Sulong

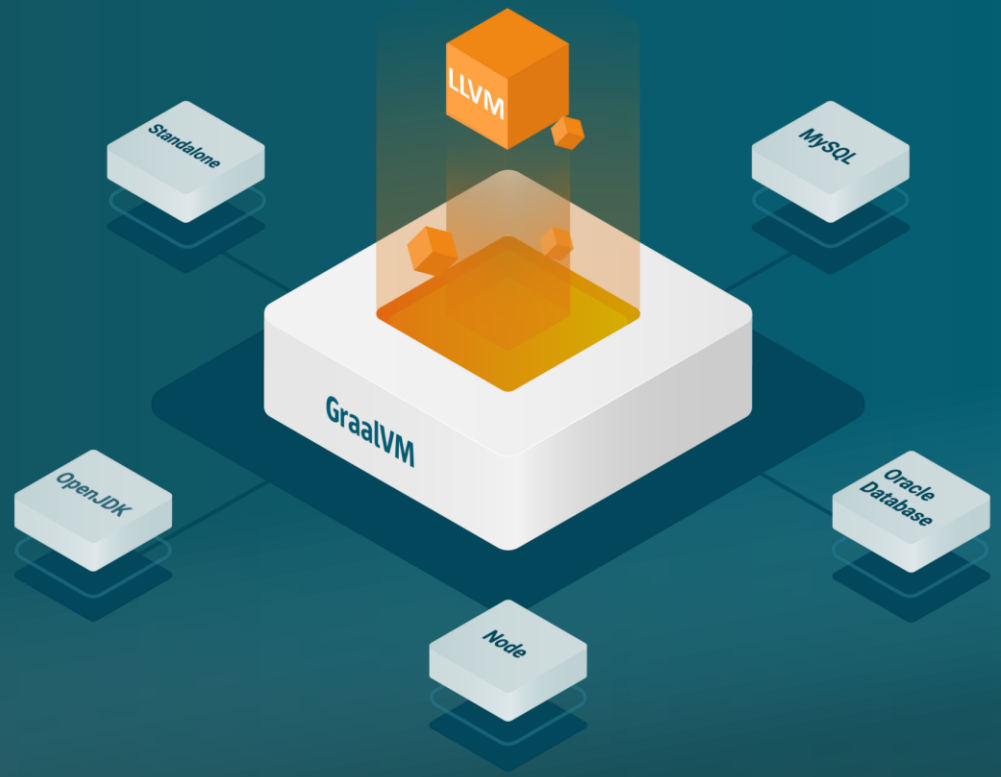
An experience report of using the "other end" of LLVM in GraalVM

Roland Schatz
Sulong Team Lead
@rschatz_at

Josef Eisl
Sulong Team
@zapstercc

GraalVM, Oracle Labs

April 9, 2019



ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

What is Sulong?

