# Ratte: Composable Semantics-Guided Program Generators

**Jacob Yu**, Nicolas Wu, Alastair Donaldson

# Testing Compilers

## Mutation based

**Finding compiler bugs via live code mutation**

Authors: Chengnian Sun, Vu Le, Zhendong Su | Authors Info & Claims

**Compiler validation via equivalence modulo inputs**

Authors: Vu Le, Mehrdad Afshari, Zhendong Su | Authors Info & Claims

Automated testing of graphics shader compilers

Authors: Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, Paul Thomson | Authors Info & Claims

## Generation based

**RustSmith: Random Differential Compiler Testing for Rust**

Authors: Mayank Sharma, Pingshi Yu, Alastair F. Donaldson | Authors Info & Claims

**Finding and understanding bugs in C compilers**

Authors: Xuejun Yang, Yang Chen, Eric Eide, John Regehr | Authors Info & Claims

**Random testing for C and C++ compilers with YARPGen**

Authors: Vsevolod Livinskii, Dmitry Babokin, John Regehr | Authors Info & Claims

# Testing compilers

## Null-pointer dereference

```
int bar()
{
    int* p = NULL;
    return *p;    // Unconditional UB
}
```

## Signed overflow if *x* = *MAX_INT*

```
int foo(int x)
{
    return x + 1 > x;
}
```

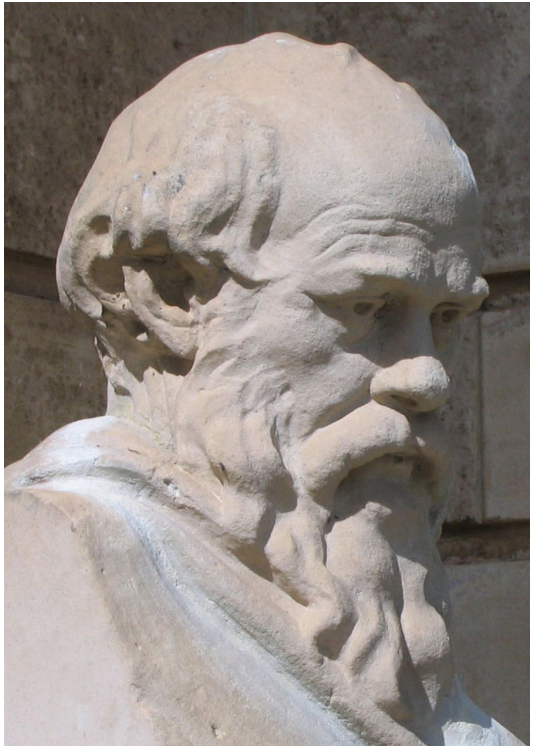
## Access out of bounds, UB if loop reaches 5 iterations

```
int table[4] = {0};
int exists_in_table(int v)
{
    // return 1 in one of the first 4 iterations or UB due to out-of-bounds access
    for (int i = 0; i <= 4; i++)
        if (table[i] == v)
            return 1;
    return 0;
}
```

