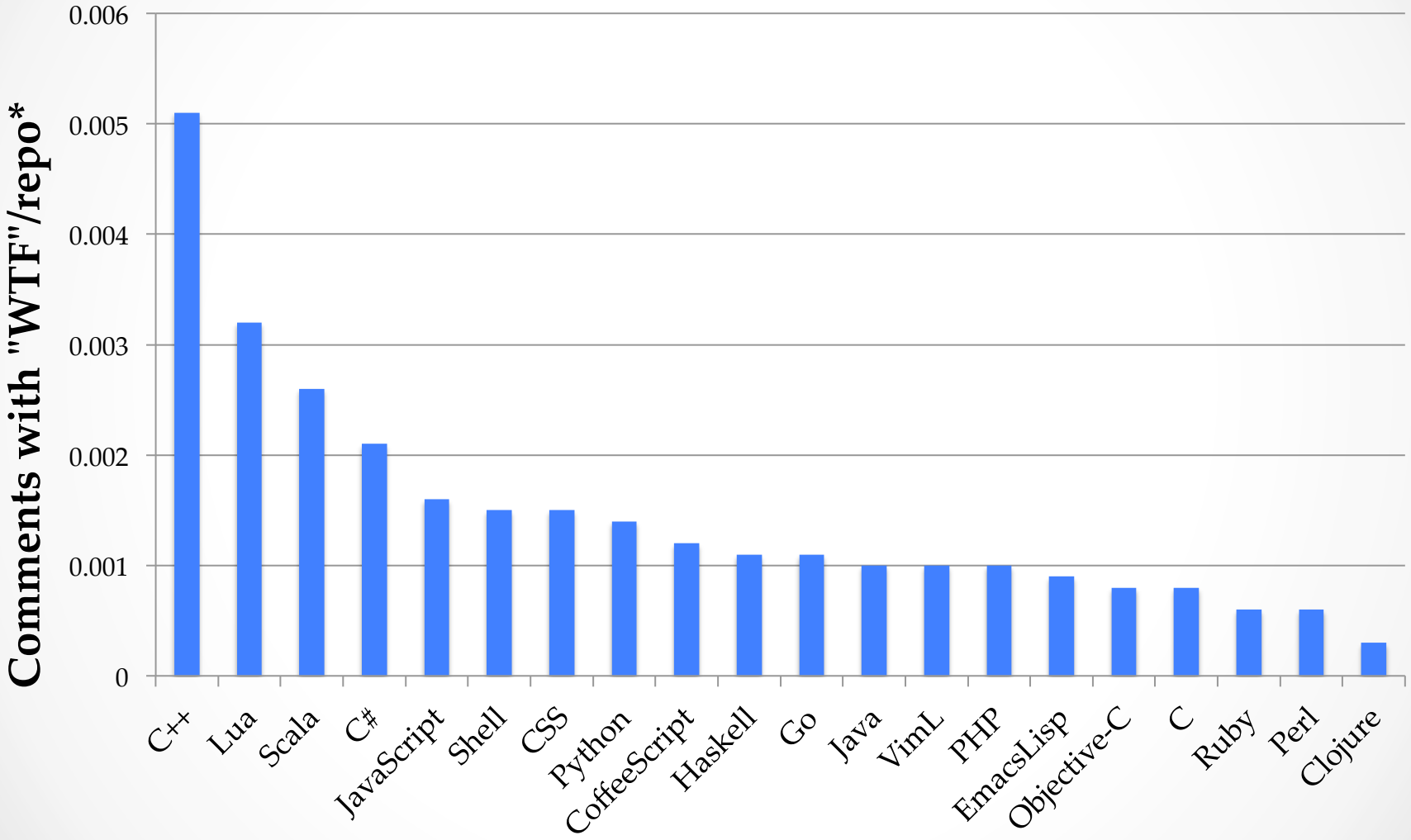
CSE 6305
Programming Languages and Systems

Programming languages suck.

wtf?



*all of these numbers are pretty meaningless

Source: Phil Johnson, IT World, 25/9/13

a little js…

```
$ jsc
> [] + []

> [] + {}
[object Object]
> {} + []
0
> {} + {}
NaN
```

Source: Gary Bernhardt CodeMash 2012

a little more js…

```
$ jsc
> Array(14)
,,,,,,,,,,,,,,
> Array(14).join("foo")
foofoofoofoofoofoofoofoofoofoofoofoofoo
> Array(14).join("foo" + 1)
foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1foo1
> Array(14).join("foo" - 1) + "Batman!"
NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN Batman!
```

Source: Gary Bernhardt CodeMash 2012

some php…

```php
<?php
$A = array();
$A[0] = 5;

echo "A[0]: $A[0]";
?>
```

```
A[0]: 5
```

some php…

```php
<?php
$A = array();
$A[0] = 5;
$C = $A;
$C[0] = 10;

echo "A[0]: $A[0]";
?>
```

```
A[0]: 5
```

