Vellvm: Verifying Transformations of the LLVM IR

Steve Zdancewic
Jianzhou Zhao
Milo M.K. Martin
University of Pennsylvania

Santosh Nagarakatte
Rutgers University

Motivation: SoftBound/CETS

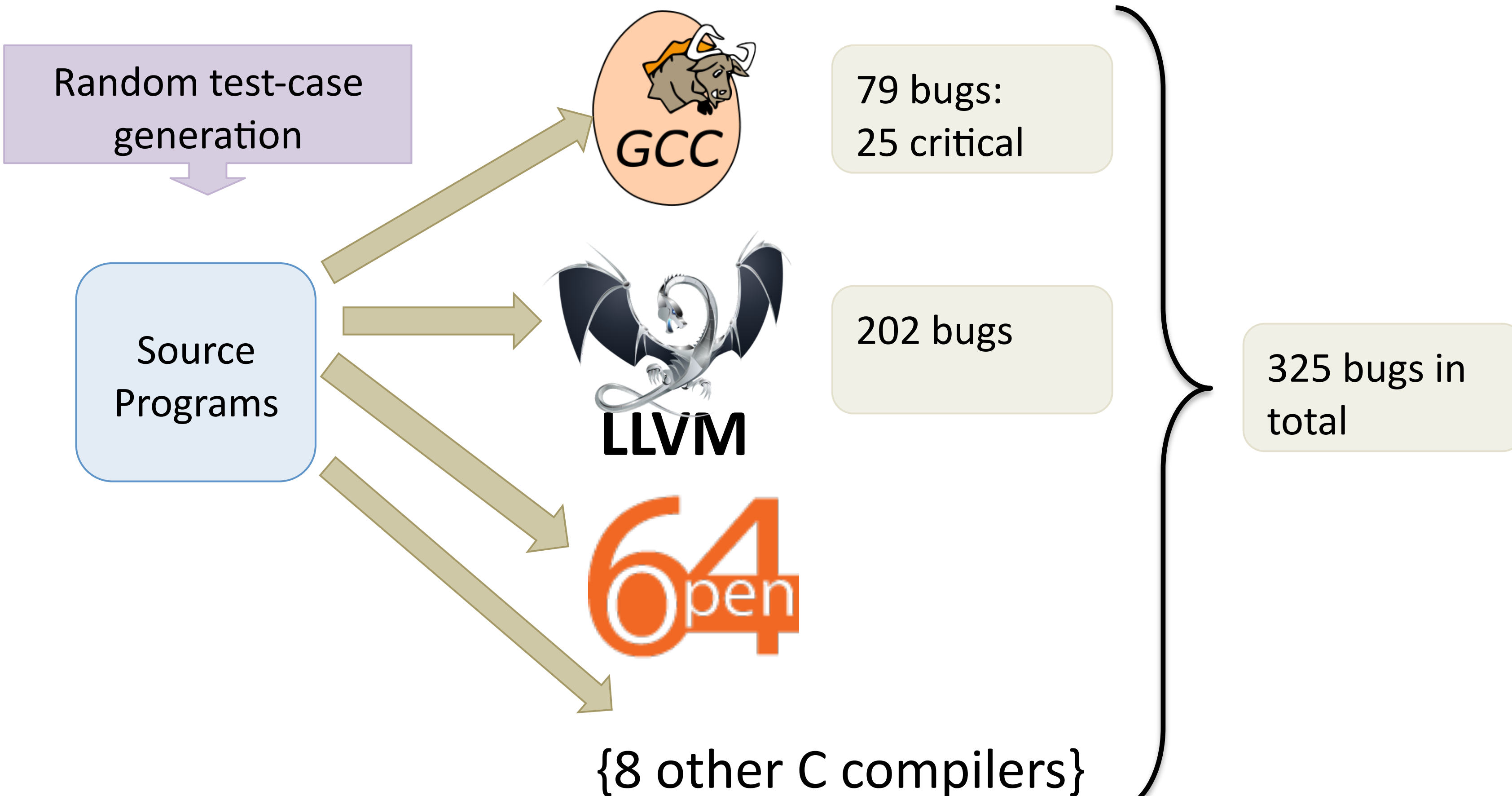
[Nagarakatte, et al. *PLDI ’09, ISMM ‘10*]

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

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

Motivation:Compiler Bugs

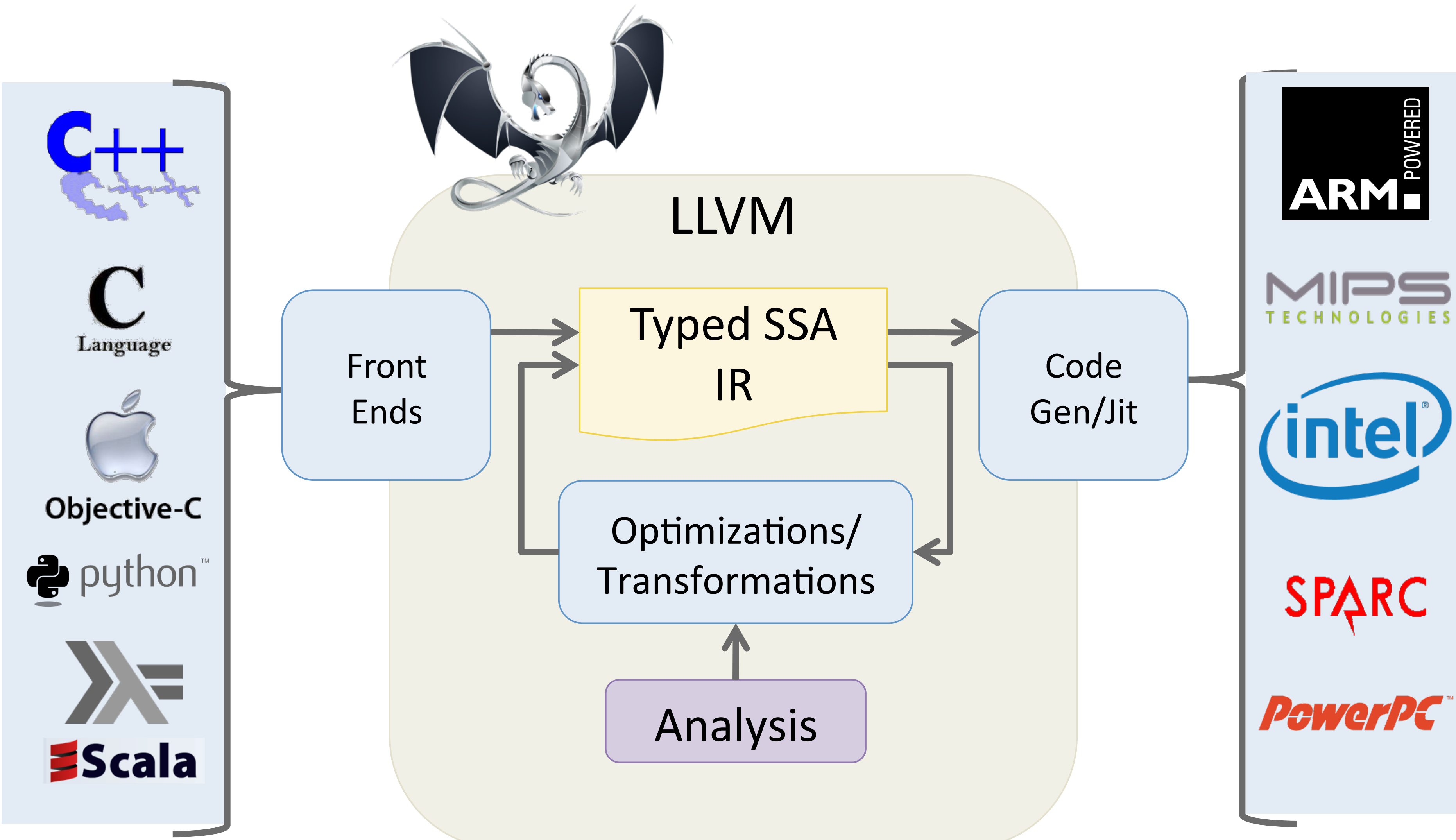
[Yang et al. PLDI 2011]



Verified Compilation: Compcert [Leroy et al.]
(Not directly applicable to LLVM)

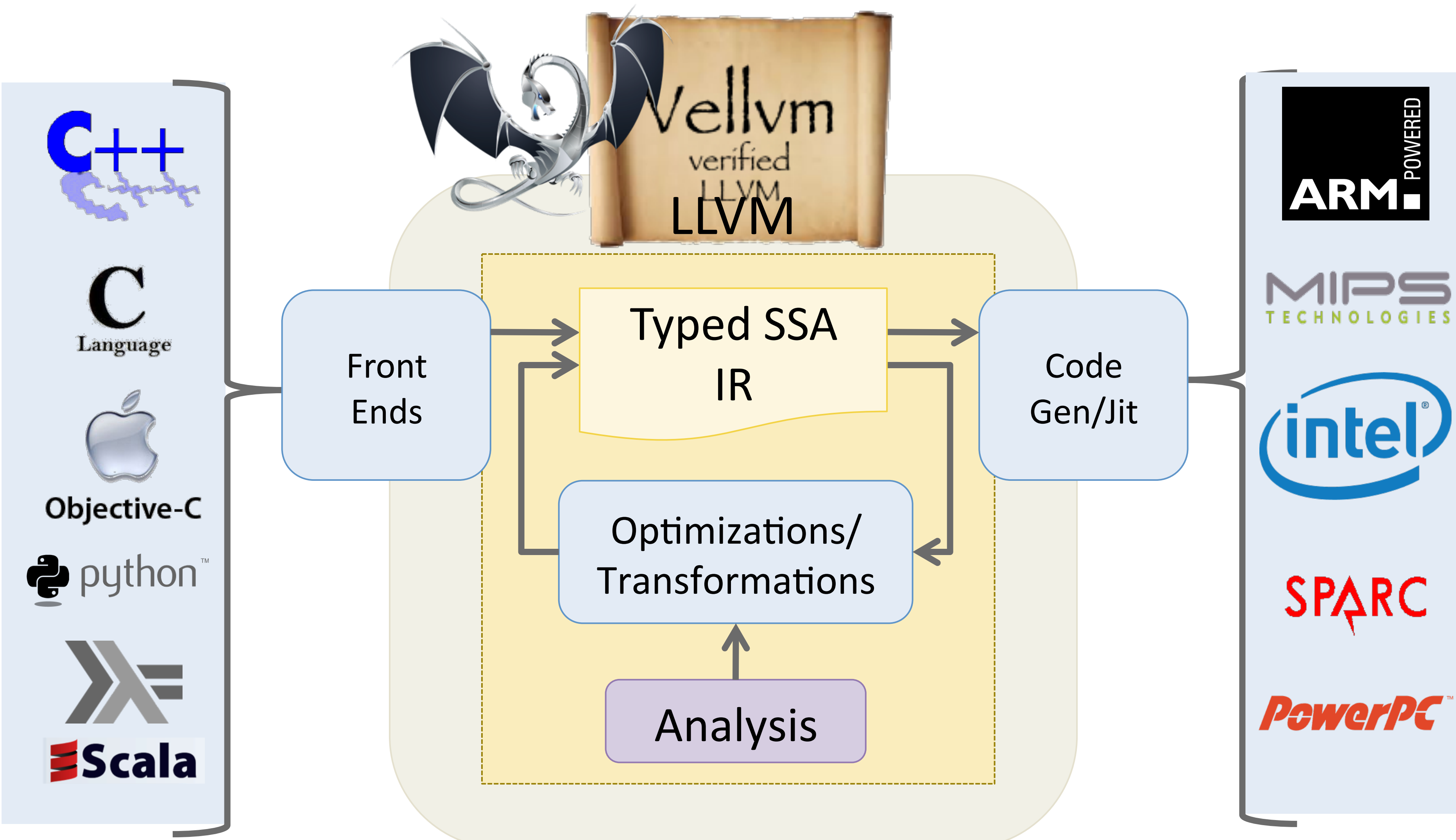
LLVM Compiler Infrastructure

[Lattner et al.]



