Verifying the LLVM

Steve Zdancewic

DeepSpec Summer School 2017

Thanks To

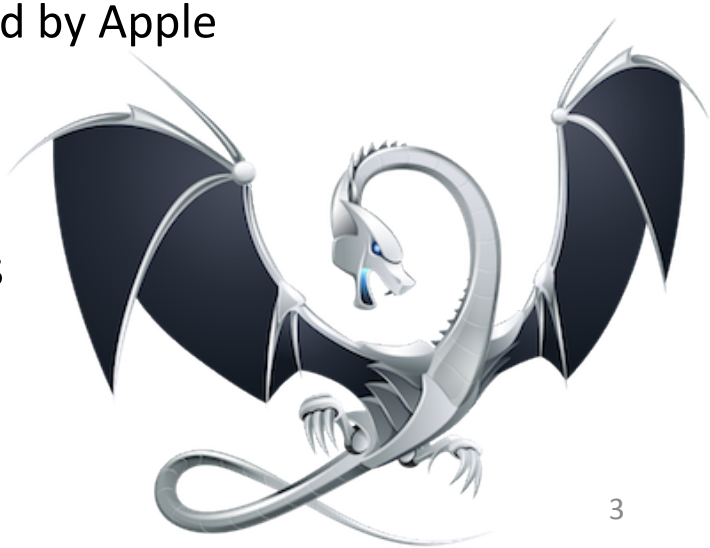
- Dmitri Garbuzov
- Nicolas Koh
- Olek Gierczak

And… collaborators on Vellvm

- Jianzhou Zhao
  - developed the "legacy" Vellvm Coq framework
- Santosh Nagarakatte
- Milo Martin
- William Mansky
- Christine Rizkallah

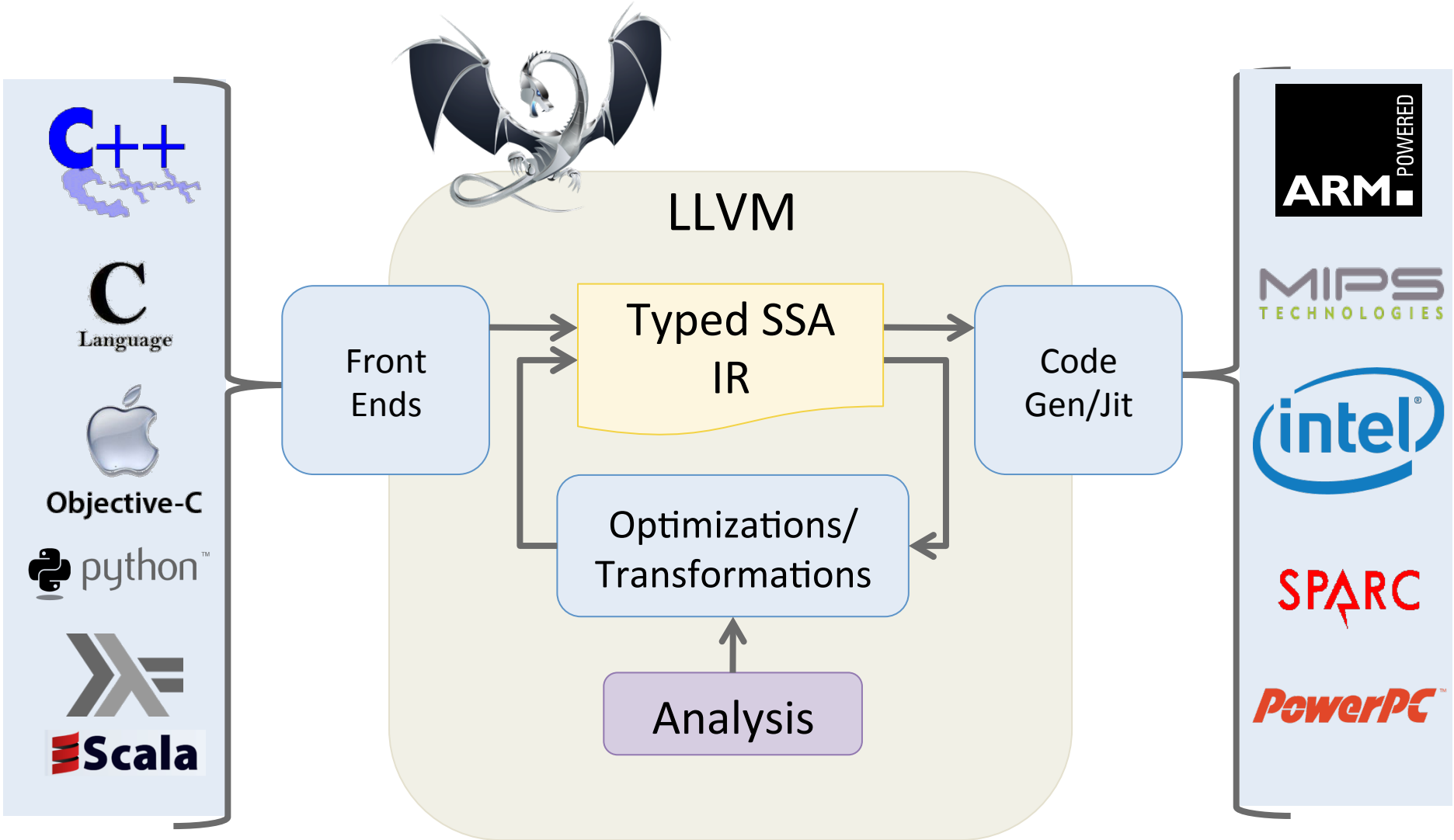
Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Multi-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
  - llvm-gcc (drop-in replacement for gcc)
  - Clang: C, objective C, C++ compiler supported by Apple
  - various languages: ADA, Scala, Haskell, …
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects
  - Here at Penn: SoftBound, Vellvm



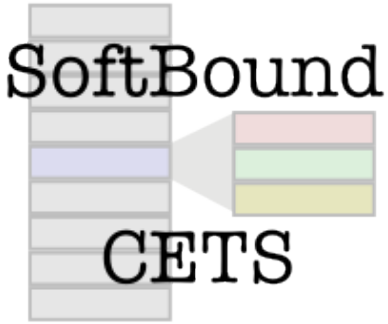
LLVM Compiler Infrastructure

[Lattner et al.]



