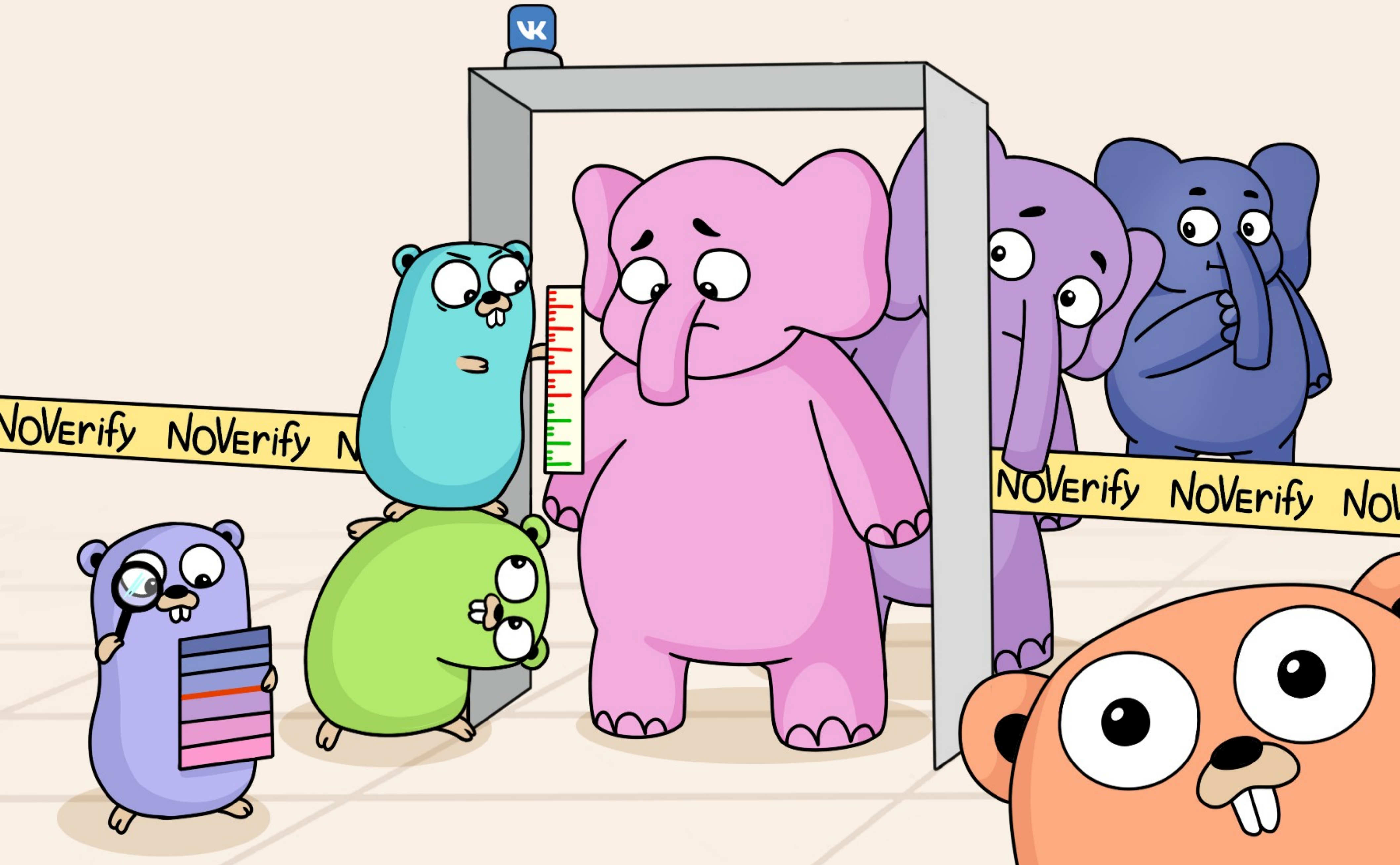
The Missing Static Type Ballad

quasilyte @ PHP Yoshkar-Ola 2019

vk.com infrastructure team

Before we begin...

This presentation is created in LibreOffice Impress. I didn't like the experience at all.


NoVerify NoVerify N
NoVerify NoVerify No

Few words about NoVerify

- **Fast:** several times faster than most linters.
- **Extensible:** extensions in Go and PHP.
- **Language server:** supports LSP.

Telegram group: https://t.me/noverify_linter

What **type** of a presentation this is?

I want to convince you that PHP needs more type facilities even though it get a few new features recently.

Why do we need types info?

- **Documentation:** API contracts
- **IDE:** navigation, autocomplete, refactoring, etc.
- **Static analysis:** find more bugs.
- **JIT** and **AOT:** more optimization space.
- **Meta:** more info for API/schema/code gen.