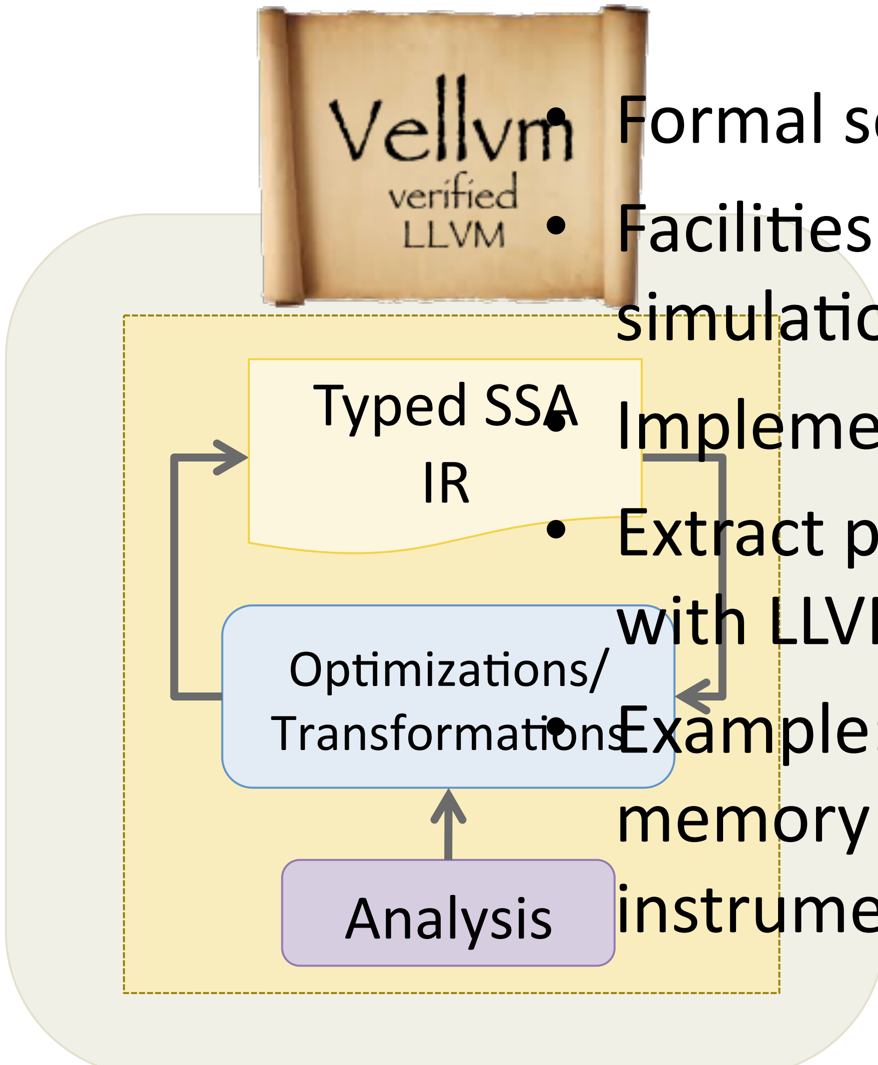
LLVM Compiler Infrastructure

[Lattner et al.]



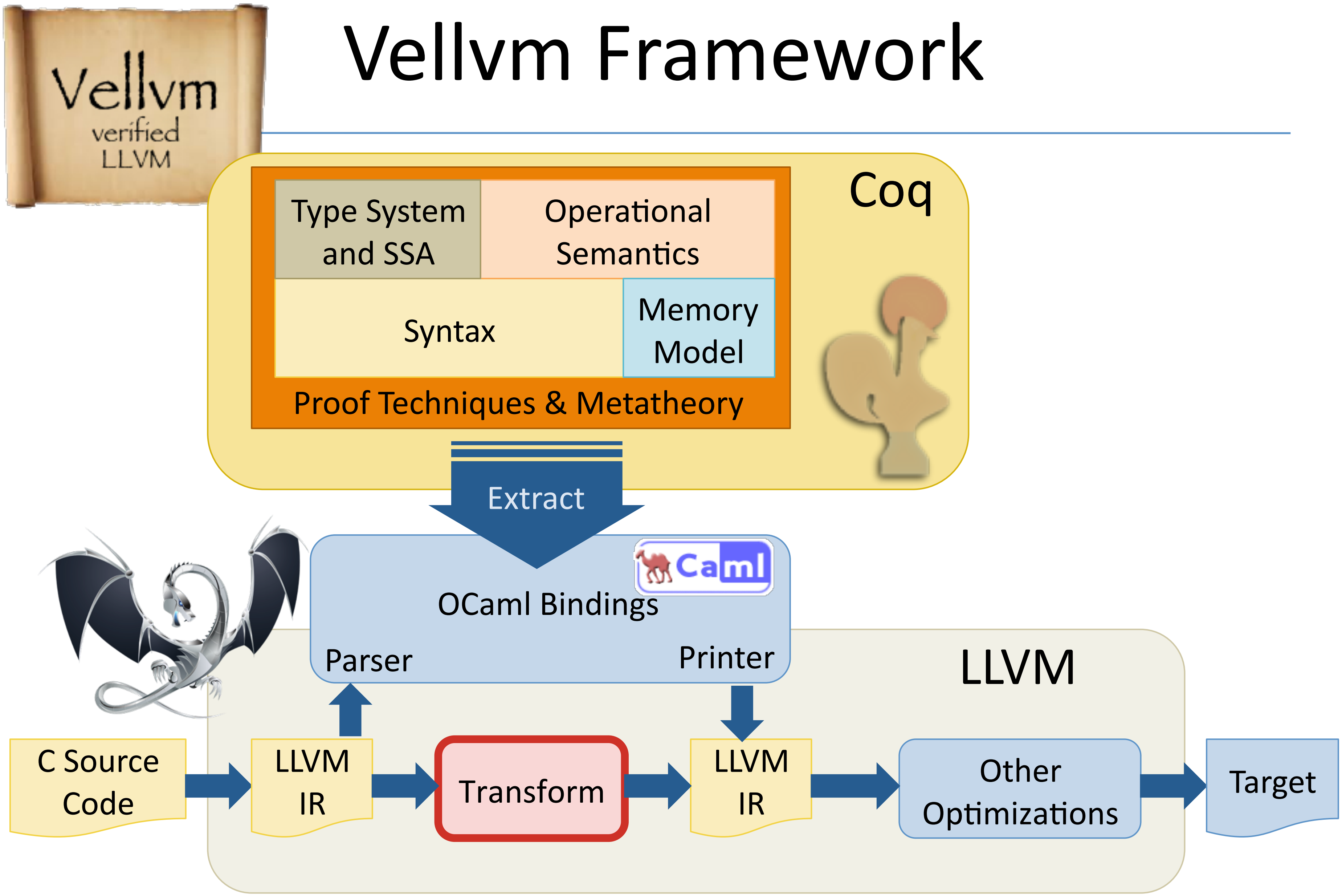
The Vellvm Project

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

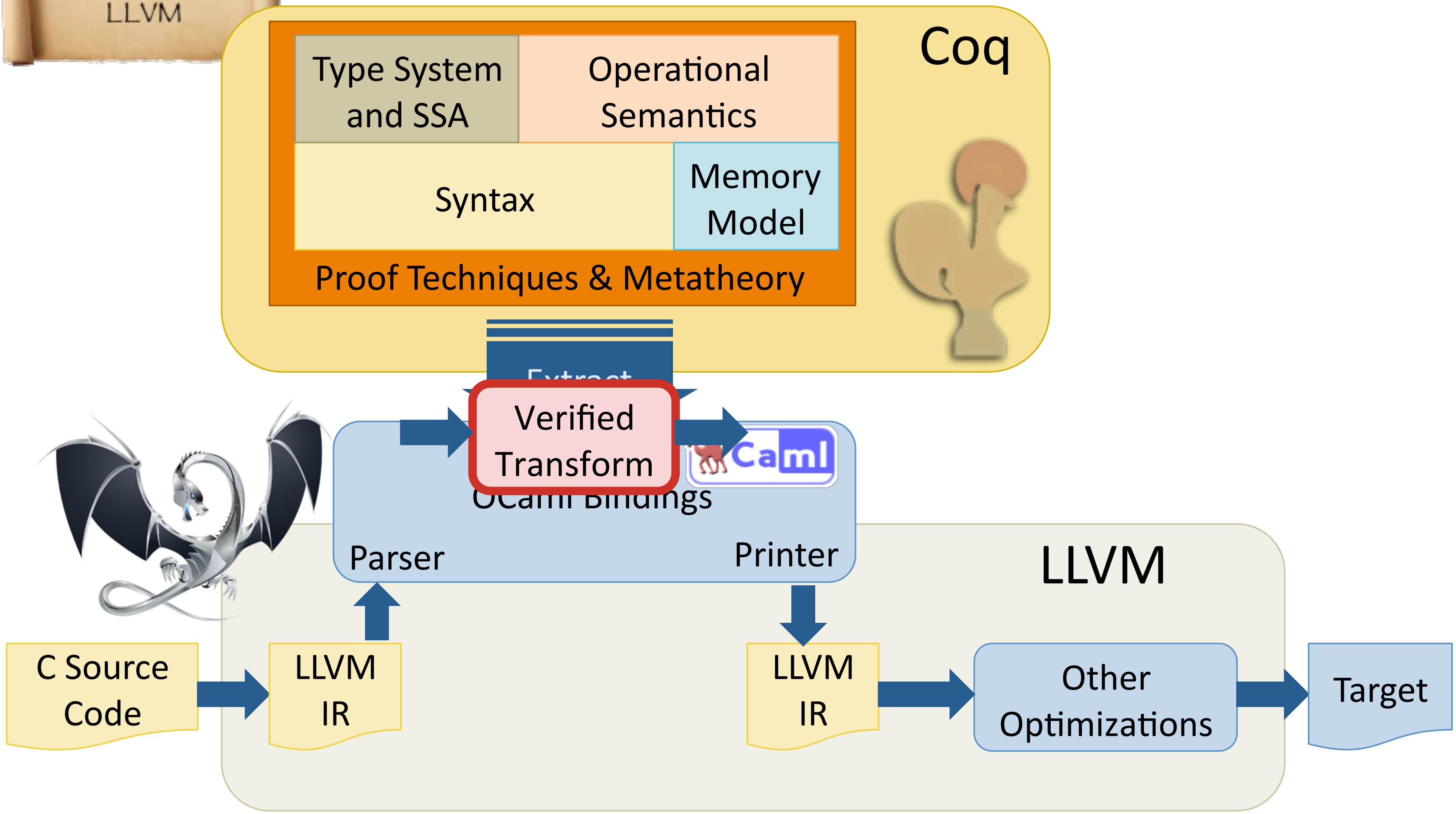


- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation

Vellvm Framework



Vellvm Framework



Plan

- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

LLVM IR by Example

```
entry:
```

```
loop:
```

```
exit:
```

Control-flow Graphs:
+ Labeled blocks

LLVM IR by Example

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

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

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

Control-flow Graphs:

