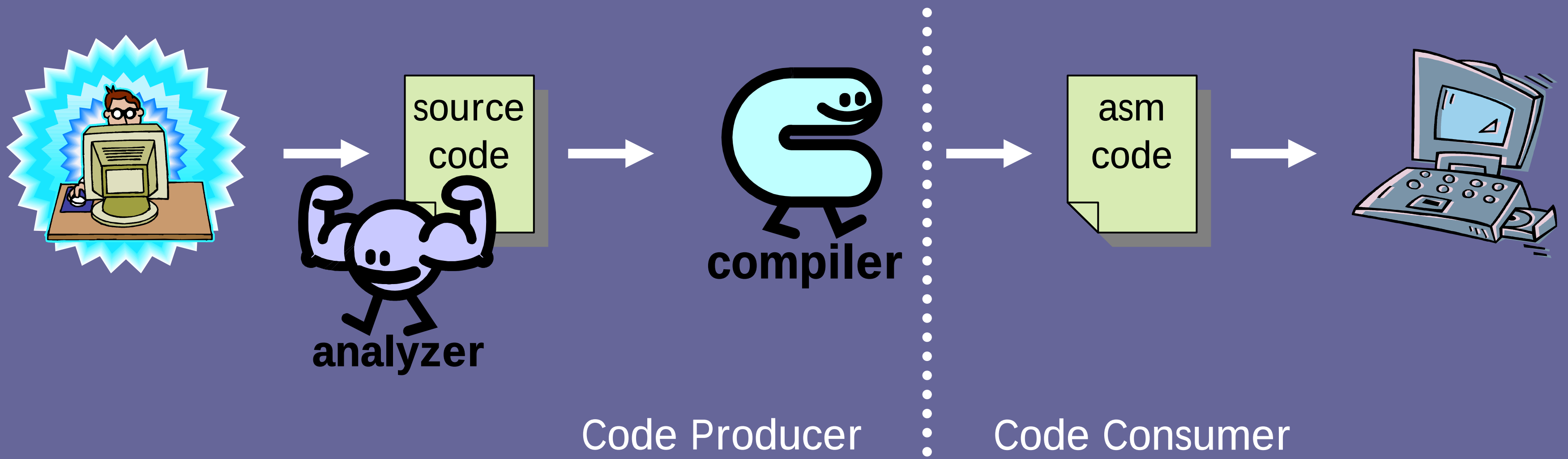
Analysis of Low-Level Code Using Cooperating Decompilers

Bor-Yuh Evan Chang
Matthew Harren
George C. Necula

University of California, Berkeley

SAS 2006

Why Analyze Low-Level Code?



- Analyze what is executed
  - Avoids issues with compiler bugs and underspecified source-language semantics
- Analyze when source is unavailable
  - Applications in mobile code safety assurance

Motivation

Analyzers for low-level code are more difficult and tedious to build

Example: Java Type Analysis

```
class C extends P {
  void m() { ... }
...
P p = new P();
P c = ... ? new C() : p;
...
c.m();
```

$r_c := \mathbf{m}[r_{sp}]$ — $\langle r_c : P, \ldots \rangle$

if $(r_c = 0)\ L_{exc}$ — $\langle r_c : \text{nonnull } P, \ldots \rangle$

$r_1 := \mathbf{m}[r_c]$ — $\ldots \rangle$

$r_1 :=$ … — $\ldots \rangle$

$r_{sp} :=$ …

$\mathbf{m}[r_{sp}] :=$ … $[r_{sp}+4]$ — $\langle \mathbf{m}[r_{sp}] : \text{nonnull } P, \ldots \rangle$

icall $[r_1]$

Unsound: Dependencies must be carefully tracked

Equal

Type analysis intermixed with low-level reasoning
- e.g., args on stack

Observations

- Handling low-level implementation details is common to many low-level analyzers
  - call stack (provides “local variables”)
  - register allocation
  - dependencies across instructions
- Ad-hoc modularization attempts failed

