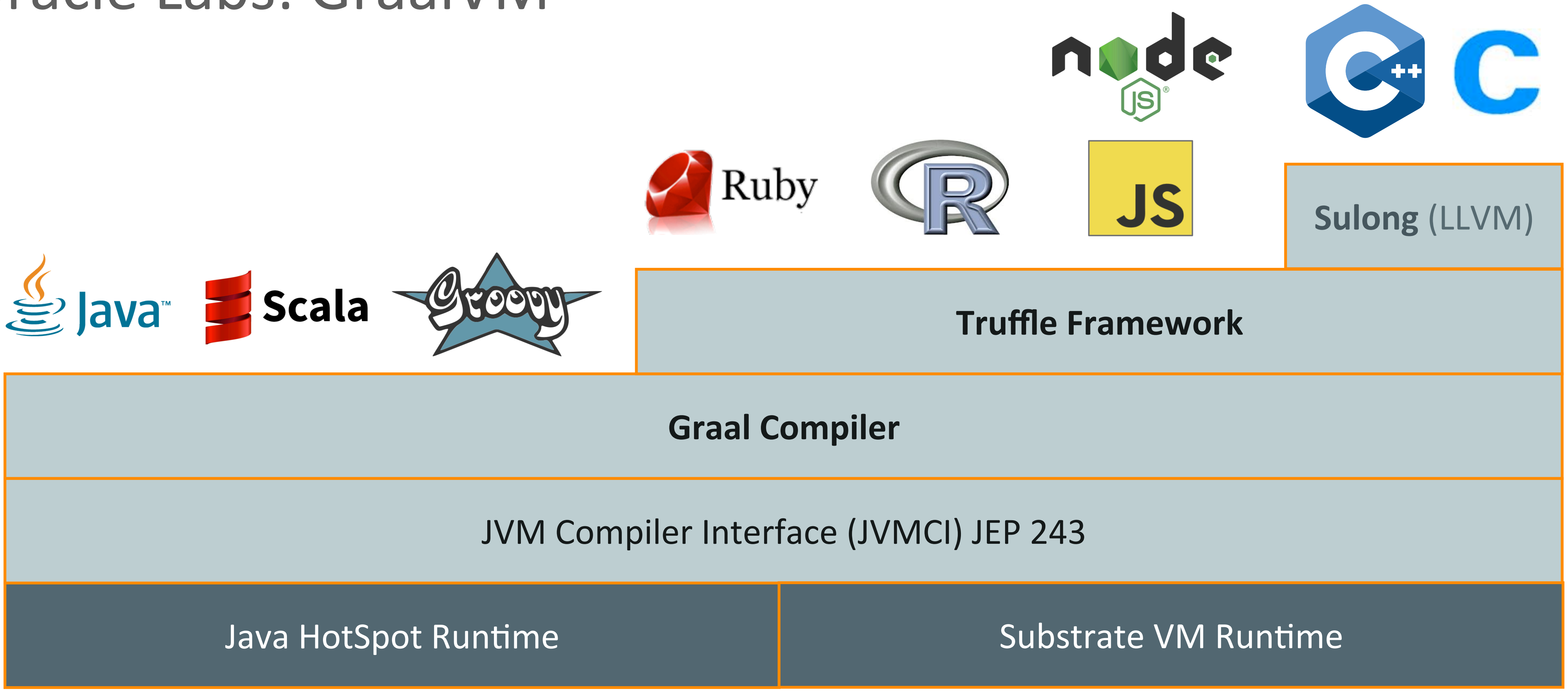
Polyglot Native: Scala, Kotlin, and Other JVM-Based Languages with Instant Startup and low Footprint

Vojin Jovanovic
VM Research Group, Oracle Labs

Github: @vjovanov
Twitter: @vojjov

Safe Harbor Statement

Oracle Labs: GraalVM



Sulong (LLVM)

Truffle Framework

Graal Compiler

JVM Compiler Interface (JVMCI) JEP 243

Java HotSpot Runtime

Substrate VM Runtime

Managed Runtimes: Slow Startup and High Footprint

- Slow startup and high footprint
  - Class loading
  - Bytecode interpretation or baseline compilation
  - Just-in-time compilation

Program	Time	Instructions
“Hello, World!” in C	0.005s	154,127
“Hello, World!” in Java	0.109s	162,673,275
“Hello, World!” in JS on the JVM	1.268s	3,272,118,178

