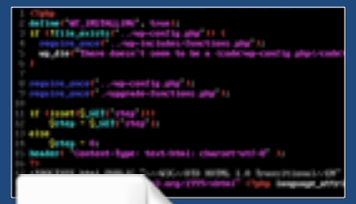
facebook

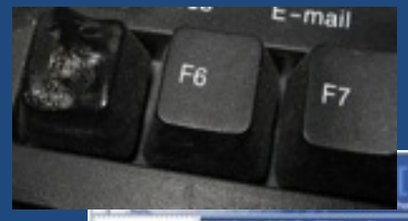
facebook

Intro to HHVM

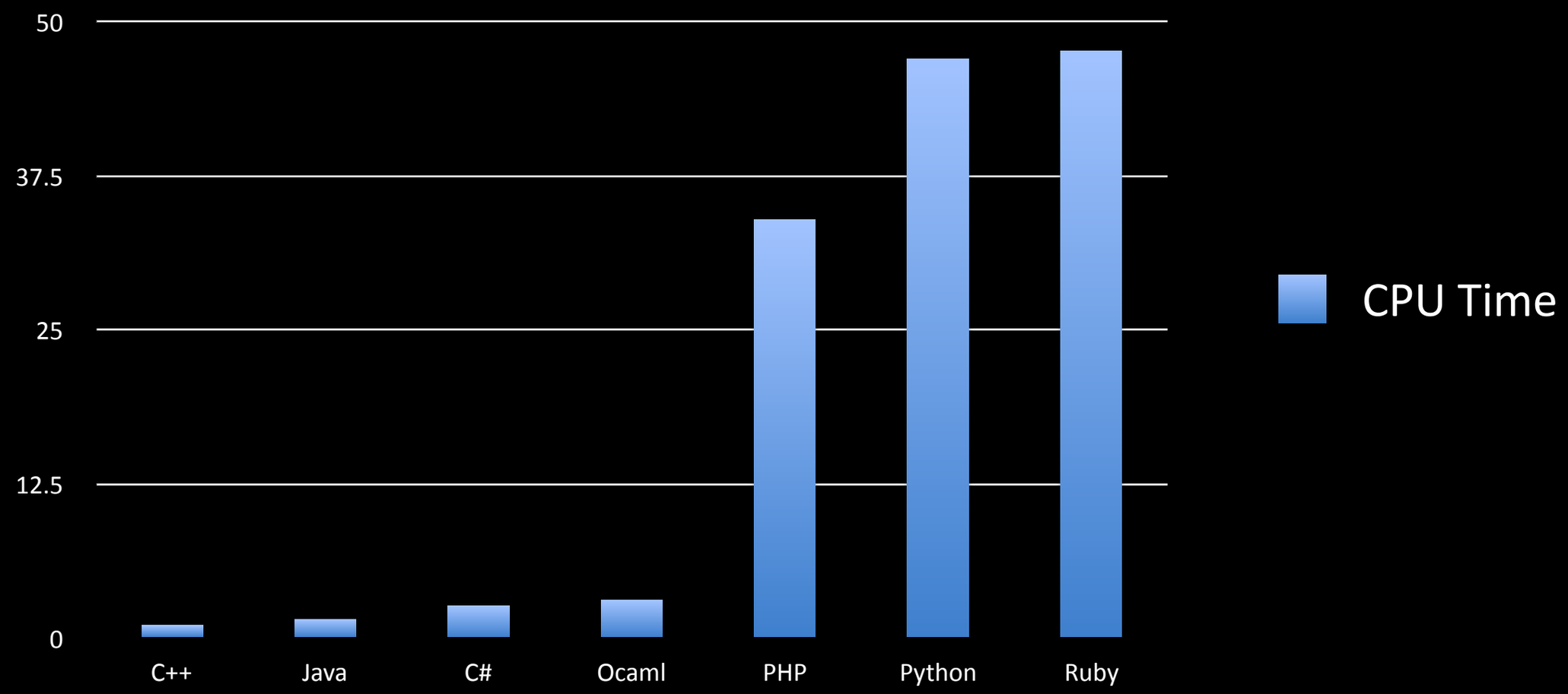
Matt Clarke-Lauer

PHP @ Facebook





Performance... ☹



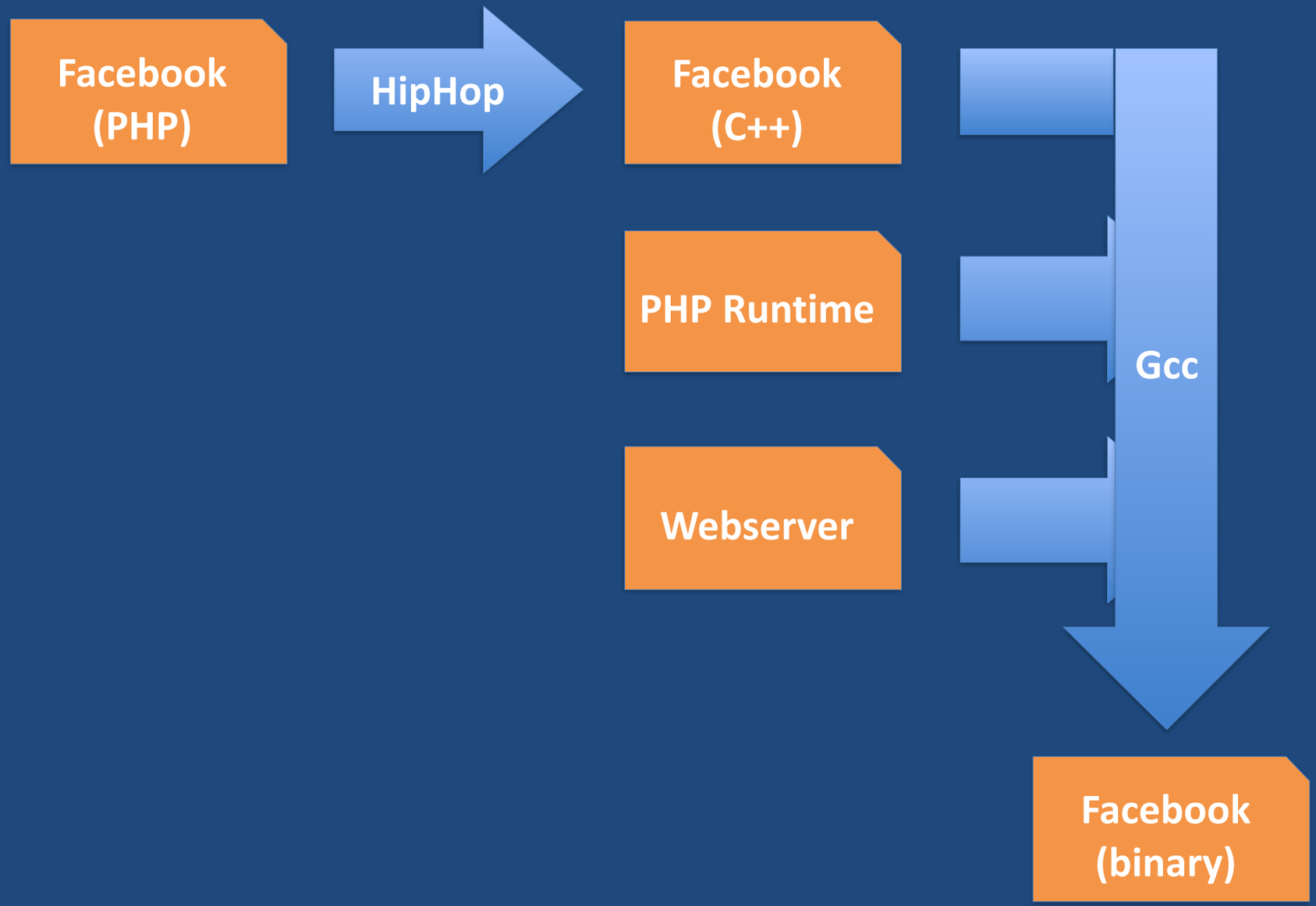
Source: http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=all

Implications for Facebook

1. Bad performance can limit user features
2. Poor efficiency requires lots of re$ource$

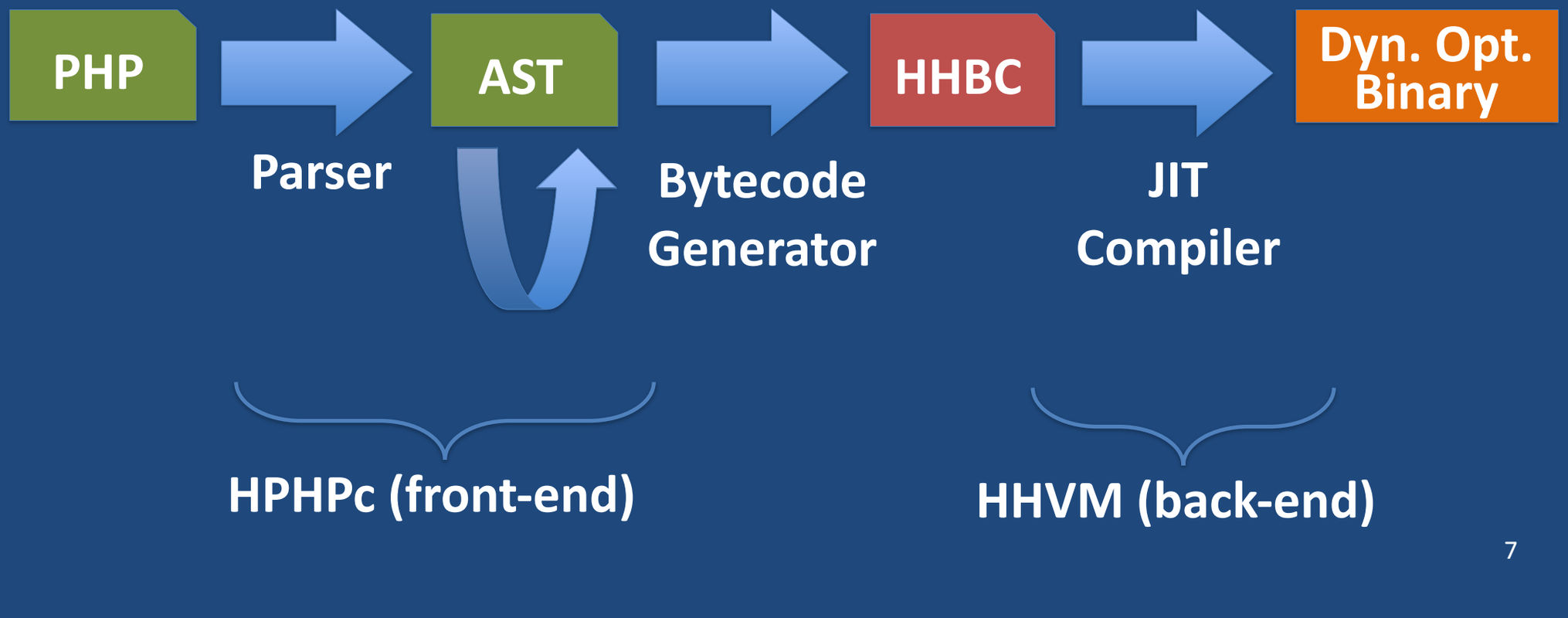
HipHop v1.0: Static Compiler (HPHPc)

