Sulong: Fast LLVM IR Execution on the JVM with Truffle and Graal

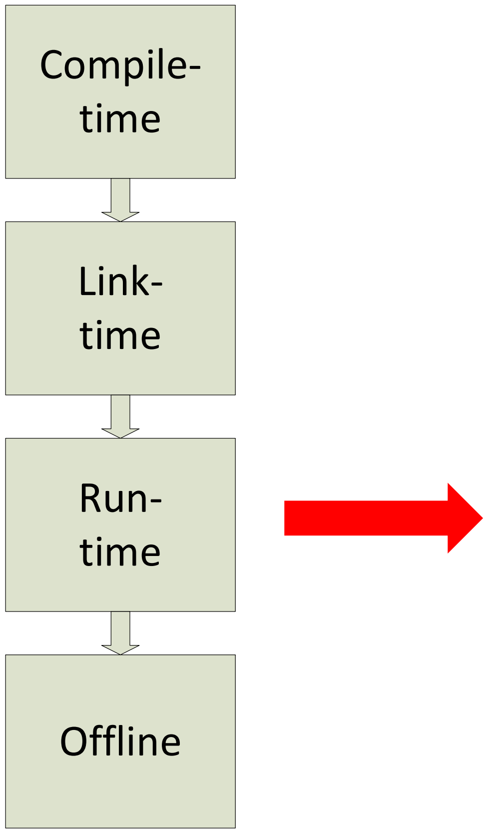
FOSDEM 2016: 31. January 2016

Manuel Rigger @RiggerManuel

PhD student at Johannes Kepler University Linz, Austria

JOHANNES KEPLER UNIVERSITY LINZ

Why Do We Need A(nother) LLVM IR Interpreter?



Speculative optimizations?

Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004.

Motivation Example: Function Pointer Calls

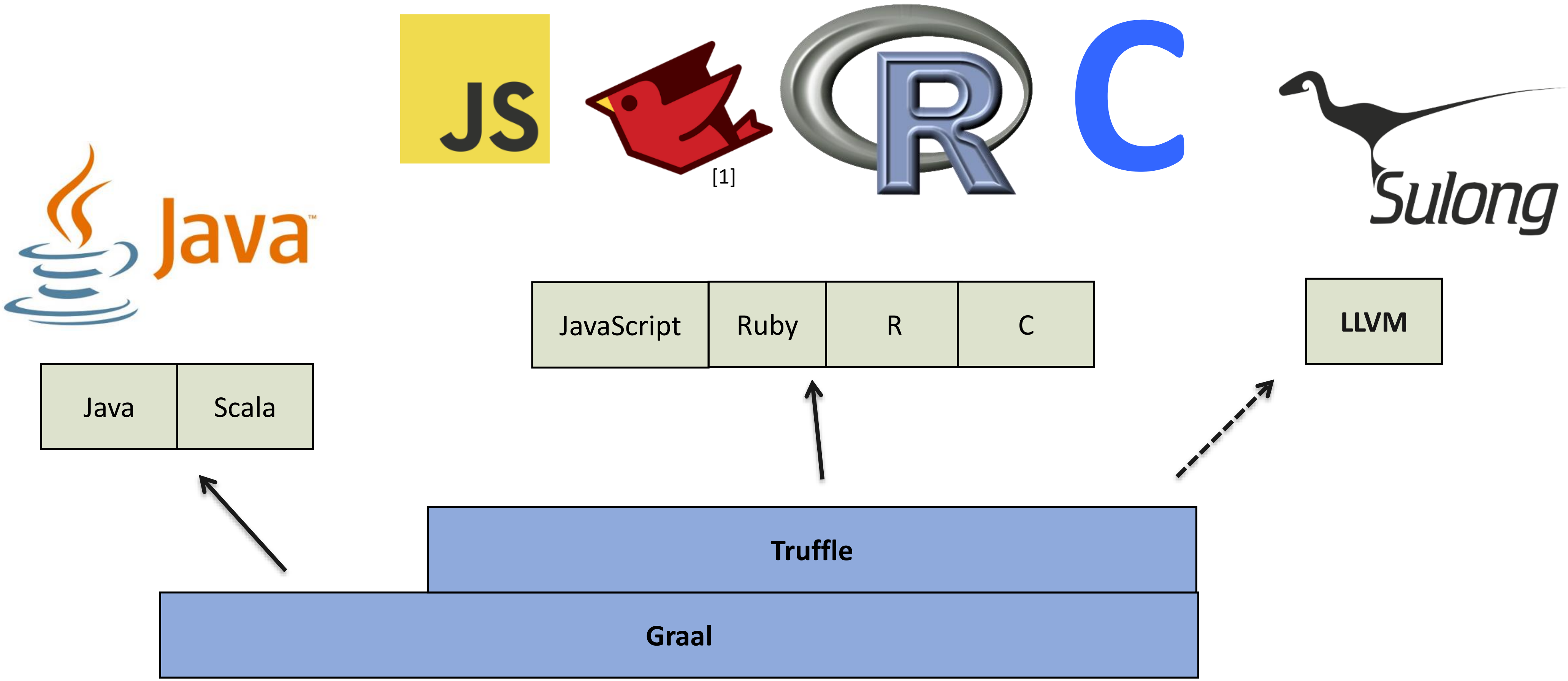
```
int ascending(int a, int b){ return a - b; }

int descending(int a, int b){ return b - a; }
```

```
void bubble_sort(int *numbers, int count, (*compare)(int a, int b)) {
    for (int  i = 0; i < count; i++) {
        for (int j = 0; j < count - 1; j++) {
            if (compare(numbers[j], numbers[j+1]) > 0) {
                swap(&numbers[j], &numbers[j+1]);
            }
        }
    }
}
```