Examples from cppreference.com

# Testing compilers

Know ground truth: exactly how the code *should* behave

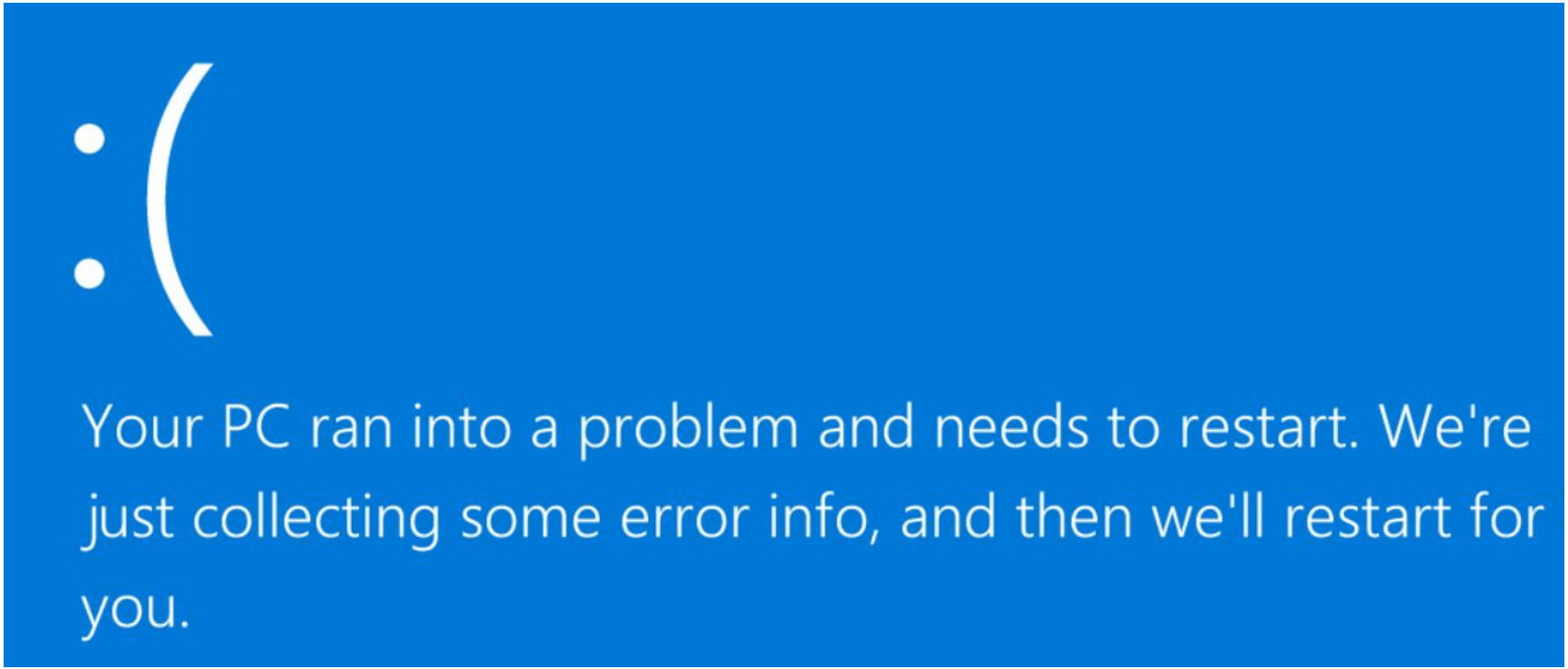


# Testing compilers

Differential testing across implementations or configurations

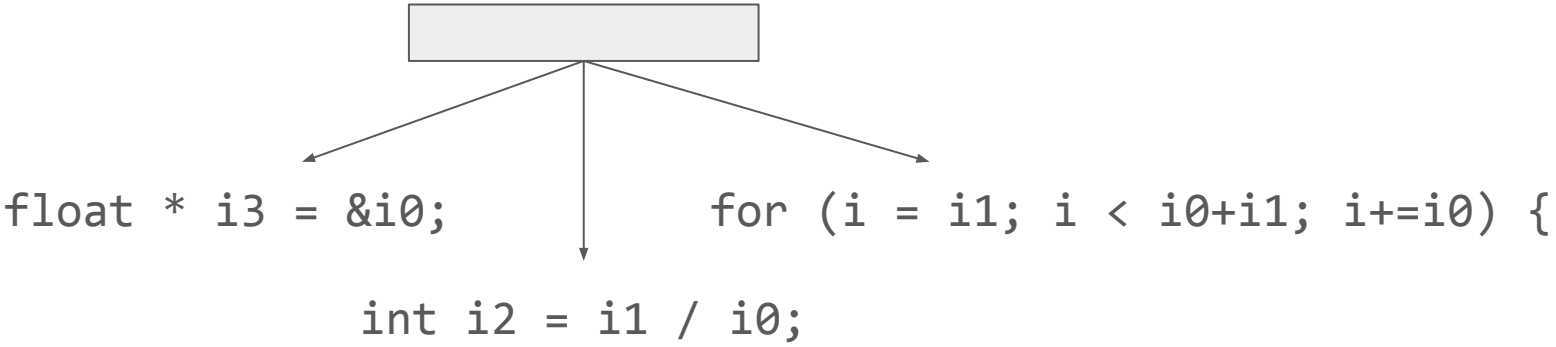
# Testing compilers

Compilers should never crash!