[OOPSLA'2012]

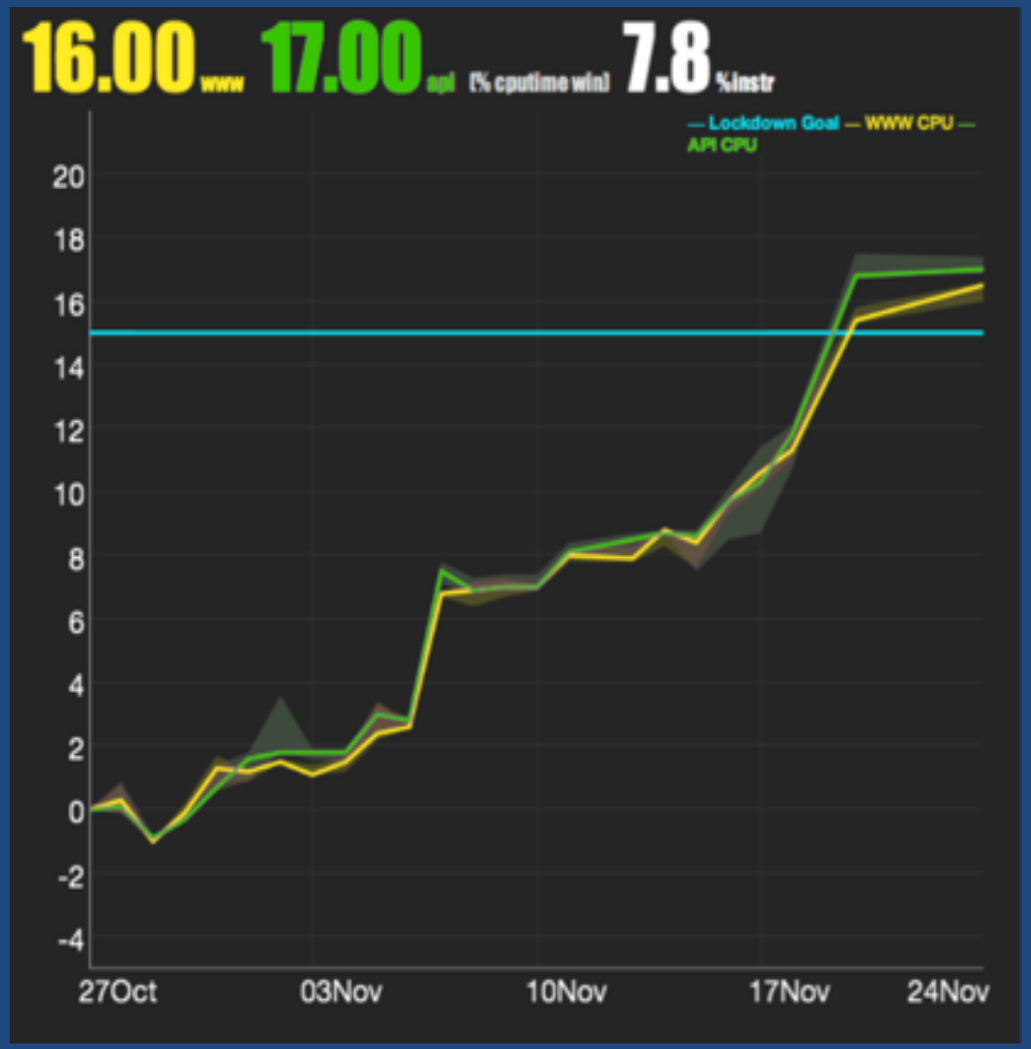


HipHop v2.0: Virtual Machine (HHVM)

- Ambitious goals:
  1. Serve both development and production
  2. Generate better code than the static compiler



Performance!!!



HHVM is Open Source!

- http://github.com/facebook/hiphop-php
- http://hhvm.com
- #hhvm freenode

Contributions are appreciated!

facebook

Three Hack talks butchered into a single talk

Butchery by Gabe Levi

Hi, I'm Gabe

- Software Engineer at Facebook since 2010
- Has worked on
  - News Feed
  - Platform
  - Notifications
  - Product Infrastructure
  - Hack
- HHVM noob

Today I'll be butchering these talks:

1	"Introducing Hack" by Julien Verlaguet
2	"Convert Your PHP Code to Hack" by Josh Watzman
3	"A Tour of New Language and Library Features" by Drew Paroski and Eugene Letuchy

http://www.youtube.com/playlist?list=PLb0IAmt7-GS2fdbb1vVdP8Z8zx1l2L8YS


hack

Hack: a sweet spot

dynamic

FAST

static

g++ -02 -whatever

Type checking

C++

Linking

./a.out

DEMO!

https://github.com/gabelevi/Boston-PHP-Meetup-Examples

Hack

- A **statically typed** language for HHVM
- Compatible with PHP:
  - Interoperates with no overhead
  - Same representation at runtime
- Evolved from PHP:
  - If you know PHP, you know Hack!
- Designed for **incremental adoption**:
  - Gradual typing

Hack Types

- All PHP types: `int`, `MyClassName`, `array`
- Nullable: `?int`, `?MyClassName`
- Mixed: anything (careful)
- Tuples: `(int, bool, X)`
- Closures: `(function(int): int)`
- Collections: `Vector<int>`, `Map<string, int>`
- Generics: `A<T>`, `foo<T>(T $x): T`
- Constraints: `foo<T as A>(T $x): T`

Hack Type System

- What must be annotated?
  - Class members
  - Function parameters
  - Return types
- What is inferred?
  - All the rest

Hack Modes

- Where we want to be: `<?hh // strict`
  - Type-checks everything.
- The default: `<?hh`
  - Assumes the code is correct if an annotation is missing.
  - Assumes unknown functions/classes are defined in PHP (even when they are not)
- The minimum: `<?hh // decl`

DEMO!

https://github.com/gabelevi/Boston-PHP-Meetup-Examples

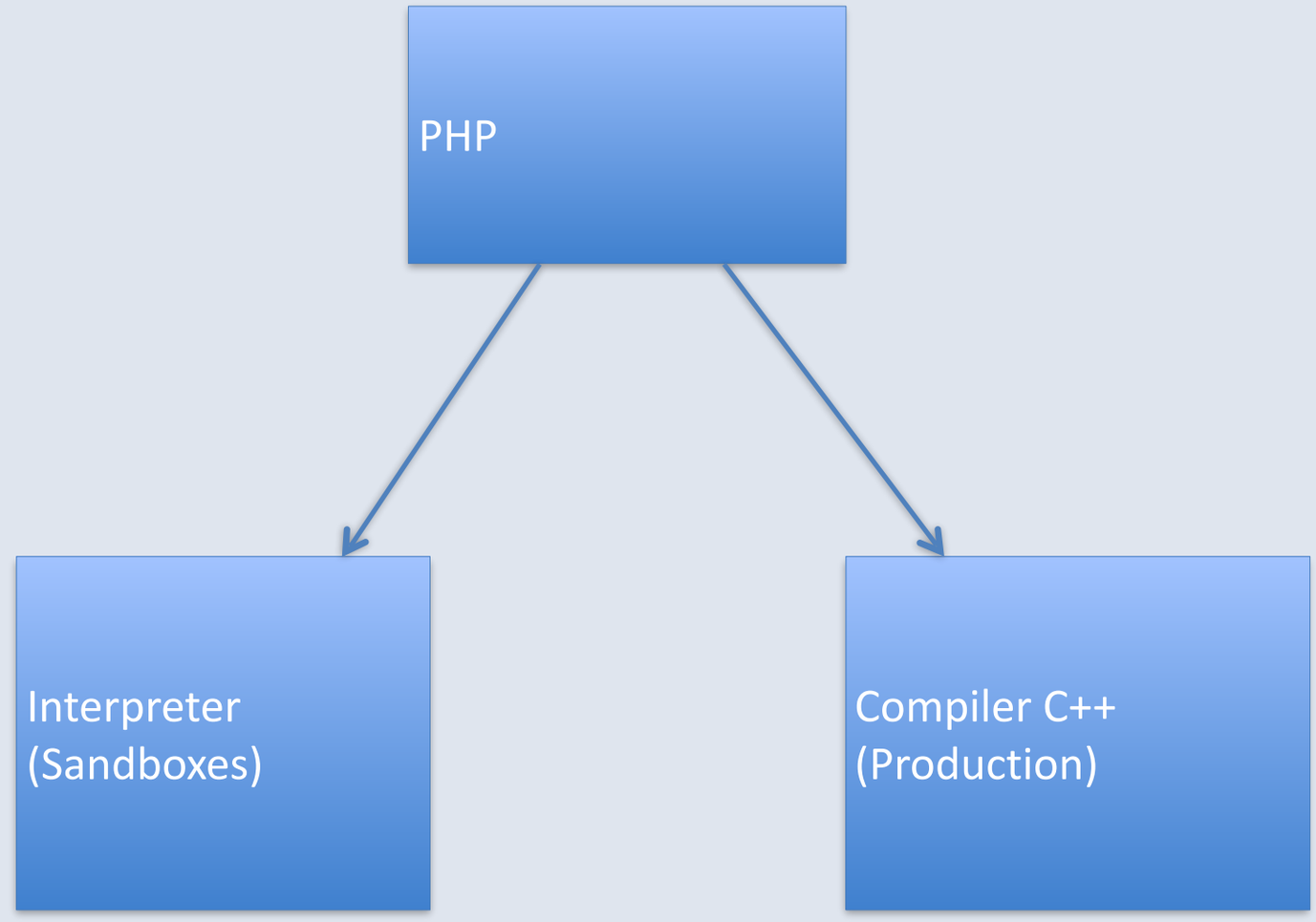
Error messages

- We can't expect the user to understand all the type-inference
- The solution: keep the reason why we deduced a type and expose it to the user