Sulong

- LLVM IR interpreter running on the JVM
  - With dynamic optimizations and JIT compilation!
- Available under a BSD 3-Clause License
  - https://github.com/graalvm/sulong
  - Contributions are welcome!
- Sulong: Chinese for velocisaurus
  - 速: fast, rapid
  - 龙: dragon

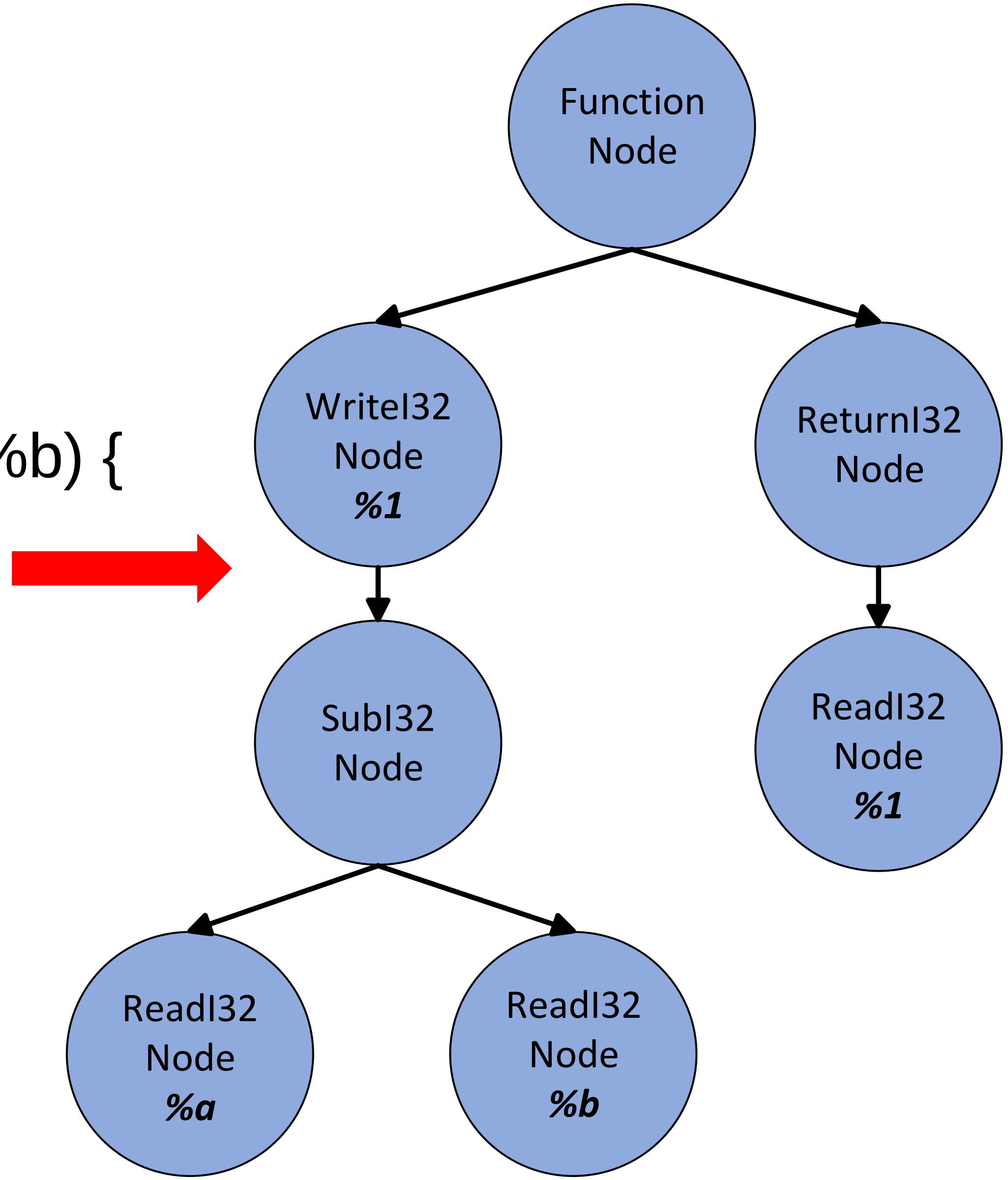
Truffle Multi-Language Environment



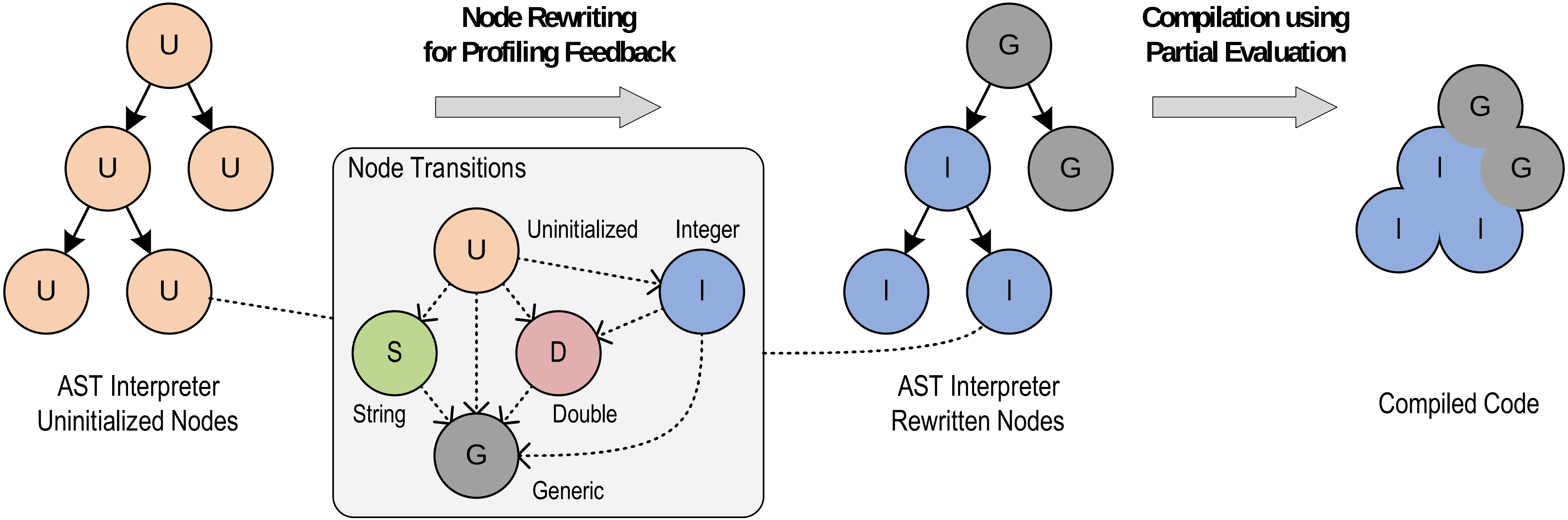
http://www.github.com/graalvm

AST Interpreter

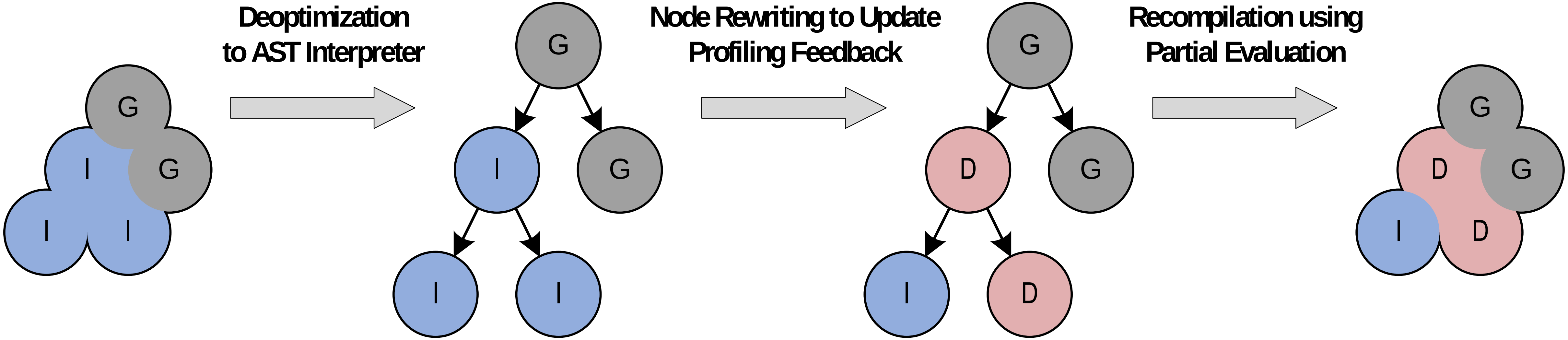
```
define i32 @ascending(i32 %a, i32 %b) {
  %1 = sub nsw i32 %a, %b
  ret i32 %1
}
```