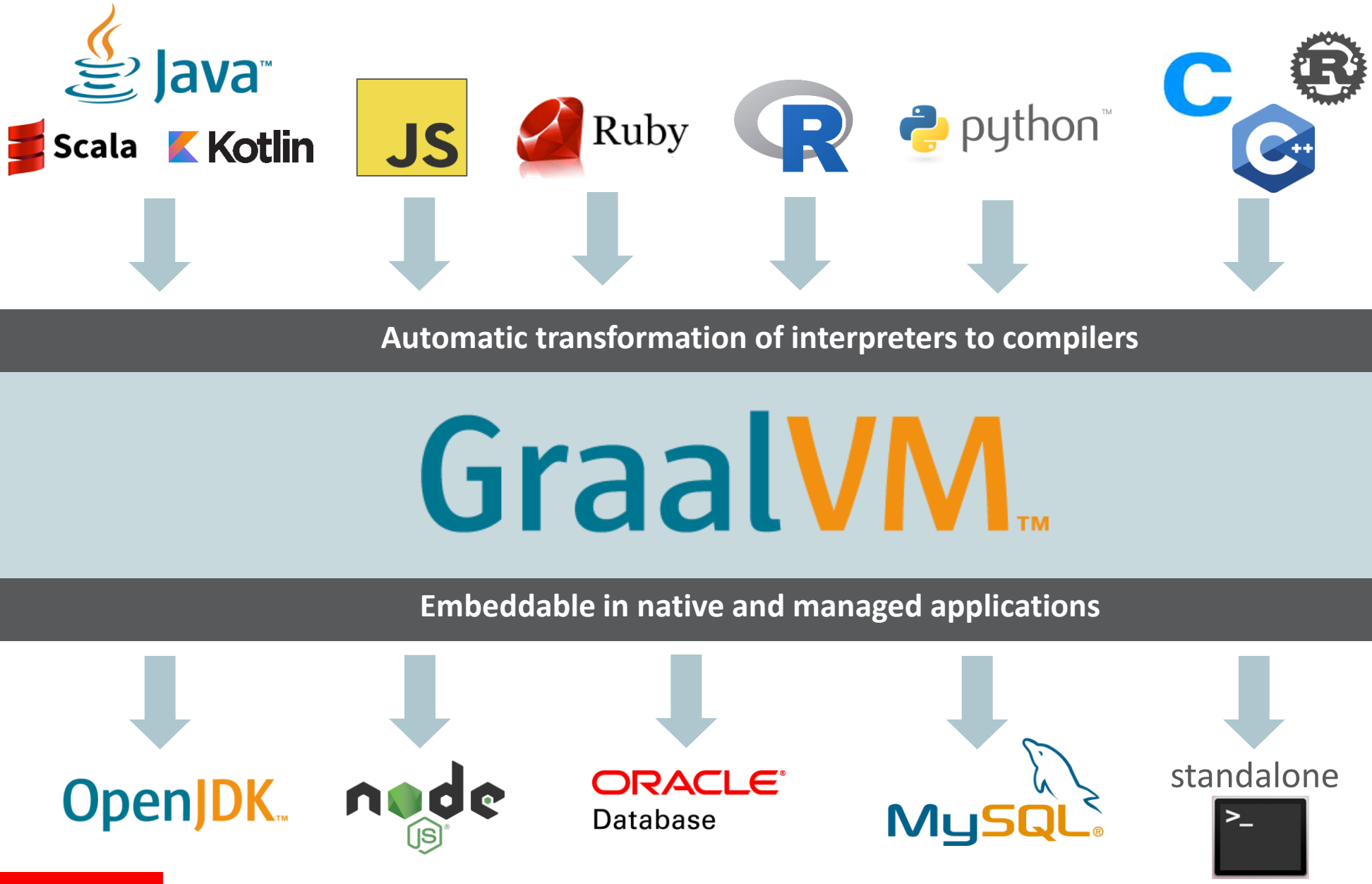
LLVM Bitcode execution engine

- Think: `lli`
- Interpretation and JIT-compilation

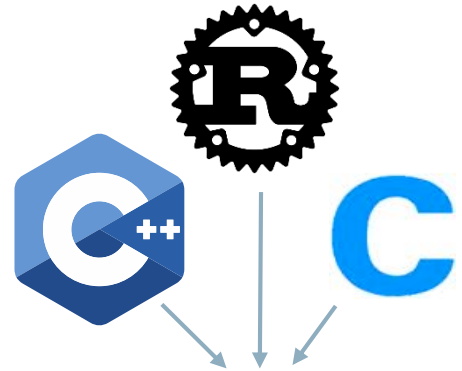
What is the Goal of Sulong?

Execute “low-level/unsafe” languages on GraalVM

- C, C++, Fortran, Rust, (Swift?)



GraalVM Stack



Truffle Framework

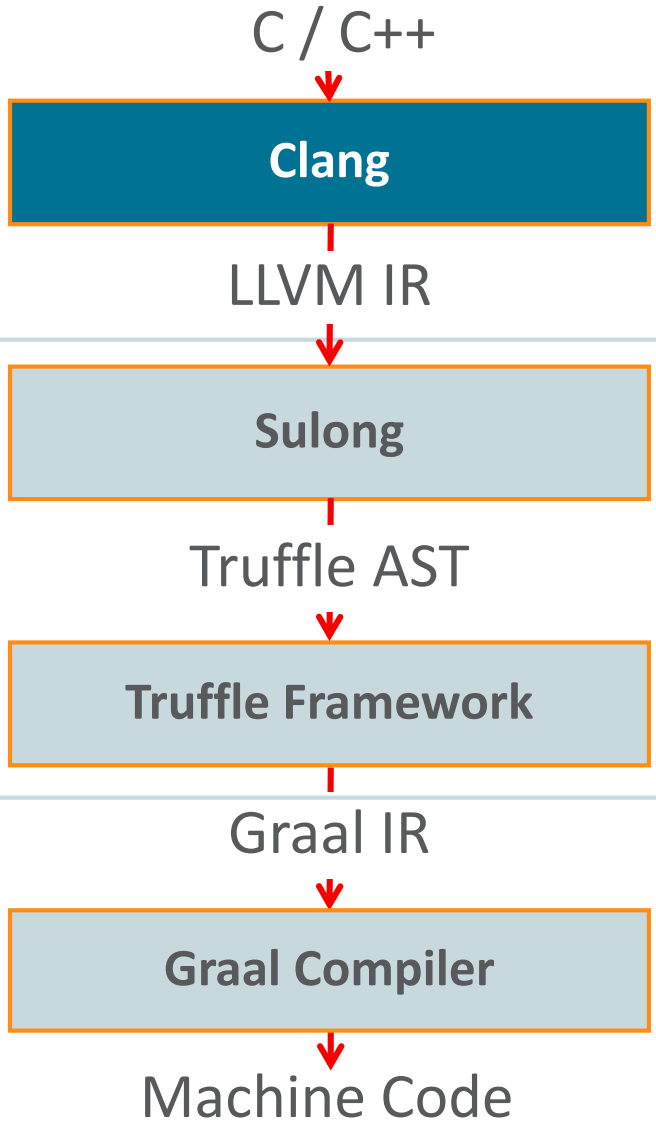
Graal Compiler

Sulong Pipeline

Ahead of Time

Run Time

Just in Time
(hot code only)