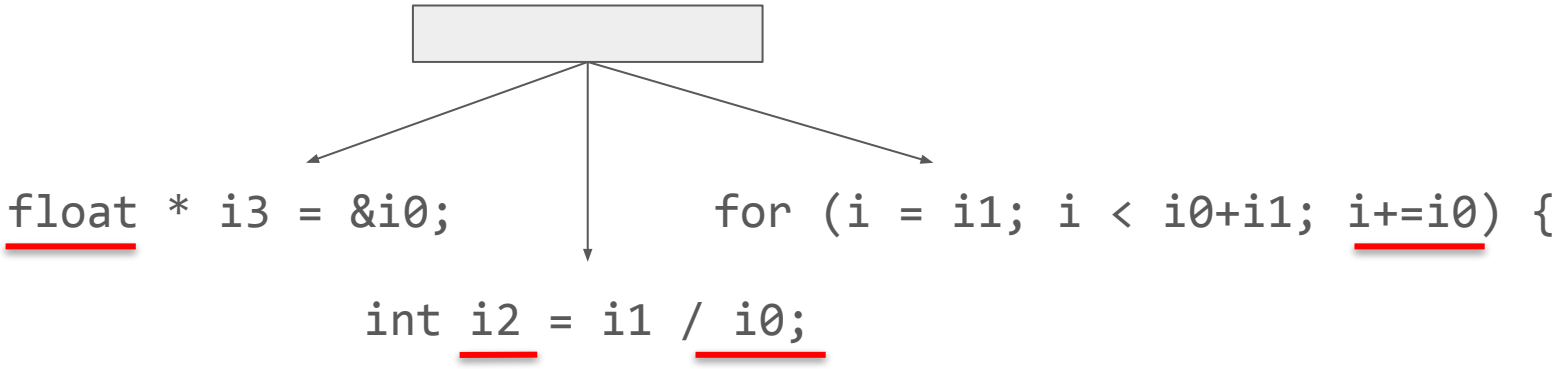
# What is a program generator

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```



# What is a program generator

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```



# What is a program generator

*Csmith*

**Symbol table**
i0 : int
i1: int
c : bool
i2 : int

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```

# What is a program generator

*Csmith*

**Symbol table**
i0 : int
i1: int
c : bool
i2 : int

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```

```
int * i3 = &i0;
```

```
i0’ = i0 == 0 ? 1 : i0;
int i3 = i1 / i0’;
```

# What is a program generator

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```

```
int * i3 = &i0;
```

```
i0’ = i0 == 0 ? 1 : i0;
int i3 = i1 / i0’;
```

*Csmith*

**Symbol table**
i0 : int
i1: int
c : bool
i2 : int

*YARPGen*

**Value table**
i0 = 0
i1 = 1
c = false
i2 = 0

# What is a program generator

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```

```
int * i3 = &i0;
```

```
int i3 = i0 / i1;
```

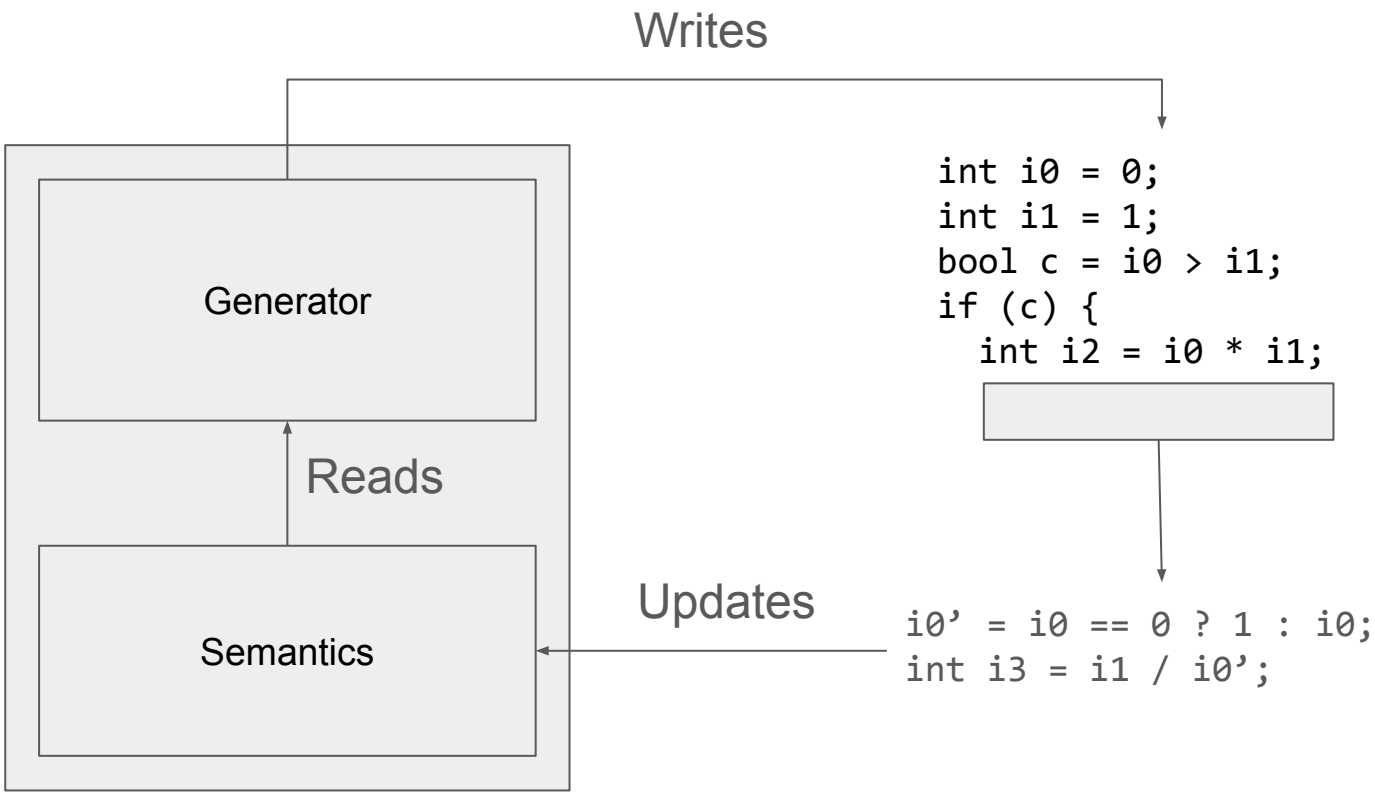
*Csmith*

**Symbol table**
i0 : int
i1: int
c : bool
i2 : int

*YARPGen*

**Value table**
i0 = 0
i1 = 1
c = false
i2 = 0

# What is a program generator



# What is a program generator

Semantics?