Motivation: SoftBound/CETS

[Nagarakatte, et al. *PLDI '09, ISMM '10*]



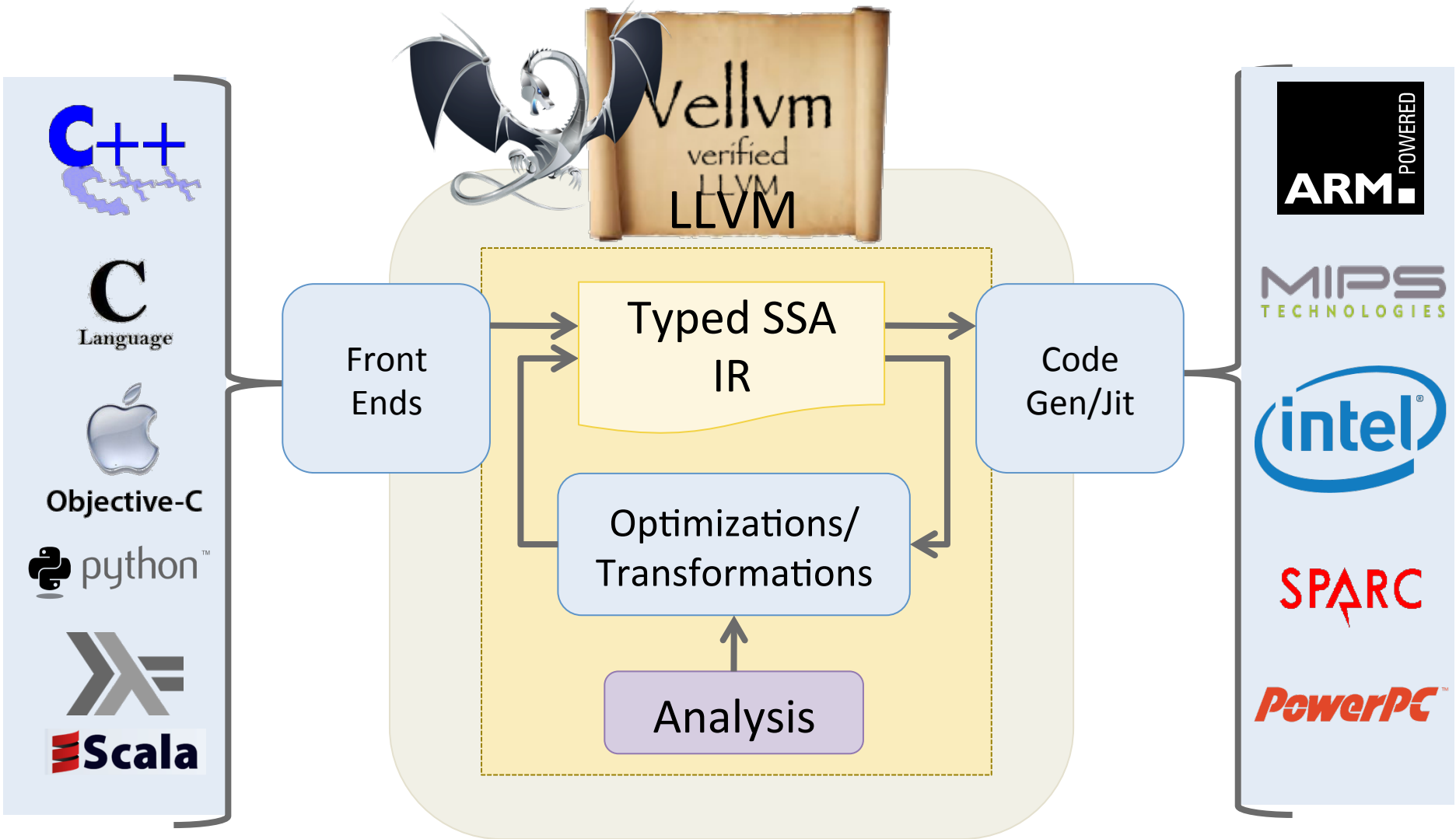
- Buffer overflow vulnerabilities.
- Detect spatial/temporal memory safety violations in legacy C code.
- Implemented as an LLVM pass.
- What about correctness?

Context:
Penn's POPLMark challenge: using Coq was becoming cool
Xavier Leroy's CompCert: provided inspiration!