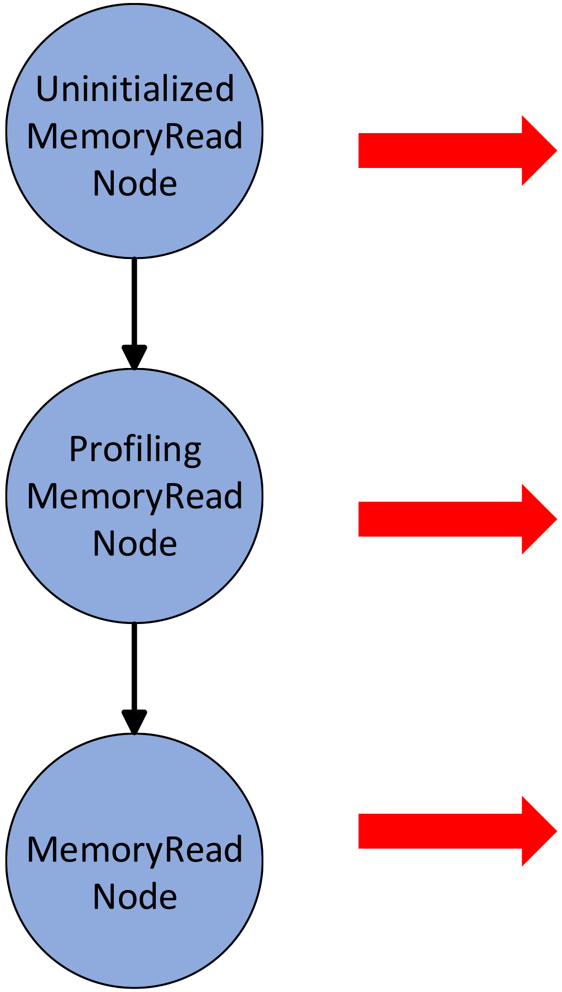
Truffle and Graal



Truffle and Graal



Example 1: Value Profiling



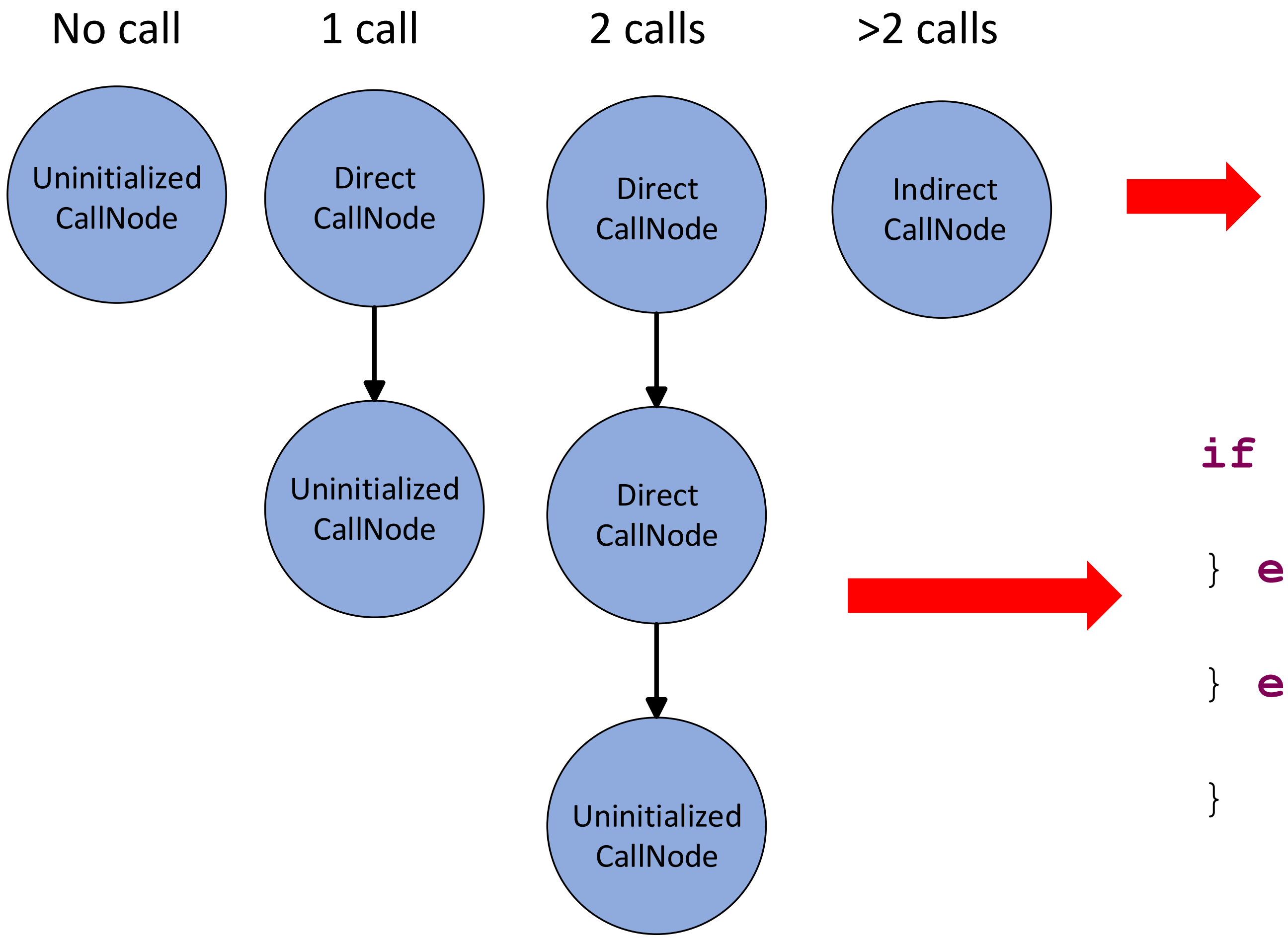

```
expectedValue = memory[ptr];
deoptimizeAndRewrite();
```

```
currentValue = memory[ptr];
if (currentValue ==  expectedValue) {
    return expectedValue;
} else {
    deoptimizeAndRewrite();
}
```

```
return memory[ptr];
```