+ Labeled blocks
+ Binary Operations

LLVM IR by Example

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

   br r0 loop exit
```

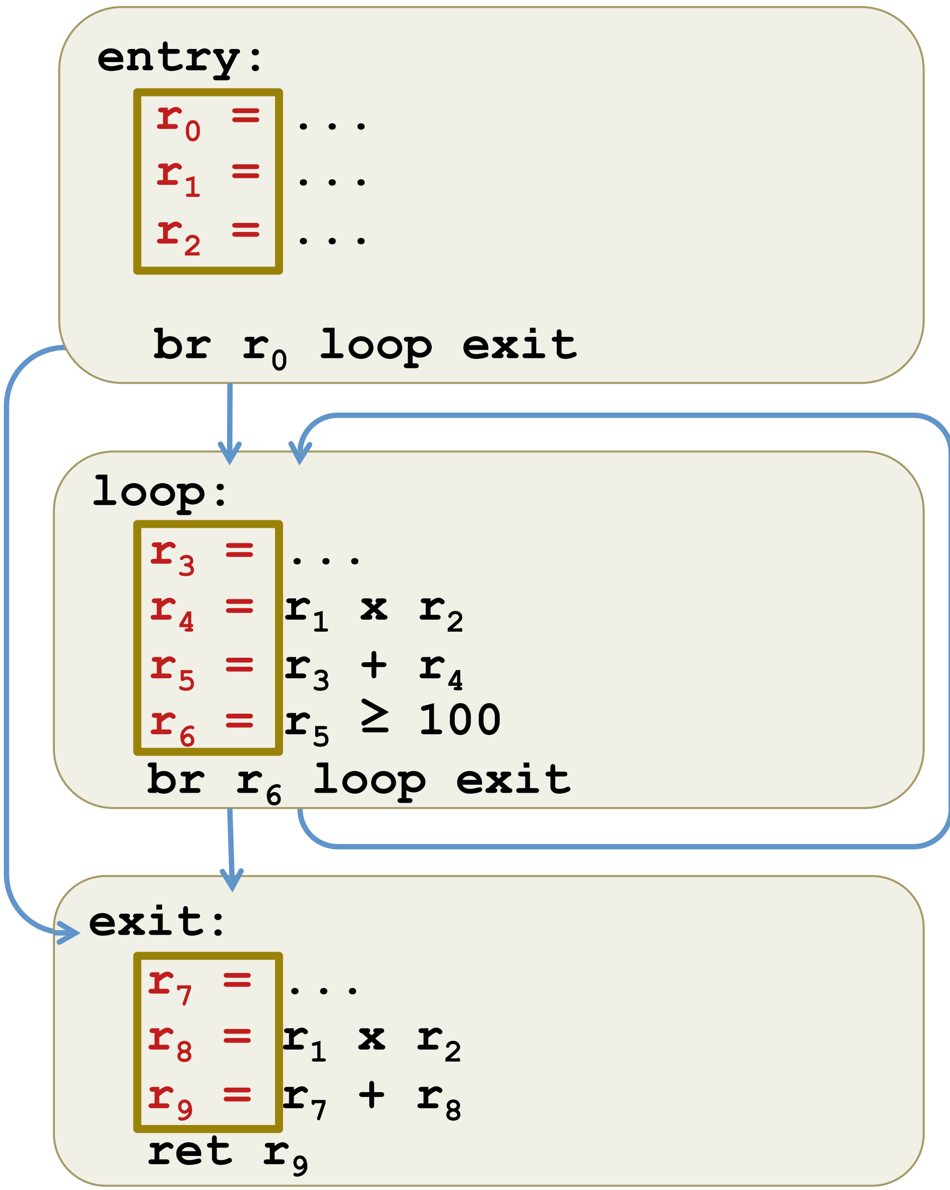
```
loop:
   r3 = ...
   r4 = r1 x r2
   r5 = r3 + r4
   r6 = r5 ≥ 100
   br r6 loop exit
```

```
exit:
   r7 = ...
   r8 = r1 x r2
   r9 = r7 + r8
   ret r9
```

Control-flow Graphs:

+ Labeled blocks
+ Binary Operations
+ Branches/Return

LLVM IR by Example



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

(each variable assigned only *once*, statically)

LLVM IR by Example

entry:

r_0 = ...

r_1 = ...

r_2 = ...

br r_0 loop exit

loop:

r_3 = ϕ[0;entry][r_5;loop]

r_4 = r_1 x r_2

r_5 = r_3 + r_4

r_6 = r_5 $\geq$ 100

br r_6 loop exit

exit:

r_7 = ϕ[0;entry][r_5;loop]

r_8 = r_1 x r_2

r_9 = r_7 + r_8

ret r_9

Control-flow Graphs:

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

LLVM IR by Example

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

   br r0 loop exit
```

```
loop:
   r3 = φ[0;entry][r5;loop]
   r4 = r1 x r2
   r5 = r3 + r4
   r6 = r5 ≥ 100
   br r6 loop exit
```

```
exit:
   r7 = φ[0;entry][r5;loop]
   r8 = r1 x r2
   r9 = r7 + r8
   ret r9
```

Control-flow Graphs:

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

(choose values based on predecessor blocks)

Plan

- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

Structured Data in LLVM

- LLVM's IR is uses types to describe the structure of data.

```
ty ::=
  | i1 | i8 | i32 |…            N-bit integers
  | [<#elts> x t]               arrays
  | r (ty1, ty2, … , tyn)       function types
  | {ty1, ty2, … , tyn}         structures
  | ty*                         pointers
  | %Tident                     named (identified) type

r ::=          Return Types
    ty         first-class type
    void       no return value
```

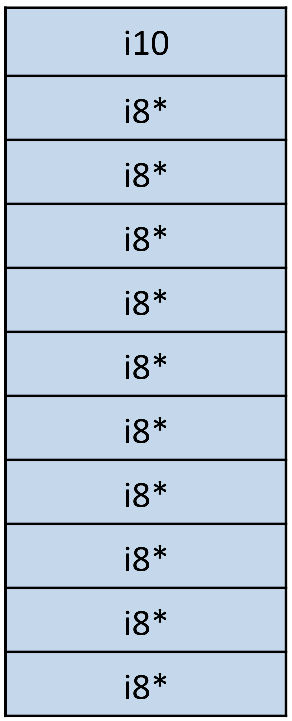
- <#elts> is an integer constant >= 0
- (Recursive) Structure types can be named at the top level:

```
%T1 = type {ty1, ty2, … , tyn}
```

LLVM’s memory model

```
%ST = type {i10,[10 x i8*]}
```

High-level Representation

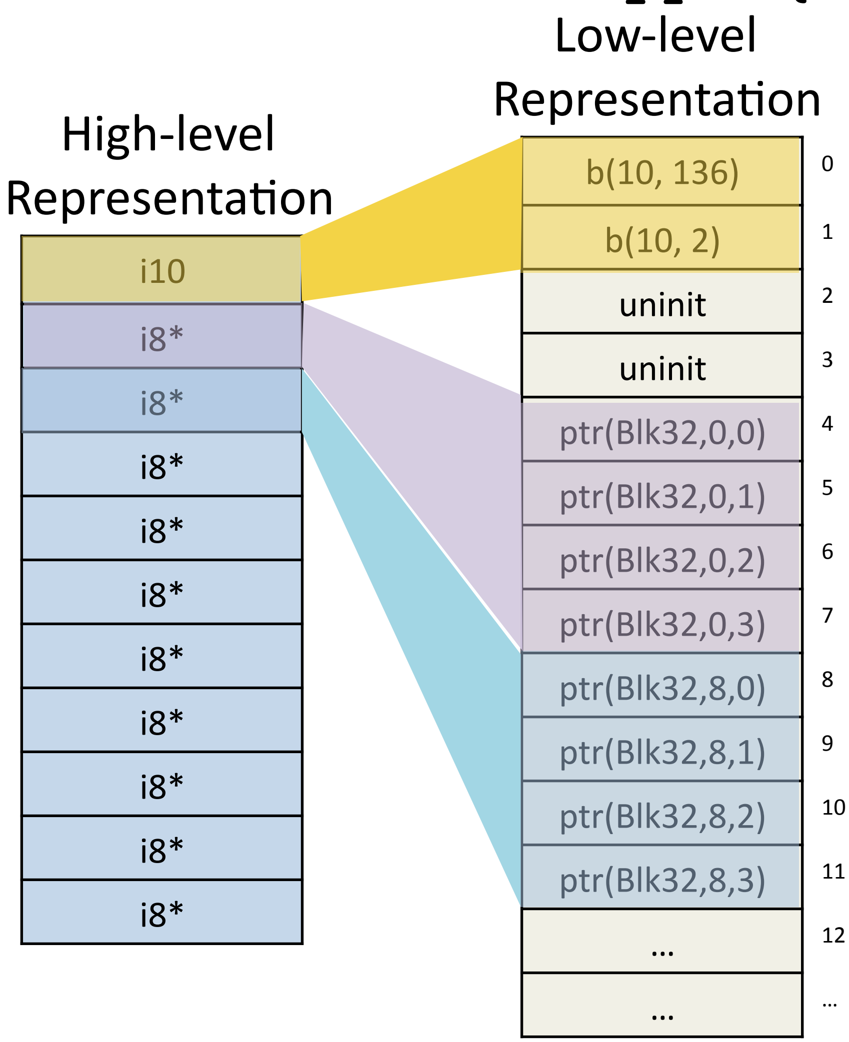


- Manipulate structured types.

```
%val = load %ST* %ptr
…
store %ST* %ptr, %new
```

LLVM's memory model

```
%ST = type {i10,[10 x i8*]}
```

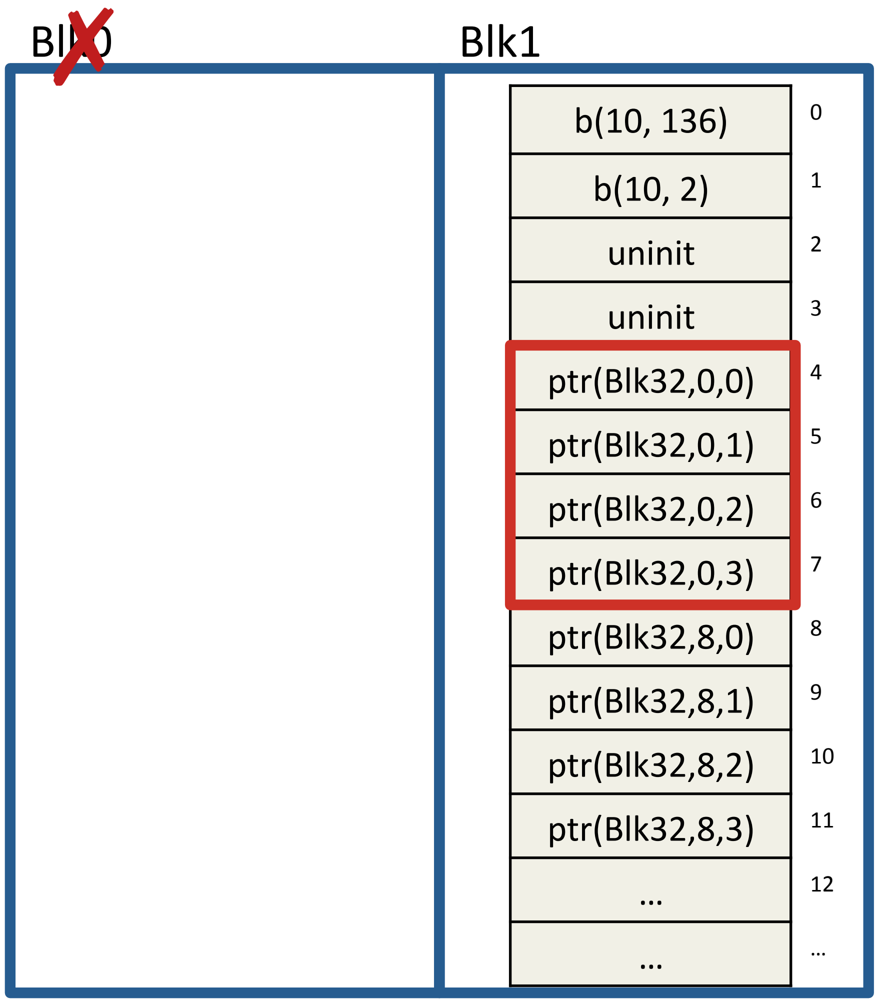


- Manipulate structured types.

```
%val = load %ST* %ptr
…
store %ST* %ptr, %new
```