Goal: Design a modular framework that makes it easy to write high-level analyses
  - for different architectures
  - for the output of different compilers

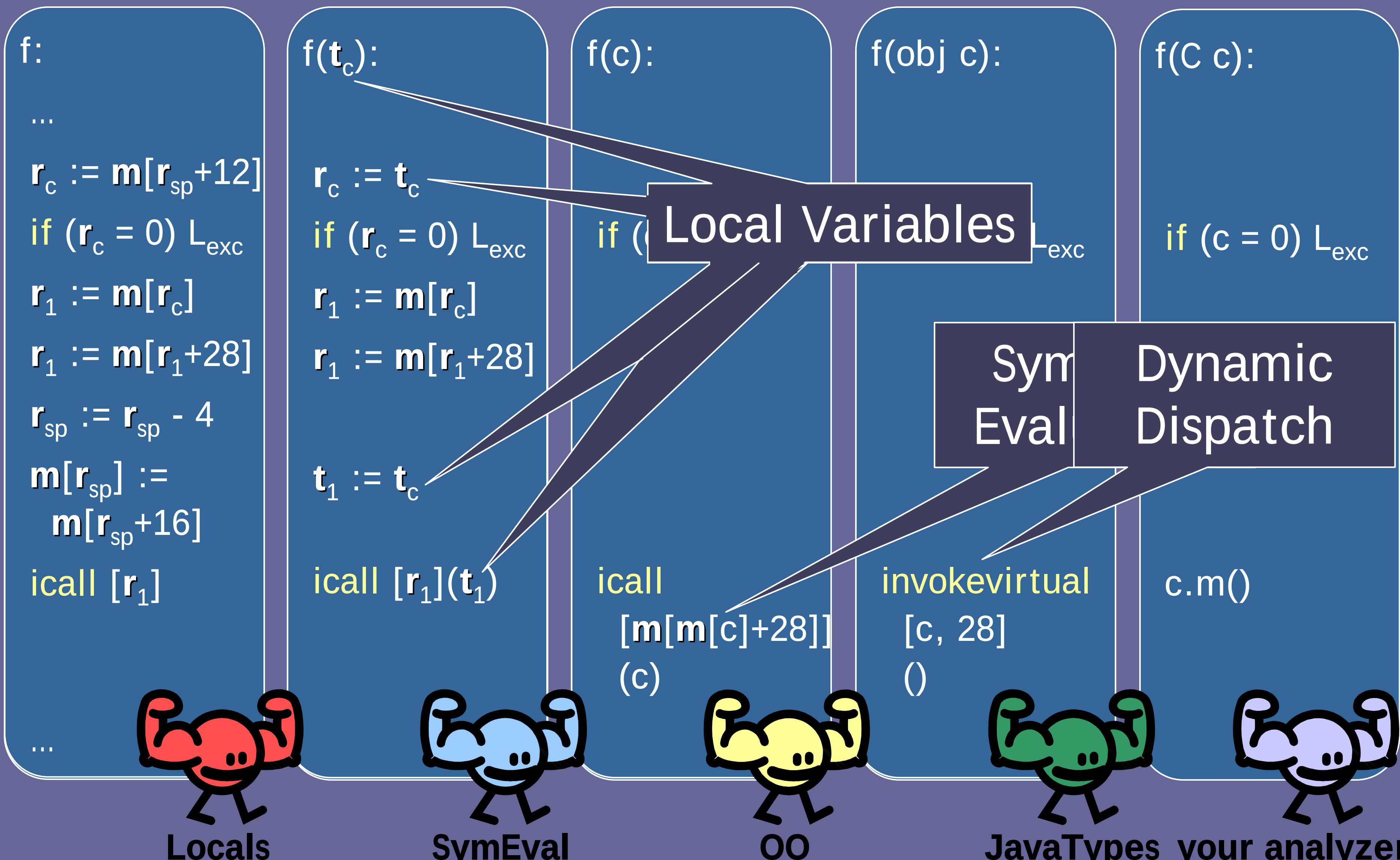
Observations

- Intermediate languages abstract varying levels of detail
  - E.g., source language hides compiler details
  - Provides well-specified interface between (de)compiler phases