Example 2: Polymorphic Function Pointer Inline Caches

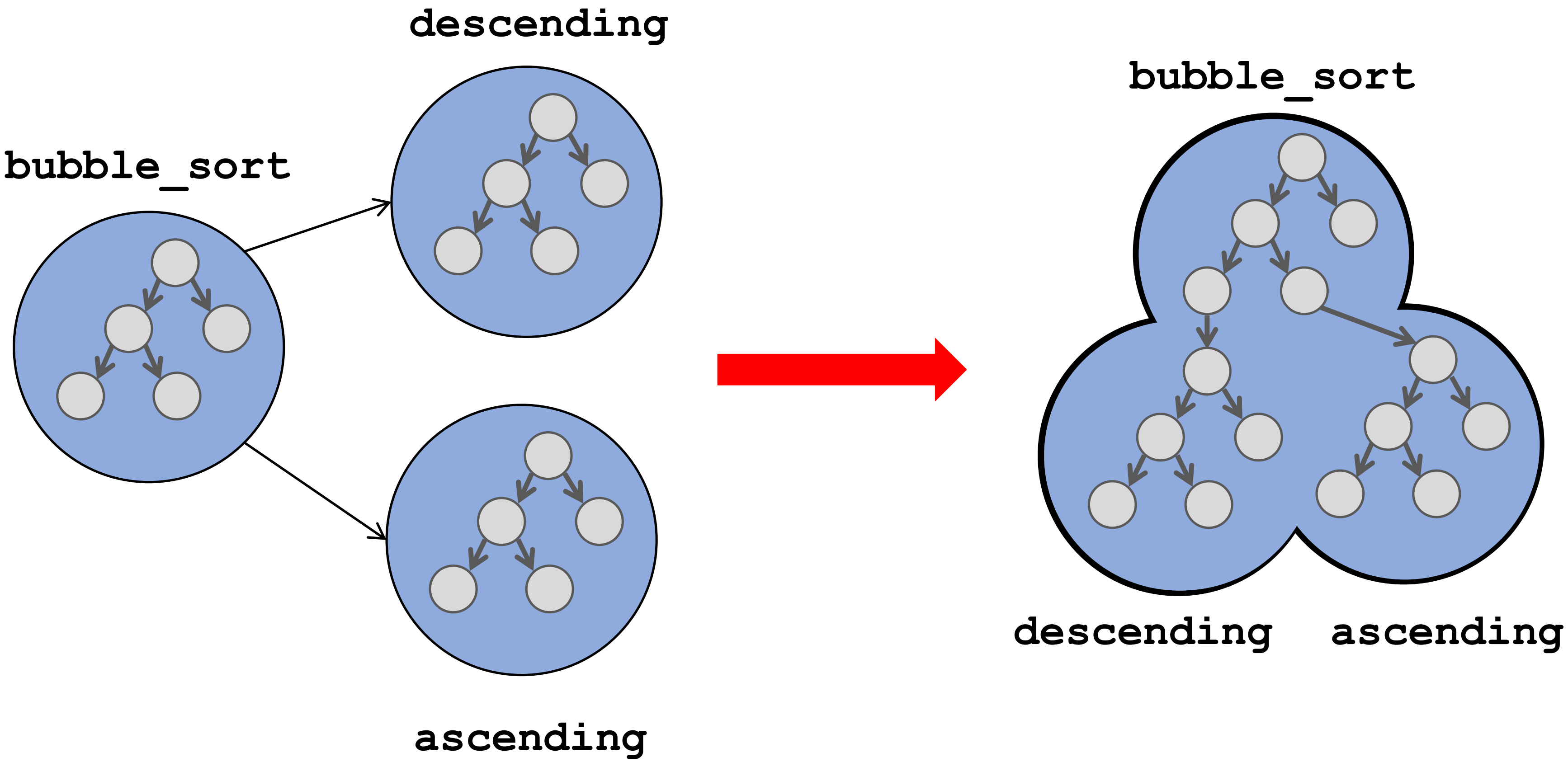
```
compare(a, b) > 0
```




```
compare(a, b);
```

```
if (compare == &ascending) {
        return ascending(a, b);
} else if (compare == &descending) {
        return descending(a, b);
} else {
        deoptimizeAndRewrite();
}
```