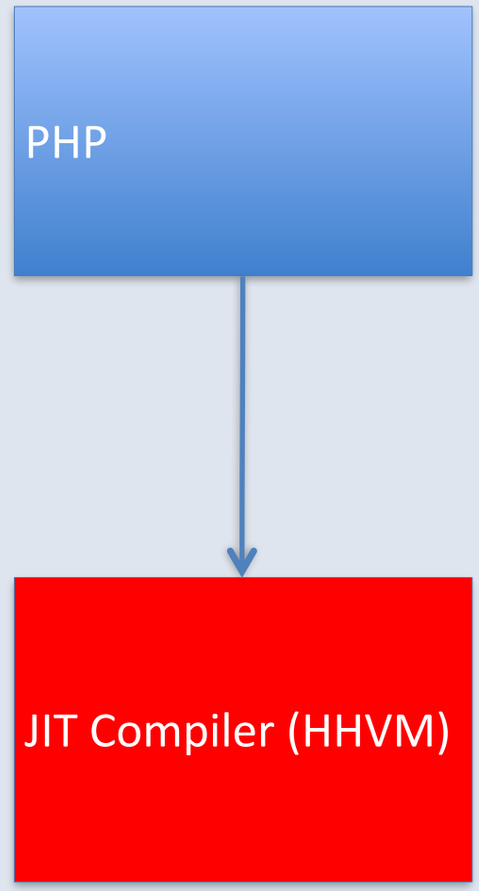
```
File "test.php", line 6, characters 10-11:
Invalid return type
File "test.php", line 3, characters 24-26:
This is an int
File "test.php", line 5, characters 10-11:
It is incompatible with a string
```

Under the hood: architecture

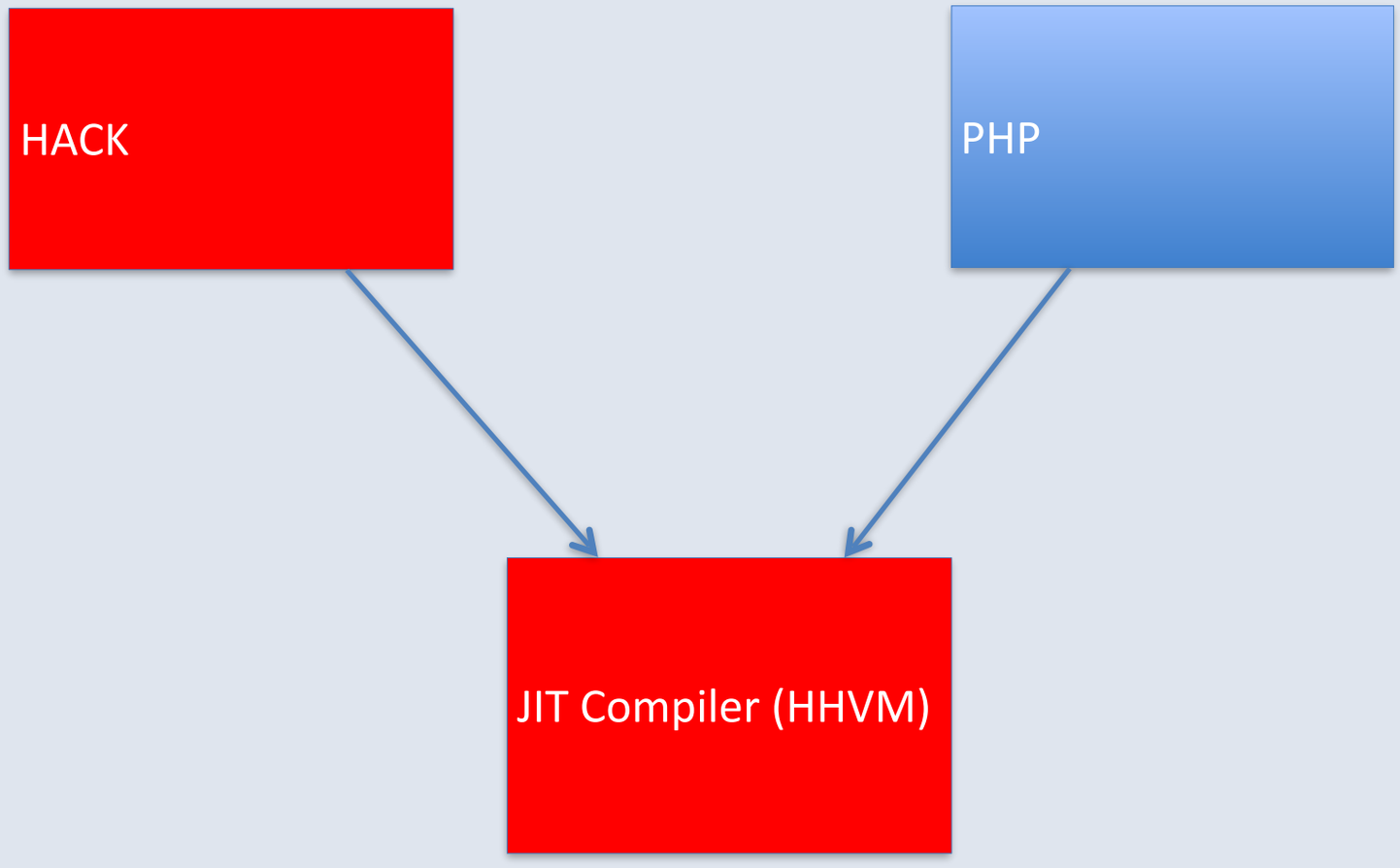
HipHop



HHVM



Hack for HHVM



Traditional architecture

- A compiler: takes input files produces .o
- A Makefile: recomputes the object files when something changed
- A linker: produces a binary file from .o
- IDE: talks to the compiler
- This design came to be for historical reasons

Hack

- We knew we wanted an IDE from day one
- Big code base
- The solution, a server:
  - The server type-checks all the files
  - Keeps track of the dependencies
  - Recomputes types when something changed

The Constraints

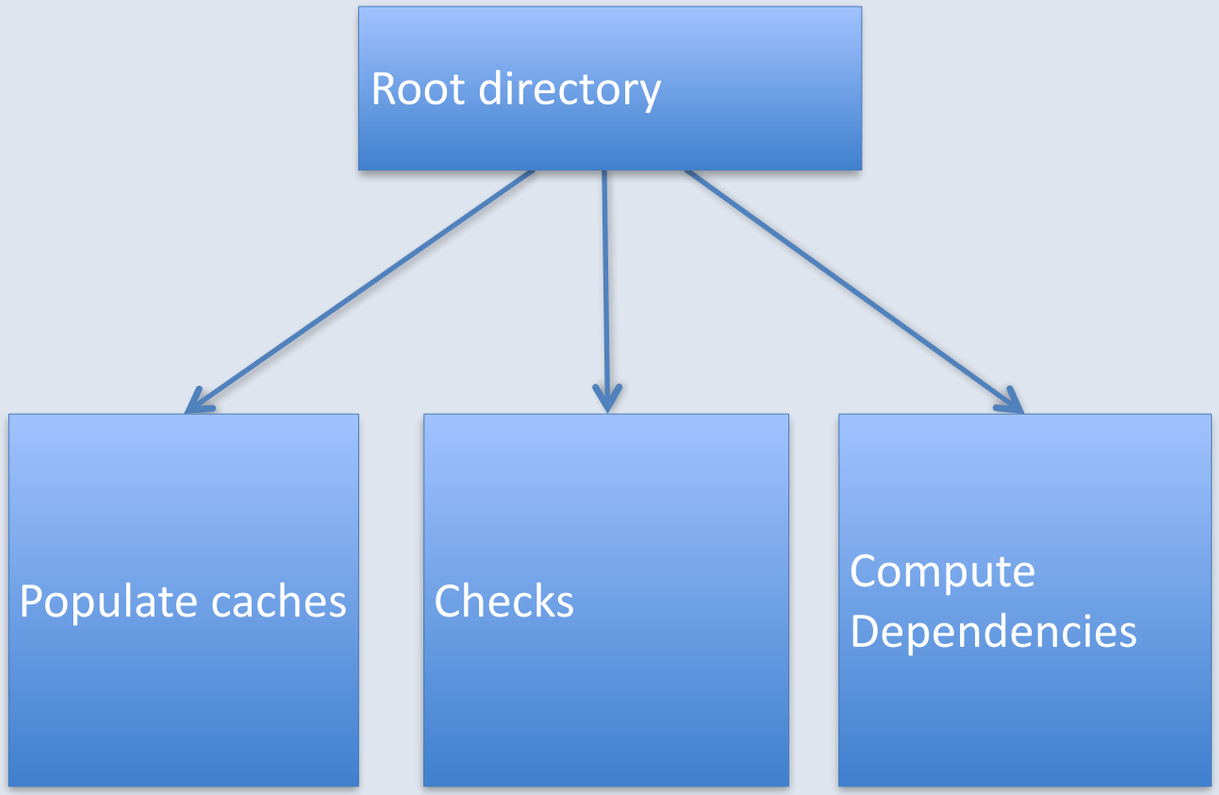
- Low latency makes things more fun!
- Users use version control (e.g., switching between branches)
- We must use a reasonable amount of RAM
- We must have a reasonable initialization time
- Must be **stable**

