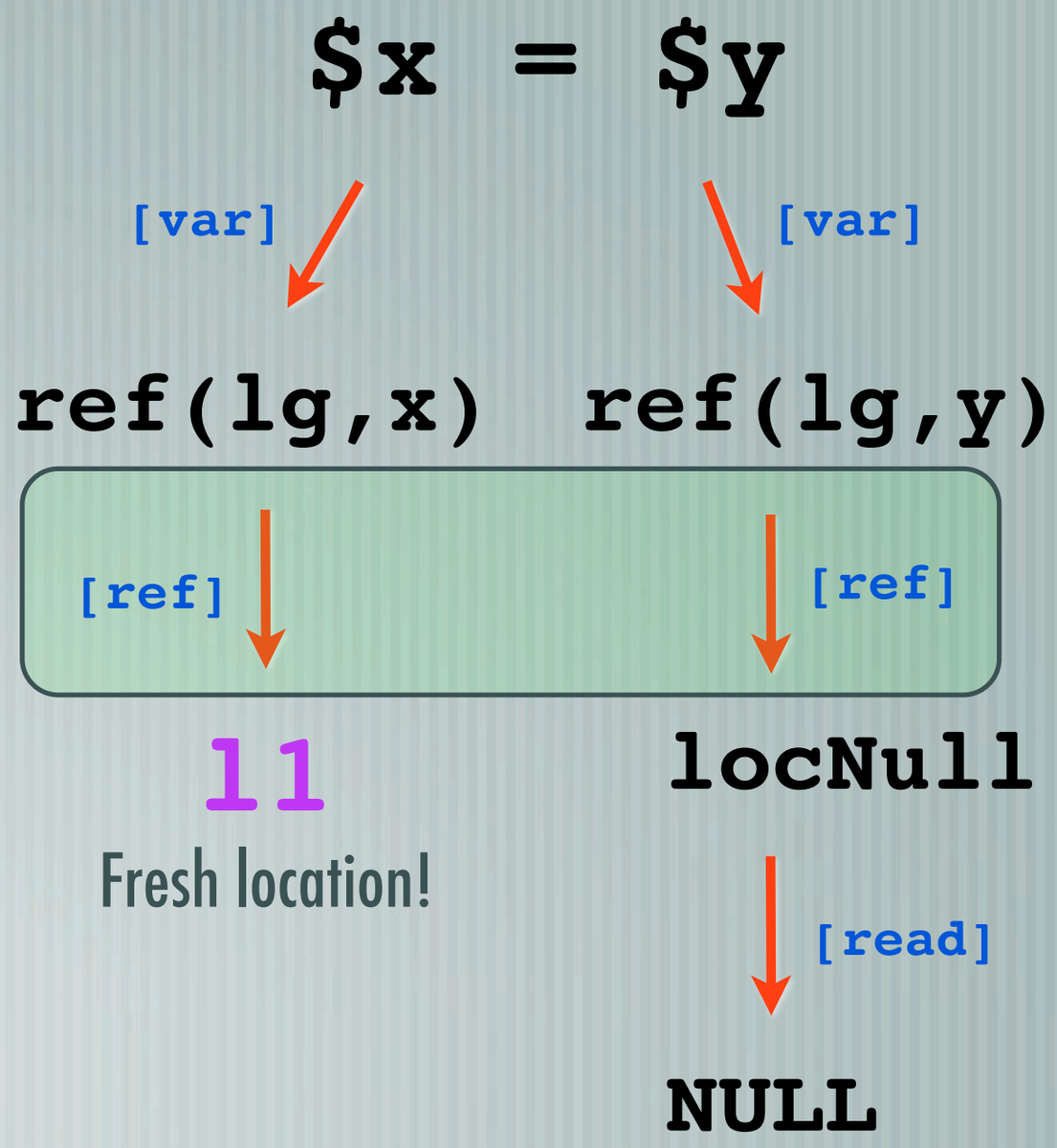
References and Locations #2

- Variables evaluate to references
- When needed, further reduction apply
- Different processing for LHS or RHS references
- More complex cases!



References and Locations #2

- Variables evaluate to references
- When needed, further reduction apply
- Different processing for LHS or RHS references
- More complex cases!