In PHP, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable. This means, for instance, that after assigning one variable's value to another, changing one of those variables will have no effect on the other.

some php…

```php
<?php
$A = array();
$A[0] = 5;
$b = &$A[0];
$C = $A;
$C[0] = 10;

echo "A[0]: $A[0]";
?>
```

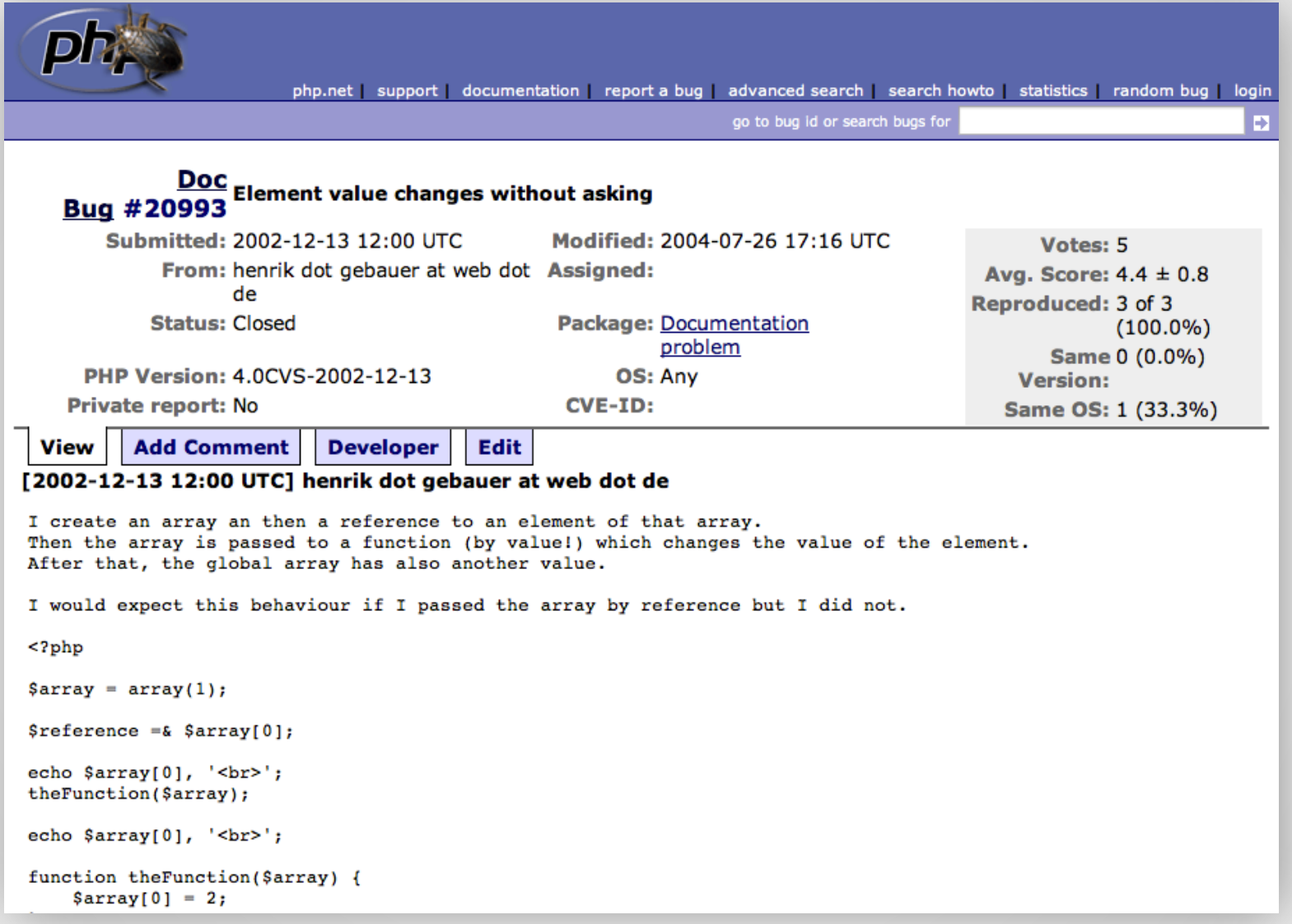
```
A[0]: 10
```

some php…

```php
<?php
$A = array();
$A[0] = 5;
$b = &$A[0];
$C = $A;
unset($b);
$C[0] = 10;

echo "A[0]: $A[0]";
?>
```

```
A[0]: 5
```



php.net | support | documentation | report a bug | advanced search | search howto | statistics | random bug | login

go to bug id or search bugs for

Doc Bug #20993 Element value changes without asking

Submitted: 2002-12-13 12:00 UTC
From: henrik dot gebauer at web dot de
Status: Closed
PHP Version: 4.0CVS-2002-12-13
Private report: No