Interpreter execution

Compiled Code execution

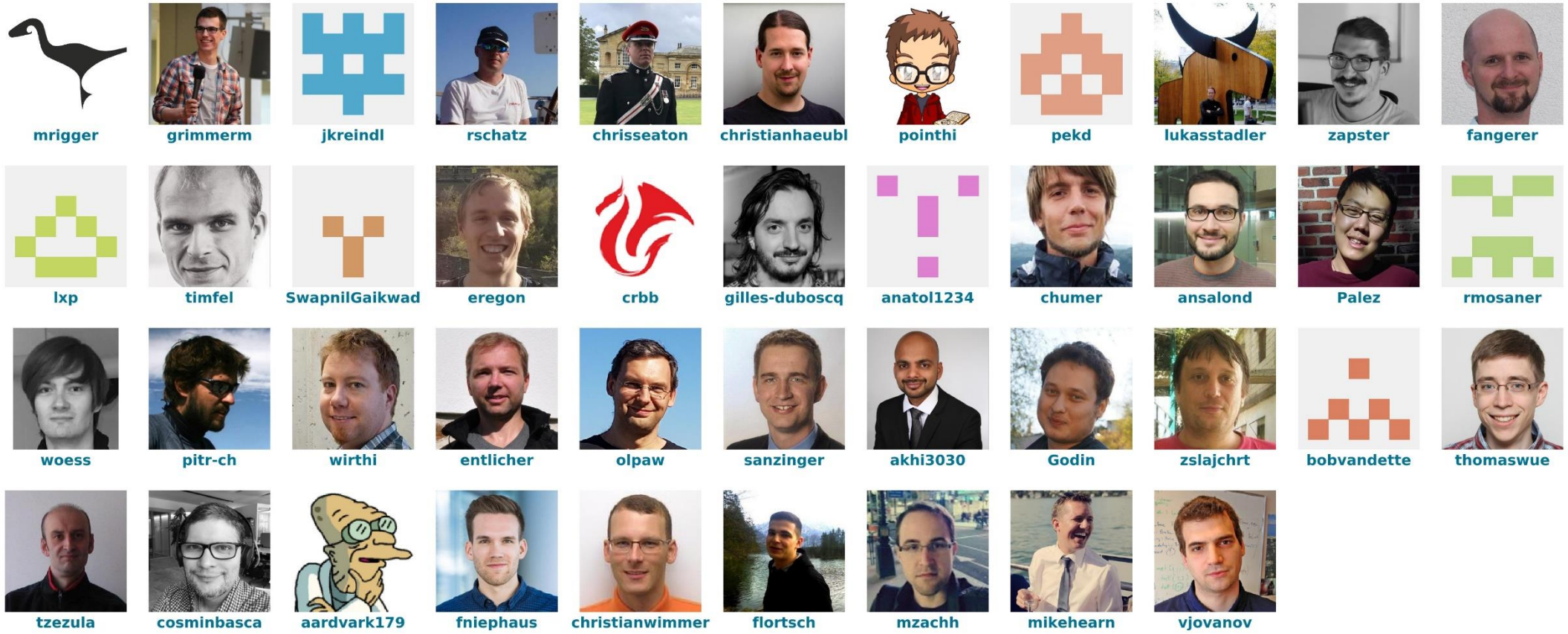
ORACLE

Sulong on Github

ORACLE Labs

JOHANNES KEPLER UNIVERSITY LINZ

MANCHESTER 1824 The University of Manchester



https://github.com/oracle/graal/tree/master/sulong

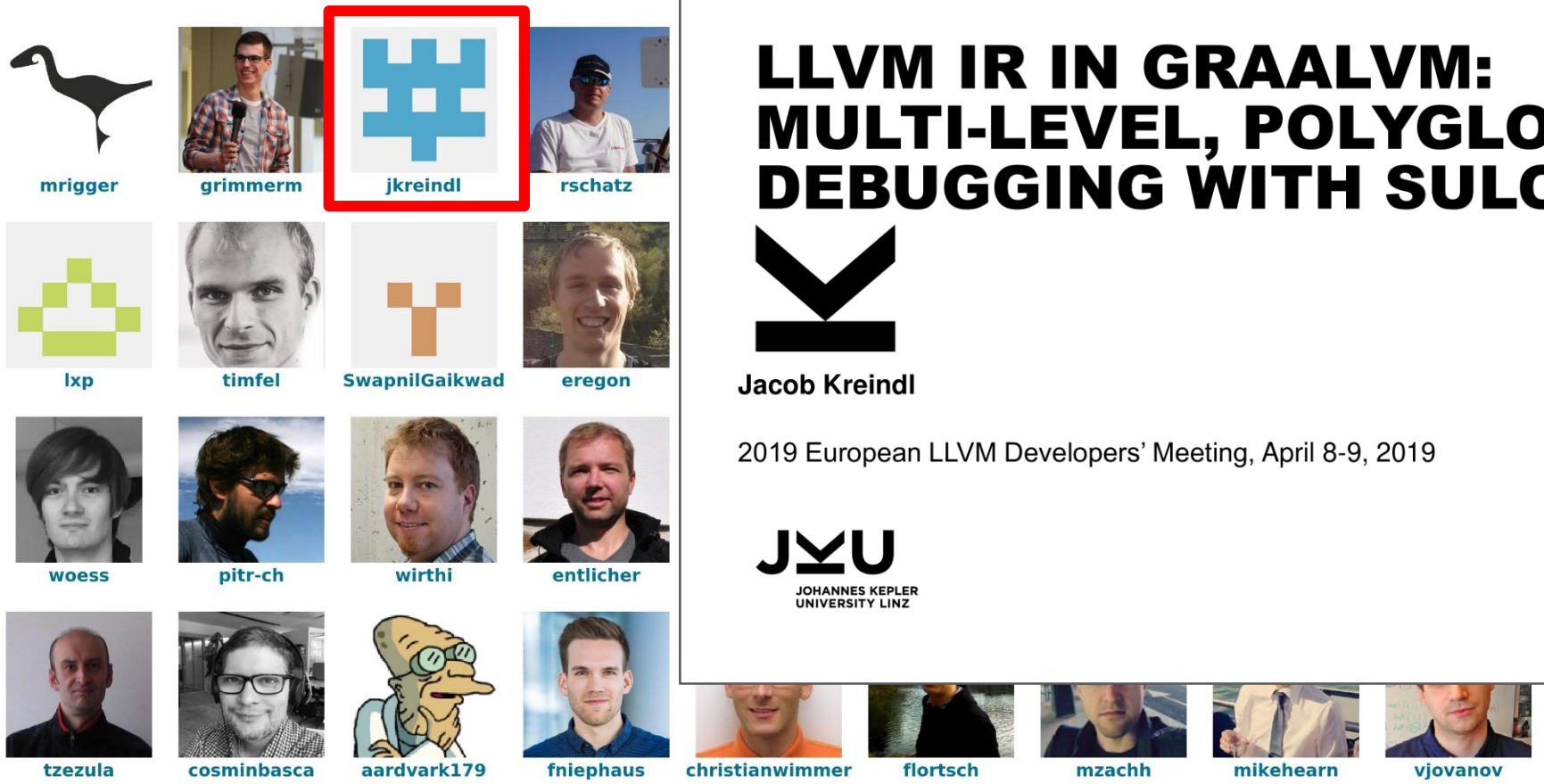
ORACLE

Sulong on Github