Complexity of JNI

JVM

Process Boundary

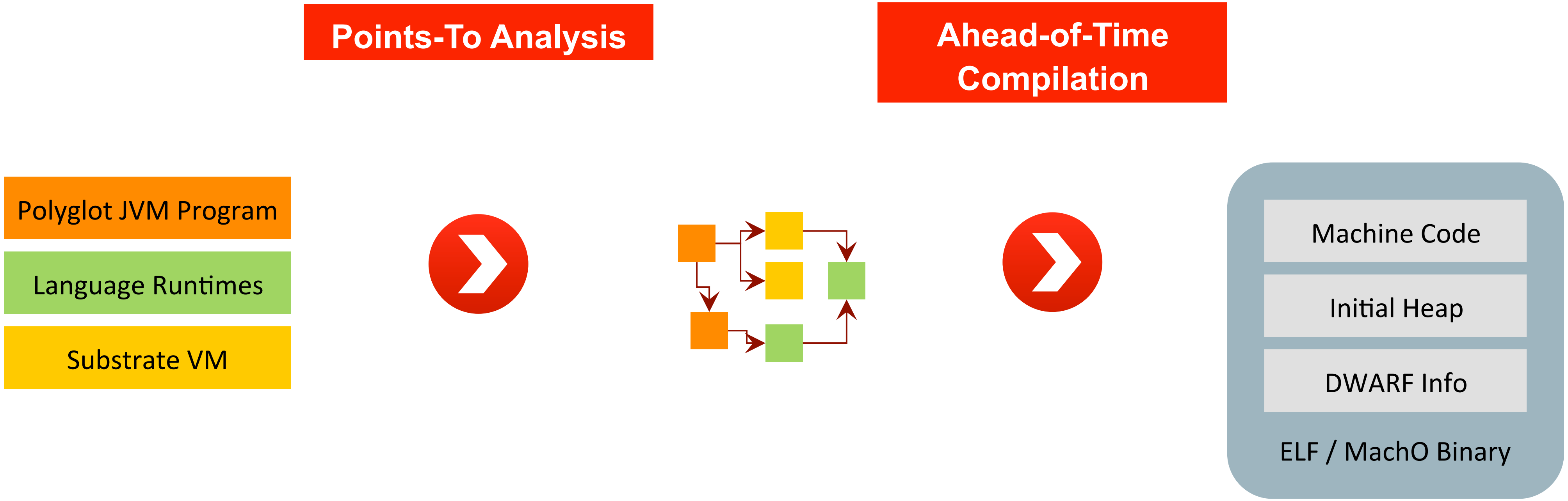
Native Code

Java AOT (JEP 235) and Project Panama

- Java AOT addresses startup performance on the JVM
  - Ahead-of-time compiles Java bytecode
  - Code is JIT compiled with profiling information
  - Ahead-of-time compiled code contains additional instructions for profiling

- Project Panama
  - Makes it faster and easier to call native code from Java
  - Allows changes the Java object layout for native code
  - JVM still can not be easily embedded in native projects

Polyglot Native: Execution Model



Points-To Analysis

Ahead-of-Time Compilation

Polyglot JVM Program

Language Runtimes

Substrate VM

Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

All classes from
the user application,
all language runtimes,
and Substrate VM

Reachable methods,
fields, and classes

Application running
without dependency on JDK
and without Java class loading

The Substrate VM is …

… an **embeddable** VM

for, and written in, a **subset of Java**

optimized to **execute Truffle** languages

ahead-of-time compiled using Graal

integrating with **native development tools.**

Substrate VM Building Blocks

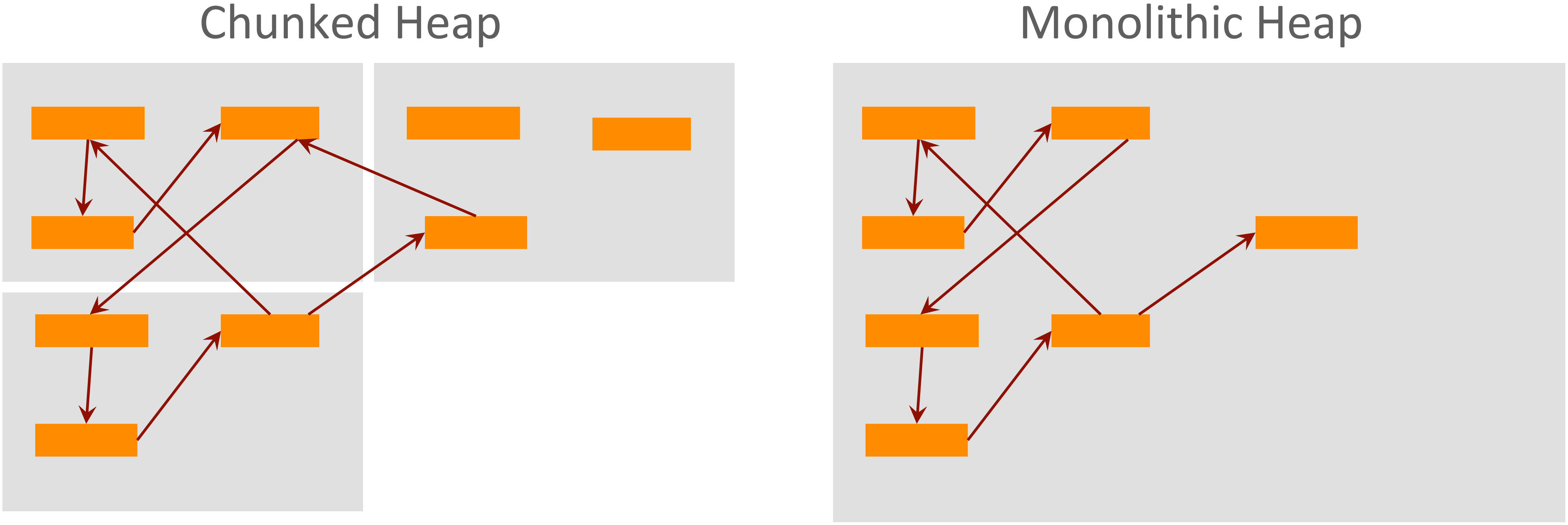
- Reduced runtime system, all written in Java
  - Stack walking, exception handling, garbage collector, deoptimization
  - Graal for ahead-of-time compilation and dynamic compilation