Proposal: Structure low-level analyses as small, incremental decompilation phases

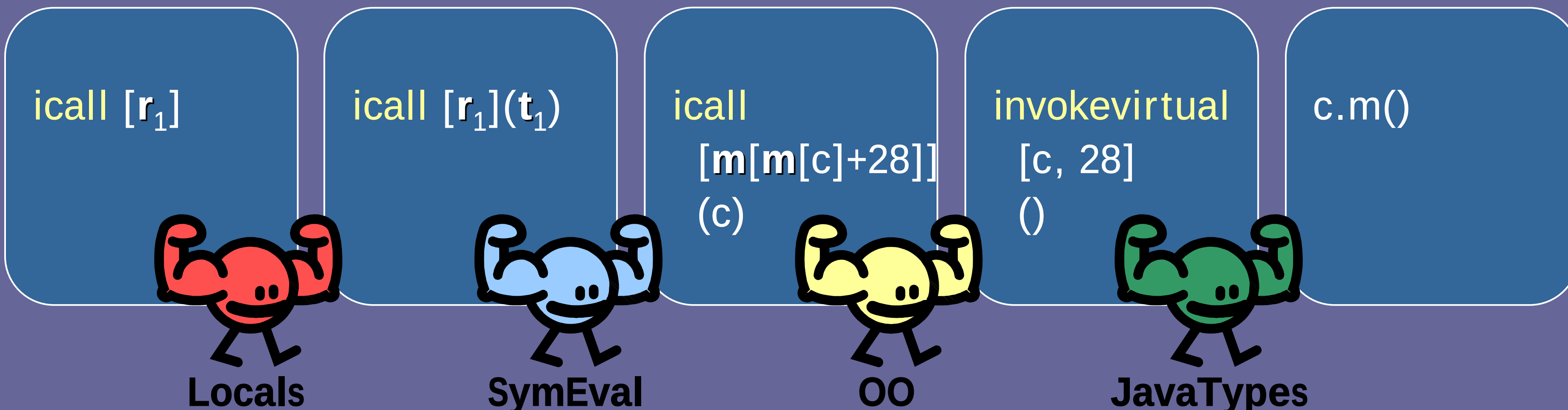
Basic Idea

static void f(C c) { c.m(); }



Difficulties

- Unidirectional communication + analysis is insufficient



<u>Needs to know</u> "call with 1 arg"