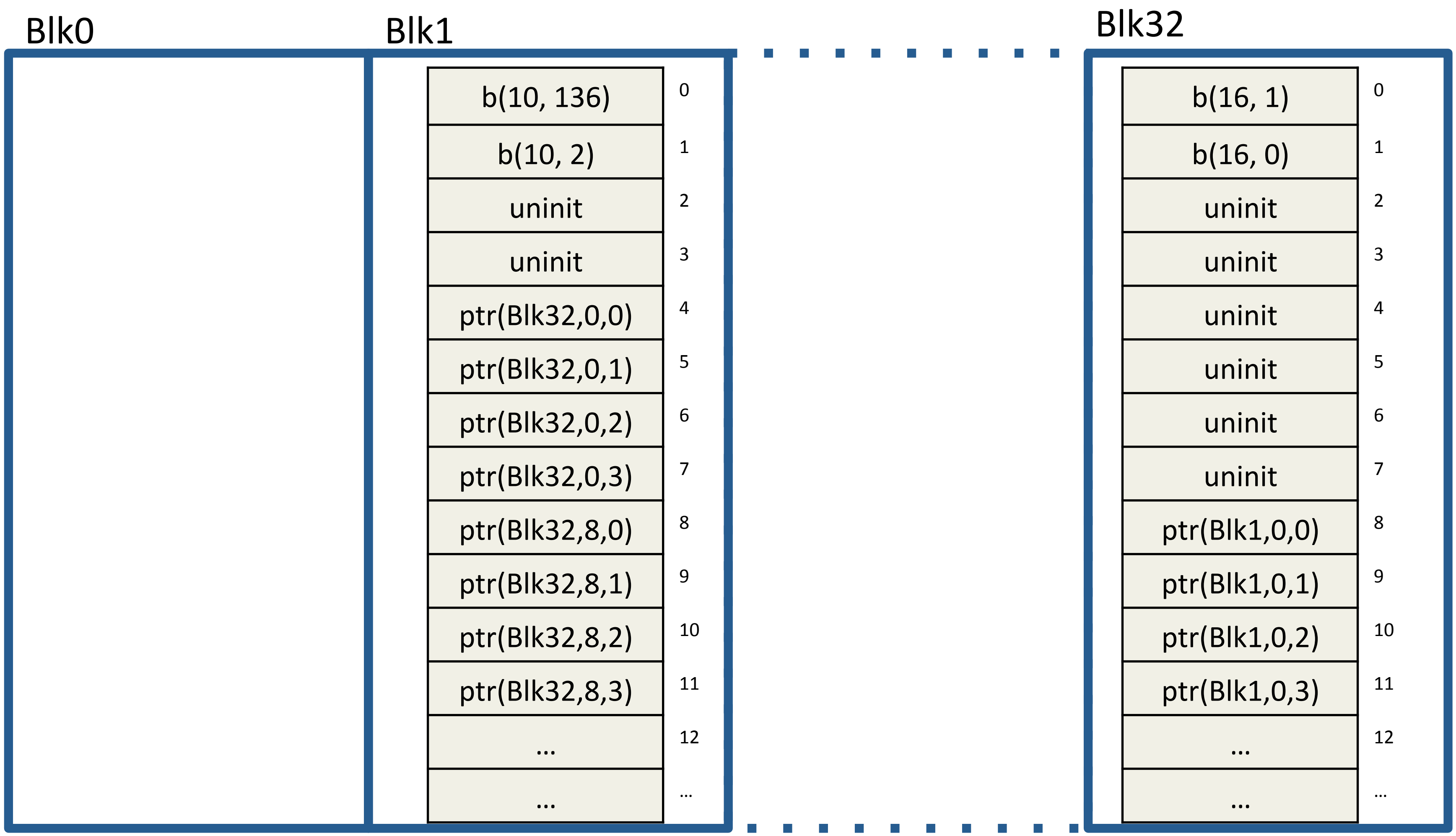
- Semantics is given in terms of byte-oriented low-level memory.
  - padding & alignment
  - physical subtyping

Adapting CompCert's Memory Model



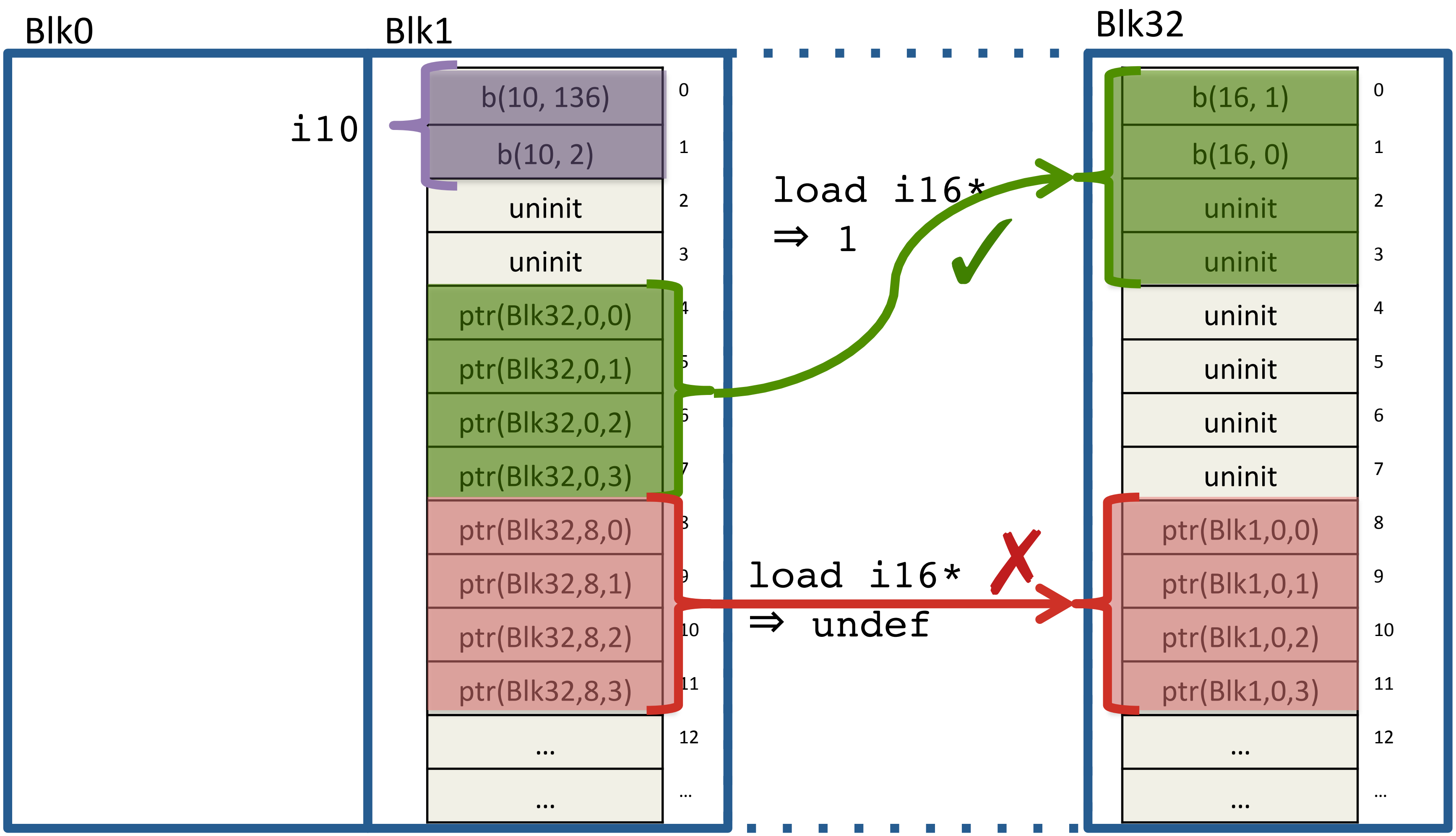
- Code lives in blocks
- Represent pointers abstractly
  - block + offset
- Deallocate by invalidating blocks
- Allocate by creating new blocks
  - infinite memory available

Adapting CompCert's Memory Model



Dynamic Physical Subtyping

[Nita, et al. *POPL '08*]



Sources of Undefined Behavior

Target-dependent Results

- Uninitialized variables:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage

Nondeterminism

Fatal Errors

- Out-of-bounds accesses
- Access dangling pointers
- Free invalid pointers
- Invalid indirect calls

Stuck States

Sources of Undefined Behavior

Target-dependent Results

- Uninitialized variables:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage

Nondeterminism

Defined by a predicate on the program configuration.

$$
\begin{aligned}
\mathrm{Stuck}(f, \sigma) = \; & \mathrm{BadFree}(f, \sigma) \\
& \vee\ \mathrm{BadLoad}(f, \sigma) \\
& \vee\ \mathrm{BadStore}(f, \sigma) \\
& \vee\ \ldots \\
& \vee\ \ldots 0
\end{aligned}
$$

Stuck States

`undef`

- What is the value of `%y` after running the following?

```
%x = or i8 undef, 1
%y = xor i8 %x %x
```

- One plausible answer: 0
- Not LLVM's semantics!