http://www.cis.upenn.edu/acg/softbound/

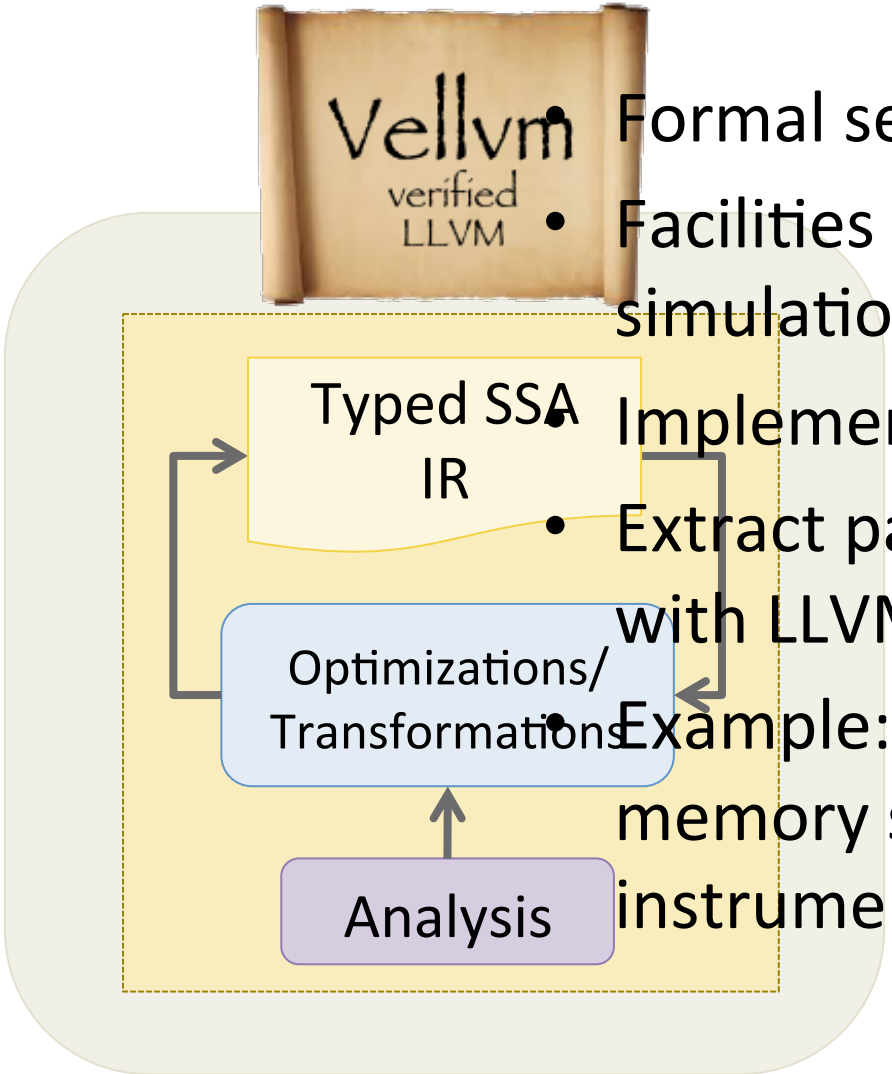
LLVM Compiler Infrastructure

[Lattner et al.]



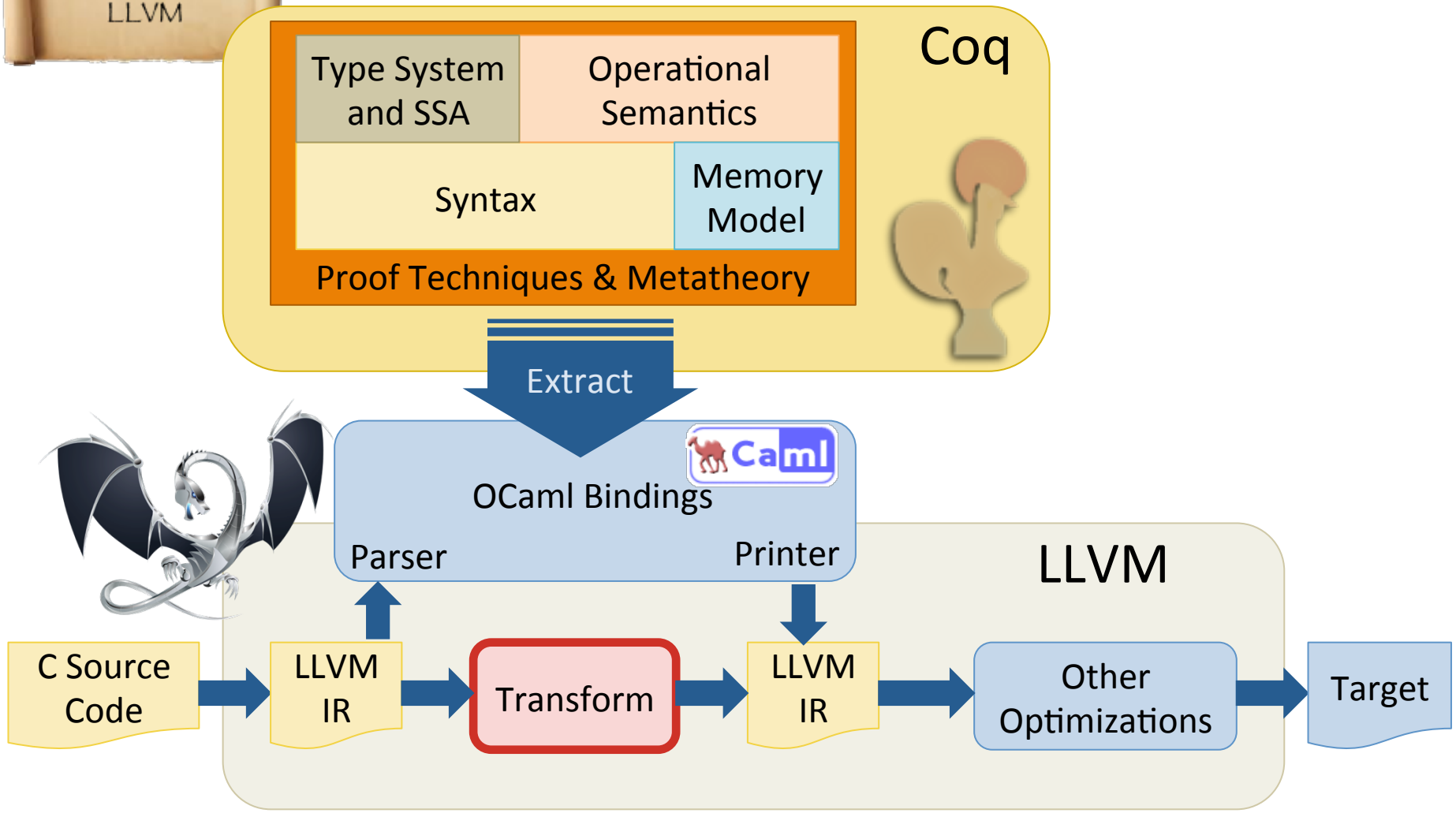
The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013]