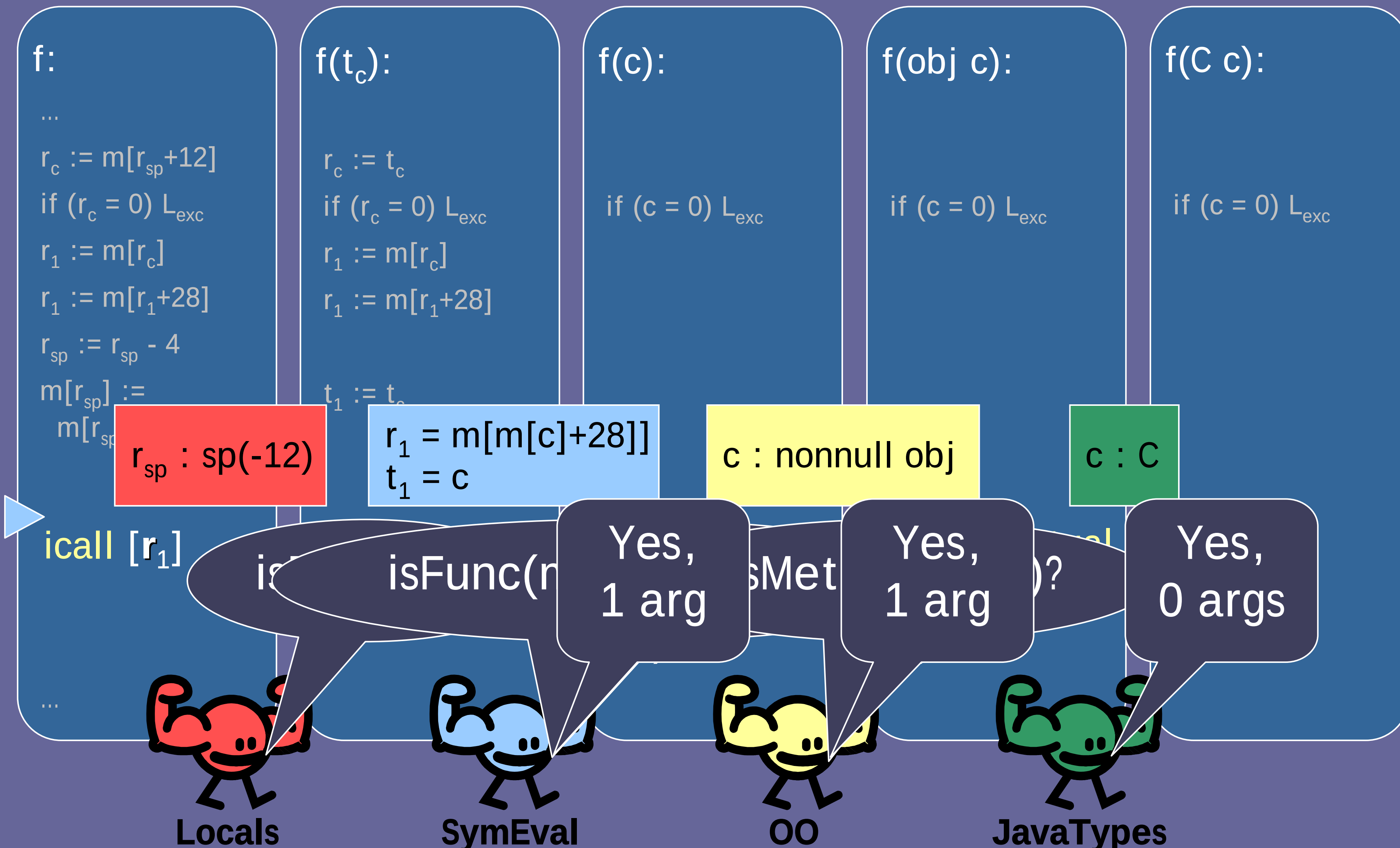
<u>Can answer</u> "method call to C.m()" because <u>knows</u> about the class table but <u>needs</u> previous instrs for analysis

Overview of the Framework

- To enable bidirectional communication
  - cannot decompile in stages
  - decompile at all levels simultaneously
- Each decompiler analyzes the preceding instructions before decompiling the next
- ~~"Pipeline"~~ "Reduced product"