Modified: 2004-07-26 17:16 UTC
Assigned:
Package: Documentation problem
OS: Any
CVE-ID:

Votes: 5
Avg. Score: 4.4 ± 0.8
Reproduced: 3 of 3 (100.0%)
Same Version: 0 (0.0%)
Same OS: 1 (33.3%)

View | Add Comment | Developer | Edit

[2002-12-13 12:00 UTC] henrik dot gebauer at web dot de

```
I create an array an then a reference to an element of that array.
Then the array is passed to a function (by value!) which changes the value of the element.
After that, the global array has also another value.

I would expect this behaviour if I passed the array by reference but I did not.

<?php

$array = array(1);

$reference =& $array[0];

echo $array[0], '<br>';
theFunction($array);

echo $array[0], '<br>';

function theFunction($array) {
    $array[0] = 2;
```

[2002-12-18 03:25 UTC] msopacua@php.net

We have discussed this issue and it will put a considerable slowdown on php's performance, to fix this properly.

Therefore this behavior will be documented.

Copy-on-Write in the PHP Language

Akihiko Tozawa Michiaki Tatsubori
Tamiya Onodera
IBM Research, Tokyo Research Laboratory
atozawa@jp.ibm.com,
mich@acm.org,tonodera@jp.ibm.com

Yasuhiko Minamide
Department of Computer Science
University of Tsukuba
minamide@cs.tsukuba.ac.jp

Abstract

PHP is a popular language for server-side applications. In PHP, assignment to variables copies the assigned values, according to its so-called *copy-on-assignment* semantics. In contrast, a typical PHP implementation uses a *copy-on-write* scheme to reduce the copy overhead by delaying copies as much as possible. This leads us to ask if the semantics and implementation of PHP coincide, and actually this is not the case in the presence of sharings within values. In this paper, we describe the copy-on-assignment semantics with three possible strategies to copy values containing sharings. The current PHP implementation has inconsistencies with these semantics, caused by its naïve use of copy-on-write. We fix this problem by the novel *mostly copy-on-write* scheme, making the copy-on-write implementations faithful to the semantics. We prove that our copy-on-write implementations are correct, using bisimulation with the copy-on-assignment semantics.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Design, Languages

1. Introduction

Assume that we want to maintain some data locally. This local data is mutable, but any change to it should not affect the global, master data. So, we may want to create and maintain a copy of the master data. However such copying is often costly. In addition, the copied data may not be modified after all, in which case the cost of copy is wasted. This kind of situation leads us to consider the *copy-on-write* technique.

Copy-on-write is a classic optimization technique, based on the idea of delaying the copy until there is a write to the data. The name of the technique stems from the copy of the original data being forced by the time of the write. One example of copy-on-write is found in the UNIX *fork*, where the process-local memory corresponds to the local data, which should be copied from the address space of the original process to the space of the new process by the fork operation. In modern UNIX systems, this copy is usually delayed by copy-on-write.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

Another example is found in the PHP language, a popular scripting language for server-side Web applications. Here is an example with PHP's associative arrays.

```
$r["box"] = "gizmo";
$l = $r;          // assignment from $r to $l
$l["box"] = "gremlin";
echo $r["box"];   // prints out gizmo
```

The change of `$l` at Line 3, following the assignment `$l = $r`, only has local effects on `$l` which cannot be seen from `$r`. The behavior or semantics in PHP is called *copy-on-assignment*, since the value of `$r` seems to be copied before it is passed to `$l`. We can consider the copy-on-write technique to implement this behavior. Indeed, the by far dominant PHP runtime, called the Zend runtime[1], employs copy-on-write and delays the above copy until the write at Line 3.

For readers in the functional or declarative languages community, the semantics of PHP arrays may first sound like a familiar one, e.g., PHP arrays are similar to *functional arrays*. However their similarity becomes less clear as we learn how we can *share* values in PHP. In PHP, we have the reference assignment statement, `=&`, with which we can declare a sharing between two variables. Such a sharing breaks the locality of mutation. For example, the write to `$y` is visible from `$x` in the following program.

```
$x[0] = "shares me";
$y =& $x;                 // creates sharing
$y[0] = "shared you";
echo $x[0];               // shared you
```

Now, our question is as follows. The copy-on-write is considered as a runtime optimization technique to reduce useless copies. Then, does the use of copy-on-write preserve the equivalence to the original semantics, in which we did not delay copying? This equivalence might be trivial without a sharing mechanism as above, but is not clear when we turn to PHP. In PHP, we can even share a location *inside a value*. This is where the problem gets extremely difficult.

```
$r["box"] = "gizmo";
$x =& $r["box"];     // creates sharing inside $r
$l = $r;             // copies $r
$l["box"] = "gremlin";
echo $r["box"];      // what should it be ?
```

The result of this program should reflect how exactly PHP copies arrays when they contain sharings. Our discussion will start from clarifying such PHP's copy semantics.