ORACLE Labs

JOHANNES KEPLER UNIVERSITY LINZ

MANCHESTER 1824 The University of Manchester



LLVM IR IN GRAALVM: MULTI-LEVEL, POLYGLOT DEBUGGING WITH SULONG

Jacob Kreindl

2019 European LLVM Developers' Meeting, April 8-9, 2019

JOHANNES KEPLER UNIVERSITY LINZ

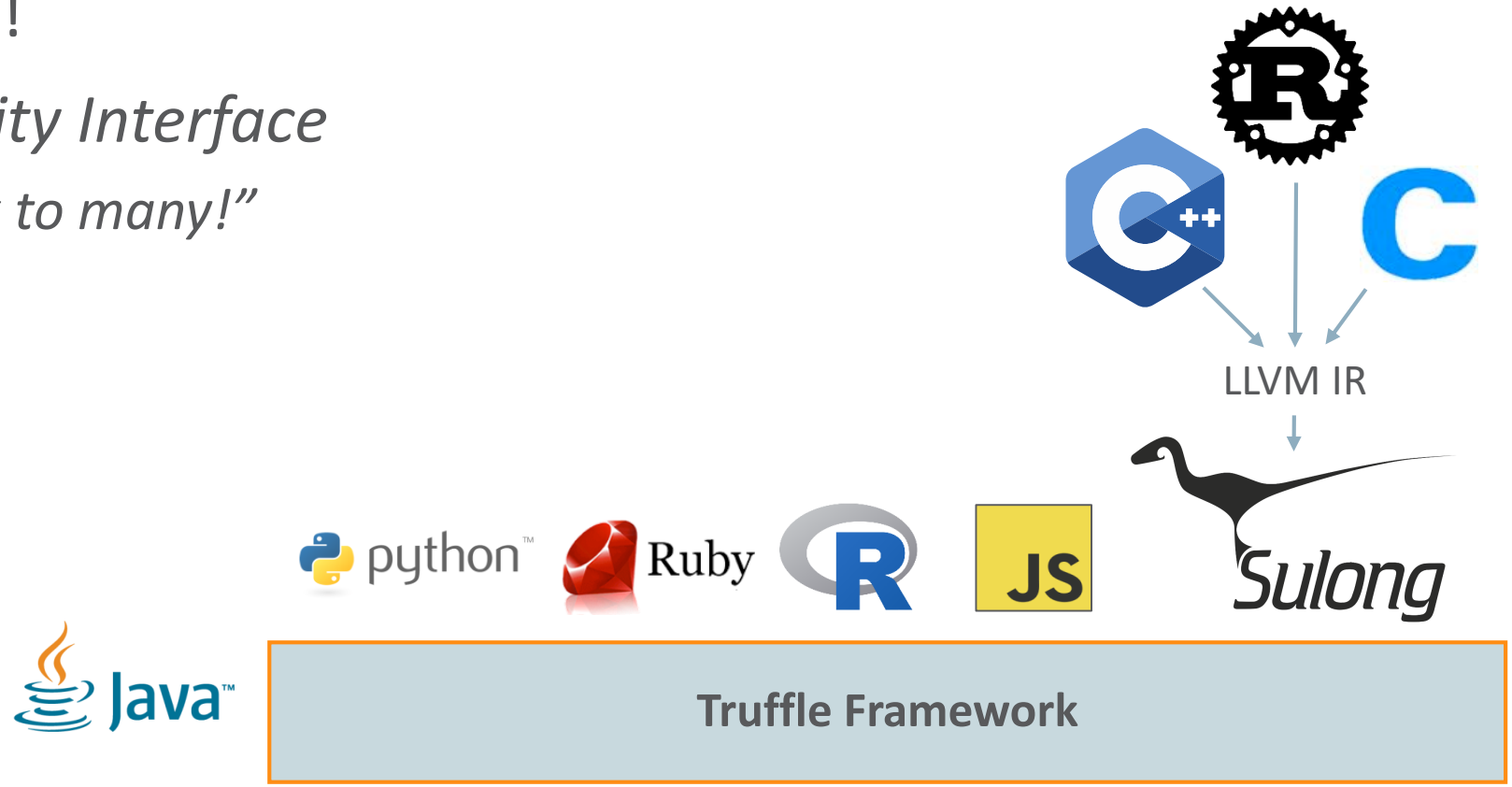
ORACLE Labs