Working at scale: The Solution

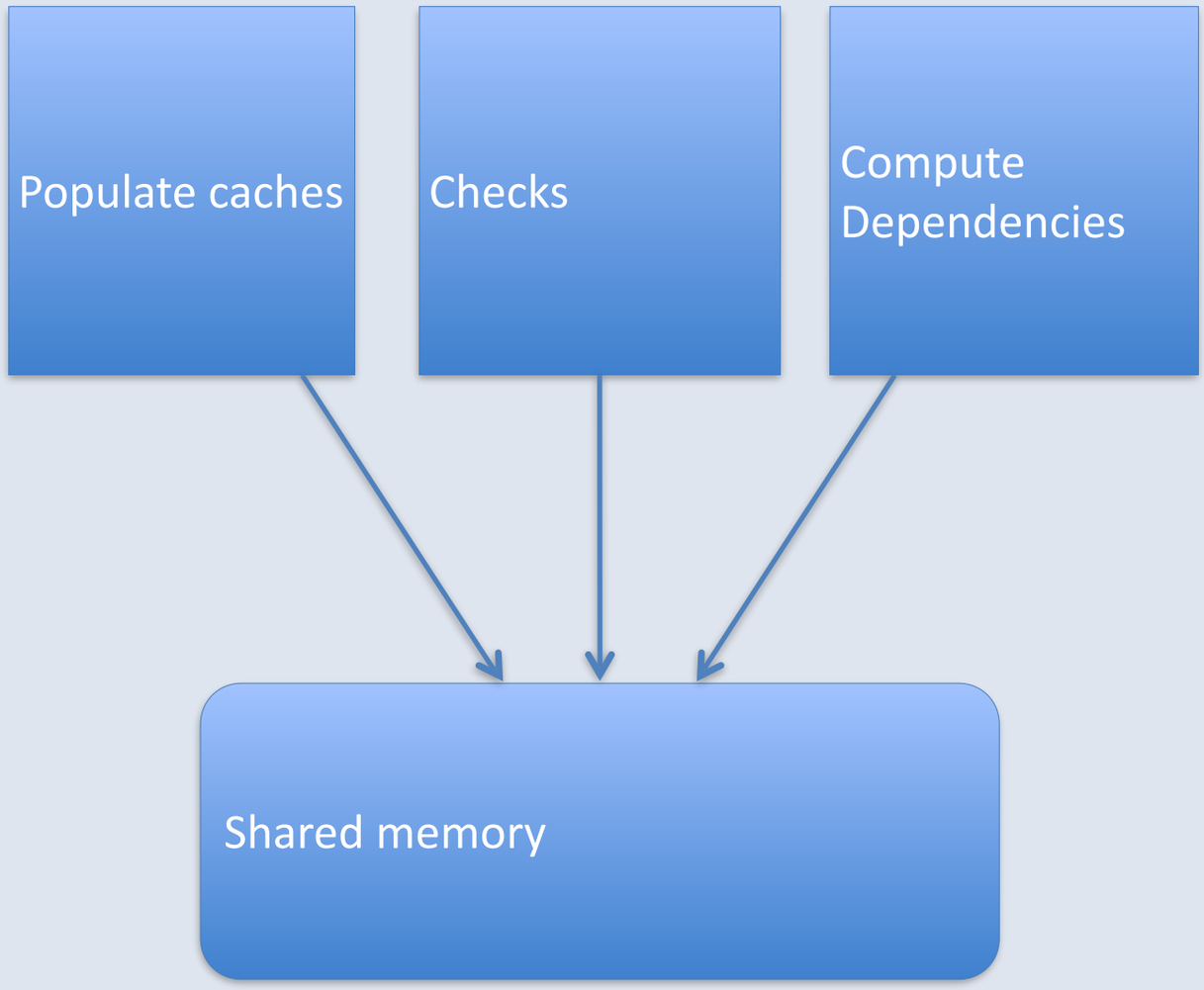
- How do we make this work:
  - We parallelize
  - We cache
  - We do things incrementally

What happens under the hood?

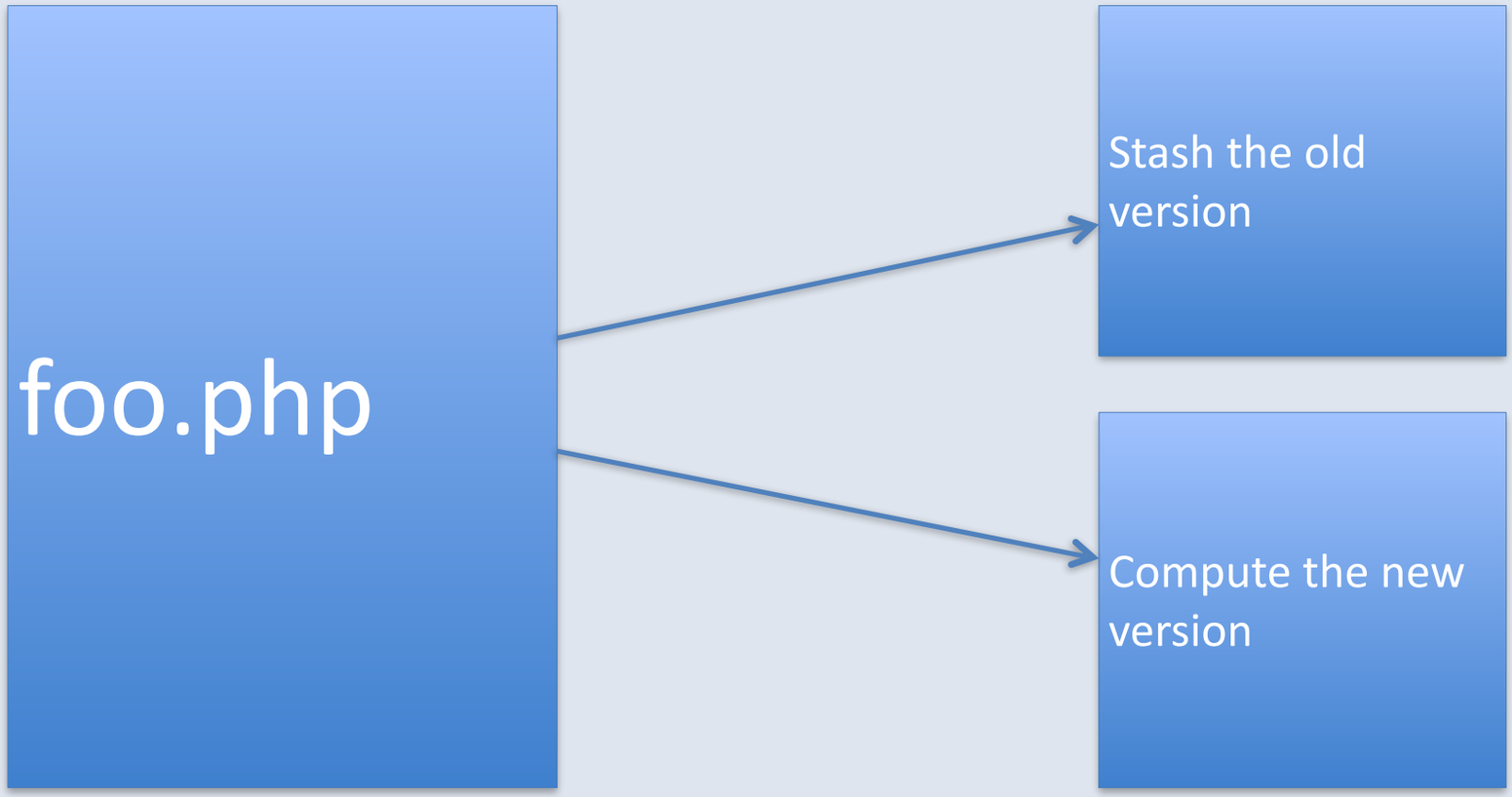
- At initialization time



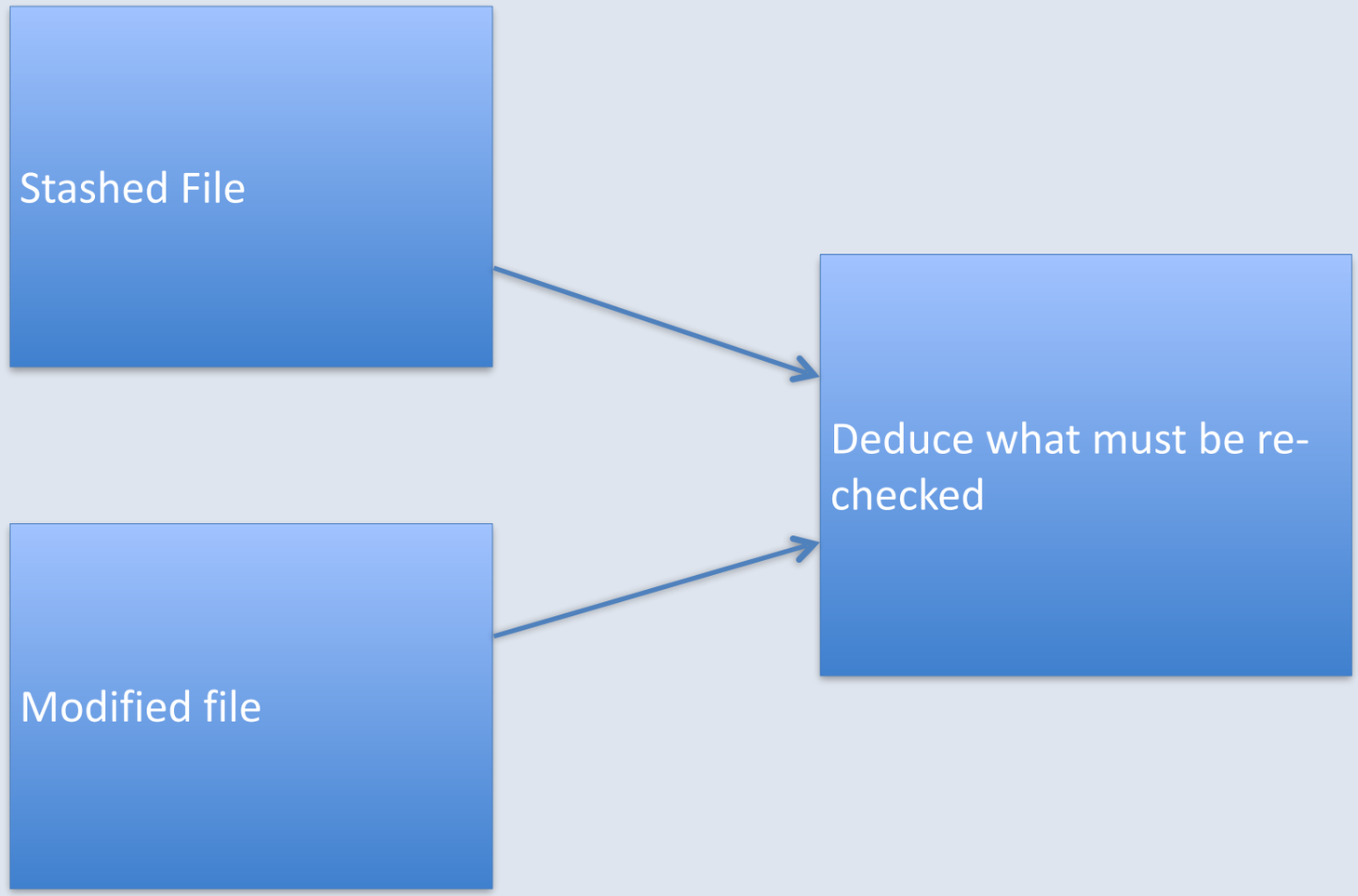
What happens under the hood? (2)