  (LLVM is more liberal to permit more aggressive optimizations)

undef

- Partially defined values are interpreted *nondeterministically* as sets of possible values:

```
%x = or i8 undef, 1
%y = xor i8 %x %x
```

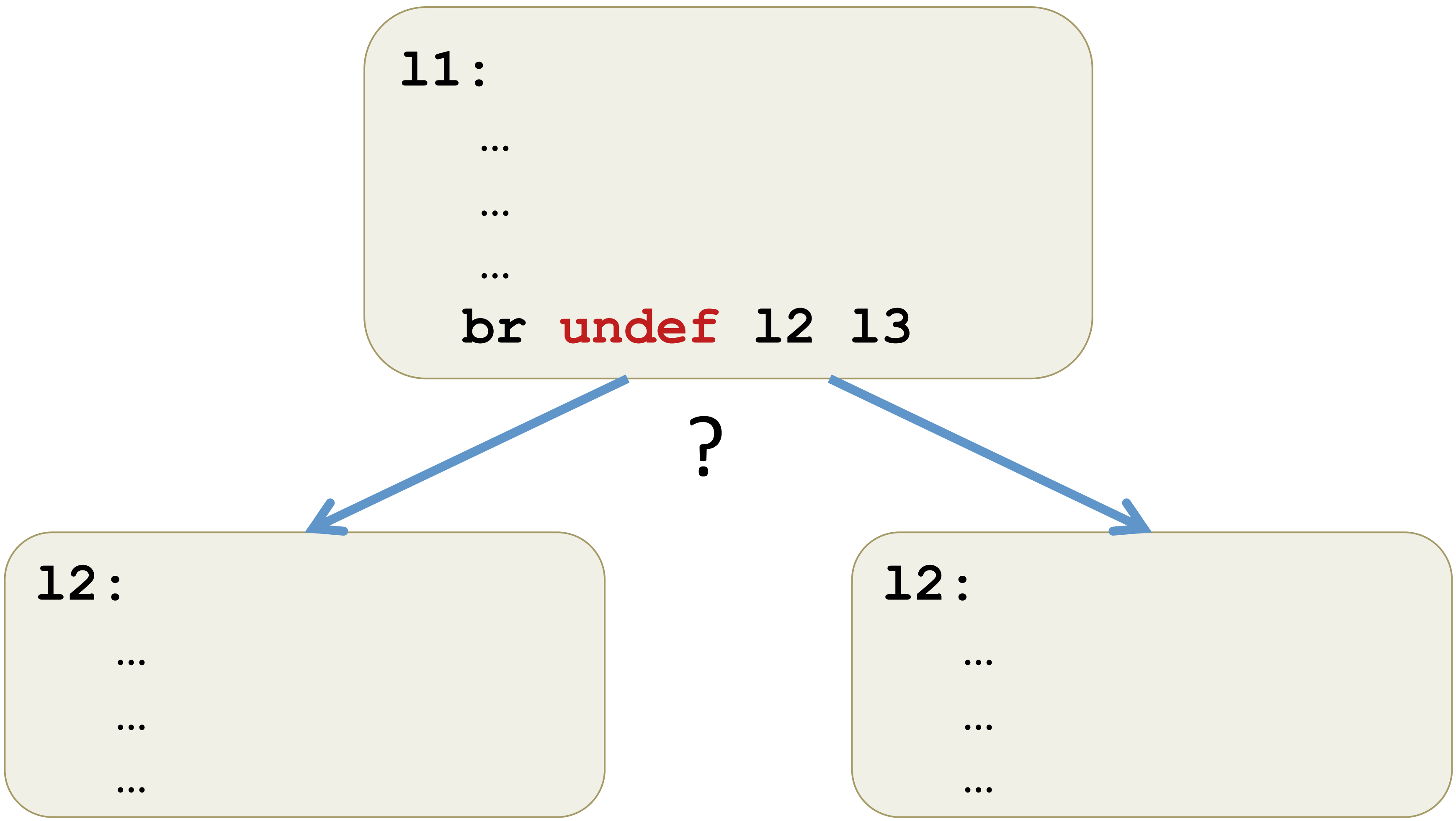
$$[\![\texttt{i8 undef}]\!] = \{0,\ldots,255\}$$

$$[\![\texttt{i8 1}]\!] = \{1\}$$

$$[\![\%x]\!] = \{a \text{ or } b \mid a \in [\![\texttt{i8 undef}]\!],\ b \in [\![1]\!]\}$$
$$= \{1,3,5,\ldots,255\}$$

$$[\![\%y]\!] = \{a \text{ xor } b \mid a \in [\![\%x]\!],\ b \in [\![\%x]\!]\}$$
$$= \{0,2,4,\ldots,254\}$$

Nondeterministic Branches



LLVM_{ND} Operational Semantics

- Define a transition relation:

$$f \vdash \sigma_1 \longmapsto \sigma_2$$

  - f is the program
  - σ is the program state: pc, locals(δ), stack, heap
- Nondeterministic
  - δ maps local `%uids` to sets.
  - Step relation is nondeterministic
- Mostly straightforward (given the heap model)
  - One wrinkle: phi-nodes exectuted atomically

Operational Semantics

	Small Step	Big Step
Nondeterministic	LLVM_{ND}	
Deterministic		

Deterministic Refinement

	Small Step	Big Step
Nondeterministic	$LLVM_{ND}$	
Deterministic	⋓ $LLVM_D$	

Instantiate ‘undef’ with default value (0 or null) ⇒ deterministic.

Big-step Deterministic Refinements

	Small Step	**Big Step**
Nondeterministic	LLVM_{ND} $\pitchfork$	
Deterministic	$\mathrm{LLVM}_{Interp} \approx \mathrm{LLVM}_{D}$	

Bisimulation up to "observable events":

- external function calls

Big-step Deterministic Refinements

	Small Step	Big Step
Nondeterministic	LLVM_{ND} $\uplus$	
Deterministic	$\mathrm{LLVM}_{Interp} \approx \mathrm{LLVM}_{D}$	$\gtrsim \mathrm{LLVM}^{*}_{DFn} \gtrsim \mathrm{LLVM}^{*}_{DB}$

Simulation up to "observable events":

- useful for encapsulating behavior of function calls
- large step evaluation of basic blocks

[Tristan, et al. *POPL '08*, Tristan, et al. *PLDI '09*]

Plan

- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

Reasoning about SSA Transforms

- Dynamic semantics of LLVM
  - Memory model
  - Nondeterminism
  - Handle groups of phi-nodes atomically

- Static semantics of LLVM
  - Computing dominators is crucial

  [Zhao, et al. *POPL '12*] [Zhao & Zdancewic *CPP '12*]

- Use them to justify correctness of program transformations
  - Simulation proofs

Key SSA Invariant

entry:
r_0 = ...
r_1 = ...
r_2 = ...

br r_0 **loop exit**

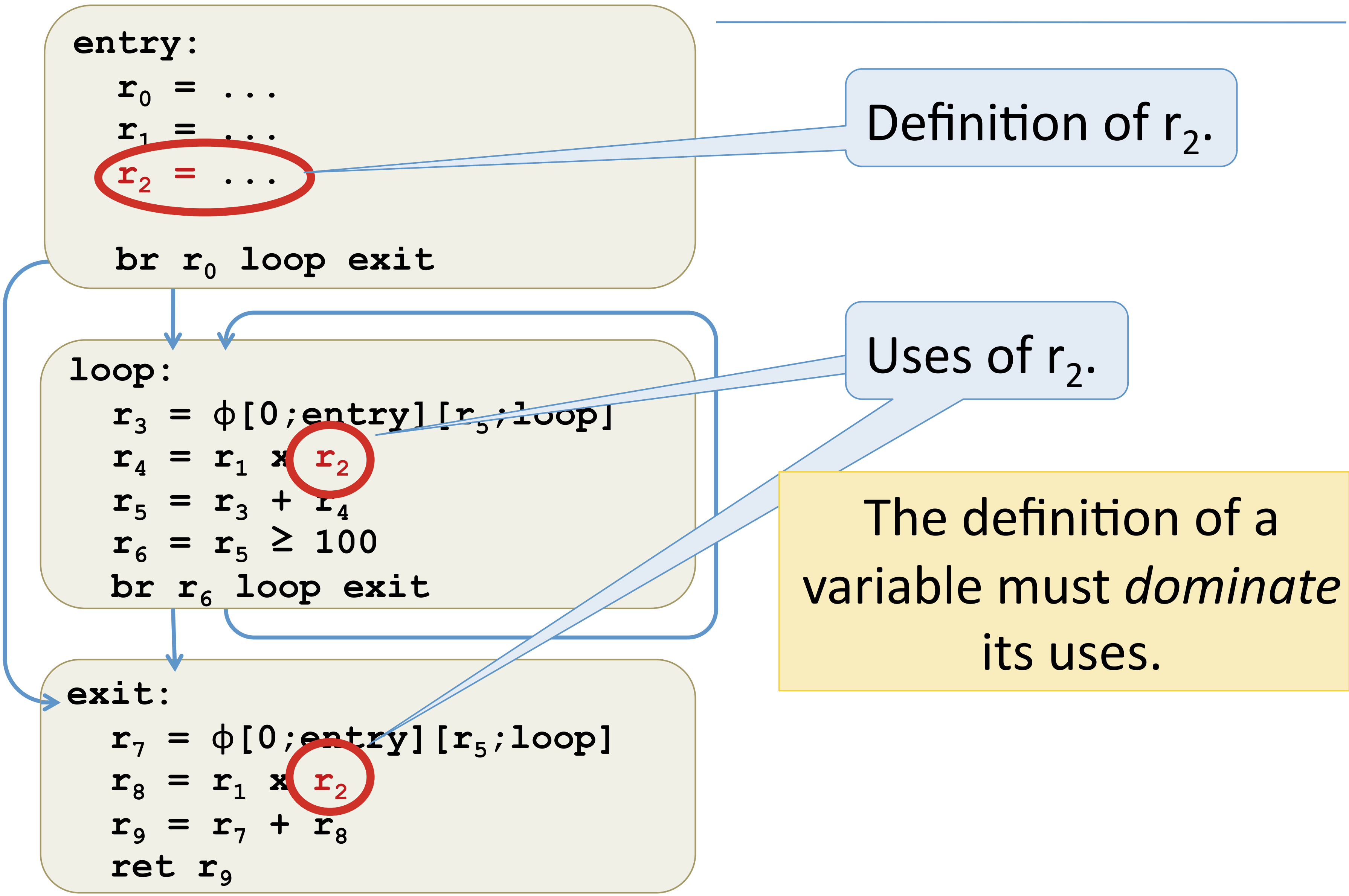
loop:
$r_3 = \phi[0;\text{entry}][r_5;\text{loop}]$
$r_4 = r_1 \times r_2$
$r_5 = r_3 + r_4$
$r_6 = r_5 \geq 100$
br r_6 **loop exit**

exit:
$r_7 = \phi[0;\text{entry}][r_5;\text{loop}]$
$r_8 = r_1 \times r_2$
$r_9 = r_7 + r_8$
ret r_9

Definition of r_2.

Uses of r_2.

Key SSA Invariant



Safety Properties

- A well-formed program never accesses undefined variables.

$$\text{If } \vdash f \text{ and } f \vdash \sigma_0 \longmapsto^* \sigma \text{ then } \sigma \text{ is not stuck.}$$

$\vdash f$	program f is well formed
σ	program state
$f \vdash \sigma \longmapsto^* \sigma$	evaluation of f

- *Initialization*: $\text{If } \vdash f \text{ then } wf(f, \sigma_0).$

- *Preservation*:

$$\text{If } \vdash f \text{ and } f \vdash \sigma \longmapsto \sigma' \text{ and } wf(f, \sigma) \text{ then } wf(f, \sigma')$$

- *Progress*:

$$\text{If } \vdash f \text{ and } wf(f, \sigma) \text{ then } f \vdash \sigma \longmapsto \sigma'$$

Safety Properties

- A well-formed program never accesses undefined variables.

If $\vdash f$ and $f \vdash \sigma_0 \longmapsto^* \sigma$ then σ is not stuck.

$\vdash f$	program f is well formed
σ	program state
$f \vdash \sigma \longmapsto^* \sigma$	evaluation of f

- *Initialization*: If $\vdash f$ then $wf(f, \sigma_0)$.
- *Preservation*:

If $\vdash f$ and $f \vdash \sigma \longmapsto \sigma'$ and $wf(f, \sigma)$ then $wf(f, \sigma')$

- *Progress*:

If $\vdash f$ and $wf(f, \sigma)$ then $done(f,\sigma)$ or $stuck(f,\sigma)$ or $f \vdash \sigma \longmapsto \sigma'$

Well-formed States

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

  br r0 loop exit

loop:
  r3 = φ[0;entry][r5;loop]
  r4 = r1 x r2
  r5 = r3 + r4
  r6 = r5 ≥ 100
  br r6 loop exit

exit:
  r7 = φ[0;entry][r5;loop]
  r8 = r1 x r2
  r9 = r7 + r8
  ret r9
```

pc

State σ is:

- pc = program counter
- δ = local values

Well-formed States

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

  br r0 loop exit
```

```
loop:
  r3 = φ[0;entry][r5;loop]
  r4 = r1 x r2
  r5 = r3 + r4
  r6 = r5 ≥ 100
  br r6 loop exit
```

pc

```
exit:
  r7 = φ[0;entry][r5;loop]
  r8 = r1 x r2
  r9 = r7 + r8
  ret r9
```

State σ is:

- pc = program counter
- δ = local values

sdom(f,pc) = variable defns. that *strictly dominate* pc.