https://github.com/oracle/graal/tree/master/sulong

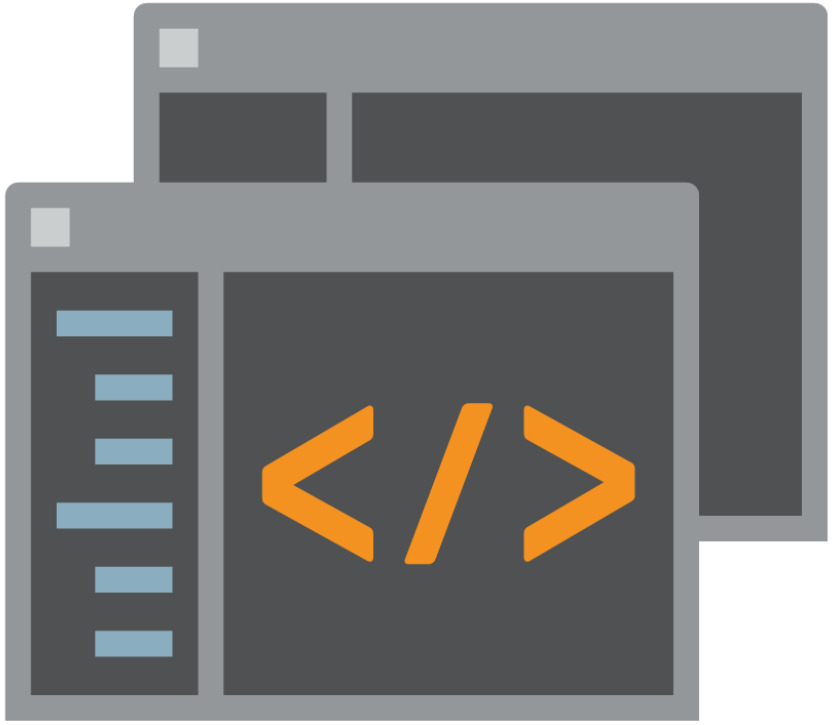
ORACLE

Fast Cross-language Interoperability

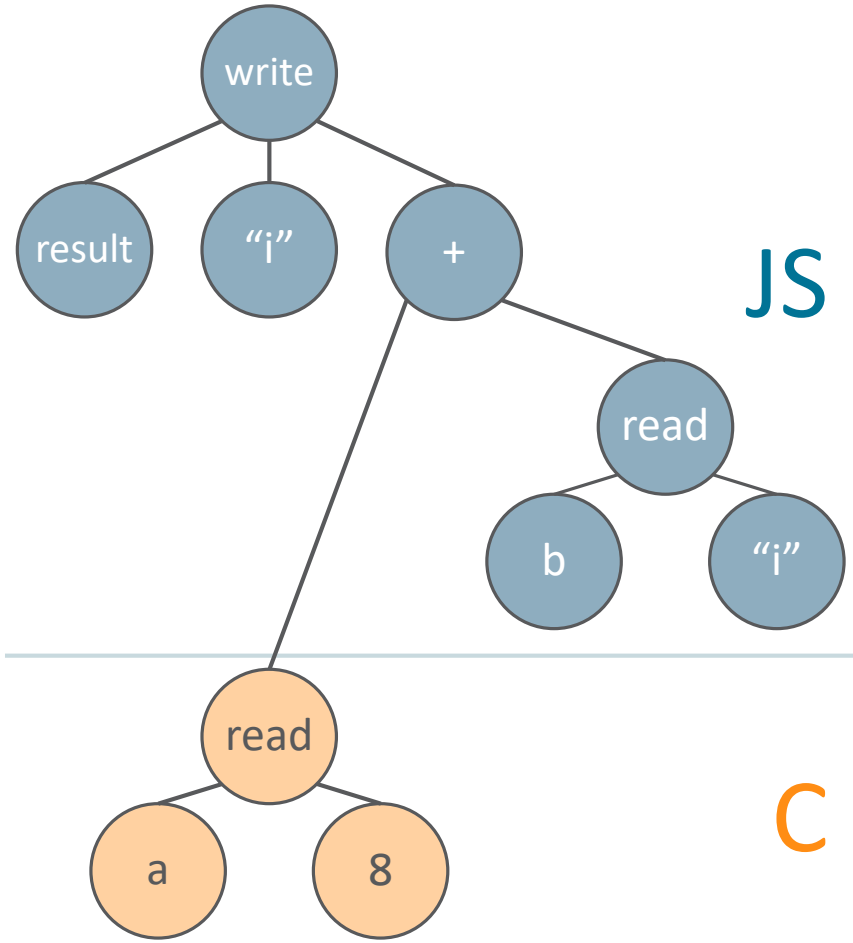
- The world is polyglot!
- Shared *Interoperability Interface*
  - *"Implement once, talk to many!"*



Live Demo



Interoperability (Truffle Approach)