Function Pointer Call Inlining



Demo

Getting started

- Download the mx build tool

```
$ hg clone https://bitbucket.org/allr/mx
$ export PATH=$PWD/mx:$PATH
```

- Clone the repo and build the project

```
$ git clone https://github.com/graalvm/sulong
$ cd sulong
$ mx build
```

- Compile and run a program

```
$ mx su-clang -S -emit-llvm -o test.ll test.c
$ mx su-run test.ll
```

Developing with mx

- Generate Eclipse project files (also available for other IDEs)

```
$ mx eclipseinit
```

- Quality tools

```
$ mx checkstyle/findbugs/pylint/...
```

- run Sulong tests

```
$ mx su-tests
```

- Eclipse remote debugging (port 5005)

```
$ mx su-debug test.ll
```

Compilation

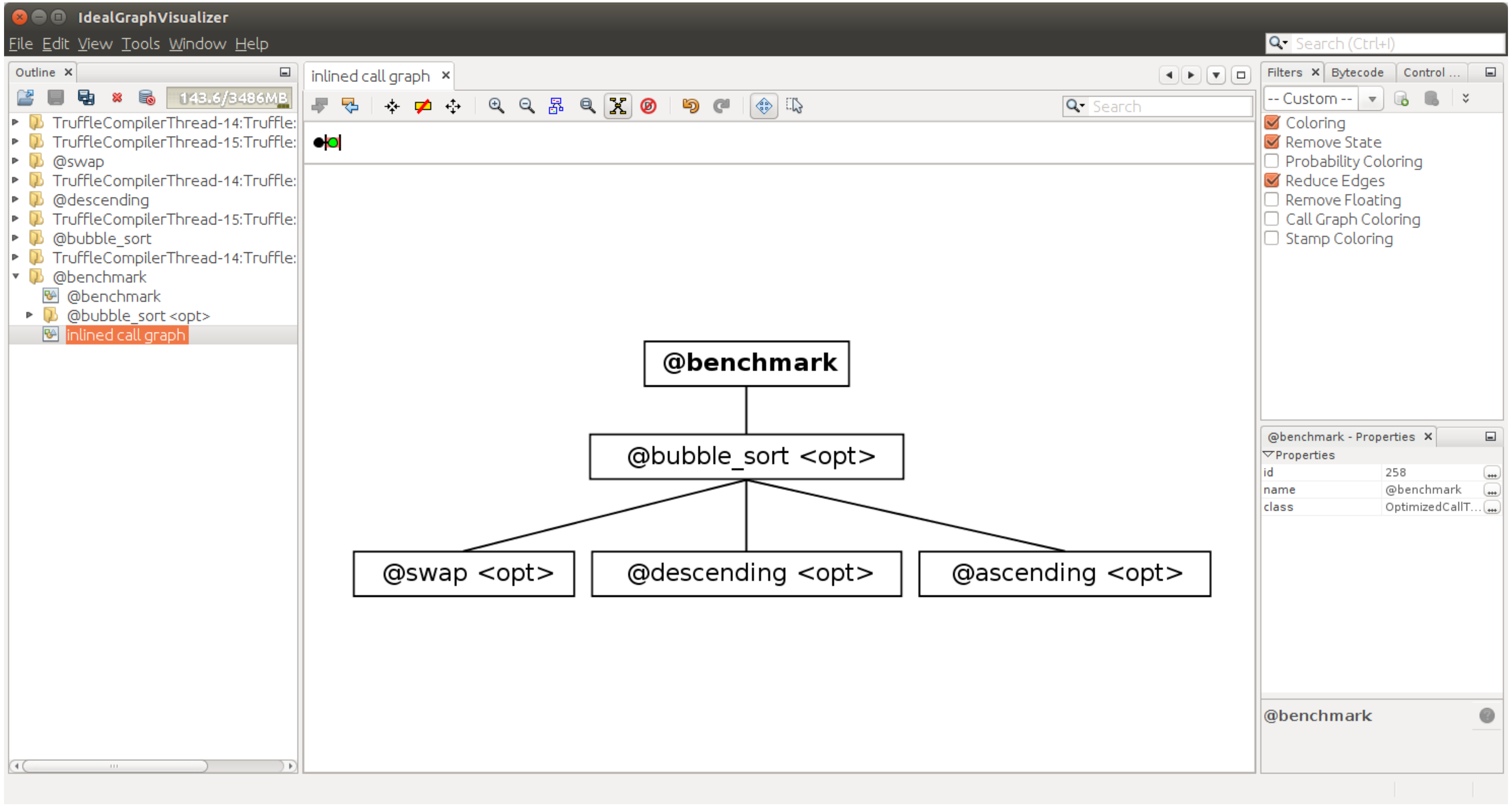
- Textual information about which LLVM functions are compiled

```
$ mx su-run test.ll -Dgraal.TraceTruffleCompilation=true
```