In this paper, we investigate the semantics and implementation of PHP focusing on the copy-on-write technique and its problems. Our contributions in this paper are as follows.

[1] Available at `http://www.php.net`.

PHP

"I don't know how to stop it, there was never any intent to write a programming language [...] **I have absolutely no idea how to write a programming language**, I just kept adding the next logical step on the way."

- Rasmus Lerdorf, creator of PHP

some Python…

C

```
int NUM = 111181111;

int is_prime(int n) {
   int
   for(
     if

     }
   }
   return 1;
 }
```

Python

```
NUM = 111181111

def is_prime(n):

    return True
```

C (GCC, O0): **0.624s**

Python: **15.609s**

25x difference!

some java…

```
List<int> il = new ArrayList<int>();
```

Design and Implementation of Generics for the .NET Common Language Runtime

Andrew Kennedy Don Syme

Microsoft Research, Cambridge, U.K.
{akenn,dsyme}@microsoft.com

Abstract

The Microsoft .NET Common Language Runtime provides a shared type system, intermediate language and dynamic execution environment for the implementation and inter-operation of multiple source languages. In this paper we extend it with direct support for parametric polymorphism (also known as generics), describing the design through examples written in an extended version of the C# programming language, and explaining aspects of implementation by reference to a prototype extension to the runtime.

Our design is very expressive, supporting parameterized types, polymorphic static, instance and virtual methods, "F-bounded" type parameters, instantiation at pointer and value types, polymorphic recursion, and exact run-time types. The implementation takes advantage of the dynamic nature of the runtime, performing just-in-time type specialization, representation-based code sharing and novel techniques for efficient creation and use of run-time types.

Early performance results are encouraging and suggest that programmers will not need to pay an overhead for using generics, achieving performance almost matching hand-specialized code.

1 Introduction

Parametric polymorphism is a well-established programming language feature whose advantages over dynamic approaches to generic programming are well-understood: safety (more bugs caught at compile time), expressivity (more invariants expressed in type signatures), clarity (fewer explicit conversions between data types), and efficiency (no need for run-time type checks).

Recently there has been a shift away from the traditional compile, link and run model of programming towards a more dynamic approach in which the division between compile-time and run-time becomes blurred. The two most significant examples of this trend are the Java Virtual Machine [11] and, more recently, the Common Language Runtime (CLR for short) introduced by Microsoft in its .NET initiative [1].

The CLR has the ambitious aim of providing a *common* type system and intermediate language for executing programs written in a variety of languages, and for facilitating inter-operability between those languages. It relieves compiler writers of the burden of dealing with low-level machine-specific details, and relieves programmers of the burden of describing the data marshalling (typically through an interface definition language, or IDL) that is necessary for language interoperation.

This paper describes the design and implementation of support for parametric polymorphism in the CLR. In its initial release, the CLR has no support for polymorphism, an omission shared by the JVM. Of course, it is always possible to "compile away" polymorphism by translation, as has been demonstrated in a number of extensions to Java [14, 4, 6, 13, 2, 16] that require no change to the JVM, and in compilers for polymorphic languages that target the JVM or CLR (MLj [3], Haskell, Eiffel, Mercury). However, such systems inevitably suffer drawbacks of some kind, whether through source language restrictions (disallowing primitive type instantiations to enable a simple erasure-based translation, as in GJ and NextGen), unusual semantics (as in the "raw type" casting semantics of GJ), the absence of separate compilation (monomorphizing the whole program, as in MLj), complicated compilation strategies (as in NextGen), or performance penalties (for primitive type instantiations in PolyJ and Pizza). The lesson in each case appears to be that if the virtual machine does not support polymorphism, the end result will suffer.

The system of polymorphism we have chosen to support is very expressive, and, in particular, supports instantiations at reference and value types, in conjunction with exact runtime types. These together mean that the semantics of polymorphism in a language such as C# can be very much "as expected", and can be explained as a relatively modest and orthogonal extension to existing features. We have found the virtual machine layer an appropriate place to support this functionality, precisely because it is very difficult to implement this combination of features by compilation alone. To our knowledge, no previous attempt has been made to design and implement such a mechanism as part of the infrastructure provided by a virtual machine. Furthermore, ours is the first design and implementation of polymorphism to combine exact run-time types, dynamic linking, shared code and code specialization for non-uniform instantiations, whether in a virtual machine or not.