K Rules: building blocks

- Internal, low level rules, operating at ZVal level
  - increment/decrement ref counter, write value/type...
- building more complex operations
  - write => write_value + write_type + init_ref_count etc.
  - reference counting must be done correctly

Rule tags

[step]: execution of language constructs

[internal]: operations which are not part of the language (e.g. incrementing reference counter)

[structural]: heating/ cooling, desugaring etc.

[intermediate]: do auxiliary work before step or internals rule can apply (e.g. type conversion on argument)

[mem]: low level, writing to memory

[error]: cause transition to error state

Rule tags #2

- General pattern of evaluation
  - structural: rearrange state so that other rules apply
  - intermediate: pre-processing of args. e.g. read reference
  - step: evaluate the construct (once the args are in the required form)
- Fine-tuning the transition system
  - kompile –transition = {step, internal, mem,...}

Example - assignment

(A) CONTEXT 'Assign(□,_)

(B) CONTEXT 'Assign(_:KResult,□)

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

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

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

when ¬ isLiteral(V) [intermediate]

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

when currentOverflow(L1) [intermediate]

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

when ¬ currentOverflow(L1) [intermediate]

Validation

- **Testing** against the Zend test suite
  - standard practice (C semantics in K, JS in Coq etc.)
- we pass all tests for core language...
  - ... in the subset we support
- test suite quite small (~200 tests for core language)
  - our own tests => 100% coverage
  - challenge: **generating test suite from K semantics?**

Limitations

- Parts of the language not modeled (yet)
  - abstract classes, interfaces, bit-wise operators, etc.
- Library and internal functions
  - we model only a few
- **Parsing** (currently using external parser)
  - outdated, no line number in AST (for analysis)

Applications

symbolic LTL model check

- Using existing K/Maude support for LTL and symbolic execution
  - Defining our own predicates, specific for PHP
- Mix of symbolic and concrete inputs (via input cell)
- verify 2 real world example in the paper

LTL predicates, example

(the value of the) global variable 'usr'

The (string) value "admin"

```
□¬eqTo(gv(var('usr')),val('admin'))

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

label('login')⇒ □eqTo(gv(var(sec)),val(1))
```

Annotating pgm with labels

LTL predicates, example

```
$y = 0;
function foo() {
  global $y;
  $x = &$y;
}
foo();
```

Can also be **symbolic** input

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

(the value of the) local variable ‘x’ in ‘foo’

(the value of the) global variable ‘y’

Real-world example: Input validation (from PHP My Admin)

```
PMA_isValid(0, "identical", 1);          // false
PMA_isValid(0, "equal", 1);              // true
PMA_isValid("hello", "similar", 1);      // false
```

```
PMA_isValid(0, array(0,1,2));    // true
PMA_isValid(true, "length");     // true, as (string) true = "1"
PMA_isValid(false, "length");    // false, as (string) false = ""
```

```
PMA_isValid("hello", "scalar");          // true
PMA_isValid("hello", "numeric");         // false
PMA_isValid("123", "numeric");           // true
PMA_isValid("anything", false);          // always true
```

- Informal 'spec' written as a comment in source (truth table style)
- correct w.r.t. such spec

Taint analysis

(Student Project with Shijiao Yuwen)

- Turning the semantics into a checker for taint-style vulnerabilities (XSS, SQLi, command injection etc.)
  - Good performance, comparable to existing tools
  - Relatively small effort (master project)
    - once the semantics is done, we can "easily" make tools

Future Work

- systematic extraction of abstract interpreters
- static analysis framework based on abstract interpretation
  - support different checkers: XSS, types, bug patterns, etc.
  - extensible (i.e. people can write plugins)
- other forms of verification?

Some ideas

JavaScript + PHP

- PHP outputs HTML + JavaScript
- Both semantics in K

Reachability logic

How can we use it for verification of PHP code?

Extracting test suites

From the semantics in K, generate a test suite that covers all the rules of the semantics!

Extracting code

- From K definition to OCaml/Java/...
  - fast interpreters
  - fast tools
  - ...

Thanks!

References

- ECOOP'14 - "An Executable Formal Semantics for PHP"
- www.phpsemantics.org
  - sources
  - paper
  - online interface