Well-formed States

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

  br r0 loop exit

loop:
  r3 = ϕ[0;entry][r5;loop]
  r4 = r1 x r2
  r5 = r3 + r4
  r6 = r5 ≥ 100
  br r6 loop exit

exit:
  r7 = ϕ[0;entry][r5;loop]
  r8 = r1 x r2
  r9 = r7 + r8
  ret r9
```

pc

State σ contains:

pc = program counter

δ = local values

sdom(f,pc) = variable defns. that *strictly dominate* pc.

$$\mathrm{wf}(f,\sigma) = \forall r \in \mathrm{sdom}(f,pc).\ \exists v.\ \delta(r) = \lfloor v \rfloor$$

“All variables in scope are initialized.”

Generalizing Safety

- Definition of wf:

$$wf(f,(pc, \delta)) \quad = \quad \forall r \in sdom(f,pc).\ \exists v.\ \delta(r) = \lfloor v \rfloor$$

- Generalize like this:

$$wf(f,(pc, \delta)) \quad = \quad P\ f\ \ (\delta|_{sdom(f,pc)})$$

$$\text{where} \quad P : \text{Program} \longrightarrow \text{Local} \longrightarrow \text{Prop}$$

- Methodology: for a given P prove thre

  Initialization(P)
  Preservation(P)
  Progress(P)

Consider only variables in scope ⇒ P defined relative to the dominator tree of the CFG.

Instantiating

- For usual safety:

$$P_{safety}\ f\ \delta \quad = \quad \forall r \in dom(\delta).\ \exists v.\ \delta(r) = \lfloor v \rfloor$$

- For semantic properties:

$$P_{sem}\ f\ \delta \quad = \quad \forall r.\ f[r] = \lfloor rhs \rfloor \Rightarrow \delta(r) = [\![rhs]\!]_{\delta}$$

- Useful for verifying correctness of:
  - code motion, dead variable elimination, common expression elimination, etc.

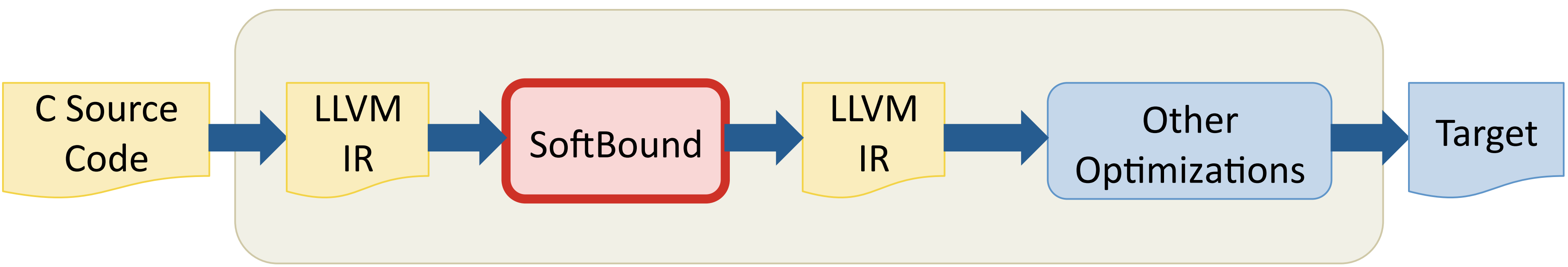
Plan

- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

SoftBound

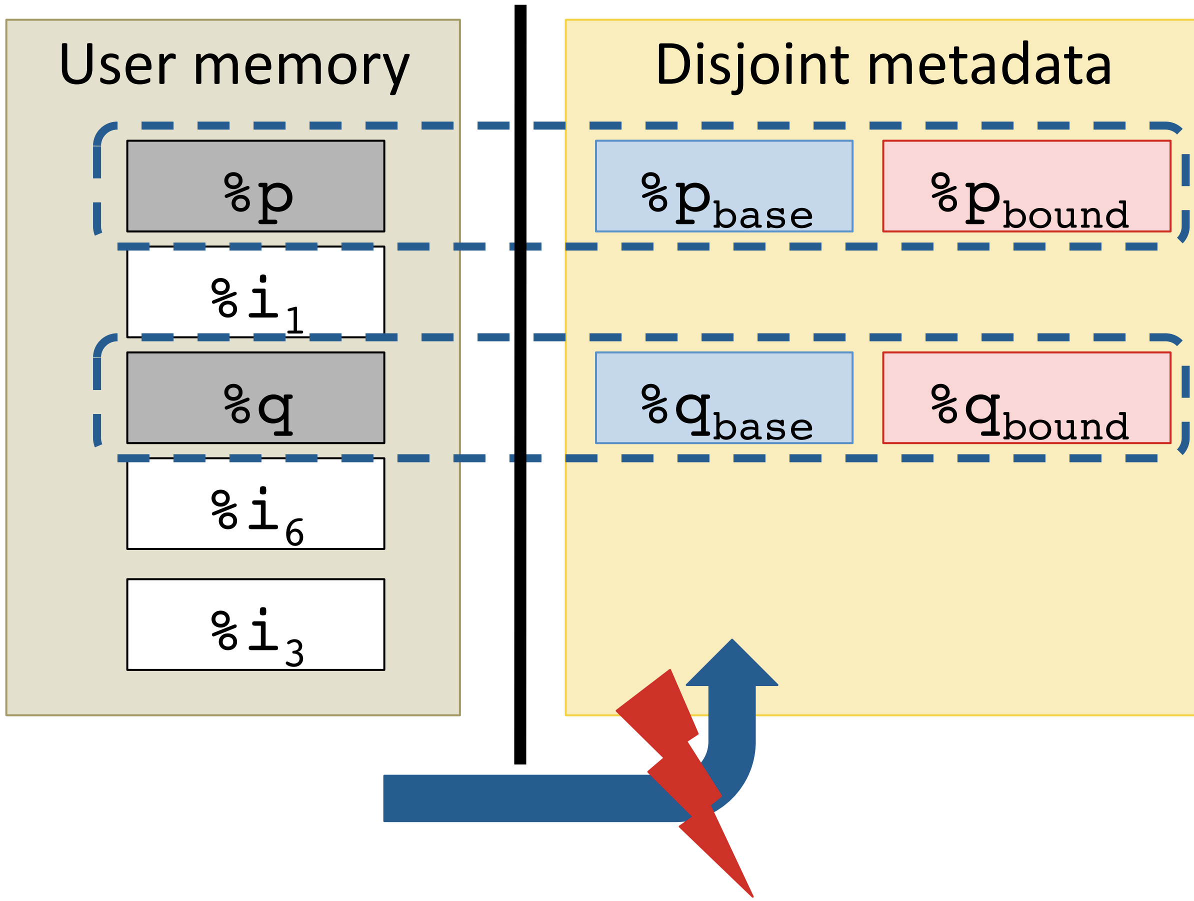
SoftBound
CETS

- Implemented as an LLVM pass.
- Detect spatial/temporal memory safety violations in legacy C code.
- Good test case:
  - Safety Critical ⇒ Proof cost warranted
  - Non-trivial Memory transformation



Disjoint Metadata

- Maintain pointer bounds in a separate memory space.
- Key Invariant: Metadata cannot be corrupted by bounds violation.



SoftBound

```
%p = call malloc [10 x i8]

%q = gep %p, i32 0, i32 255

store i8 0, %q
```

Maintain base and bound for all pointers

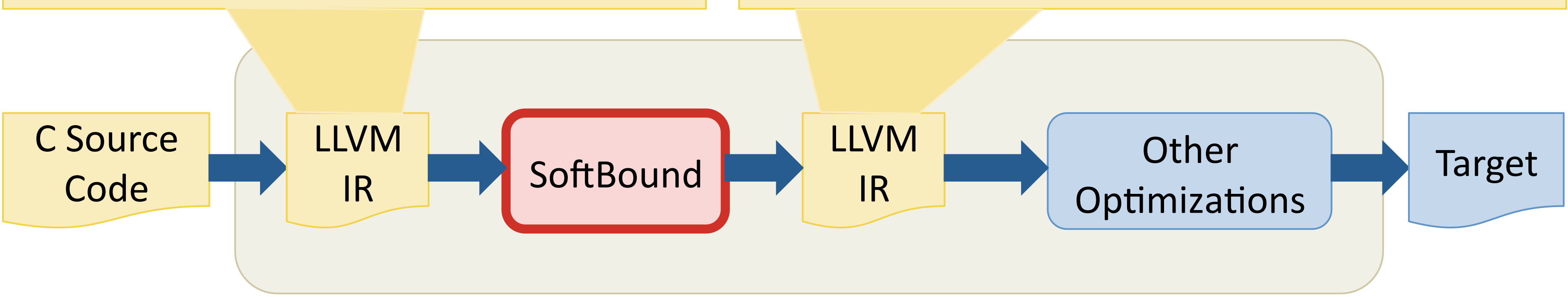
Propagate metadata on assignment

Check that a pointer is within its bounds when being accessed

```
%p = call malloc [10 x i8]
%p_base  = gep %p, i32 0
%p_bound = gep %p, i32 0, i32 10