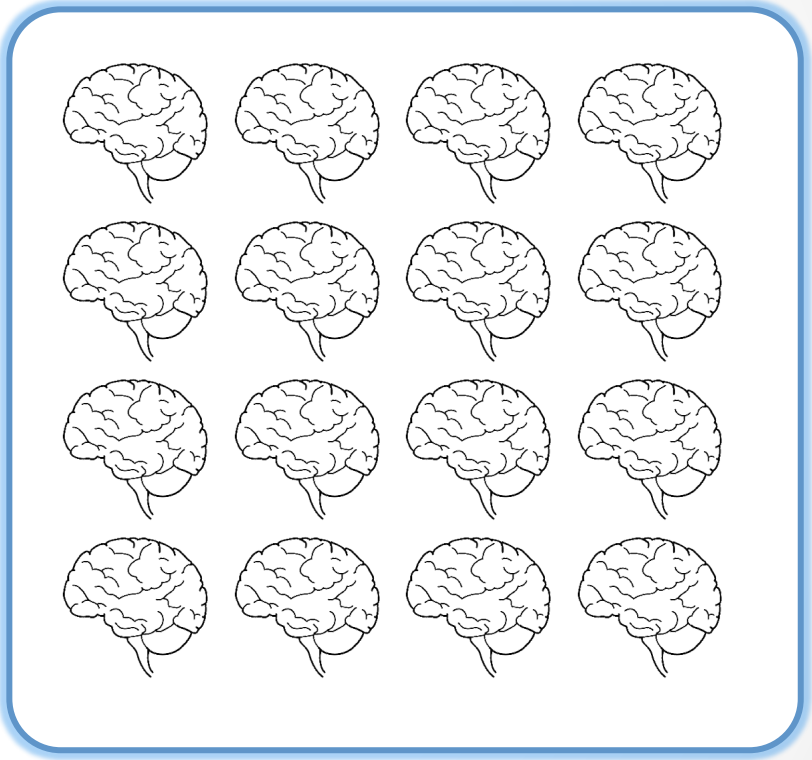
1.1 What is the CLR?

The .NET Common Language Runtime consists of a typed, stack-based intermediate language (IL), an Execution Engine (EE) which executes IL and provides a variety of runtime services (storage management, debugging, profiling, security, etc.), and a set of shared libraries (.NET Frameworks). The CLR has been successfully targeted by a variety of source languages, including C#, Visual Basic, C++, Eiffel, Cobol, Standard ML, Mercury, Scheme and Haskell.

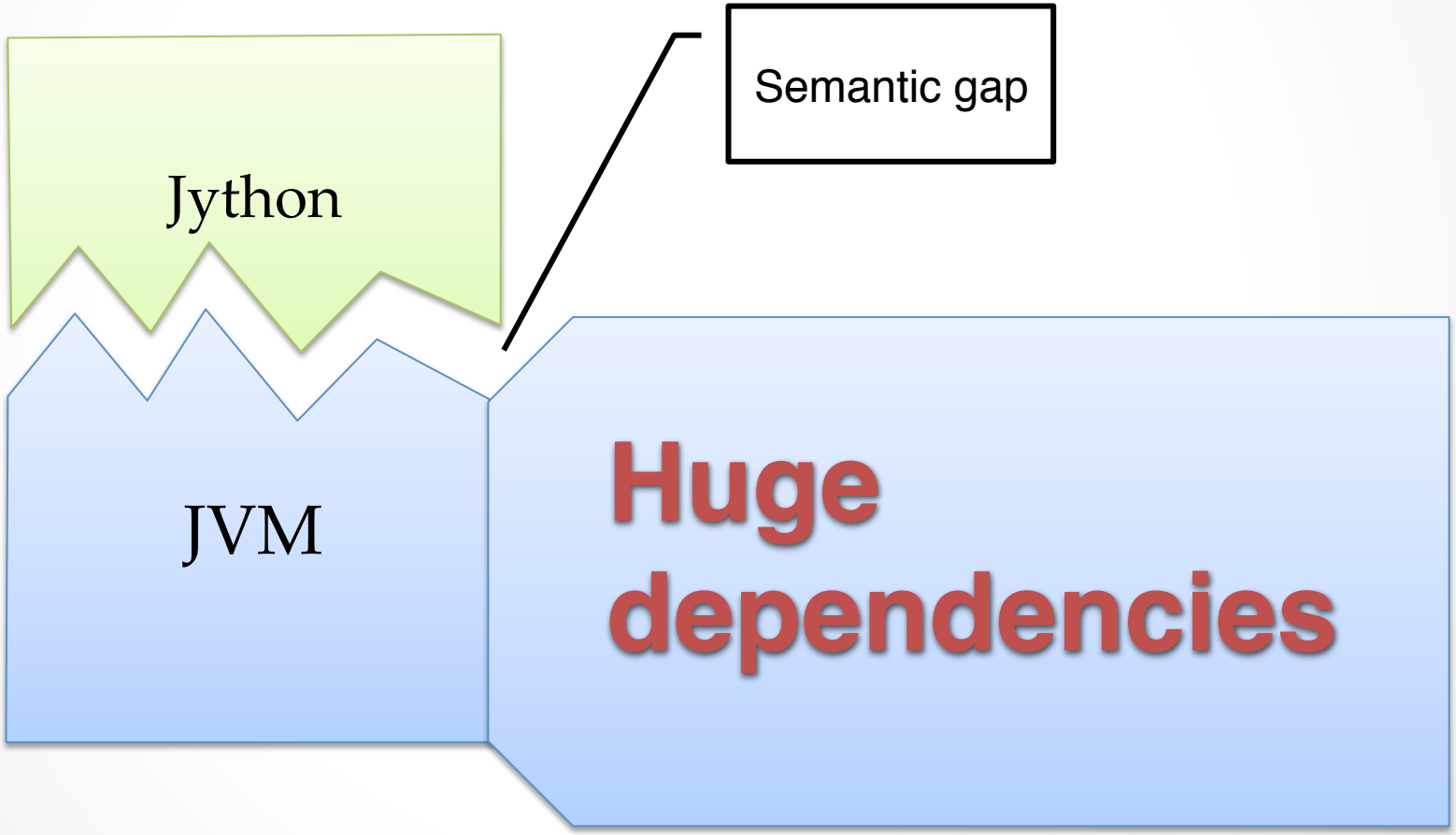
The primary focus of the CLR is object-oriented languages, and this is reflected in the type system, the core of which is the defini-