- Points-to analysis
  - Closed-world assumption: no dynamic class loading, no reflection
  - Using Graal for bytecode parsing
  - Fixed-point iteration: propagate type states through methods

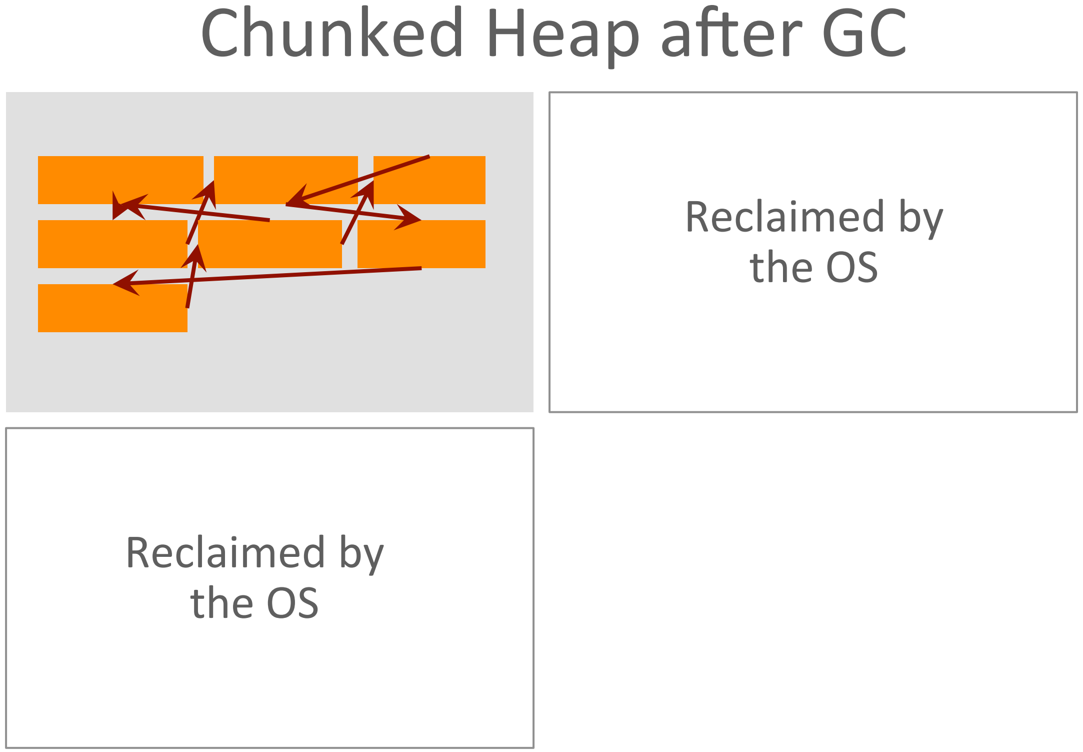
- SystemJava for integration with C code
  - Machine-word sized value, represented as Java interface, but unboxed by compiler
  - Import of C functions and C structs to Java

- Substitutions for JDK methods that use unsupported features
  - JNI code replaced with SystemJava code that directly calls to C library