Partial programs: programs obtained by

- Doing a DFS on some valid (e.g. well-defined) program
- Take the prefix of that DFS

# What is a program generator

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
  int i2 = i0 * i1;
```

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
if (c) {
```

```
int i0 = 0;
int i1 = 1;
```

```
int i0 = 0;
int i1 = 1;
bool c = i0 > i1;
```

```
int i0 = 0;
```

```
... and more
```

$\subseteq$ PartialProrgams

# What is a program generator

- Fuzzers *extend* partial programs
- To some larger partial program, using operations
- `[e1, e2, ... en]`
- `extend : PartialProgram ~> PartialProgram`

# What is a program generator

Semantics for fuzzers: defined on partial programs

- `S : PartialProgram -> A`

Efficient semantics:

- `S(p+e) = f(S(p), e)`

# What is a program generator

Program generators are an interface:

```
generate : S -> RandomGen(E)

evaluate : (S, E) -> S
```

# What is a program generator

Program generators are an interface:

```
generate : (S, [Int]) -> E

evaluate : (S, E) -> S
```

# Applying to MLIR


```
%0 = “arith.constant”() {value = 5 : i64} : () -> i64
```

Dialect

Operation

Op type

# Applying to MLIR

MLIR

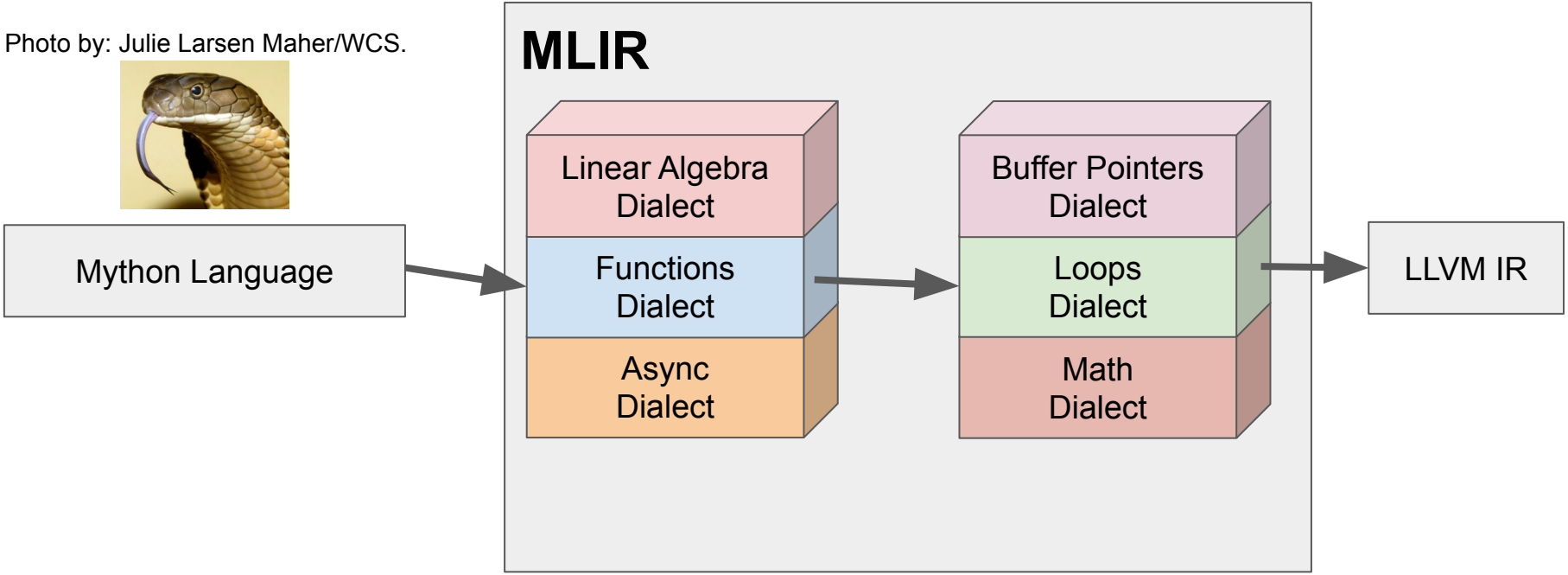
```
"func.func"()<{function_type = () -> i1, sym_name = "one"}> ({
  %0 = "arith.constant"() <{value = -1}> : () -> i1
  "func.return"(%0) : (i1) -> ()
}) : () -> ()
```

```
%1 = "scf.if"(%c) ({
  ...
}, {
  ...
}) : (i1) -> (f64)
```

# Applying to MLIR

Photo by: Julie Larsen Maher/WCS.