Dynamic (and implicit) types

- ~~**Documentation:** API contracts~~
- ~~**IDE:** navigation, autocomplete, refactoring, etc.~~
- ~~**Static analysis:** find more bugs.~~
- ~~**JIT** and **AOT:** more optimization space.~~
- **Meta:** more info for API/schema/code gen.
  (As long as reflection is enough for you.)

How much types do we need?

How much types do we need?

For tools, it's "as much as possible".

How much types do we need?

For tools, it's "as much as possible".

For humans, we need to strike a good balance, so people get enough information and do not feel overwhelmed.

“Loosing static types info since 1995”

© PHP

How exactly do we loose types?

Almost every PHP program has **type leaks**

Late static binding (bad-1)

```
class Foo {
  /** @return Foo */
  public static function create() {
    return new static();
  }
}
class Bar extends Foo {}

$b = Bar::create(); // $b:Foo
```

Late static binding (bad-2)

```
class Foo {
  /** @return self */
  public static function create() {
    return new static();
  }
}
class Bar extends Foo {}

$b = Bar::create(); // $b:Foo
```

Late static binding (fixed)

```
class Foo {
  /** @return static */
  public static function create() {
    return new static();
  }
}
class Bar extends Foo {}

$b = Bar::create(); // $b:Bar
```

array type hint (bad)

```
function first_value(array $xs) {
  foreach ($xs as $x) {
    return $x->value; // $x:mixed
  }
  return null;
}
```

array type hint (fixed)

```
/** @param $xs WithValue[] */
function first_value(array $xs) {
  foreach ($xs as $x) {
    return $x->value; // $x:WithValue
  }
  return null;
}
```

Mixed type propagation 1

```
function identity($x) { return $x; }

$x = 10;                // $x:int
$y = identity($x); // $y:mixed
```

Mixed type propagation 2

```
$i = 1; // $i:int

$mixed = [$i];

$i2 = $mixed[0]; // $i2:mixed
```

Guide: how not to loose types info

“You don’t know what you have until it’s gone”

Docs: human-only types

```
// $xs is expected to be an array
// of integers (only int keys).
function last($xs) {
  return $xs[count($xs)-1];
}
// Loosing all types info.
```

Add type hints

```
function last(array $xs) : int {
  return $xs[count($xs)-1];
}
// Still missing a lot of info...
```

Add phpdoc tags

```
/**
 * @param int[] $xs
 *
 * @return int
 */
function last(array $xs) : int {
  return $xs[count($xs)-1];
}
// No int-keys restriction...
```

Add generics-aware tags

```
/**
 * @param int[] $xs
 * @psalm-param array<int,int>
 * @return int
 */
function last(array $xs) : int {
  return $xs[count($xs)-1];
}
```

It doesn't look good...

Can we do better?

If we had generic arrays

```
function last(array<int,int> $xs) : int {
  return $xs[count($xs)-1];
}
```

PHP 10 ʕ⊙ϖ⊙ʔ

```
func last(xs []int) int {
  return xs[len(xs)-1]
}
```

Seriously, we need changes

- Educate people, explain why we need them or at least “generic arrays”.
- Get noticeable community support.
- Work on the v2 proposal for generics.
- Convince PHP devs that this feature is needed.
- Find people who will implement generics.

Type information sources

Type inference and related problems

Type inference

Since types information is mostly implicit, we need to infer it from expressions.

It's not always possible to get a precise result, since we almost always loose at least some types info along the way.

Type can also depend on the run-time information that we don't have.

Type guessing game

~~Since types information is mostly implicit, we need to infer it from expressions.~~

~~It's not always possible to get a precise result, since we almost always loose at least some types info along the way.~~

~~Type can also depend on the run-time information that we don't have.~~

Trying to guess types

Precise type info

The most reliable types info sources

Scalars and literals

```
$i = 1;             // $i:int
$f = 1.3;           // $f:float
$s = "x";           // $s:string
$o = new Obj(); // $o:Obj
$ii = [1, 2];   // $ii:int[]
```

Smart-casts

```
if ($o instanceof Obj) {
  // $o is Obj inside this block.
}

if (!is_string($s)) {
  return
}
// $s is string below this if statement
```

Type hints (in strict mode)

```
function f(int $i, array $xs) {
  // $i is int.
  // $xs is an array (of mixed).
}
```

Typed properties

```
class Typed {
  public int $i = 10;
  public ?string $s;
}

$t = new Typed();
$i = $t->i; // $i:int
$s = $t->s; // $s:?string
```

Imprecise type info

Secondary types info sources

phpdoc annotations

```
/** @var DBConnection $db */
global $db;

// + function/methods phpdocs.
// + PhpStorm meta files (stubs).
```

Flow-related types

```
if ($moon_phase) {
  $x = 10;
} else {
  $x = "hello";
}
// $x:int|string
```

Optimistic inference

```
$xs = [1, 2]; // $xs:int[]
$x = $xs[$i]; // $x:int

// In reality, if $i key is not in $xs,
// $x will be null, so the exact type
// if more like ?int, but most
// programs perform optimistic inference,
// where we omit some details...
```

Pragmatical sacrifices

It can be bad to be smarter than PhpStorm when we're talking about types.