Chunked Heap for Low Footprint

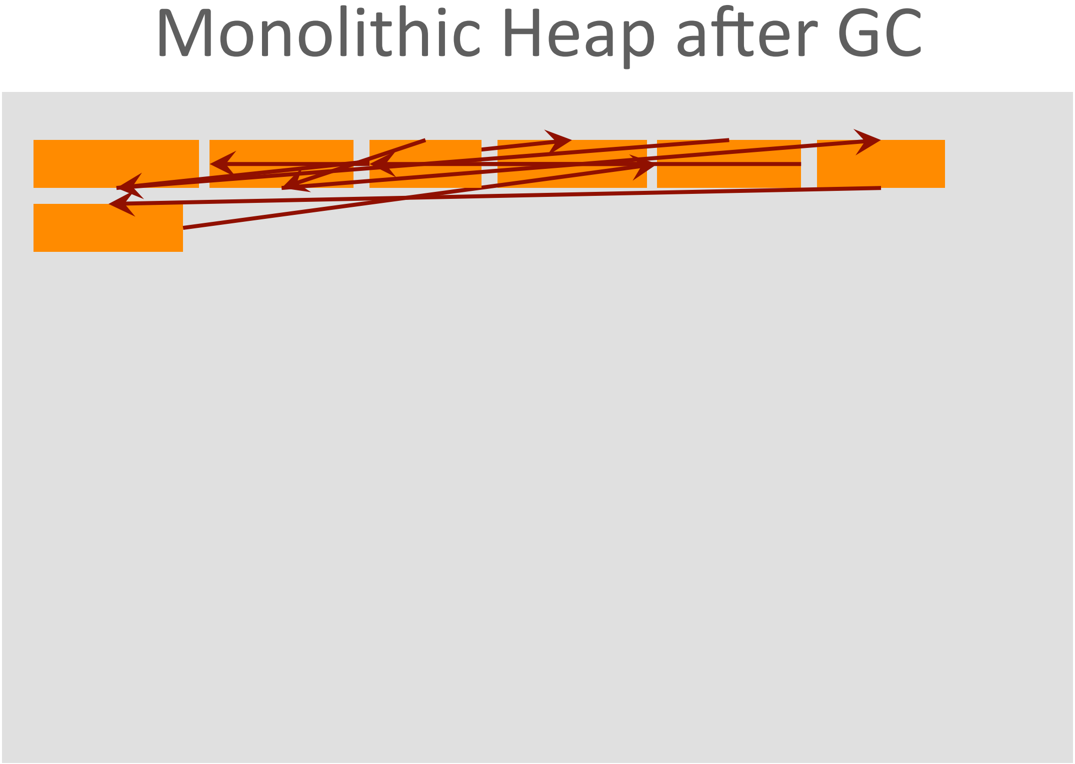


Chunked Heap for Low Footprint


Chunked Heap after GC

Reclaimed by
the OS

Reclaimed by
the OS


Monolithic Heap after GC

Polyglot Native: Execution Model

Points-To Analysis

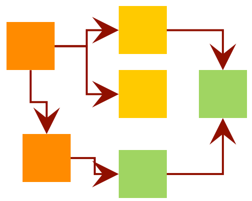
Ahead-of-Time Compilation

Polyglot JVM Program

Language Runtimes

Substrate VM

All Java classes from
Truffle language
(or any application),
JDK, and Substrate VM