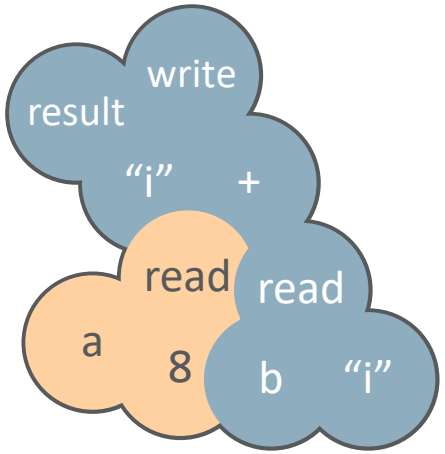


JS AST Nodes

C AST Nodes

```
function add(a, b) {
   var result = {r:0, i:0};

   result.r = a->r  +   b.r;

   result.i = a->i  +   b.i;

   return result;
}
```

Interoperability (Truffle Approach)



Single machine code function

```
function add(a, b) {
   var result = {r:0, i:0};

   result.r = a->r  +   b.r;

   result.i = a->i  +   b.i;

   return result;
}
```

Foreign Function Interfaces (FFI)

- Most non-trivial languages have an FFI
  - Usually native code (C/C++/Fortran)
- Accessing interpreter data structures
  - Implementation details become API ☹
- Sulong to the rescue!
  - Access language objects instead of C structs


```c
/* Dictionary object type */
typedef struct {
    PyObject_HEAD

    Py_ssize_t ma_used;

    uint64_t ma_version_tag;

    PyDictKeysObject *ma_keys;

    PyObject **ma_values;
} PyDictObject;
```

What is the Goal of Sulong? (cont.)

Execute “low-level/unsafe” languages on GraalVM

- C, C++, Fortran, Rust, (Swift?)

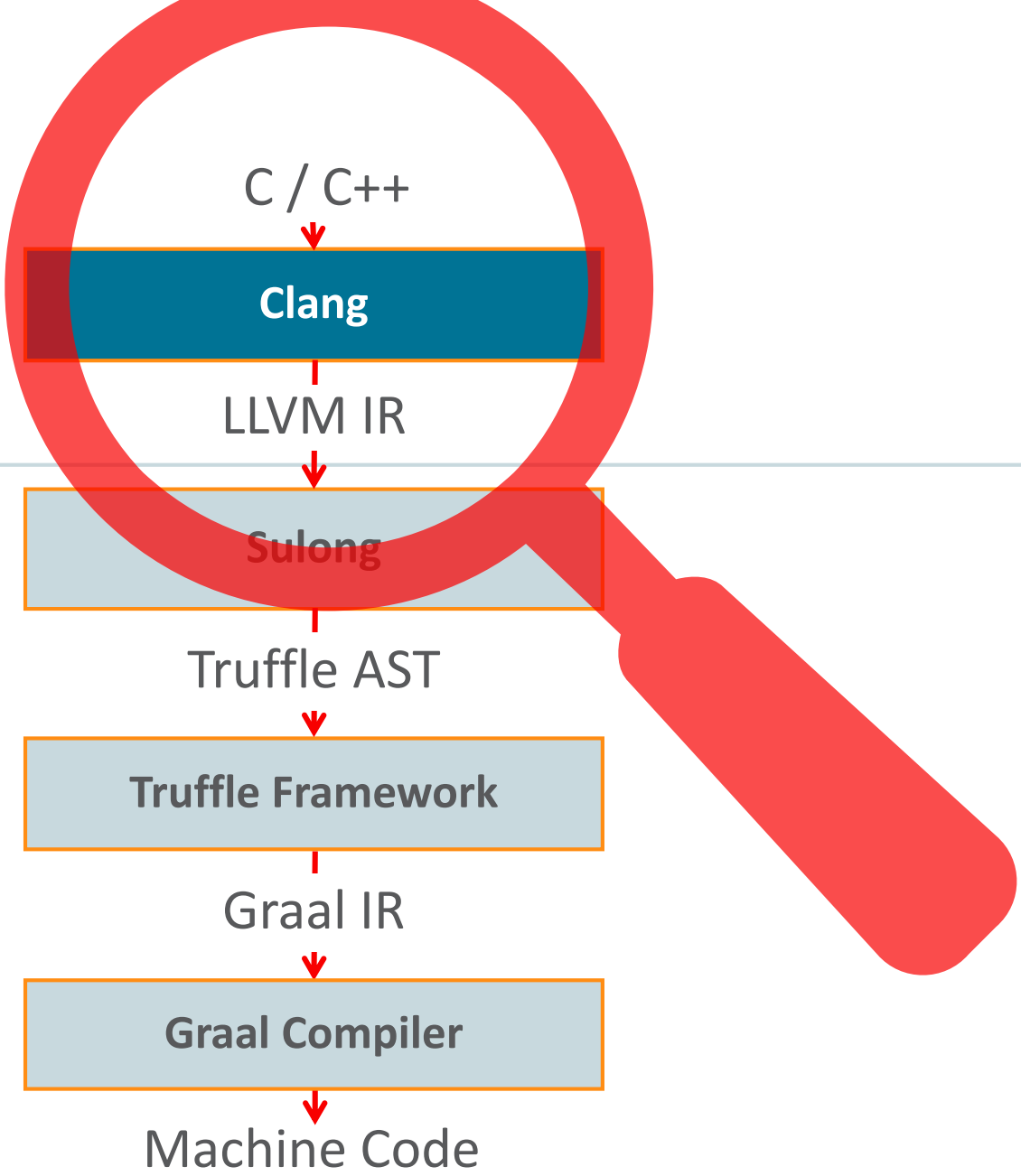
Support native language extensions

- Python, Ruby, NodeJS, R, ...