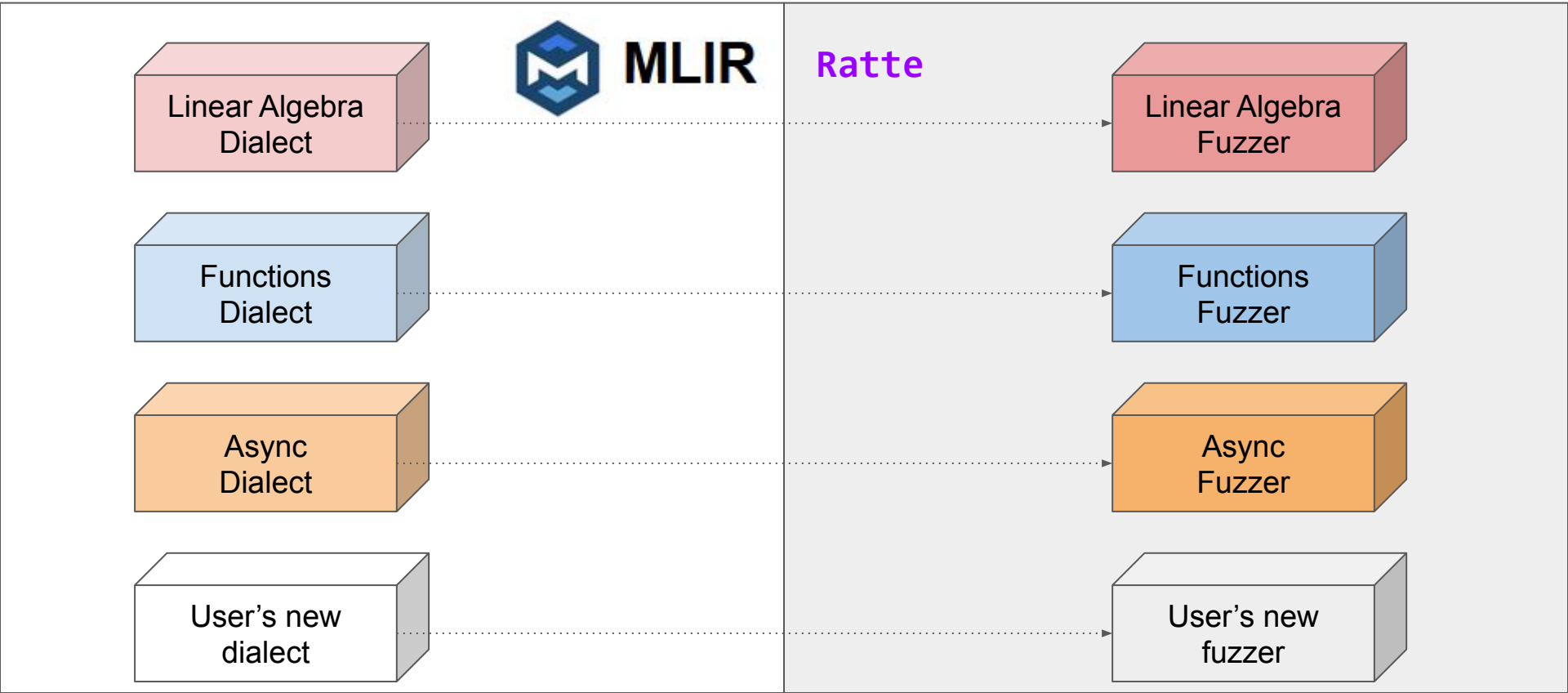


# Applying to MLIR

Building MLIR fuzzing infrastructure. Wishlist:

- Library of composable fuzzers
- User’s new dialect fuzzers can be combined with existing ones
- Should write fuzzers able to find deep bugs

# Applying to MLIR



# Applying to MLIR

E: MLIR Operation

```
generate : (S, [Int]) -> E

evaluate : (S, E) -> S
```

S: MLIR semantics, e.g. types, values

# Applying to MLIR

```
generate : (S, [Int]) -> Operation
evaluate : (S, Operation) -> S
```

# Applying to MLIR

Imperative languages:

| | Types | Interp | ... |
|---|---|---|---|
| Arith | | | 😵 |
| Scf | | | 😵 |
| ... | 😊 | 😊 | |

Functional languages:

| | Types | Interp | ... |
|---|---|---|---|
| Arith | | | 😊 |
| Scf | | | 😊 |
| ... | 😵 | 😵 | |

Diagram idea from: Eli Benderski https://eli.thegreenplace.net/2016/the-expression-problem-and-its-solutions/