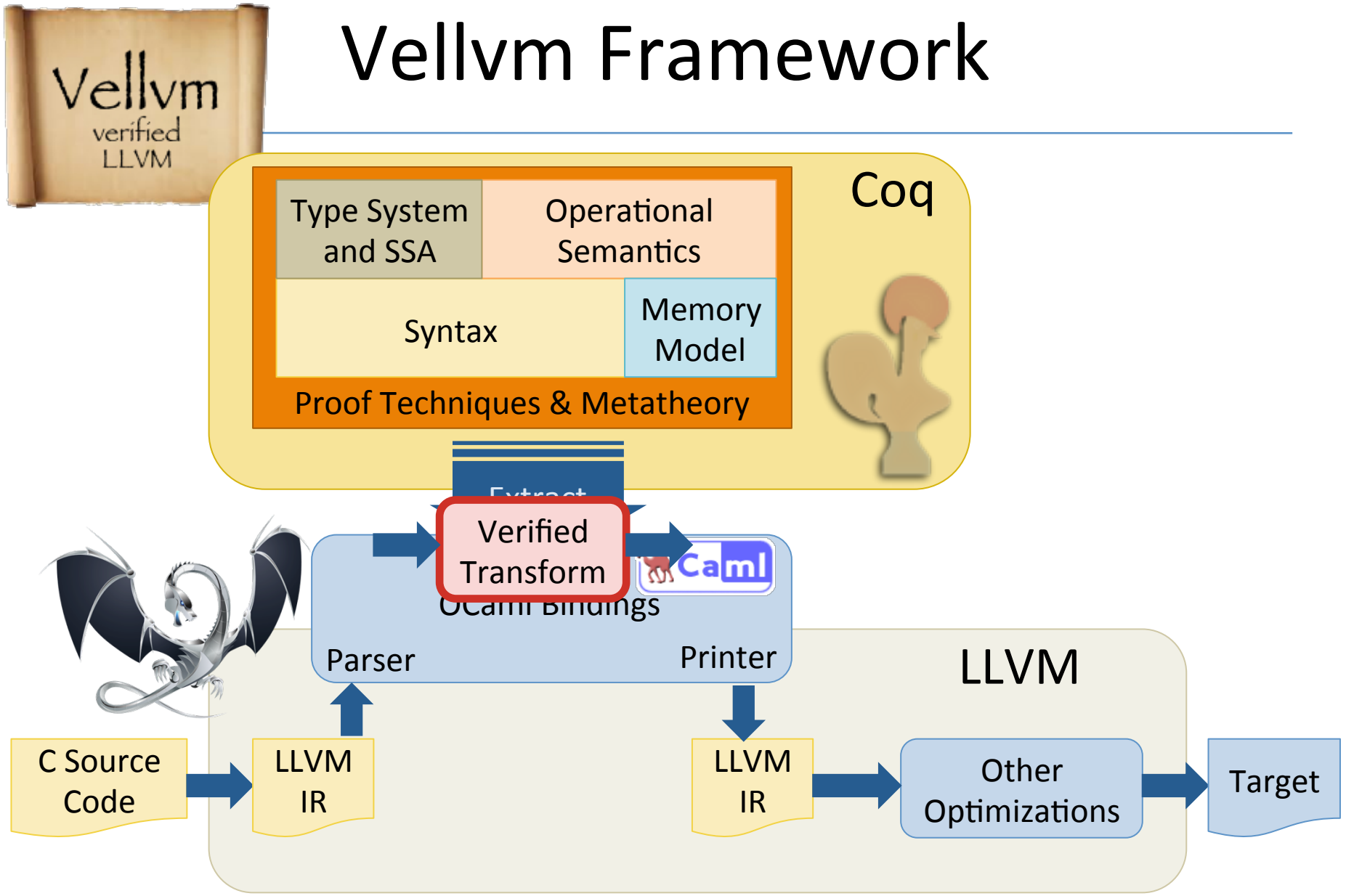


- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation

Vellvm Framework



Vellvm Framework



Plan

- Introduction to LLVM
  - static single-assignment
- Vminus: simplified SSA IR
  - Operational Semantics
  - SSA Properties
  - Static Properties
- Verified Compilation: Imp to Vminus
  - Parallel's Xavier's Imp to stack-machine compiler
  - Case study for QuickChick
  - Monotonic state (freshness!)
- Scaling up: Vellvm
  - Taste of the full LLVM IR
  - Operational Semantics
  - Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion:
  - challenges & research directions

Example LLVM Code

- LLVM offers a textual representation of its IR
  - files ending in `.ll`

factorial64.c

```
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
  int64_t acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

factorial-pretty.ll

```
define @factorial(%n) {
  %1 = alloca
  %acc = alloca
  store %n,  %1
  store 1,  %acc
  br label %start

start:
  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

else:
  %12 = load %acc
  ret %12
}
```

Real LLVM

- Decorates values with type information
  `i64`
  `i64*`
  `i1`
- Permits numeric identifiers
- Has alignment annotations
- Keeps track of entry edges for each block:
  `preds = %5, %0`

factorial64-pretty.ll

```
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2

; <label>:2                                ; preds = %5, %0
  %3 = load i64* %1, align 8
  %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                                ; preds = %2
  %6 = load i64* %acc, align 8
  %7 = load i64* %1, align 8
  %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc, align 8
  %9 = load i64* %1, align 8
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1, align 8
  br label %2