Sulong Pipeline

Ahead of Time

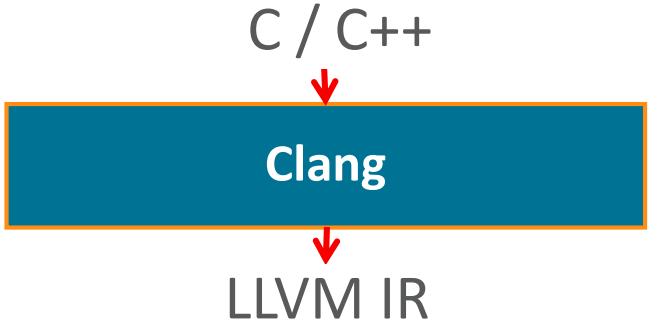
Run Time



Compile Native Projects to Bitcode

Single-file programs

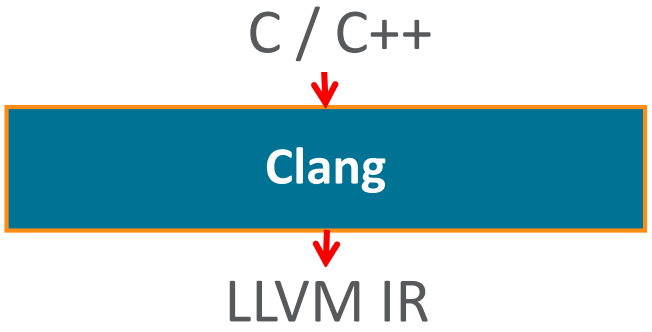
```
clang -c -emit-llvm main.c
```



Compile Native Projects to Bitcode

Mutli-file programs

```
$ clang -emit-llvm main.c foo.c

clang: error: -emit-llvm cannot be used when linking
```




```
$ clang -c -emit-llvm main.c
$ clang -c -emit-llvm foo.c

$ llvm-link main.bc foo.bc -o out.bc
```

Native Build Systems

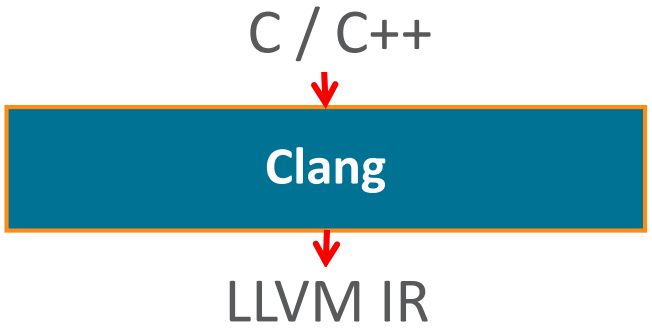
Native build systems are manifold and not under our control

- Makefile

  `CC=clang CFLAGS=-emit-llvm LD=llvm-link (?)`

- How about Python native extensions?

```
from distutils.core import setup, Extension

module1 = Extension('demo',
        define_macros = [('MAJOR_VERSION', '1'),
        ('MINOR_VERSION', '0')],
        include_dirs = ['/usr/local/include'],
        libraries = ['tcl83'],
        library_dirs = ['/usr/local/lib'],
        sources = ['demo.c'])
```

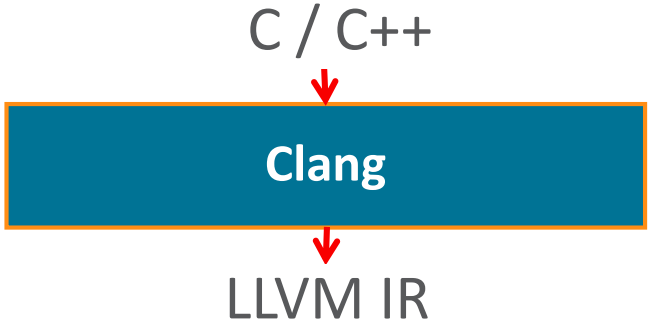


Native Build Systems

NumPy's setup is parsing object files

`setup_common.py`

```
def long_double_representation(lines):
    """Given a binary dump as given by GNU od -b,
    look for long double representation."""
```

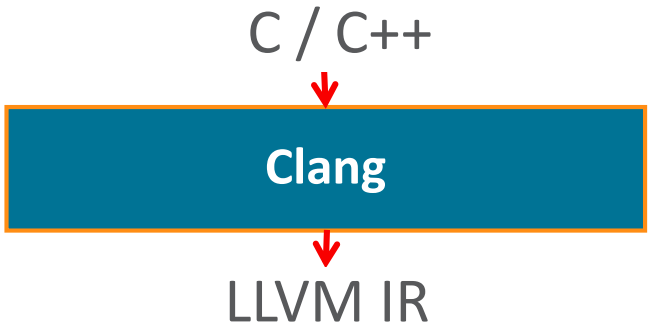


https://github.com/numpy/numpy

Compile Native Projects to Bitcode

Mutli-file programs

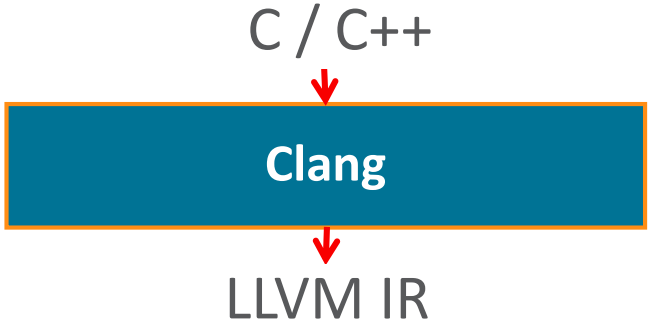
```
$ clang -c -fembed-bitcode foo.c
$ clang -c -fembed-bitcode main.c
$ clang -fembed-bitcode main.o foo.o -o a.out
$ objcopy -O binary -j .llvmbc a.out out.bc
$ lli out.bc
lli: out.bc: error: Malformed block
# bitcode section concatenated, not llvm-linked ☹
```