Existing Approaches

Approach 1: Build VM From Scratch



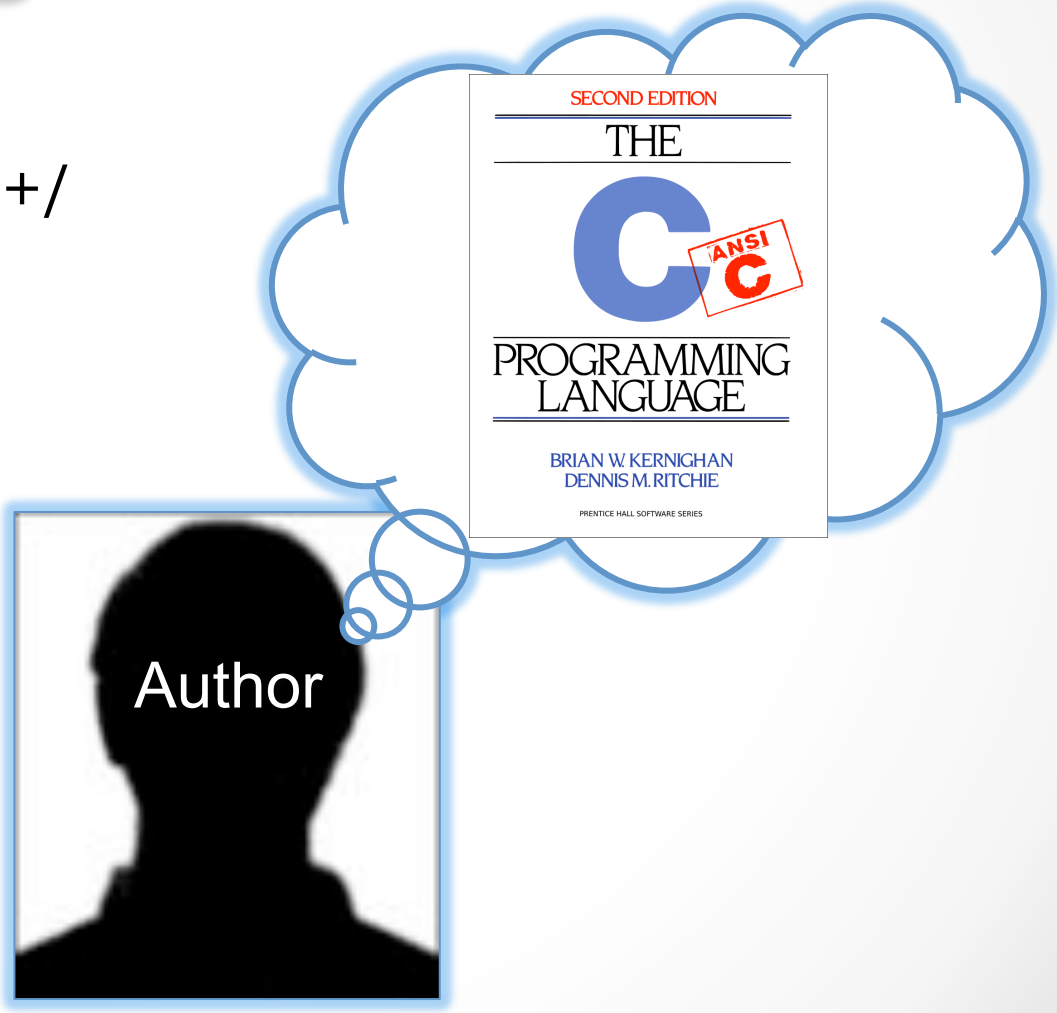
Approach 2: Existing VM



Approach 3: Existing Frameworks