; <label>:11                               ; preds = %2
  %12 = load i64* %acc, align 8
  ret i64 %12
}
```

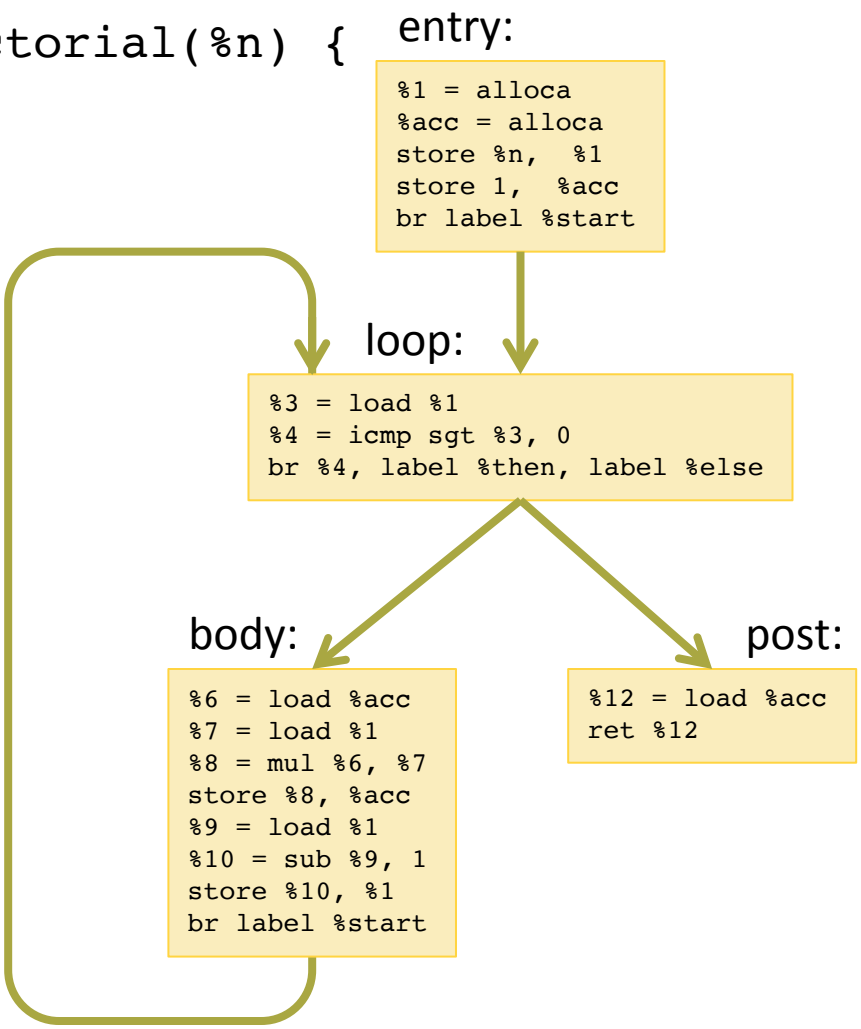
Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the "link"
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph*
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

Example Control-flow Graph

```
define @factorial(%n) {
```




```
}
```

See factorial64.ll

OPTIMIZED LLVM CODE

Static Single Assignment (SSA)

- Compiler intermediate representation developed in the late 1980's early 1990's:
  - Detecting Equality of Values in Programs
    [Alpern, Wegman, Zadeck 1988]
  - Global Value Numbers and Redundant Computations
    [Rosen, Wegman, Zadeck 1988]
  - An Efficient Method of Computing Static Single Assignment Form
    [Cytron, Ferrante, +RWZ, 1989]
  - Efficiently Computing Static Single Assignment Form and the Control Dependence Graph
    [Cytron, et. al, TOPLAS 1991]
- Makes optimizing imperative programming languages clean and efficient… by making it more purely functional
  - Used in gcc, clang, intel, Jikes, HotSpot, Open64, …

See factorial.ml

INTUITION ABOUT SEMANTICS

SSA IR's in Practice

- SSA yields an efficient representation
  - Simplifies Def-Use information needed in dataflow analysis
  - Imperative data structure to map a definition to its uses

- SSA enables good register allocation:
  - Good register allocation is (arguably) the most important optimization for performance on modern processors
  - The left-hand sides of SSA "assignments" can be thought of as "registers"
  - Register promotion − move stack-allocated data into registers

LLVM IR ⇒ Vminus

- Vastly Simplify! (For now…)

- Throw out:
  - types, complex & structured data
  - local storage allocation, complex pointers
  - functions
  - undefined values & nondeterminism

- What's left?
  - basic arithmetic
  - control flow
  - global, preallocated state (as in Imp)

Vminus by Example

Control-flow Graphs:
+ Labeled blocks

Vminus by Example