- View Truffle and Graal graphs

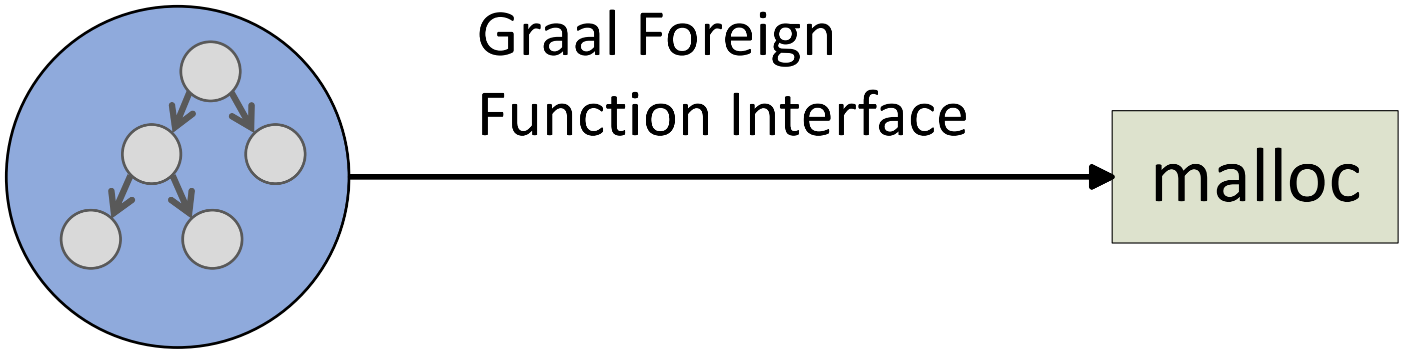
```
$ mx igv
$ mx su-run test.ll -Dgraal.Dump=Truffle
```

Example: Truffle Graph



Implementation of Memory

- Unmanaged mode
  - Heap allocation: by native standard libraries
  - Stack allocation: Java Unsafe API
- Graal Native Function Interface for library interoperability



Current State

- Performance: room for improvement on most benchmarks
- Completeness: mostly focused on C so far
  - Missing: longjmp/setjmp, inline assembly, full support of 80 bit floats
  - Can execute most of the gcc.c-torture/execute benchmarks

Outlook

- Low overhead security-related instrumentations

  → Graal is specialized to perform optimizations for operations like bounds or type checks

  - Memory safety via allocating on the Java heap
  - Tracking of integer overflows
- Full Truffle integration
  - Debugger with source code highlighting
  - Language interoperability

Q/A @RiggerManuel

- Thanks for listening!

Attributions