%q = gep %p, i32 0, i32 255
%q_base  = %p_base
%q_bound = %p_bound

assert %q_base <= %q
    /\ %q+1 < %q_bound
store i8 0, %q
```



Proving SoftBound Correct

1. Define SoftBound(f,σ) = (f_s,σ_s)
   - Transformation pass implemented in Coq.
2. Define predicate: MemoryViolation(f,σ)
3. Construct a *non-standard* operational semantics:

$$f \vdash \sigma \xmapsto{SB} \sigma'$$

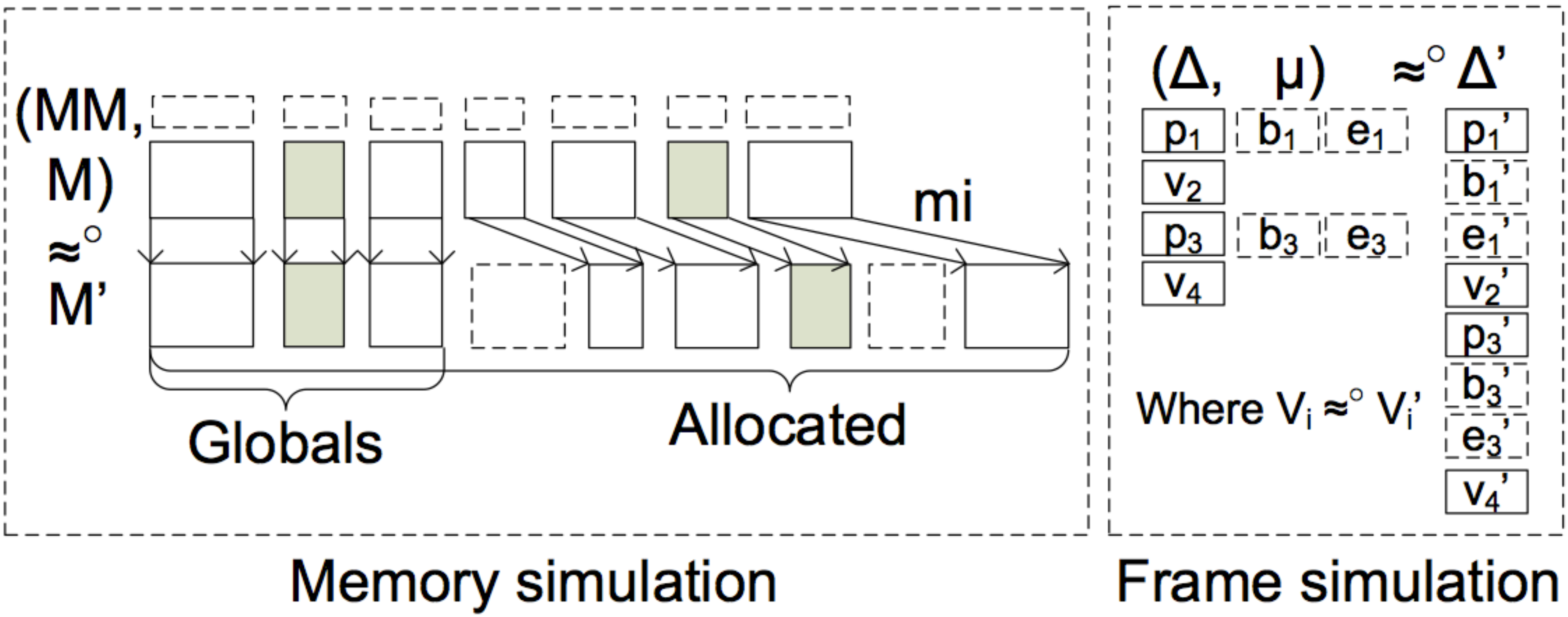
   - Builds in safety invariants "by construction"

$$f \vdash \sigma \xmapsto{SB}{}^{*} \sigma' \Rightarrow \neg\text{MemoryViolation}(f,\sigma')$$

4. Show that the instrumented code simulates the "correct" code:

$$\text{SoftBound}(f,\sigma) = (f_s,\sigma_s) \Rightarrow [f \vdash \sigma \xmapsto{SB}{}^{*} \sigma'] \gtrsim [f_s \vdash \sigma_s \longmapsto^{*} \sigma'_s]$$

Memory Simulation Relation

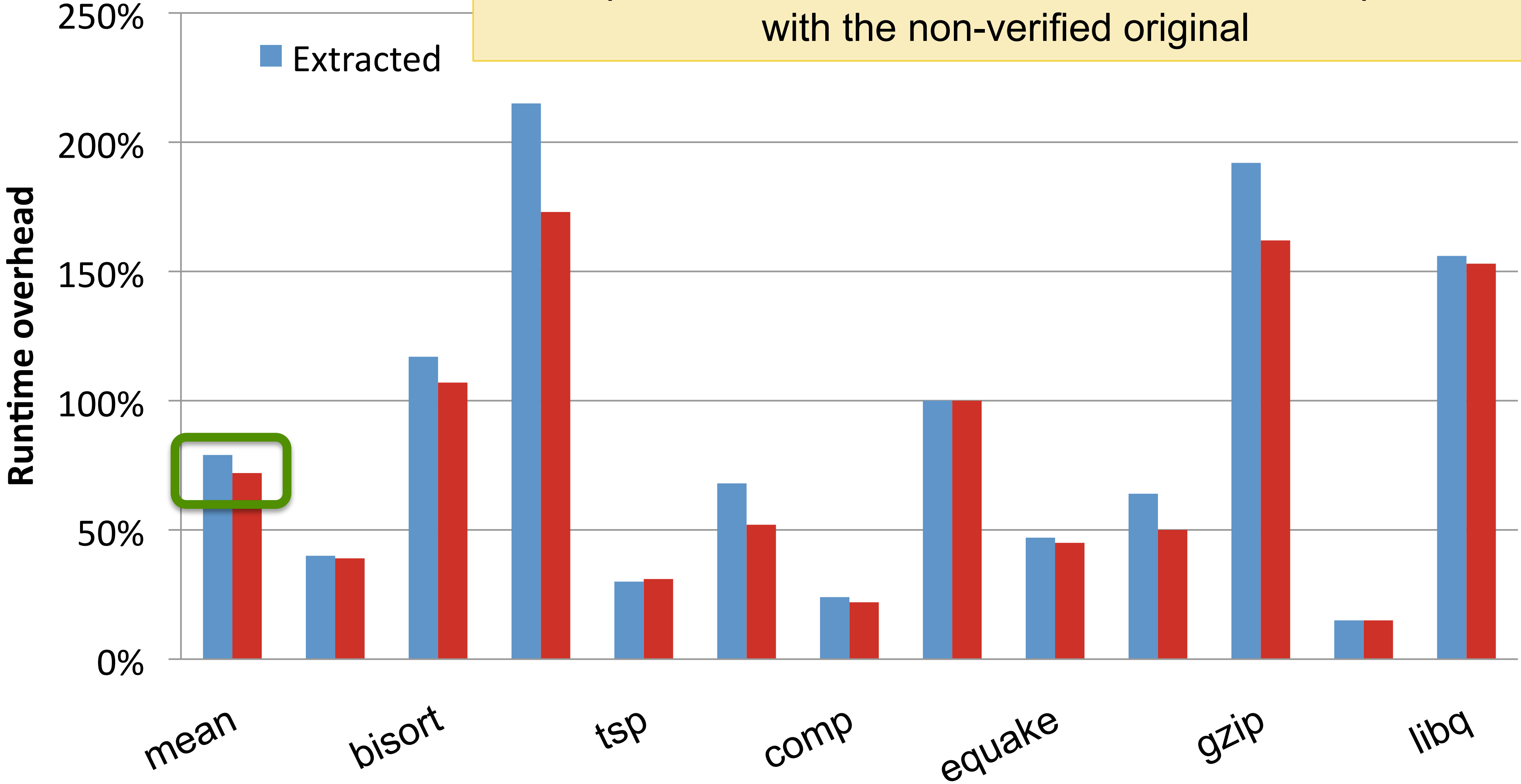


Lessons About SoftBound

- Found several bugs in our C++ implementation
  - Interaction of undef, ‘null’, and metadata initialization.
- Simulation proofs suggested a redesign of SoftBound’s handling of stack pointers.
  - Use a “shadow stack”
  - Simplify the design/implementation
  - Significantly more robust (e.g. varargs)

Competitive Runtime Overhead

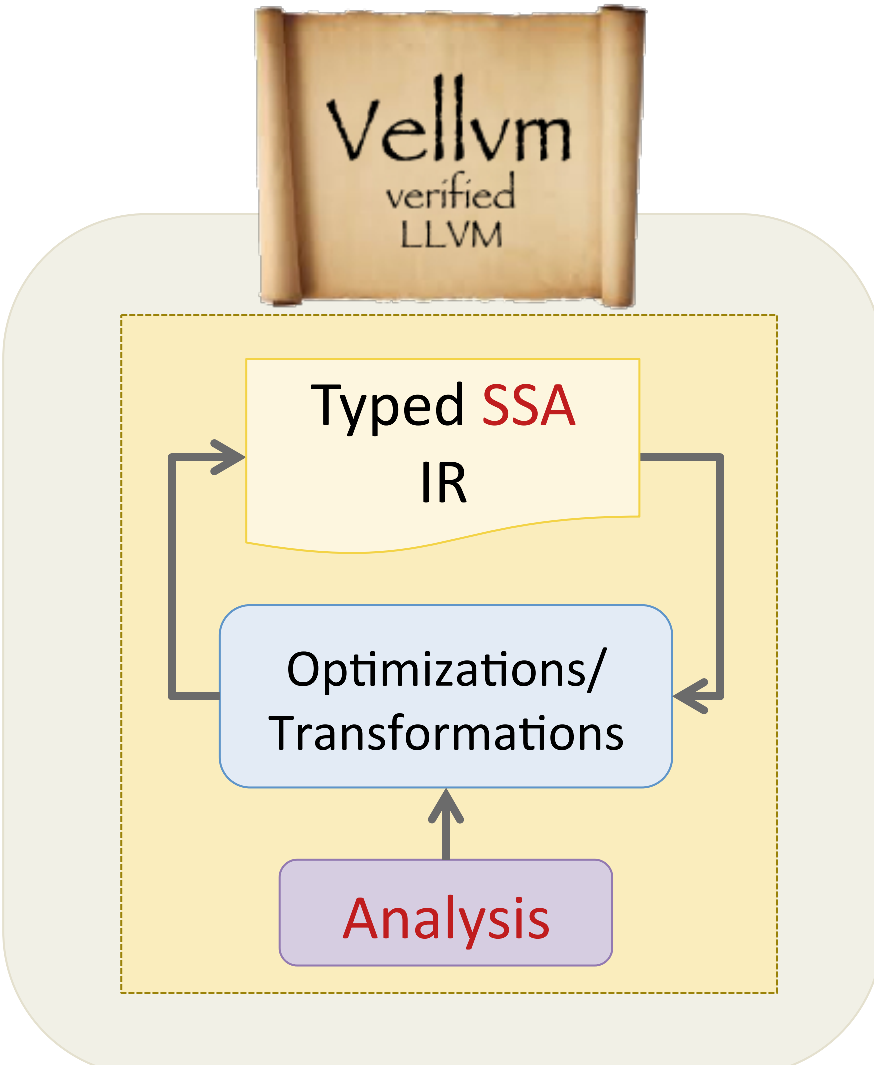
The performance of extracted SoftBound is competitive with the non-verified original



Plan

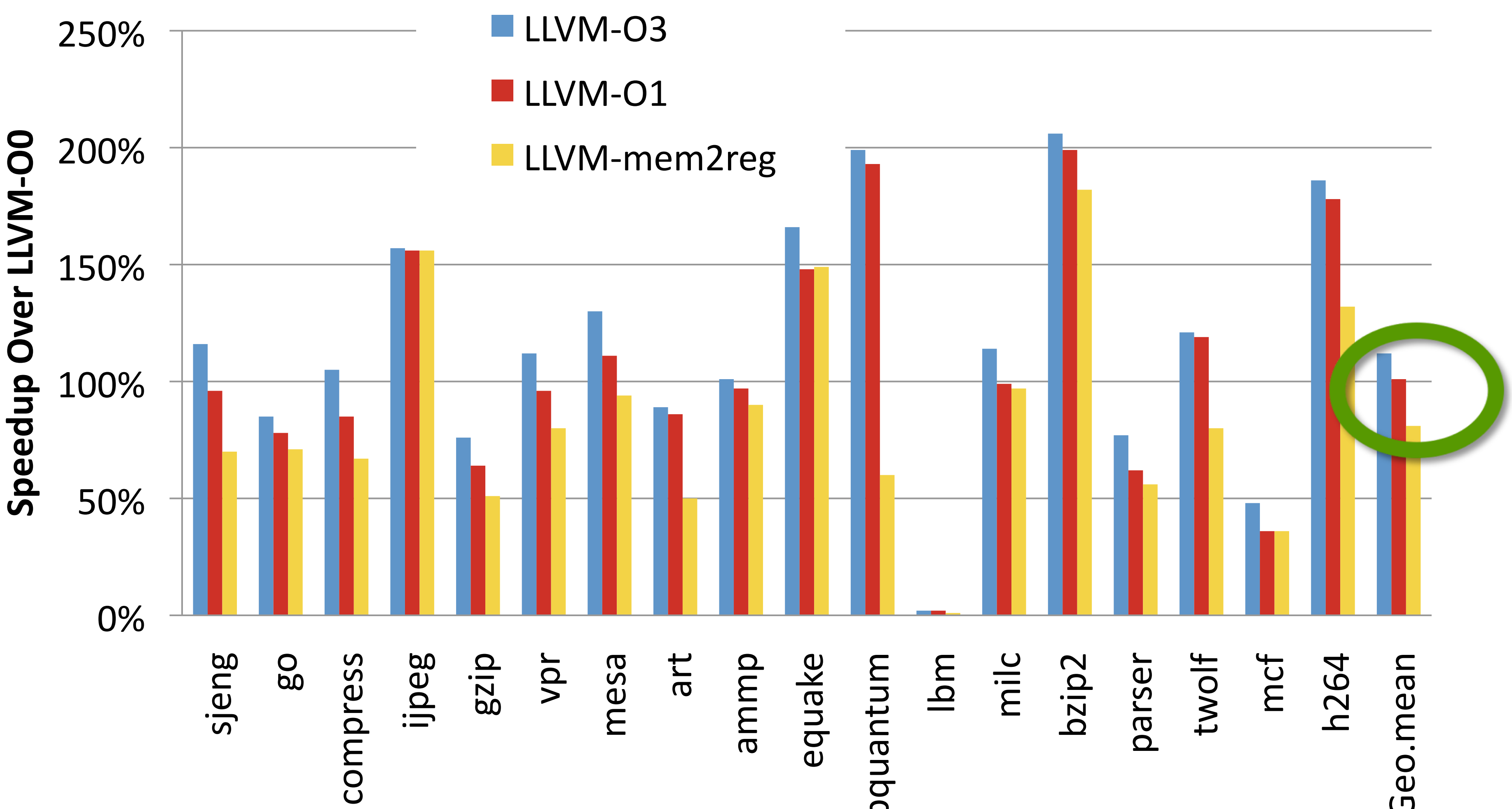
- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

Performance Critical Optimization



- LLVM compiler runs numerous optimizations
- Proving cost vs speedup
- Which optimization has the most performance impact?

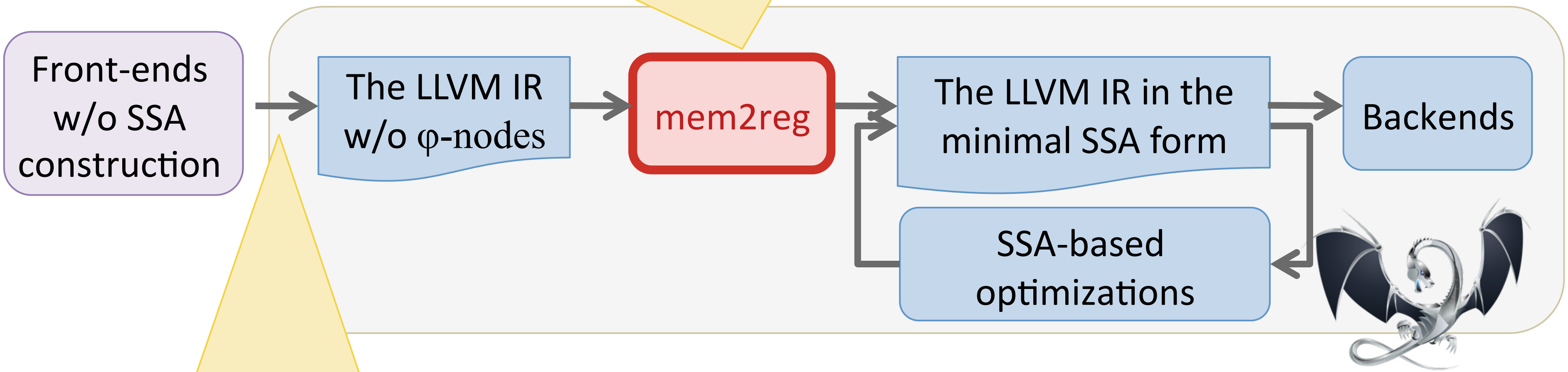
Critical Optimization in LLVM



O1 speeds up the program by 101%.
mem2reg speeds it up by 81%

mem2reg in LLVM

- Promote stack allocas to temporaries
- Insert minimal φ-nodes



- imperative variables ⇒ stack allocas
- no φ-nodes
- trivially in SSA form

mem2reg Example