```
entry:
  r0 = ...
  r1 = ...
  r2 = ...
```

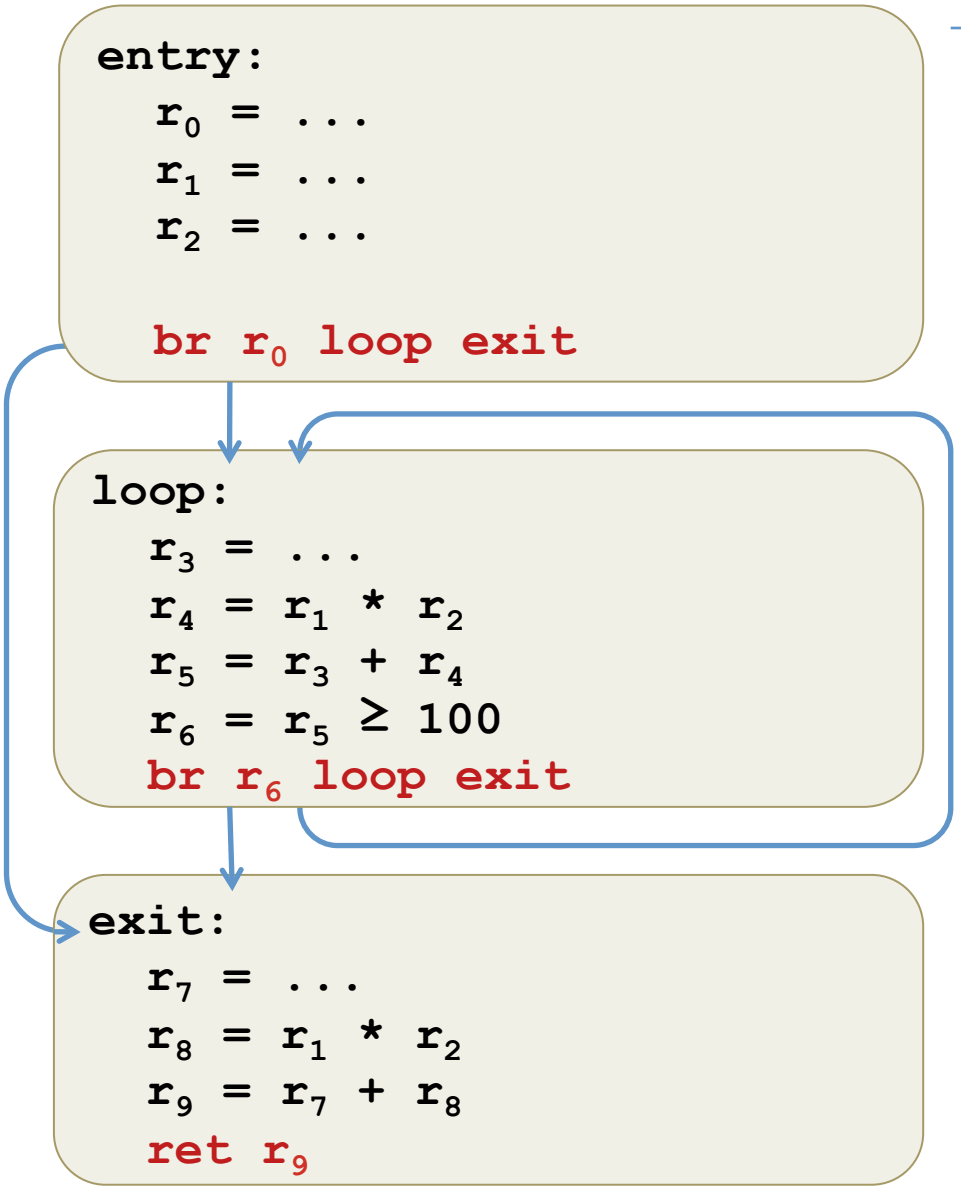
```
loop:
  r3 = ...
  r4 = r1 * r2
  r5 = r3 + r4
  r6 = r5 ≥ 100
```