When you modify a file (1)

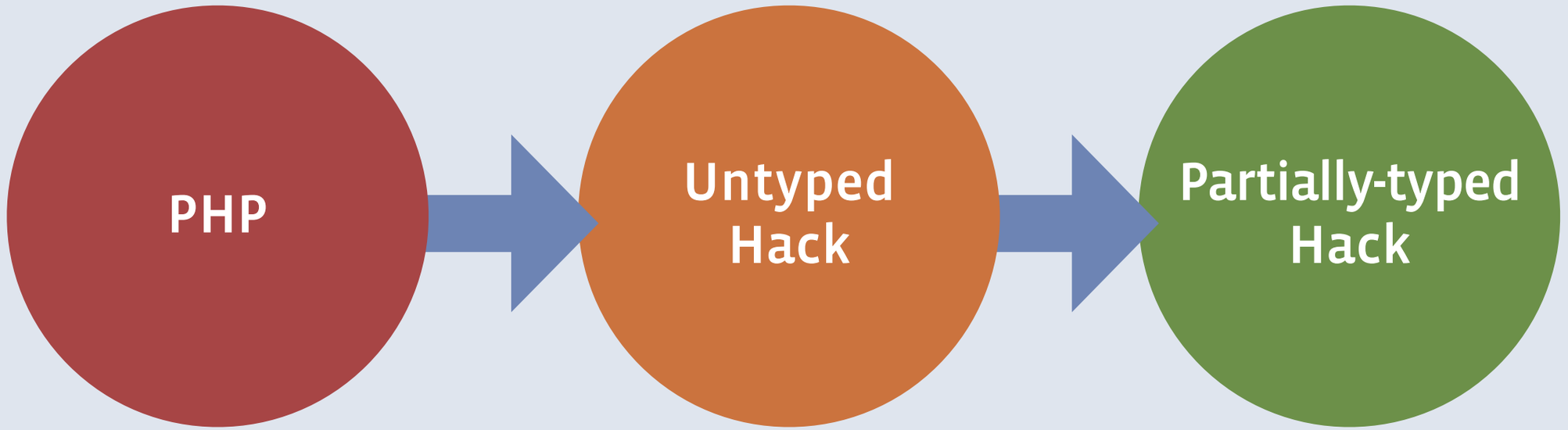


When you modify a file (2)



Conversion process

Conversion process



hackificator – basic conversion

```php
<?php

  class Foo {
    public function f() {
      return 42;
    }
  }
```

```hh
<?hh

class Foo {
  public function f() {
    return 42;
  }
}
```

hackificator – failed conversions

```php
<?php

class Foo {
  public function f() {
    // x is not declared!
    $this->x = 1;
  }
}
```

```hh
<?hh // decl

class Foo {
  public function f() {
    // x is not declared!
    $this->x = 1;
  }
}
```

hackificator – syntactic tweaks

```php
<?php

class C {
    // ...
}

function f(C $x = null) {
    // ...
}
```

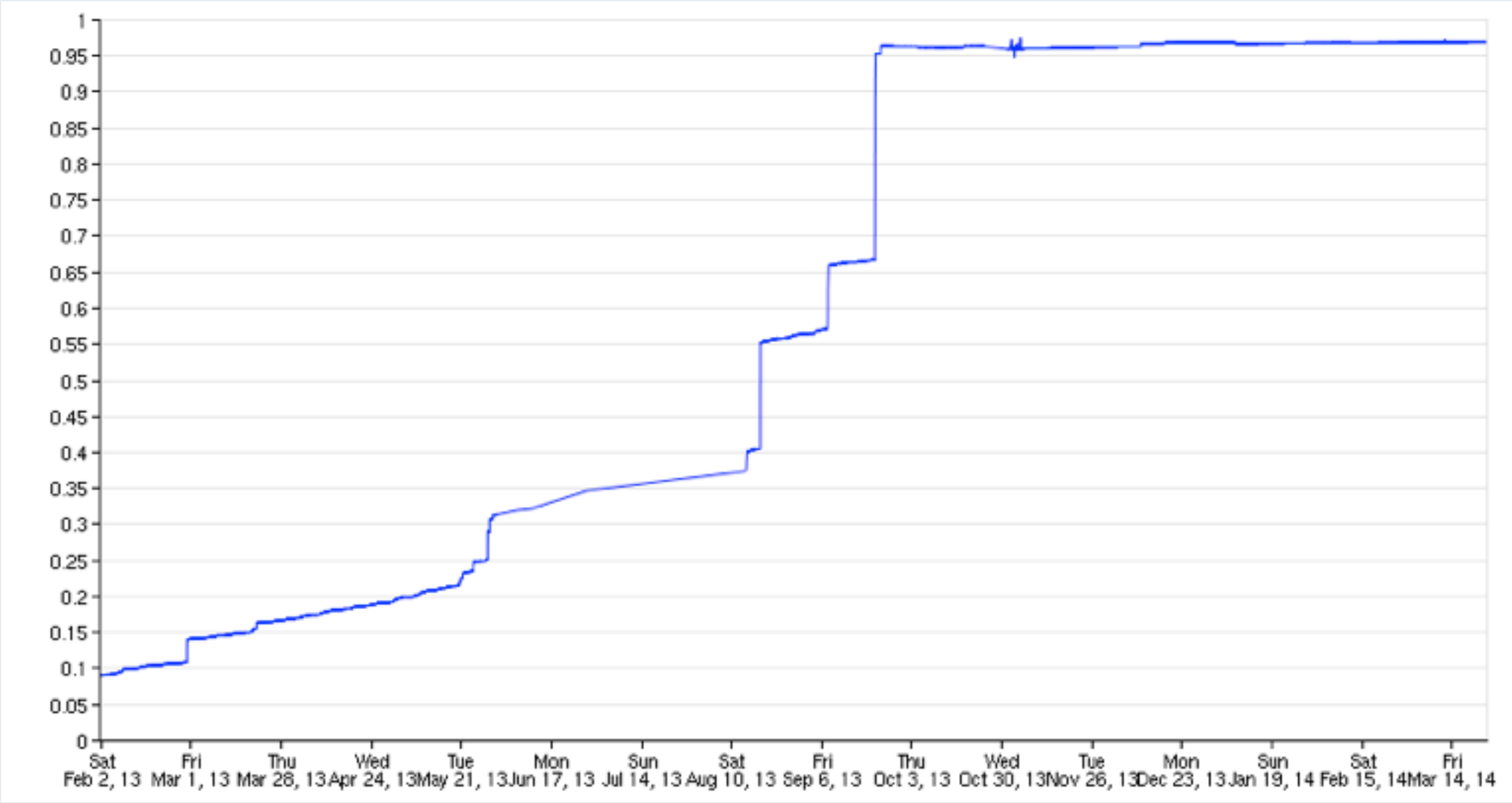
```hh
<?hh

class C {
    // ...
}

function f(?C $x = null) {
    // ...
}
```

hackificator – syntactic tweaks

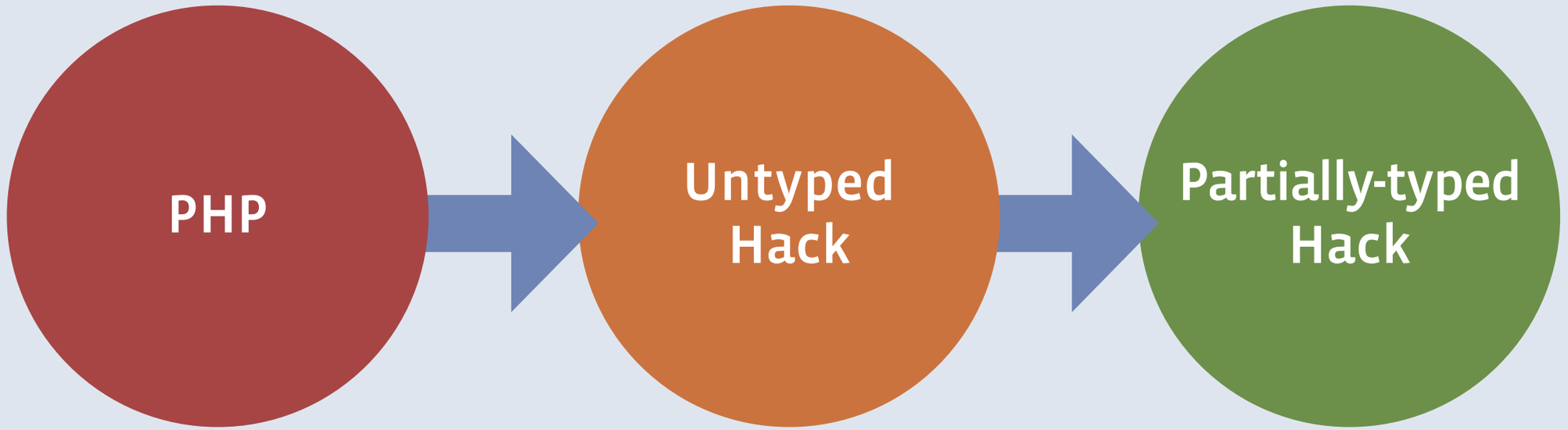
```php
<?php

class Foo {
    // ...
}

function f() {
    $x = new Foo;
}
```

```hack
<?hh

class Foo {
    // ...
}

function f() {
    $x = new Foo();
}
```