Reachable methods,
fields, and classes

Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

Application running
without dependency on JDK
and without Java class loading

Points-To Analysis

Graal as a Static Analysis Framework

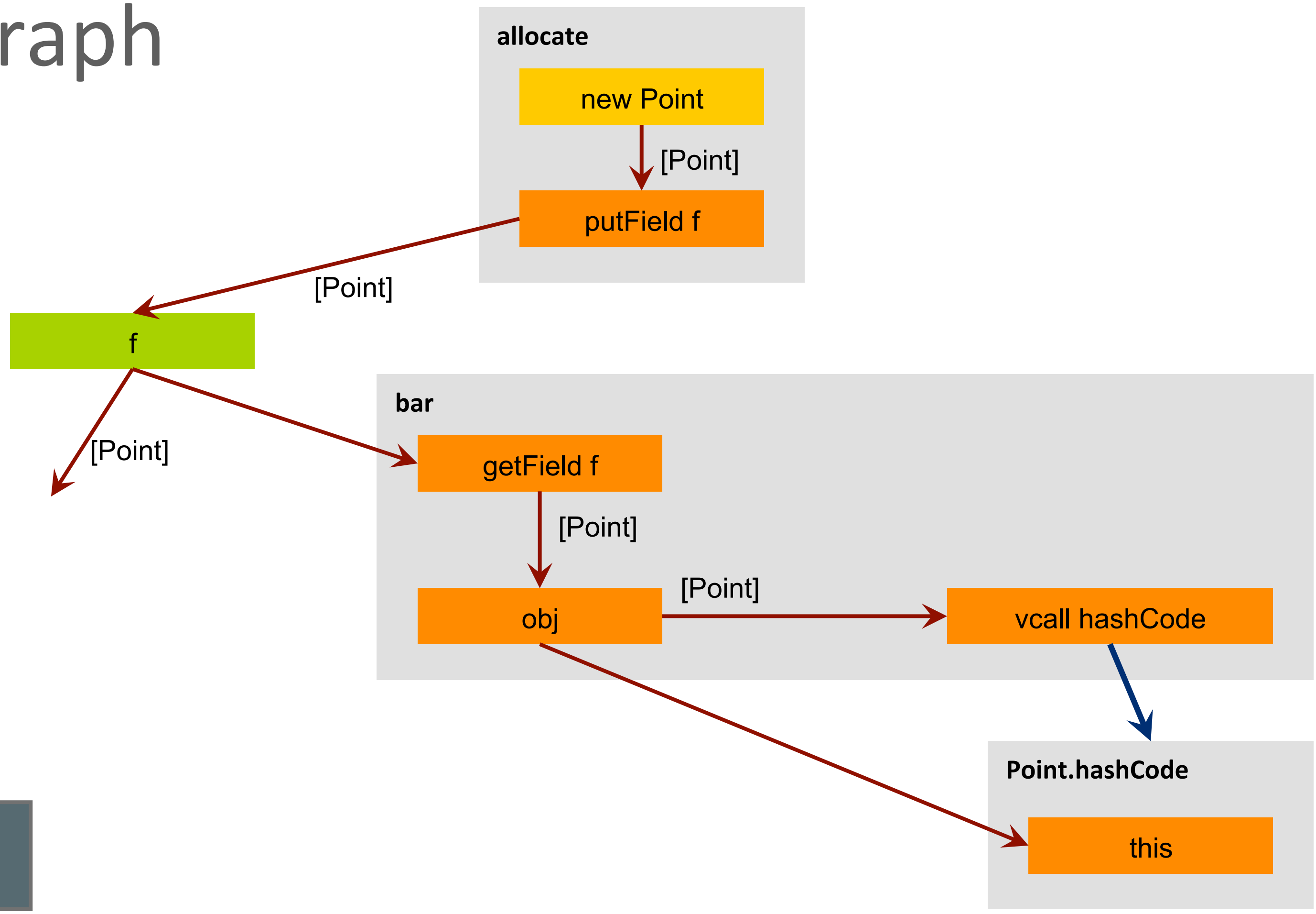
- Graal and the hosting Java VM provide
  - Class loading (parse the class file)
  - Access the bytecodes of a method
  - Access to the Java type hierarchy, type checks
  - Build a high-level IR graph in SSA form
  - Linking / method resolution of method calls

- Static points-to analysis and compilation use same intermediate representation
  - Simplifies applying the analysis results for optimizations

- Goals of points-to analysis
  - Identify all methods reachable from a root method
  - Identify the types assigned to each field
  - Identify all instantiated types
- Fixed point iteration of type flows: Types are propagated from sources (allocations) to usages