LLVM

- Excellent for C/C++/ ObjectiveC
- Not for managed languages



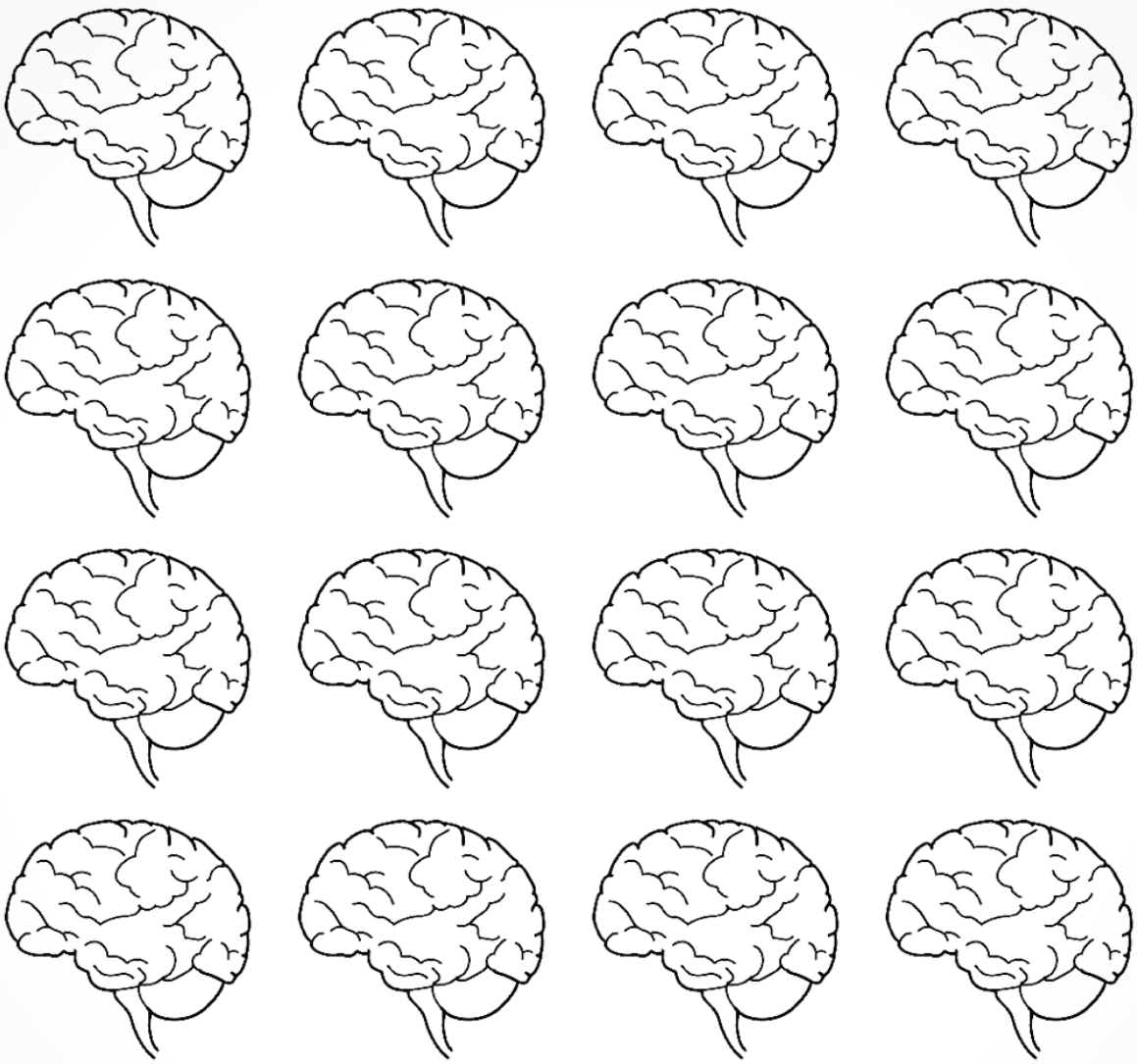
What is so difficult?