# Applying to MLIR

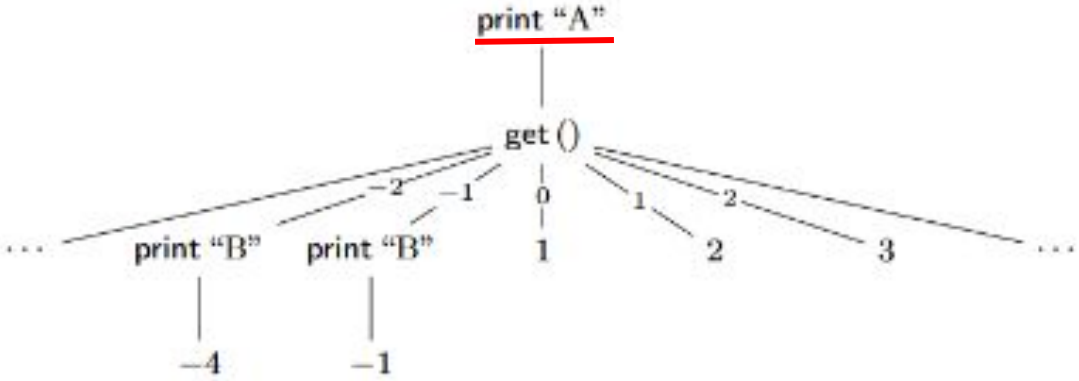
Our solution:

```
Effect Systems + QuickCheck
```

# MLIR semantics in terms of effects

*Effect constructors = Syntax*
*Effect tree = Syntax tree*

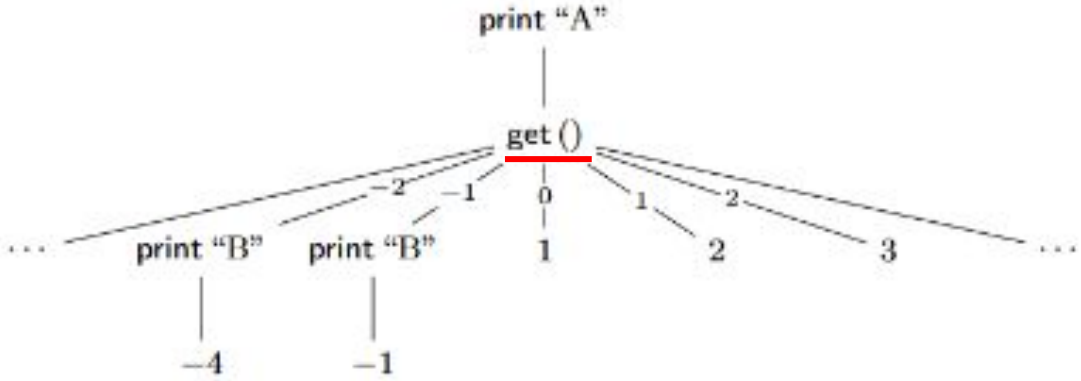
```
print "A";
do n ← get () in
if n < 0 then
   print "B";
   return −n²
 else
   return n + 1
```



# MLIR semantics in terms of effects

*Effect constructors = Syntax*
*Effect tree = Syntax tree*

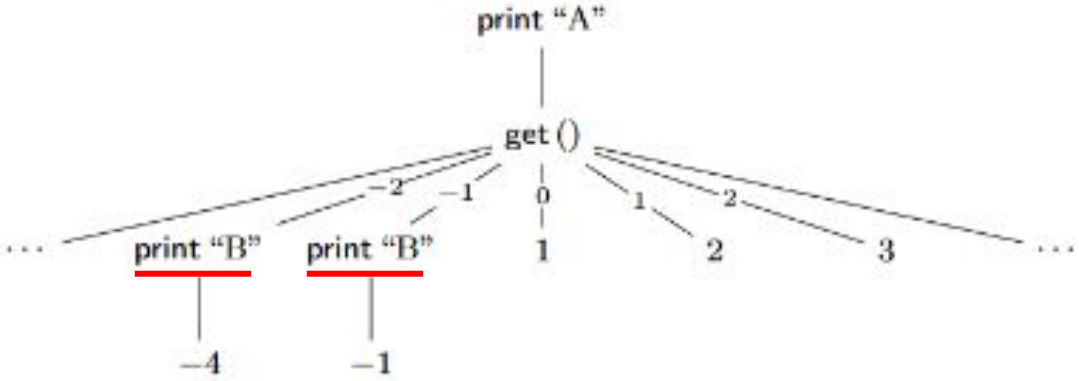
```
print "A";
do n ← get () in
if n < 0 then
   print "B";
   return −n²
 else
   return n + 1
```



# MLIR semantics in terms of effects

*Effect constructors = Syntax*
*Effect tree = Syntax tree*

```
print "A";
do n ← get () in
if n < 0 then
   print "B";
   return −n²
 else
   return n + 1
```