```
exit:
  r7 = ...
  r8 = r1 * r2
  r9 = r7 + r8
```

Control-flow Graphs:

+ Labeled blocks
+ Binary Operations

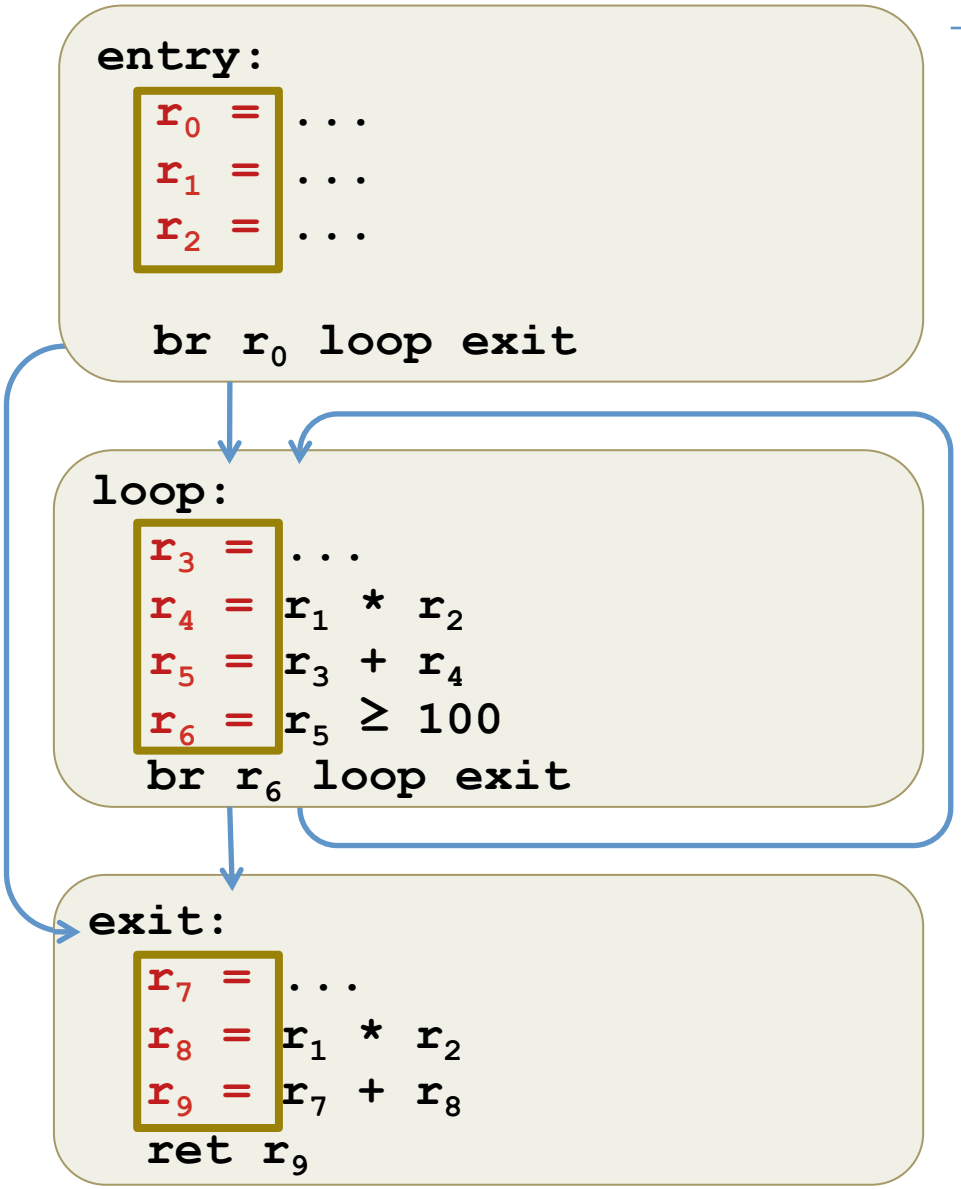
Vminus by Example



Control-flow Graphs:

+ Labeled blocks
+ Binary Operations
+ Branches/Return

Vminus by Example

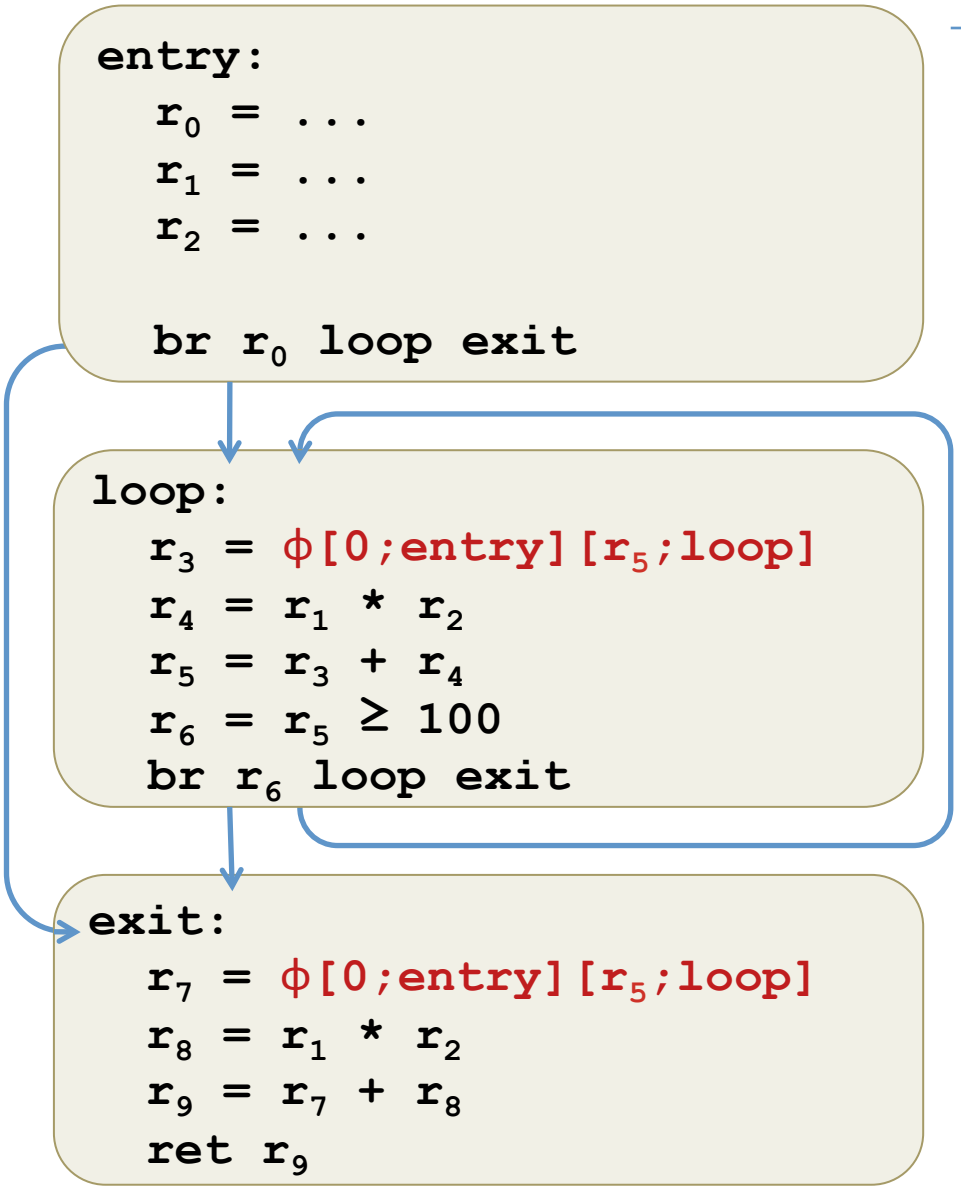


Control-flow Graphs:
+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment

(each *local identifier* assigned only *once*, statically)

local identifier a.k.a. uid or SSA variable

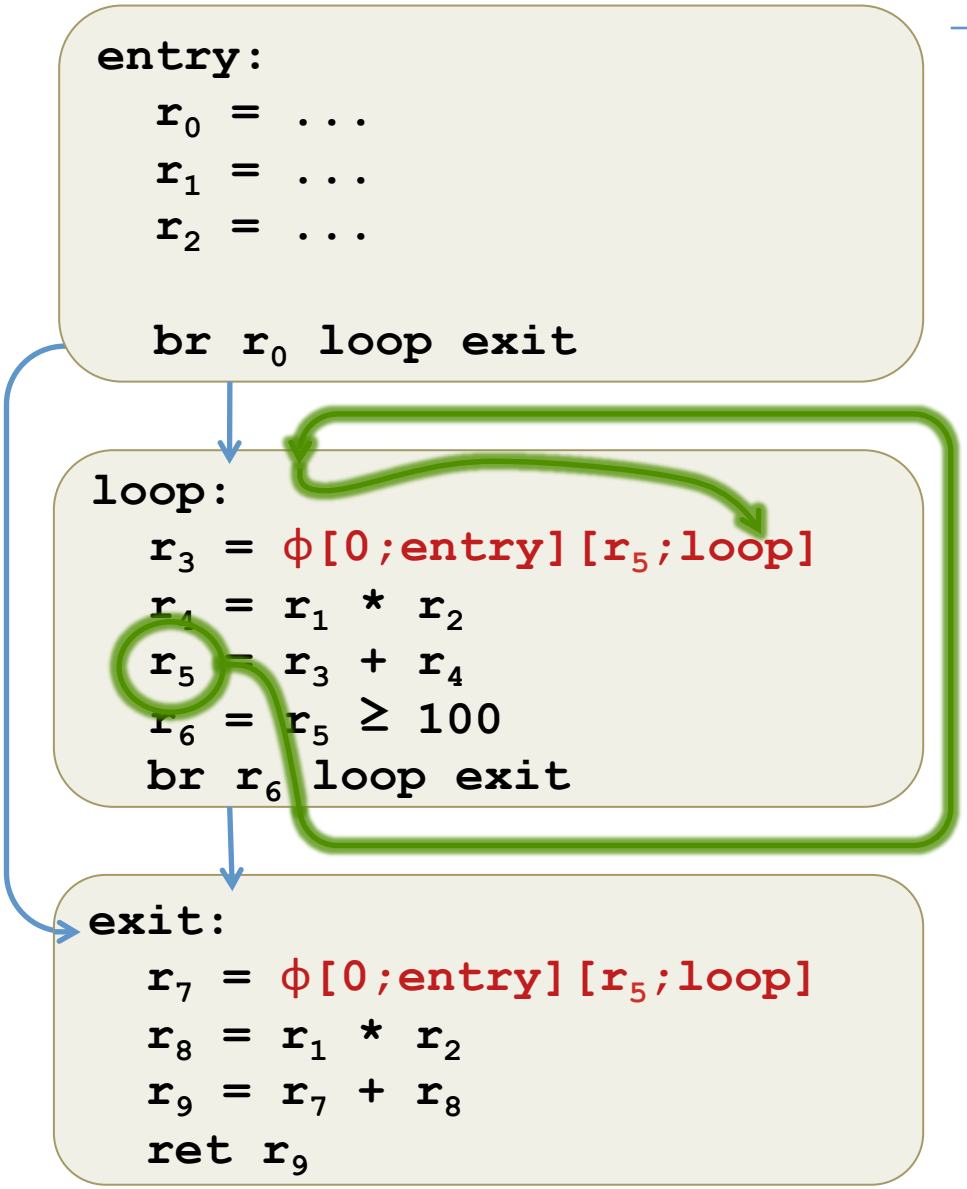
Vminus by Example



Control-flow Graphs:

+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment
+ ϕ nodes

Vminus by Example



Control-flow Graphs:

+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment
+ ϕ nodes

(choose values based on predecessor blocks)