Queries

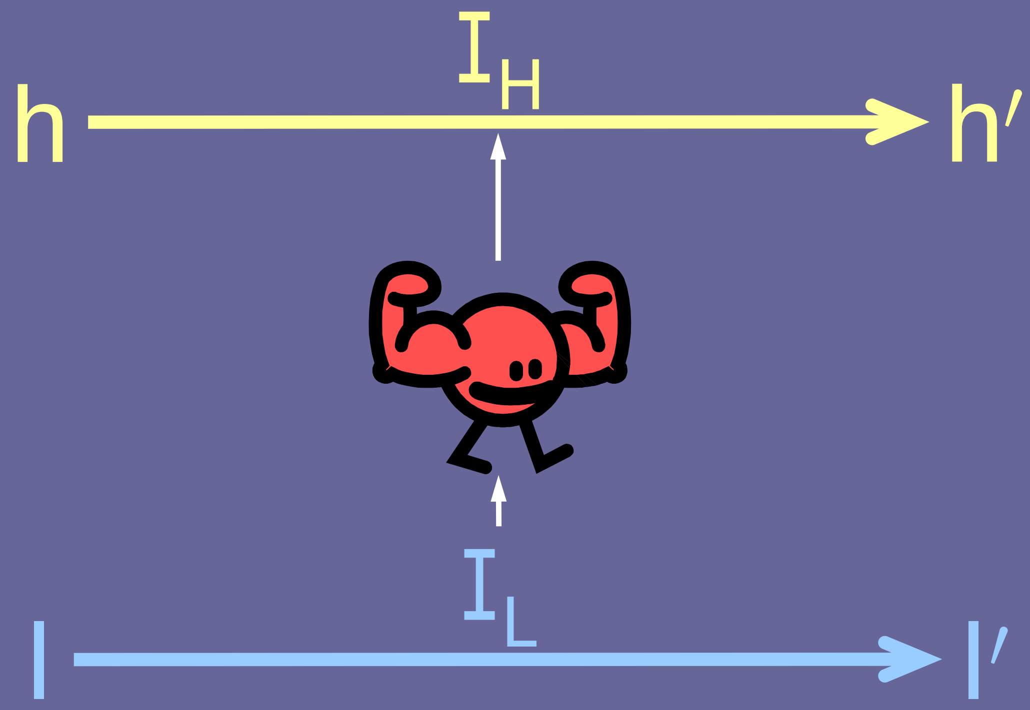
static void f(C c) { c.m(); }



Communication Summary

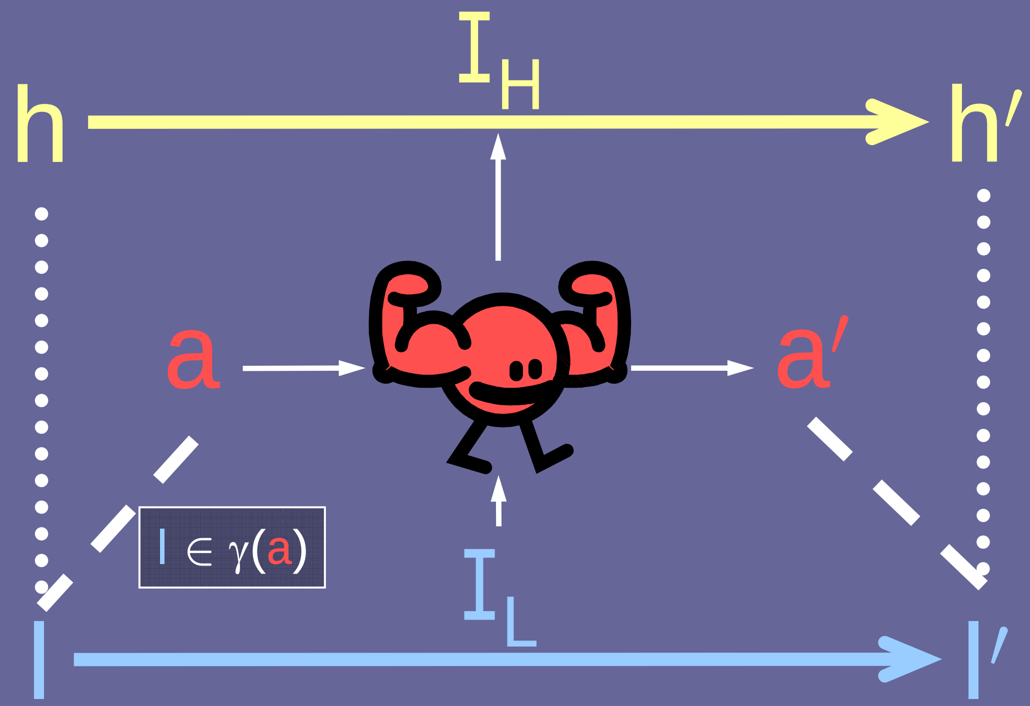
- Low-to-High (Primary)
  - Decompilation Stream
- High-to-Low
  - Queries
    - Initiated by lower-level
    - Questions decompiled
  - Reinterpretations
    - Initiated by higher-level
    - Answers decompiled