Example Type Flow Graph

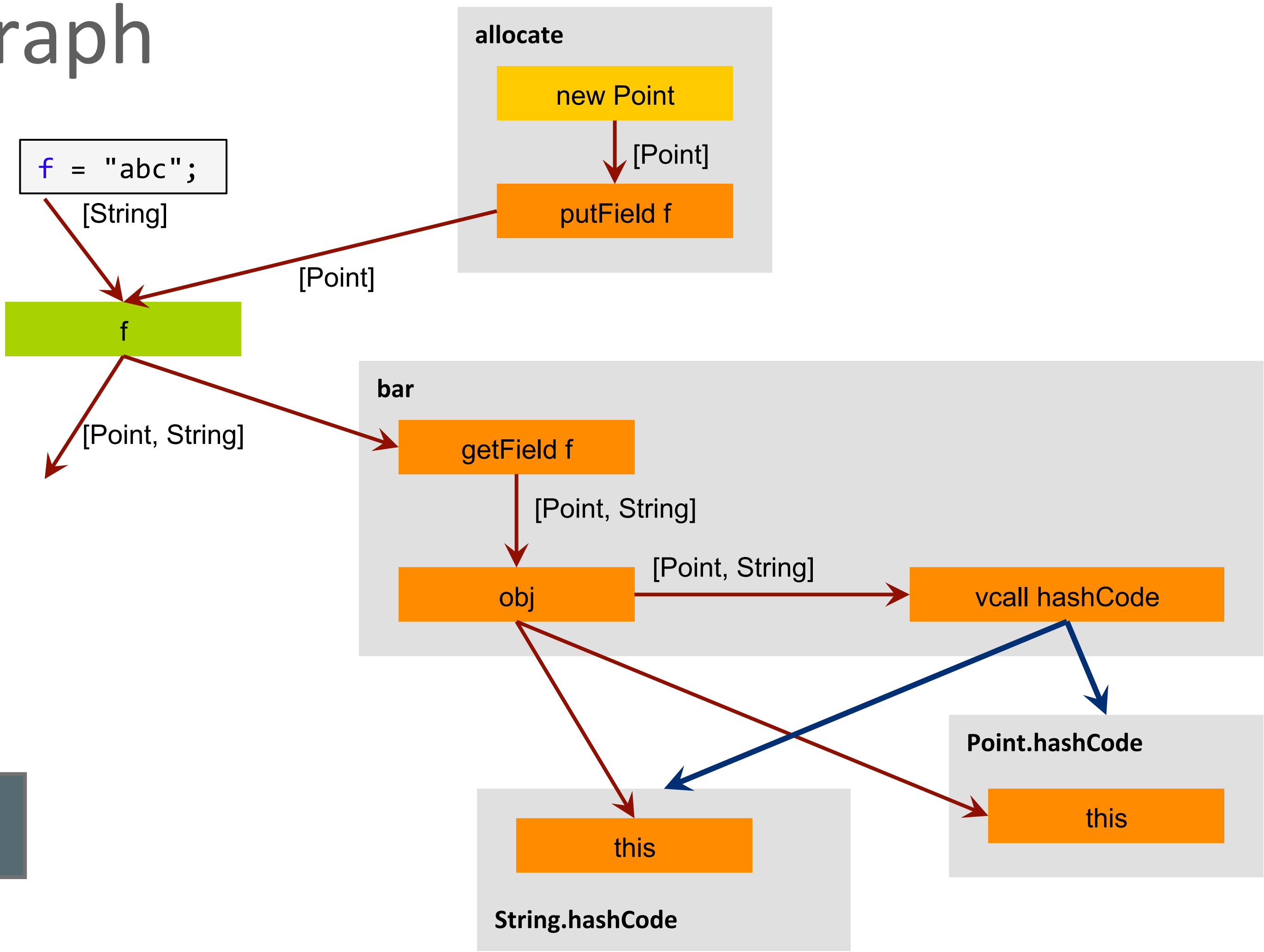
```
Object f;

void foo() {
  allocate();
  bar();
}

Object allocate() {
  f = new Point()
}

int bar() {
  return f.hashCode();
}
```



Analysis is context insensitive:
One type state per field

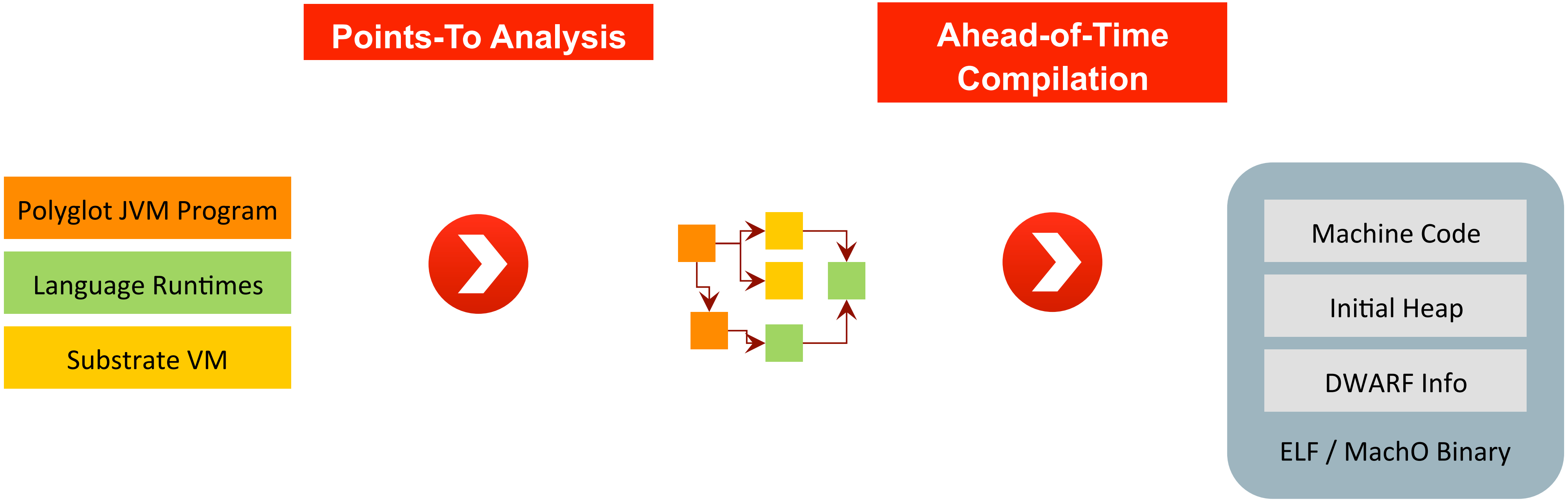
Example Type Flow Graph

```
Object f;

void foo() {
  allocate();
  bar();
}

Object allocate() {
  f = new Point()
}

int bar() {
  return f.hashCode();
}
```