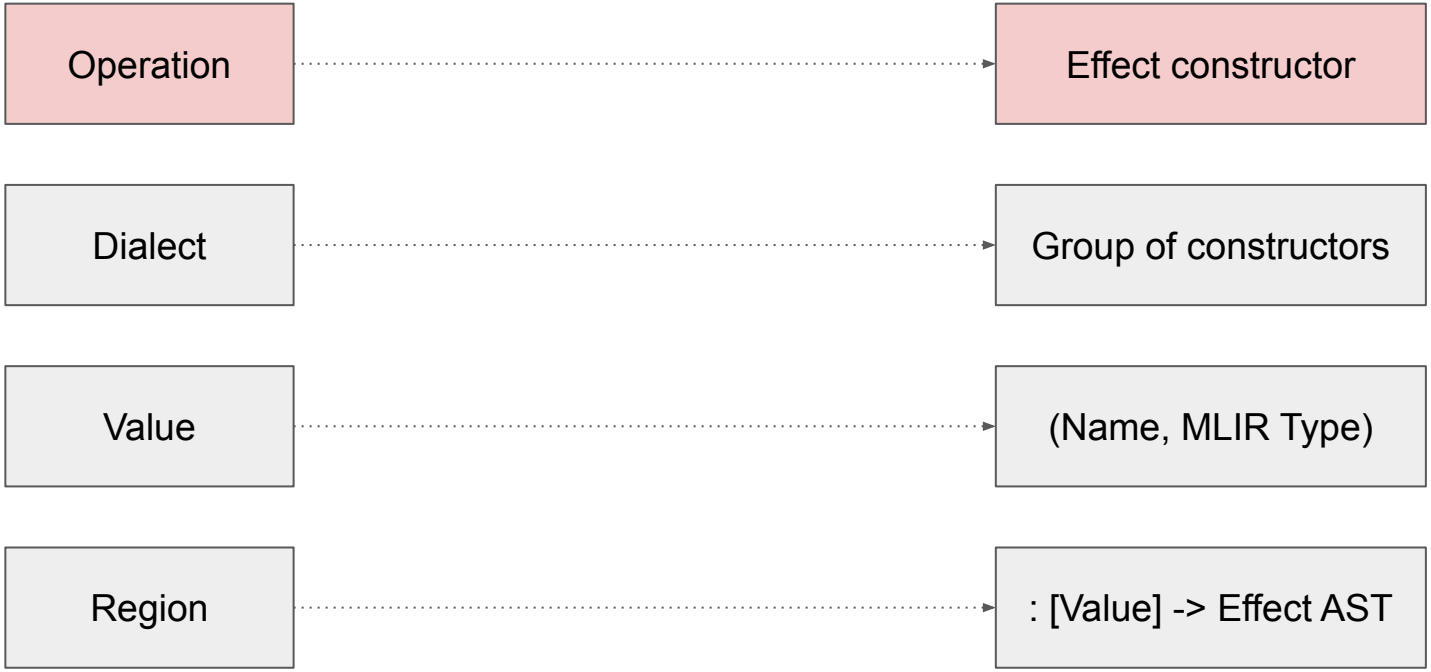


# MLIR semantics in terms of effects

*Effect constructors = Syntax*
*Effect tree = Syntax tree*
*Effect handler = Semantics by tree-rewrites*

$$
\begin{aligned}
h(a) = handler\{&\texttt{return x} \mapsto c_r(x, a);\\
&\texttt{op}_1(v; k) \mapsto c_1(v, a, k)\\
&\ldots\\
&\texttt{op}_n(v; k) \mapsto c_n(v, a, k)\}
\end{aligned}
$$

# MLIR semantics in terms of effects



# MLIR semantics in terms of effects

**MLIR Embedding**

| | | |
|---|---|---|
| **Func**<br>`func, call, return, ...` | **Tensor**<br>`constant, generate, ...` | ... |

**Interpreter Effects**

| | | |
|---|---|---|
| **Assignment**<br>`Assign, Read, ...` | **FuncTable**<br>`AddFunc, CallFunc` | ... |