You don't want to resolve less types, but it's not always desirable to resolve more.

This is especially true with global types inference.

Local type inference

To get expression type, **local type inference** only uses that expression plus the context info (variable types, functions info, etc).

Global type inference

With **global type inference,**
an expression type might depend
on very distant parts of a program.

Seemingly irrelevant code changes
can cause a lot of changes in types
inference results.

Local type inference

```
// $x       => ?
// returns => ?
function get_x($p) { return $p->x; }

// Somewhere inside a code base:
$p = new Point();
$x = get_x($p);
```

Global type inference

```
// $x       => Point
// returns  => float
function get_x($p) { return $p->x; }

// Somewhere inside a code base:
$p = new Point();
$x = get_x($p);
```

Global type inference

```
// $x       => Point|int
// returns => float|null
function get_x($p) { return $p->x; }

// Somewhere inside a code base:
$p = new Point();
$x = get_x($p);
$x2 = get_x(19);
```

Side-by-side comparison

Local	Global
+ Simplicity	+ Completeness
+ Locality	+ Precision
+ Faster	
Good enough for most static analysis tools.	Good for optimizing compilers and audit-oriented static analysis tools.

NoVerify types resolving

The problem

```
function f3() { // f3:?
  return f2();
}
function f2() { // f2:?
  return f1();
}
function f1() { // f1:int
  return 10;
}
```

The problem

- Dependent symbols can live far away from each other (different parts of a project).
- Projects can be too large to keep them in memory (several GB).
- We don’t want to make extra “passes” over the source code (too slow).
- We also don’t want to re-calculate all types when one file is changed.

Solution: lazy types

```
function f3() { // f3:f2()
  return f2();
}
function f2() { // f2:f1()
  return f1();
}
function f1() { // f1:int
  return 10;
}
```

Solution: lazy types

- **First pass:** index the entire project, record symbols info and lazy types.
- **Second pass:** do the analysis itself.

When type info is needed, it's "solved" on demand. Only files that are currently being analyzed are loaded into the memory.

Solving f3()

```
$x = f3(); // $x:f3()
```

Solving f3()

```
$x = f3(); // $x:f2()

typeof(f3()) => call(f2)
```

Solving f3()

```
$x = f3(); // $x:f1()

typeof(f3()) => call(f2)
typeof(call(f2)) => call(f1)
```

Solving f3()

```
$x = f3(); // $x:int

typeof(f3()) => call(f2)
typeof(call(f2)) => call(f1)
typeop(call(f1)) => int
```

Challenges

- 2-passes limitation make it harder to collect whole-program facts.
- Lazy types are slower than precalculated types. If we cache them, we loose some of their benefits.

Is single-pass possible?

If we have forward declarations, like in C, then yes. But that's not what you would expect from a modern programming language.

Metadata cache

- The “first pass” (indexing) is only executed if we don’t have file info. If there is none, indexing is executed and results are saved to a disk.

  So in practice it’s one-pass in some cases.

Get involved!

NoVerify is an open-source project, your contributions are welcome.

PhpStorm limitations

Imprecise suggestions (1/2)

```
function f($x) {
  if (…) {
    return false;
  }
  return (int)g();
}
```

Imprecise suggestions (2/2)

```
/** @return int|bool */
function f($x) {
  if (…) {
    return false;
  }
  return (int)g();
}
```

Union-typed array elements

```
/** @var (Foo|Bar)[] $a */
$a = [
  new Foo(),
  new Bar(),
];

$foo = $foos[0]; // $foo:mixed
```

Homogeneous array literals

```
$foos = [
  new Foo(),
  new Foo(),
];

// $foo is not inferred to be Foo.
// Need @var phpdoc.
$foo = $foos[0]; // $foo:mixed
```

Tuples

```
/** @return tuple(int,string) */
function add1($x) {
  if (!is_numeric($x)) {
    return [0, '$x must be numerical'];
  }
  return [$x + 1, ''];
}
list($v, $err) = add1($x);
// $v:mixed, $err:mixed
```

Resources

- Generic arrays RFC (2016)
- Generics RFC (2016)
- Generics and why we need them
- Typed properties RFC
- PhpStorm stubs
- PhpStorm deep-assoc plugin
- PHPDoc types format (ABNF)

NoVerify resources

- Habr: NoVerify public announcement
- Habr: NoVerify dynamic rules
- NoVerify Telegram group

Keep your **types** safe!

The Missing Static Type Ballad

quasilyte @ PHP Yoshkar-Ola 2019

vk.com infrastructure team