Analysis is context insensitive:
One type state per field

Polyglot Native: Execution Model



Points-To Analysis

Ahead-of-Time Compilation

Polyglot JVM Program

Language Runtimes

Substrate VM

Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

All Java classes from
Truffle language
(or any application),
JDK, and Substrate VM

Reachable methods,
fields, and classes

Application running
without dependency on JDK
and without Java class loading

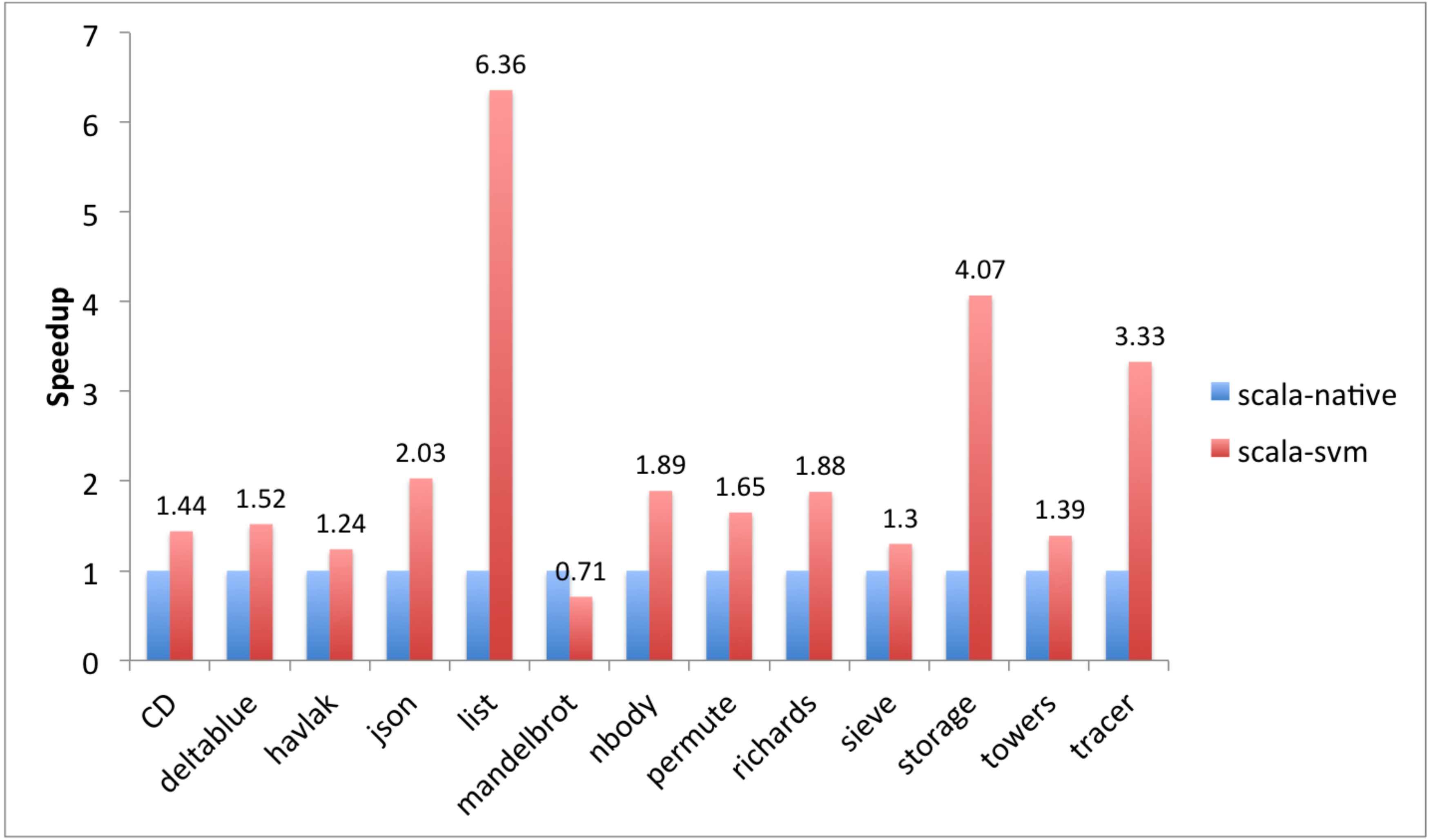
Polyglot Native Startup

- Slow startup and high footprint
  - Class loading
  - Bytecode interpretation
  - Just-in-time compilation
  - Monolithic heap

Program	Time	Instructions
“Hello, World!” in C	0.005s	154,127
“Hello, World!” in PN	0.006s	232,122
“Hello, World!” in JS with PN	0.028s	915,461