Three Foundamental Concerns

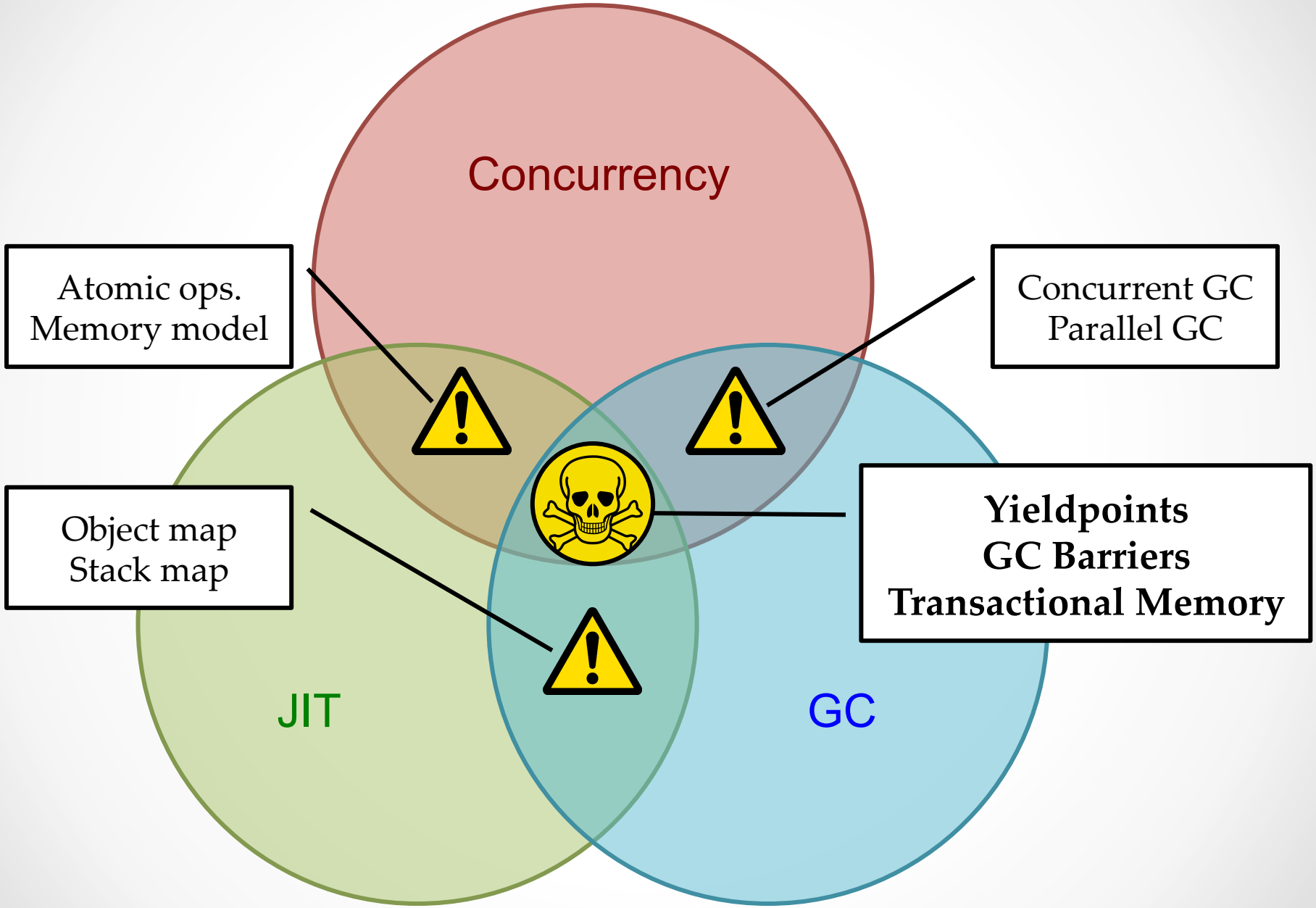
Just-in-time Compiling

Concurrency

Garbage Collection



JIT + concurrency + GC


Concurrency
Atomic ops.
Memory model
Concurrent GC
Parallel GC
Object map
Stack map
Yieldpoints
GC Barriers
Transactional Memory
JIT
GC

Language Implementations

Lang. Impl.	Concurrency	Execution	GC
HotSpot	✔ Threads	✔ JIT	✔ Generational
CPython	✖ GIL	✖ Interpreter	✖ Naïve RC
PyPy	✖ STM (exp.)	✔ JIT	✔ MMTK-like
Jython	✔ JVM Threads	✖ JVM JIT	✔ JVM GC
Unladen Swallow	✖ GIL	✖ Template JIT	✖ Naïve RC
Ruby	✖ GIL	✖ Interpreter	✖ MS
PHP	-	✖ Interpreter	✖ Naïve RC
HHVM	-	✔ JIT	✖ Naïve RC
Lua	✖ No Threads	✖ Interpreter	✖ MS
LuaJit	✖ No Threads	✔ JIT	✖ MS