Vminus.v

CFG.v

ListCFG.v

VMINUS SYNTAX

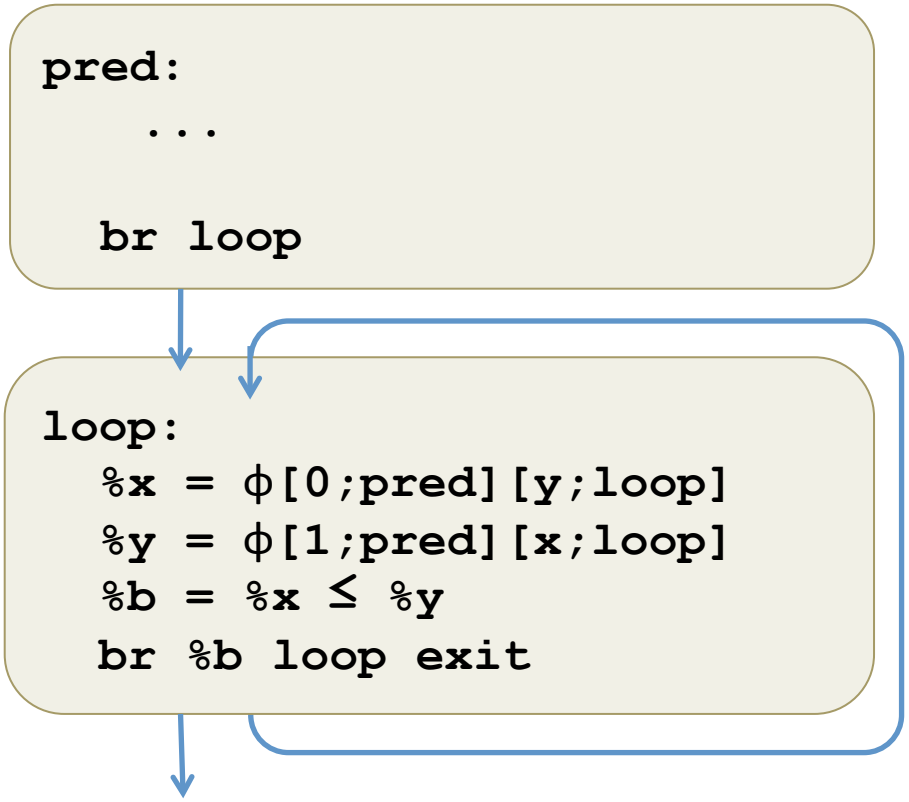
Vminus Operational Semantics

- Only 5 kinds of instructions:
  - Binary arithmetic
  - Memory Load
  - Memory Store
  - Terminators
  - Phi nodes

- What is the state of a Vminus program?

Subtlety of Phi Nodes

- Phi-Nodes admit “cyclic” dependencies:



Semantics of Phi Nodes

- The value of the RHS of a phi-defined uid is relative to the state at the entry to the block.

- Option 1:
  - Require all phi nodes to be at the beginning of the block
  - Execute them “atomically, in parallel”
  - (Original Vellvm followed this model)

- Option 2:
  - Keep track of the state upon entry to the block
  - Calculate the RHS of phi nodes relative to the entry state
  - (Vminus follows this model)