Demo: Instant Startup of JVM Code

Benchmarks: Scala Native vs SystemJava



SystemJava

SystemJava

- Legacy C code integration
  - Need a convenient way to access preexisting C functions and structures

- Legacy Java code integration
  - Leverage preexisting Java libraries
  - Example: JDK class library

- Call Java from C code
  - Entry points into JVM code

SystemJava vs. JNI

- Java Native Interface (JNI)
  - Write custom C code to integrate existing C code with Java
  - C code knows about Java types
  - Java objects passed to C code using handles

- SystemJava
  - Write custom Java code to integrate existing C code with Java
  - Java code knows about C types

Word Type for Low-Level Memory Access

- Requirements
  - Support raw memory access and pointer arithmetic
  - No extension of the Java programming language
  - Pointer type modeled as a class to prevent mixing with, e.g., `long`
  - Transparent bit width (32 bit or 64 bit) in code using it

- Base interface `Word`
  - Looks like an object to the Java IDE, but is a primitive value at run time
  - Graal does the transformation

- Subclasses for type safety
  - `Pointer`: C equivalent `void*`
  - `Unsigned`: C equivalent `size_t`
  - `Signed`: C equivalent `ssize_t`

```
public static Unsigned strlen(CharPointer str) {
  Unsigned n = Word.zero();
  while (str.read(n) != 0) {
    n = n.add(1);
  }
  return n;
}
```

Java Annotations for C Interoperability

```
@CFunction static native int clock_gettime(int clock_id, timespec tp);
```

```
int clock_gettime(clockid_t __clock_id, struct timespec *__tp)
```

```
@CConstant static native int CLOCK_MONOTONIC();
```

```
#define CLOCK_MONOTONIC 1
```

```
@CStruct interface timespec extends PointerBase {
  @CField long tv_sec();
  @CField long tv_nsec();
}
```

```
struct timespec {
  __time_t tv_sec;
  __syscall_slong_t tv_nsec;
};
```

```
@CPointerTo(nameOfCType="int") interface CIntPointer extends PointerBase {
  int read();
  void write(int value);
}
```

```
int* pint;
```

```
@CPointerTo(CIntPointer.class) interface CIntPointerPointer ...
```

```
int** ppint;
```

```
@CContext(PosixDirectives.class)
```

```
#include <time.h>
```

```
@CLibrary("rt")
```

```
-lrt
```

Implementation of `System.nanoTime()` using SystemJava:

```
static long nanoTime() {
  timespec tp = StackValue.get(SizeOf.get(timespec.class));
  clock_gettime(CLOCK_MONOTONIC(), tp);
  return tp.tv_sec() * 1_000_000_000L + tp.tv_nsec();
}
```

SystemJava from JVM Languages

- SystemJava extracts semantics from bytecode
- For all static JVM languages:
  - Function calls are propagated
  - Java annotations are propagated to bytecode
- Possible extension to language idiomatic interface
  - In Scala *sizeOf[CCharPointer]* would be preferred to *SizeOf.get(CCharPointer.class)*

SystemJava	SystemScala	SystemKotlin
SizeOf.get(CCharPointer.class)	SizeOf.get(classOf[CCharPointer])	SizeOf.get(CCharPointer::class)
StackAlloc.get(…)	StackAlloc.get(…)	StackAlloc.get(…)
@CStruct interface AStruct {…}	@CStruct trait AStruct {…}	@CStruct interface AStruct {…}

Managed Objects in Native Code

- Managed objects are different than native objects
  - in layout, as every object has a header
  - memory location, they can, at any time, be moved by the garbage collector

- To avoid these issues, when passing objects to native code
  - use handles when native code only holds a reference
  - pin objects and ignore their header when native code reads the object

Demo: Sentiment Analysis of Tweets

Limitations

- Java reflection can not be fully supported
  - Dynamic class loading is not possible
- Currently not implemented
  - Reflective access to Java fields, methods, and types

Take Polyglot Native for a Spin

- Download GraalVM

  http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html

- The Tweet Sentiment Analysis Demo

  https://github.com/vjovanov/polyglot-native-demo

Acknowledgements

Oracle
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez

Oracle (continued)
Roland Schatz
Chris Seaton
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Lukas Stadler
Codrut Stancu
Jan Štola
Jaroslav Tulach
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Interns
Brian Belleville
Miguel Garcia
Shams Imam
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
David Piorkowski
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
Hyunjin Lee
David Leibs
Chris Thalinger
Till Westmann

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Huber
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Prof. Michael Franz
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Prof. Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Prof. Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Prof. Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Prof. Walter Binder
Sun Haiyang
Yudi Zheng

Integrated Cloud

Applications & Platform Services

ORACLE®