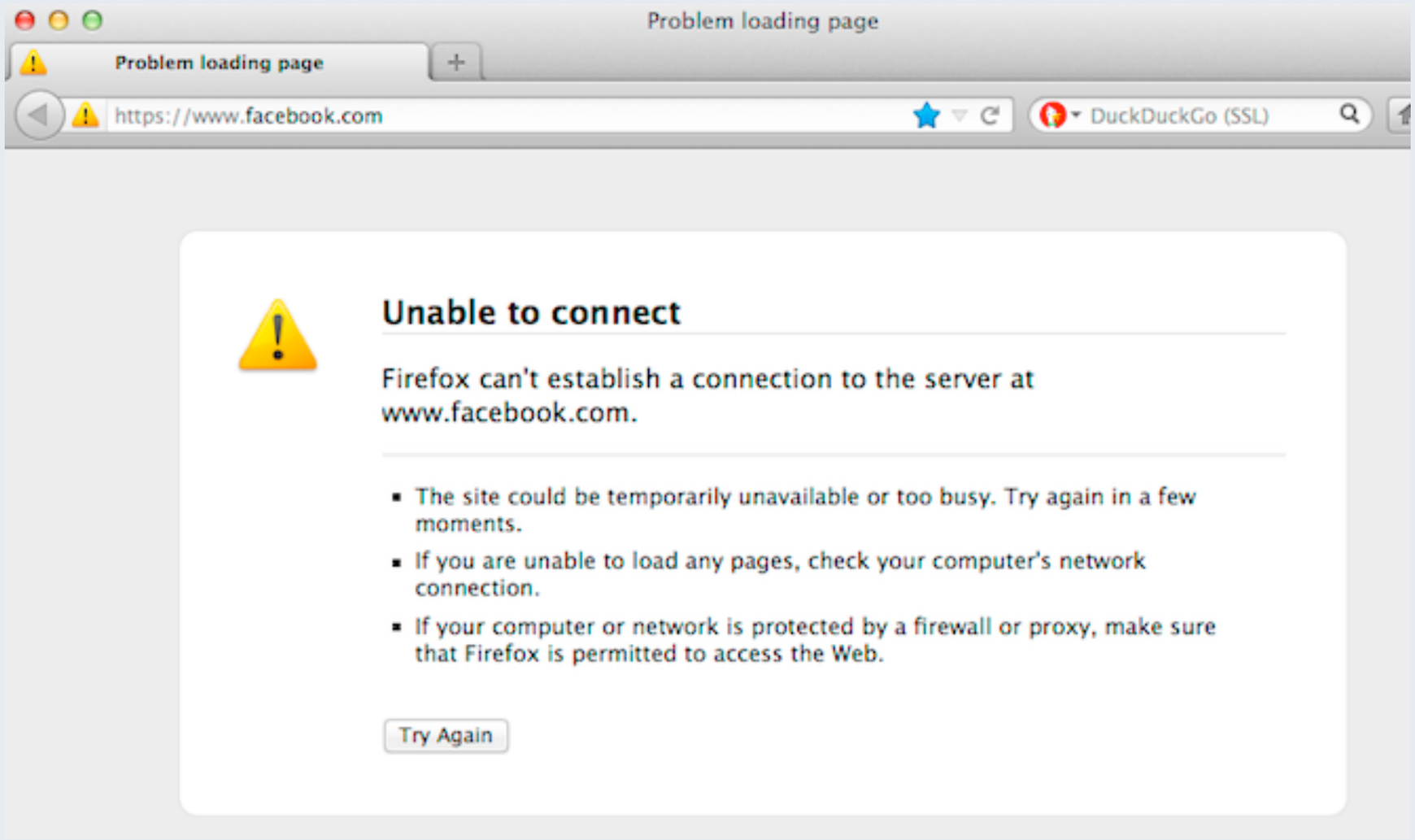
hackificator – results



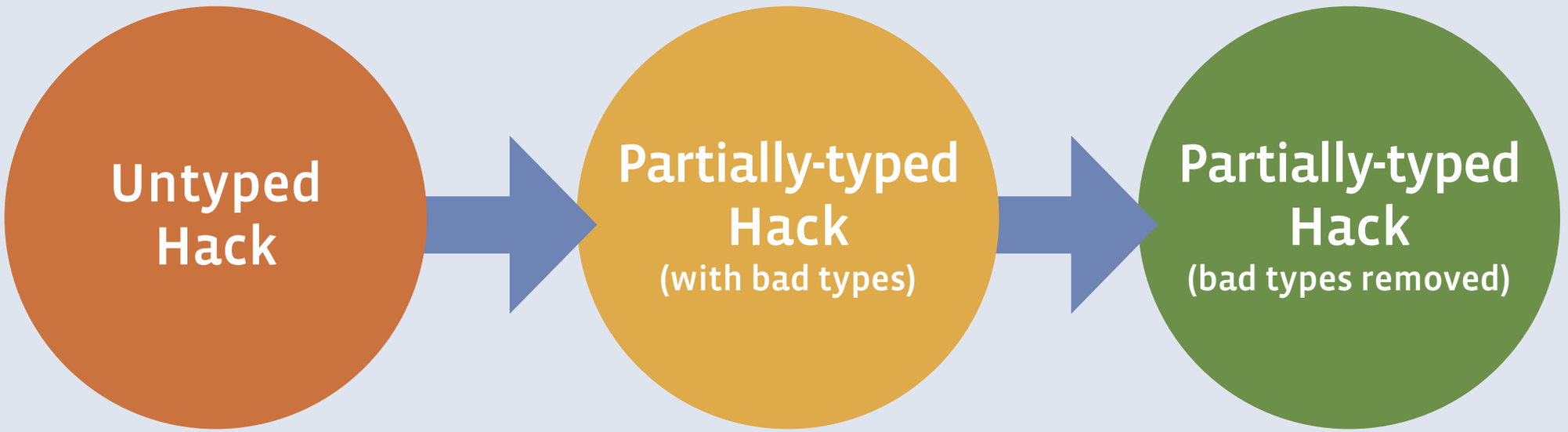
Conversion process



Adding type annotations



Adding type annotations



hh_server – convert

```
<?hh

function f($x) {
  if ($x) {
    return new Foo();
  } else {
    return null;
  }
}

function g($y) {
  f(42);
  return f(103);
}

function h() {
  g('hello world');
  g(44);
}
```

hh_server – convert

```
<?hh

function f(@int $x): @?Foo {
  if ($x) {
    return new Foo();
  } else {
    return null;
  }
}

function g($y) {
  f(42);
  return f(103);
}

function h(): @void {
  g('hello world');
  g(44);
}
```

hh_server – convert

```
<?hh

function f(@int $x): @?Foo {
  if ($x) {
    return new Foo();
  } else {
    return null;
  }
}

function g($y): @?Foo {
  f(42);
  return f(103);
}

function h(): @void {
  g('hello world');
  g(44);
}
```

More visibility is better

```
<?hh

function f($x): void {
    // ...
}
```

```
function g(): void {
    f(1);
}
```

More visibility is better

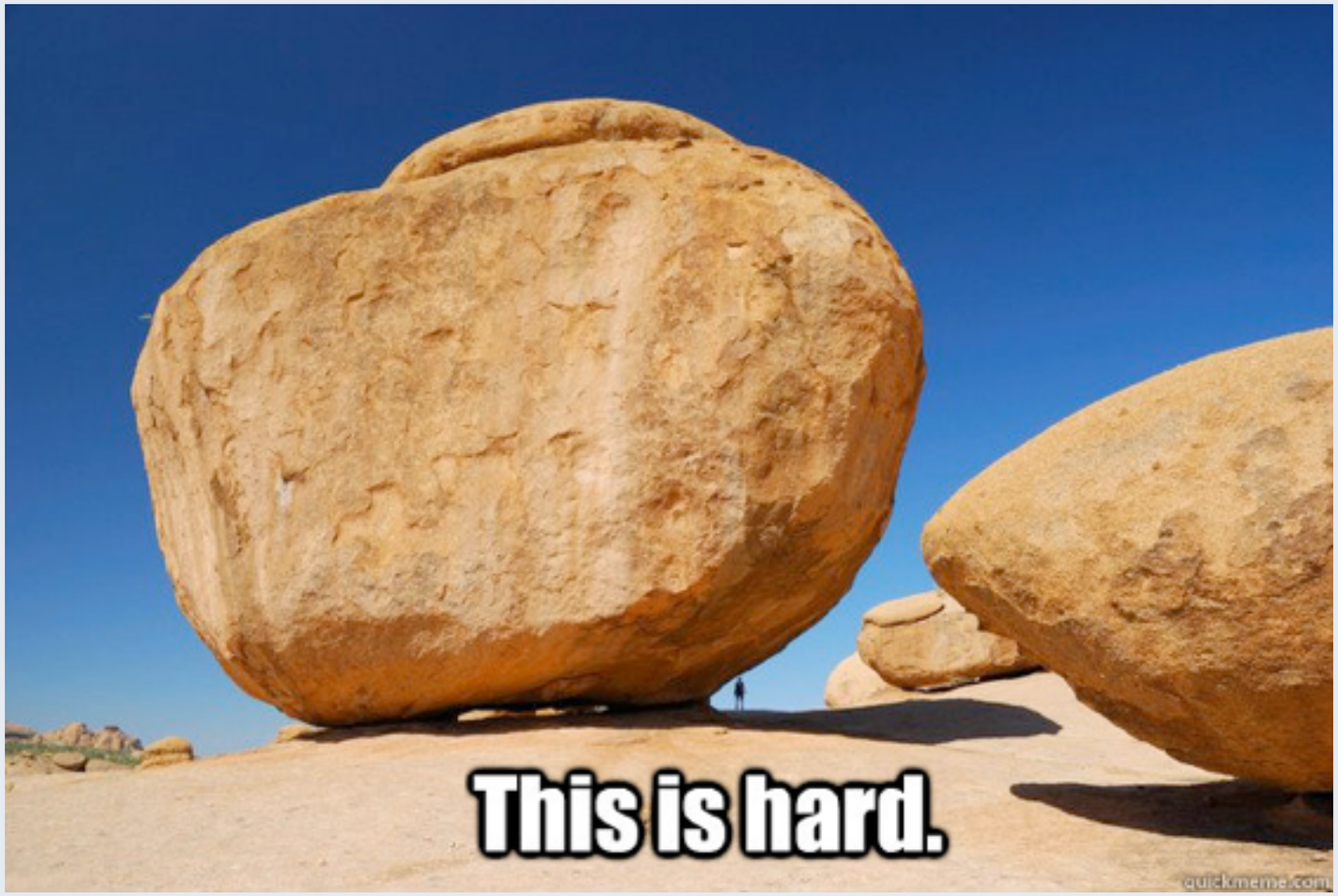
```hack
<?hh

function f($x): void {
    // ...
}
```

```hack
<?hh

function h(): void {
    f(null);
}
```