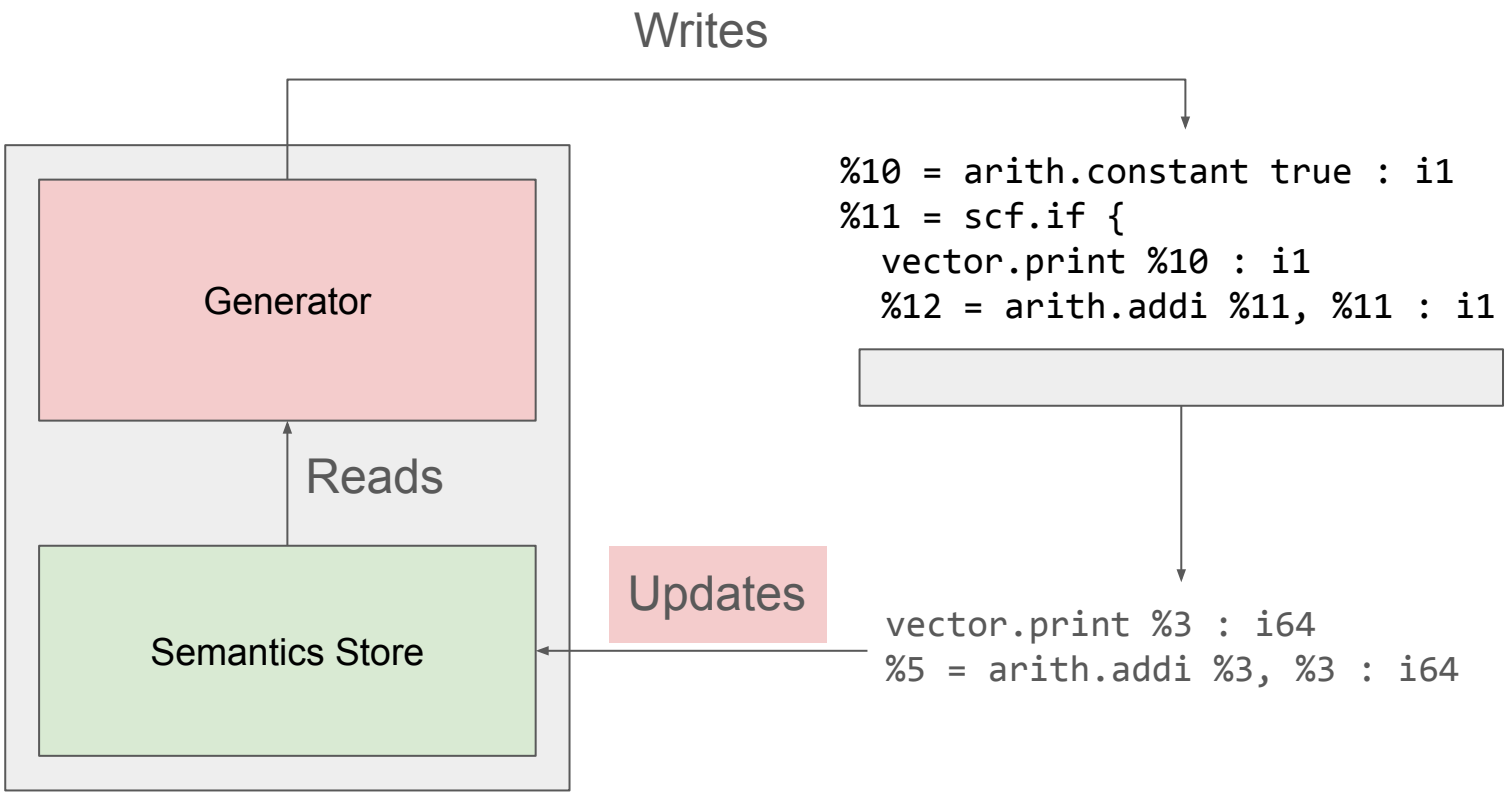
**Host Language**

| |
|---|
| **State**<br>`Read, Write, Modify, ...` |

# MLIR semantics in terms of effects

| | |
|---|---|
| **Disjoint union**<br>Updates the joint state, but no interaction otherwise<br><br>`arith.constant, tensor.fill` | **Joint interface**<br>Interface involving two or more specific types<br><br>`arith.index_cast` |
| **Open interface**<br>Public interface that allows types of other dialects to implement<br><br>`vector.print` | **Continuations (regions)**<br>Continuation that can be called, which may contain semantics from other dialects<br>`Linalg.generic, func.func` |

# MLIR semantics in terms of effects



# MLIR semantics in terms of effects

- Abstracted the structure YARPGen/Csmith program generator
- Extensible YARPGen/Csmith style fuzzers
  - Key: address the expression problem for MLIR semantics
- Way to develop QuickCheck-validated MLIR semantics

# Results from fuzzing

## Miscompilation: arith

```
func.func @main() {
    %cm, %cn1 = call @func1() : () -> (i64, i64)
    vector.print %cm : i64
    vector.print %cn1 : i64
    %1 = arith.floordivsi %cm, %cn1 : i64
    vector.print %1 : i64
    return
  }
func.func @func1() -> (i64, i64) {
    %cm = arith.constant -9223372036854775807 : i64
    %cn1 = arith.constant -1 : i64
    return %cm, %cn1 : i64, i64
}
```

# Results from fuzzing

## Rejection in func.call

```
func.func @func0(%arg0: i1, %arg1: i1) -> (i1, i1) {
  %true = arith.constant true
  return %arg1, %true : i1, i1
}
func.func @func1(%arg0: i1) -> (i1, i1) {
  %true = arith.constant true
  %0, %1 = func.call @func0(%true, %arg0) : (i1, i1) -> (i1, i1)
  return %0, %1 : i1, i1
}
```

```
<source>:11:26: error: null operand found
                %7, %8 = "func.call"(%5, %4) {callee=@func0} : (i1, i1) -> (i1, i1)
                         ^
<source>:11:26: note: see current operation: %0:2 = "func.call"(<<NULL VALUE>>, %arg0) <{callee = @func0}> : (<<NULL TYPE>>, i1) -> (i1, i1)
Compiler returned: 1
```