Soundness of Decompiler Pipelines



- Operational semantics for each language
- Safety encoded as “not getting stuck”

Soundness of Decompiler Pipelines

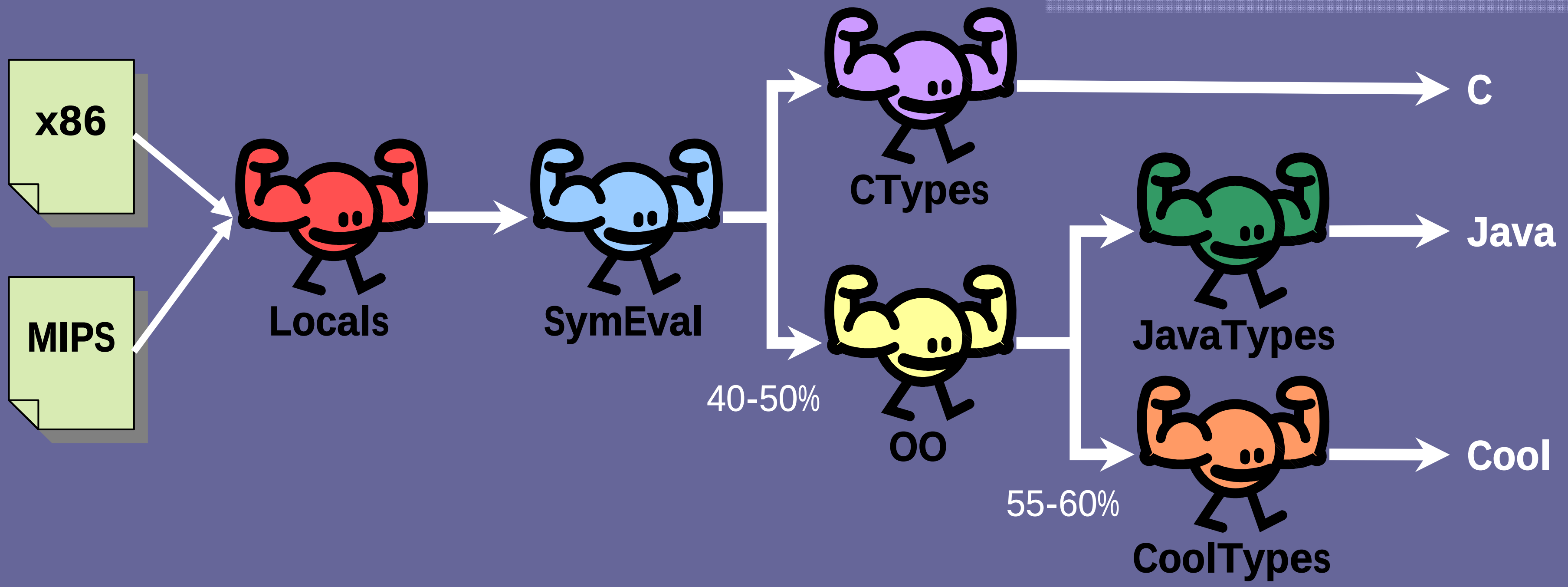


- Safe at high-level implies safe at low-level

Experiments

- Evaluate flexibility
- Evaluate modularity
- Evaluate applicability of existing source-level tools