Compile Native Projects to Bitcode

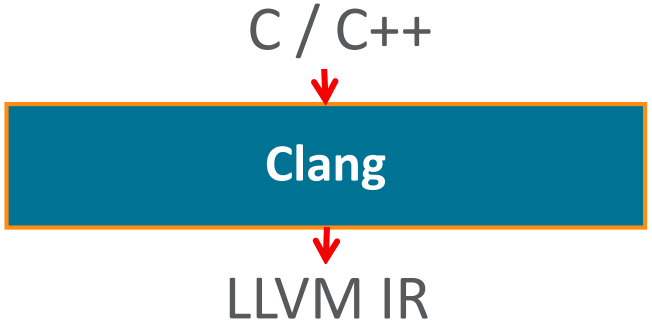
Mutli-file programs

```
$ clang -c -flto main.c

$ clang -c -flto foo.c

$ clang -fembed-bitcode -flto main.o foo.o -o a.out

# no bitcode section ☹
```



3rd Party Solutions for Compiling to Bitcode

- `wllvm`, `gllvm` wrappers \o/
  - Compiler flags fiddling is cumbersome
  - `extract-bc` hard to integrate in build scripts
  - Unsupported corner cases (e.g., cross-compilation)
- Darwin Linker supports embedding bitcodes
  - via embedded bundles
- Custom wrapper code
  - E.g. in our GraalPython

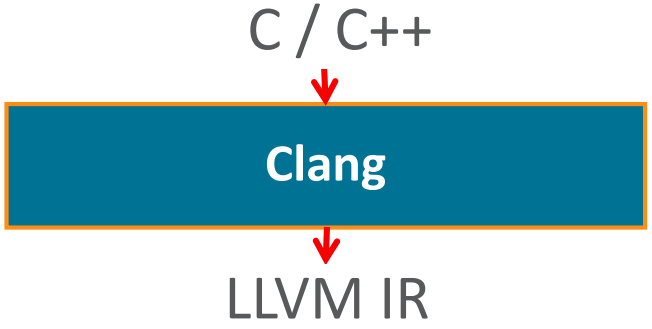



```
public class GraalPythonCC
extends GraalPythonCompiler
{...}
```

RFC: LLD + LTO + --embed-bitcode

- Teach LLD to embed bitcode during LTO

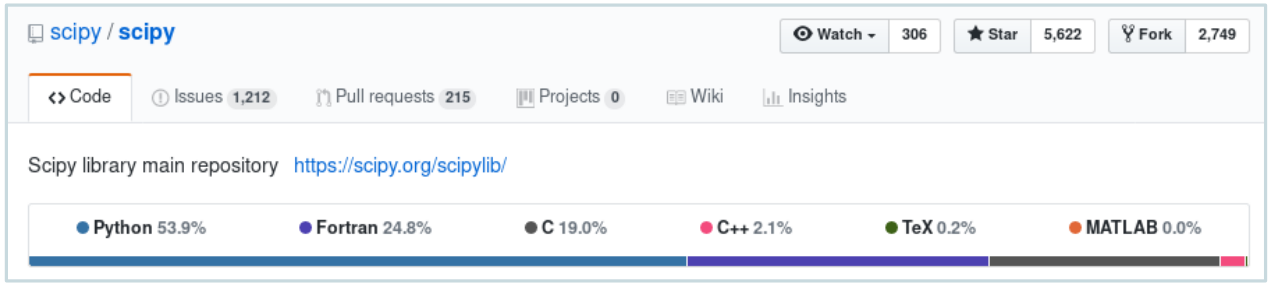
```
$ clang -c -flto main.c

$ clang -c -flto foo.c

$ clang -fuse-ld=lld -Wl,--embed-bitcode main.o
foo.o -o a.out # \o/
```

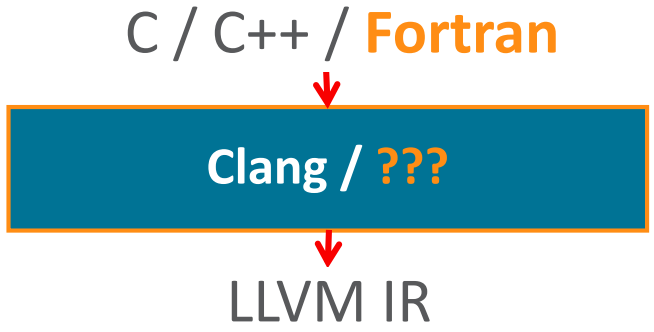


- Patch currently under evaluation
  - Planning to contribute it to upstream (if wanted)

Compiling Fortran to Bitcode

- Fortran is popular in native extensions





- DragonEgg is outdated ☹
- Looking forward to f18 ☺

Conclusion

https://www.graalvm.org

Integrated Cloud

Applications & Platform Services


ORACLE®