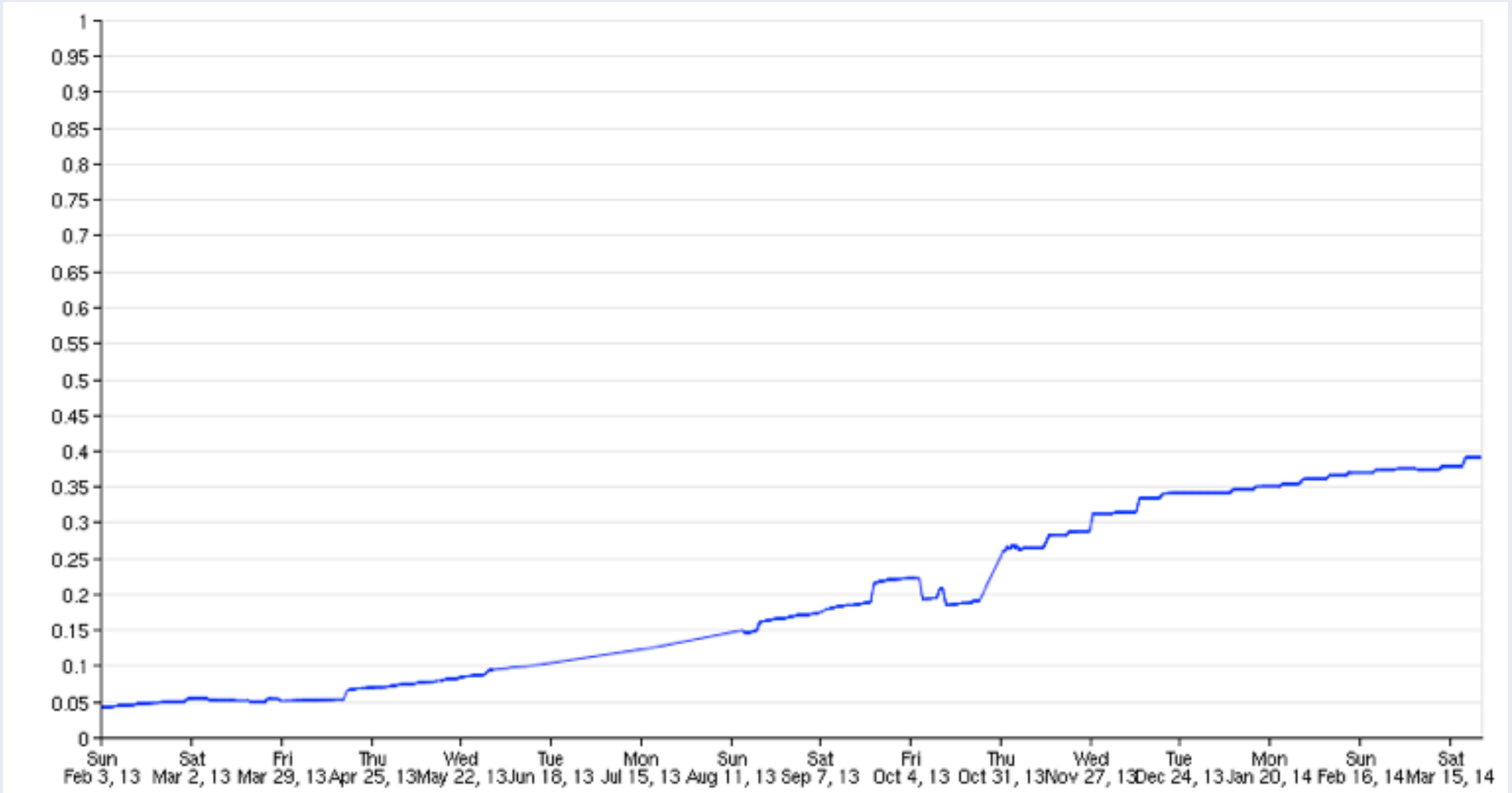
```hack
function g(): void {
    f(1);
}
```

Removing bad types

Hardening types



Conversion results



Hack Language Features and APIs

Runtime Enforcement

```
// Object type hints already in PHP
function foo($my_int, MyObject $my_obj) { return 1; }

// Hack adds runtime-enforced primitive type hints
function foo(int $my_int) { return 1; }

// Hack adds runtime-enforced return type hints
function foo(int $my_int): int { return 1; }
```

Lambda Expressions

```php
// PHP closure syntax
$b = 1;
$fn = function($a) use($b) { return $a + $b; }

// Hack lambda syntax
$b = 1;
$fn = $a ==> $a + $b;
```

More Examples

```php
// Multi-statement body
$fn = ($x, $y) ==> $x + $y;

// Multi-statement body
$fn =
  $x ==> { $y = foo($x); return bar($y,$y); };

// Transitive capture
$z = 5;
$fn1 = $x ==> $y ==> $x + $y + $z;
$fn2 = $fn1(3);
var_dump($fn2(4)); // outputs 12
```

Collections

- The Hack Collections framework is a set of language features and APIs that serve as a complete alternative to PHP arrays

```
// PHP arrays
$a = array(1, 2, 3);
$b = array('a' => 4, 'b' => 5);
function foo(array $c): void { .. }

// Hack collections
$a = Vector {1, 2, 3};
$b = Map {'a' => 4, 'b' => 5};
function foo(Map<string,int> $c): void { .. }
```

Collections

- 3 mutable collection types:
  - Vector<Tv>
  - Map<Tk, Tv>
  - Set<Tv>
- Immutable collections too (more on that later)

map/filter example

```
// PHP arrays
$x = array_filter(array_map($fn1, $a), $fn2);
```

```
// Hack Collections
$x = $a->map($fn1)->filter($fn2);
```

Collections + Lambdas = Awesome

```
// PHP arrays and closures
$x = array_filter(
  array_map(
    function($a) use($b) { return $a + $b; },
    $array
  ),
  function($a) { return $a > 10; }
);


// Hack Collections and lambdas
$x = $vector->map($a ==> $a + $b)
            ->filter($a ==> $a > 10);
```

Benefits

- Separate types for the vector, map, and set use cases
- Designed for object oriented and functional programming styles
- Work smoothly with static typing and generics
- Better API design
- Don’t have the backwards-compat baggage

One-size-fits-all approach

- PHP arrays try to be both a **vector** *and* a **map**
- Typically the programmer intends to use a given array only as a vector or only as map for its entire lifetime
- PHP arrays' one-size-fits-all approach can lead to subpar API design and cause confusion

array_merge() example

```php
$x = array_merge(
  array('x' => 3),
  array('x' => 7)
);
var_dump($x);

// Output
array("x" => 7)
```

```php
$x = array_merge(
  array(124 => 3),
  array(568 => 7)
);
var_dump($x);

// Output
array(0 => 3, 1 => 7)
```

Downsides of one-size-fits-all

- Using arrays to emulate a set is clunky:

```
$x = array('a' => true, 'b' => true);
$x['c'] = true;
if (isset($x['a'])) {
  // ...
}
```

- The PHP array data structure is a bit more complex than a basic vector or hashtable, making it harder to implement efficiently

Static Typing

```
function foo(string $s): void { .. }
function bar(array<string> $a): void {
  $s = $a[2];
  // type checker thinks $s is a string
  // at this point
  foo($s);
}
bar(array('baz'));

// Output
Notice: Undefined offset: 2
Error: expected string, got null
```

Static Typing

```
function foo(string $s): void { .. }
function bar(Vector<string> $a): void {
  // throws if key is out of bounds
  $s = $a[2];
  // ..
  foo($s);
}
bar(Vector {'baz'});

// Output
Fatal error: Uncaught exception 'OutOfBoundsException'
```

Static Typing

```
function foo(string $s): void { .. }
foreach (array('123' => 1) as $k => $v) {
  var_dump($k);
  foo($k);
}
```

```
// Output
int(123)
Error: expected string, got int
```

Static Typing

```
function foo(string $s): void { .. }
foreach (Map {'123' => 1} as $k => $v) {
  var_dump($k);
  foo($k);
}
```

```
// Output
string(3) “123”
```

PHP arrays have value-type semantics

- When PHP arrays are assigned, passed, or returned a logical copy is made. In practice, there are cases where programmer needs to use PHP references.

```
function f(array $a) { $a['x'] = 123; }
function g(array &$a) { $a['x'] = 123; }
function bar() {
  $x = array();
  f($x);
  // bar's $x was not modified by f()
  g($x);
  // bar's $x was modified by g()
}
```

Collections are objects

- Callee can modify the original collection that was passed in

```
function f(Map<string,int> $a): void {
  $a['x'] = 123;
}
function g(ConstMap<string,int> $a): void {
  // type checker will not allow mutating
  // methods or operations on $a
}
function bar(): void {
  $x = Map {};
  f($x);
  // bar's $x was modified by f()
  g($x);
}
```

Iterable Interface

- The Iterable interface includes map(), filter(), zip(), and more
- Iterable lays the foundation for building good APIs and language features for working with collections and other kinds of sequence-like things such as generators

Immutable Collections

- 3 immutable collection types:
  - ImmVector‹Tv›
  - ImmMap‹Tk,Tv›
  - ImmSet‹Tv›
- Useful when the programmer wants to pass or return a collection but wants run time protection against modification
- Covariant
  - If class Duck extends class Animal
  - ImmVector‹Duck› is a subtype of ImmVector‹Animal›

Immutable Collections Example

```
class C {
  public Vector<string> $vec = Vector {};
  public ImmMap<string,int> $map = ImmMap {};
  public function foo(): ImmVector<string> {
    return $this->vec->immutable();
  }
  public function bar(): ImmMap<string,int> {
    return $this->map;
  }
  // ..
}
```

Migrating existing code to Collections

- Many builtin array functions such as array_map() are have been updated to support collections
- Using collections with common constructs behaves the same as arrays:
  - `foreach`
  - `$c[$k]`
  - `(bool)` cast, `!` operator, and `empty()`
  - Serialization
- `array` parameter typehints implicitly convert a collection passed into a PHP array

Future Directions

- More higher order function goodness like flatMap() and groupBy()
- Run time enforcement of key types and value types
- Objects as keys for Maps (and values for Sets)
- More performance optimizations

Async/await

Code example: Sync

```
function complex_calculations($args
): Result {
  // ... query dbs, memcache, services etc.
  return new Result(..);
}

function f($id): .. {
  $args = ...;
  $result = complex_calculations($args);
  do_stuff_with($result);
}
```