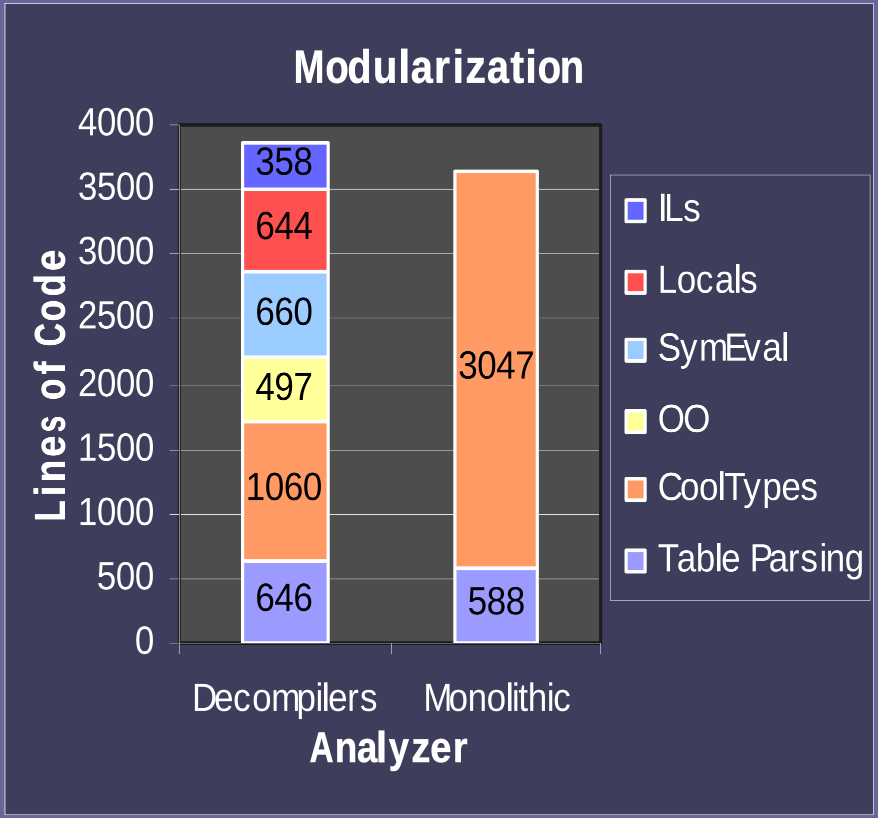
Flexibility



- For the output of gcc, gcj, and compilers for Cool (a “mini-Java”)
- To implement JavaTypes (no exns, interfaces)
  - 3-4 hours, 500 lines

Modularity: Decompilers vs. Monolithic

Type Checking Compiled Cool



- Modules approx. same size

- Compared on 10,339 tests
  - 49 tests, 211 compilers
  - 182 disagreements (pass/fail)
- Decompilers (new)
  - 1 incompletenesses
  - 0 soundness bugs
- Monolithic (used heavily)
  - 5 incompletenesses
  - 3 soundness bugs
  - mishandling of run-time library functions
    - Calls uniform - Locals
    - SSA - SymEval

Applicability of Source-Level Tools

- Experimental Setup
  - Compiled 3 previously reported benchmarks for BLAST (B) and Cqual (Q)
  - Verified presence (or absence) of bugs as in the original

Test Case		Code Size C (kloc)	Code Size x86 (kloc)	Decomp (s)	Verification Orig (s)	Verification Decomp (s)
`qpmouse.c`	(B)	8.0	1.9	0.74	0.34	1.26
`tlan.c`	(B)	10.9	10.7	8.16	41.20	94.30
`gamma_dma.c`	(Q)	11.2	5.2	2.44	0.97	1.05

- Limitations
  - Source-level tools needed types
  - Recovered types from debugging information
- On optimized code (`qpmouse.c`)
  - gcc –O2 except "merge constants"
  - reads byte in middle of word-sized field

Conclusion: Lessons Learned

- Need types for existing source analyses
  - E.g., BLAST on untyped C does not work
- Very useful low-level modules
  - Locals: recover statically-scoped local variables
  - SymEval: recover normalized expr. trees, SSA
- Defining output IL guides analysis impl.
  - IL specifies what analysis should guarantee
  - Leads to small and well-isolated modules

Thank You!

http://www.cs.berkeley.edu/~bec

High-to-Low Communication Examples

- Queries
  - "Does *e* point to a function/method?"
  - "Does *e* point to an object?"
- Reinterpretations
  - Exceptional successor for try-catch blocks