Code example: Sync => Async

```
async function complex_calculations($args
): Awaitable<Result> {
  // ... query dbs, memcache, services etc.
  return new Result(..);
}
```

```
async function f($id): Awaitable<..> {
  $args = ...;
  $result = await complex_calculations($args);
  do_stuff_with($result);
}
```

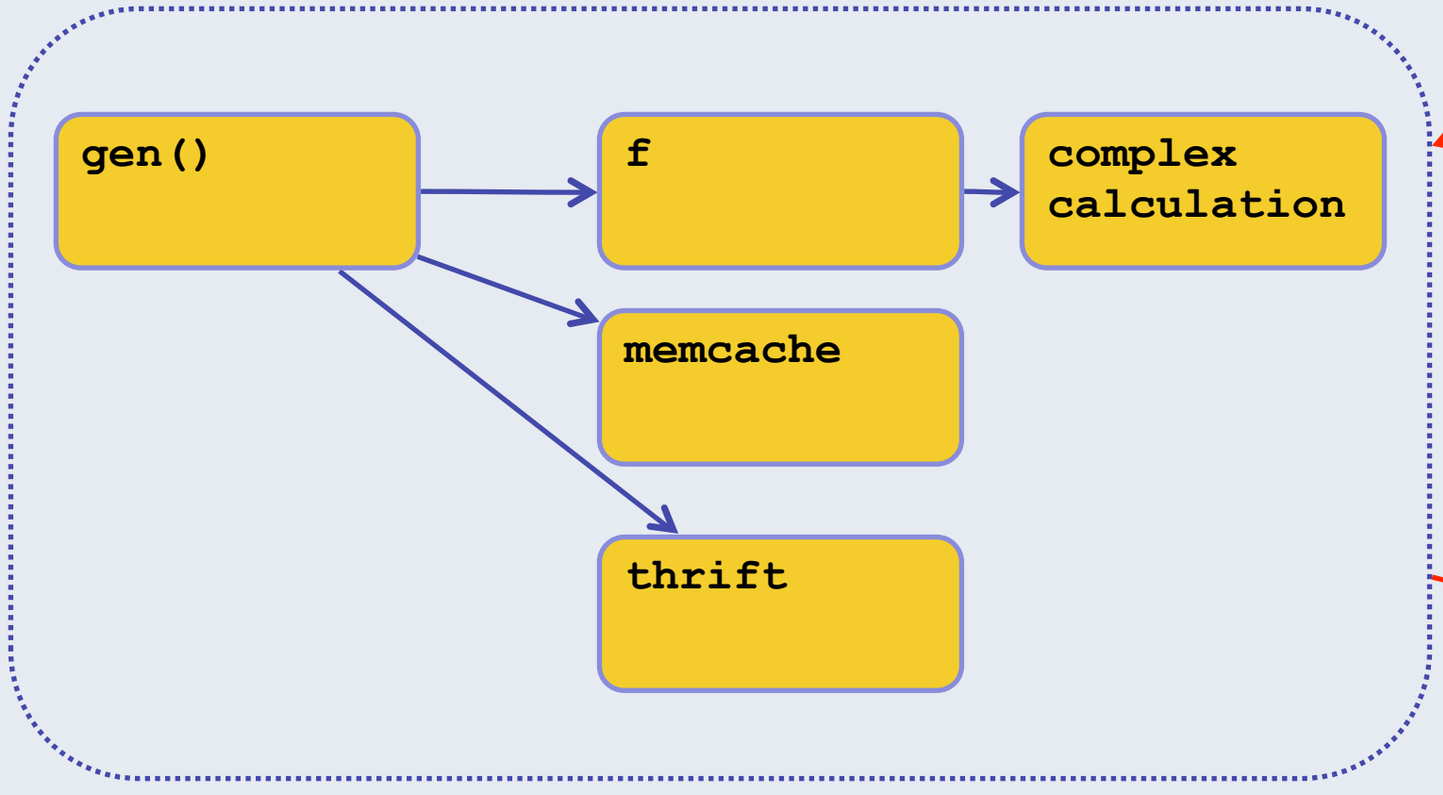
Awaitable

```
async function f($id): Awaitable<..> {
  $args = ...;

  $a = complex_calculations($args);
  // $a is an Awaitable<Result> object;
  // represents the (unstarted) state
  // of computation of complex_calculation()
  $result = await $a;
  do_stuff_with($result);
}
```

Parallel async/await

```
async function gen_response(): Awaitable<WebPage> {
  $user_id = …;
  list($a, $b, $c) = await gen(Vector {
    f($user_id),
    get_stuff_from_memcache(..),
    call_thrift_service(..),
  });
  // all of $a, $b, $c are assigned now
  return new WebPage($a, $b, $c);
}

async function get_stuff_from_memcache($b): Awaitable<int> { .. }
async function call_thrift_service($b): Awaitable<int> { .. }

function gen(Vector $v): Awaitable<..> {
  return GenVectorWaitHandle::create($v);
}
```

Mid-execution




```
async function gen_response(
): Awaitable<WebPage> {
  $user_id = …;
  list($a, $b, $c) = await gen(
    f($user_id),
    get_stuff_from_memcache(..),
    call_thrift_service(..),
  );
  return new WebPage($a, $b, $c);
}
```

```
async function gen_response(
): Awaitable<WebPage> {
  $user_id = …;
  list($a, $b, $c) = await gen(
    f($user_id),
    get_stuff_from_memcache(..),
    call_thrift_service(..),
  );
  return new WebPage($a, $b, $c);
}
```

Entering async

```
..
$response_awaitable = gen_response();
$web_page = $response_awaitable->join();
..
```

State of async/await

… a work in progress

- Batching/Rescheduling usable today
  - see “Coalesced Fetching” example on docs.hhvm.com
- More async-compatible APIs “coming soon”™

Goodies!: Constructor Promotion

```php
<?php
class C {
  private $prop1;
  private $prop2;
  .. // more lines
  private $propN;
  public function __construct(
    A $p1,
    B $p2,
    .. // more lines
    $pN = 1
  ) {
    $this->prop1 = $p1;
    $this->prop2 = $p2;
    .. // more lines
    validate($pN);
    $this->propN = $pN;
}
```

```hh
<?hh
class C {
  public function __construct(
    private A $prop1,
    private B $prop2,
    .. // more lines
    private int $propN = 1
  ) {
    validate($this->propN);
  }
}
```

Goodies!: Trailing Commas

```php
<?php

$a = array(
  'one',
  'two', // optional
);

f(
  $arg1,
  $arg2, // fatal!
);

function f(
  $arg1 = 'one',
  $arg2 = 'two', // fatal!
) { … }
```

```hh
<?hh

$a = array(
  'one',
  'two', // optional
);

f(
  $arg1,
  $arg2, // optional!
);

function f(
  $arg1 = 'one',
  $arg2 = 'two', // optional!
) { … }
```

Traits

Traits in PHP5

- Code reuse without inheritance
- Model: “copy-paste stuff into a scope”
- Pretty sweet, but hard on static analysis

Detecting Errors in Traits

```
trait MyTrait {
  final public function callF(
  ): void {
    $this->f('string');
  }

  final public function callG(
  ): int {
    return $this->g();
  }
}
```

```
class Hypothetical {

  use MyTrait;

  protected function f(
    string $x
  ): void {}

  public function g(): int { .. }
}
```

No runtime error!

Detecting Errors in Traits

```
class C {
  protected function f(
   int $x,
  ): void { .. }
}
```

```
interface I {
  public function g(): void { .. }
}
```

```
class Hypothetical
  extends C
  implements I {

  use MyTrait;
}
```

Runtime error!

Trait requirements

```php
<?php

/* This trait is supposed to be used
 * with classes that extend C. */
trait MyTrait {

  final public function callF(
  ) {
    $this->f('string');
  }

  // Pretty please implement I
  // or this code will, like, fatal
  final public function callG(
  ) {
    return $this->g();
  }
}
```

```hack
<?hh

trait MyTrait {
  require extends C;
  require implements I;

  final public function callF(
  ): void {
    $this->f('string');
  }

  final public function callG(
  ): int {
    return $this->g();
  }
}
```

Trait requirements: solution

```hack
<?hh

class C {
  protected function f(
    int $x,
  ): void { .. }
}

interface I {
  public function g(): void { .. }
}
```

```hack
<?hh

trait MyTrait {

  require extends C;
  final public function callF(
  ): void {
    $this->f('string');
  }

  require implements I;
  final public function callG(
  ): int {
    return $this->g();
  }
}
```

Trait requirements

- `require extends Superclass`
- `require implements IFace`

- Enforced at `use` sites
- Imposes constraints on `$this` in trait definition
  - … which means we can check it independently!

```
<?hh

trait MyTrait {

  final public function callF(
  ): void {
    $this->f('string');
  }

  final public function callG(
  ): int {
    return $this->g();
  }
}
```

Summing up: Hack Traits

- Model: “Mix in encapsulated functionality”
  - Provide internal & external API
  - Independently type-checkable

- Does it work? … look at trait definition
- Type errors in the trait methods? … look at trait definition
- Traits tied to type hierarchies? … via trait requirements

Goodies!: Trait Interfaces

```hh
<?hh

interface IDoStuff {
  public function f(): int {}
}

trait StuffTrait implements IDoStuff{
  public function f(): int {
    return 42;
  }
}

class C {
  use StuffTrait;
  // C has the interface!
}
```

```hh
<?hh

function f(IDoStuff $arg) { .. }

f(new C()); // works!
```

User Attributes

‹‹User('Attributes')›› Syntax

```
..
<<Awesome>>
class C {
  <<Special>>
  function f(<<Cool>> $a): int { .. }
  ..
}
..
```

User Attributes:

- Annotations without grammar changes, code-in-comments, etc.
- No runtime effect beyond parsing
- Reflection: `getUserAttributes()`
- Parameterized: `<<Attribute('expr')>>`

Some Uses

- HHVM System Builtins (HNI): `<<__Native>>`
- Unit Tests: declaring mockability
- Code generation
- Arbitrary tooling …

Example: <<Override>>

```
class CParent {
  public final function doStuff() {
    $this->impl();
  }
  protected function impl() {
    ..
  }
}

class CChild extends CParent {
  <<Override>>
  protected function impl().. {
    .. // plugs into CParent::doStuff
  }
}
```

- Child-class counterpart to **`abstract`**
- Prevents refactoring mistakes
- Catches typos

Getting started

- **http://www.hacklang.org**

facebook