```
int x = 0;
if (y > 0)
  x = 1;
return x;
```

```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3
```

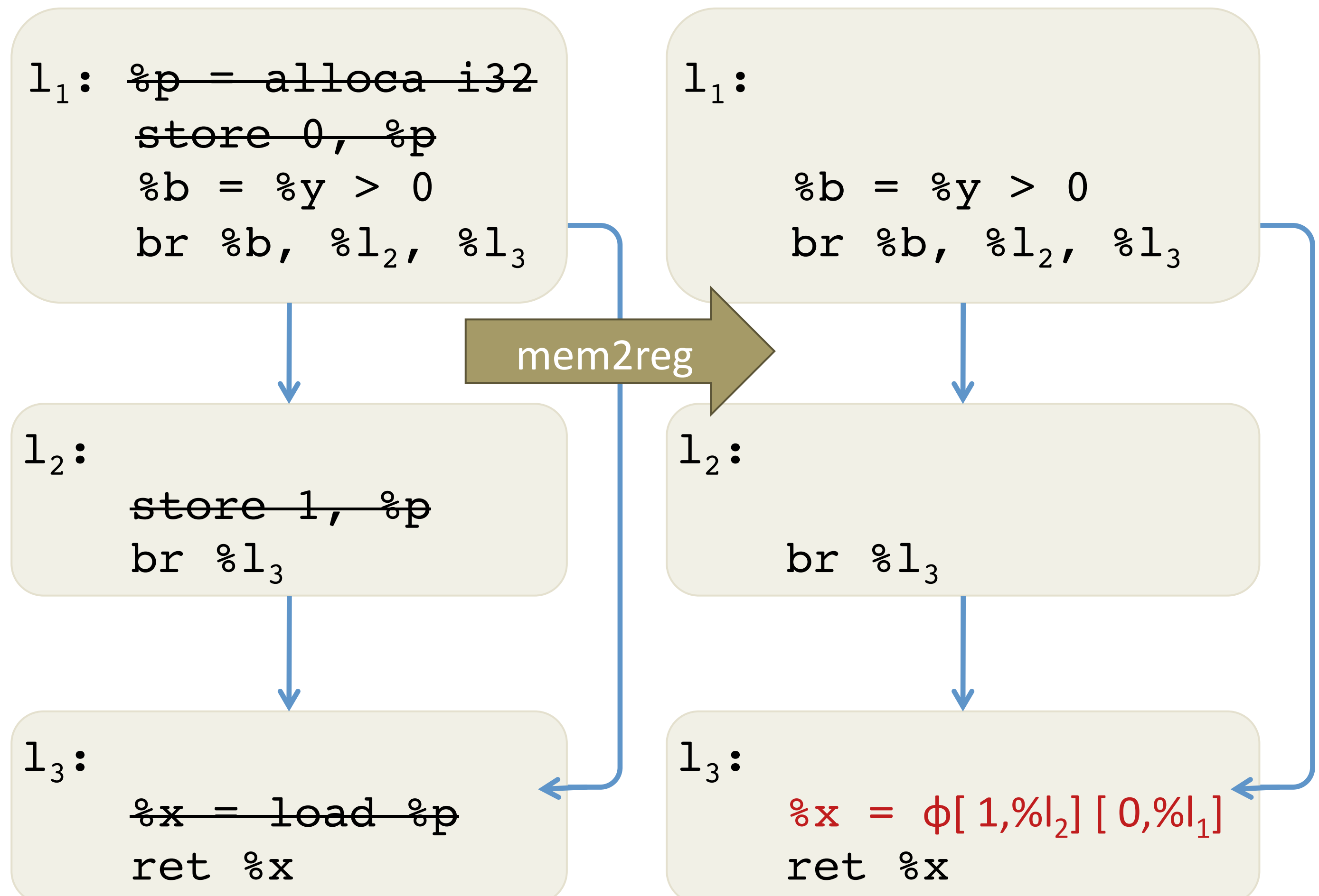
```
l2:
    store 1, %p
    br %l3
```

```
l3:
    %x = load %p
    ret %x
```

The LLVM IR in the trivial SSA form

mem2reg Example

```
int x = 0;
if (y > 0)
  x = 1;
return x;
```



The LLVM IR in the trivial SSA form

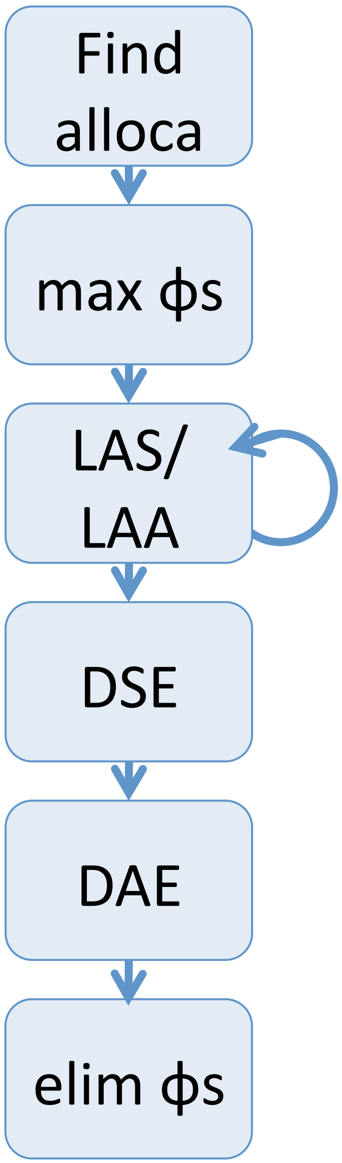
Minimal SSA after mem2reg

mem2reg Algorithm

- Two main operations
  - Phi placement (Lengauer-Tarjan algorithm)
  - Renaming of the variables

- Intermediate stage breaks SSA invariant
  - Defining semantics & well formedness non-trivial

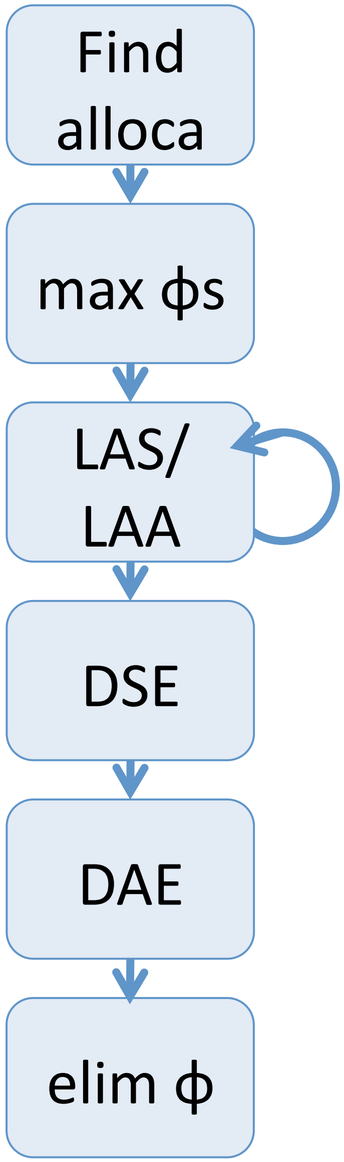
vmem2reg Algorithm



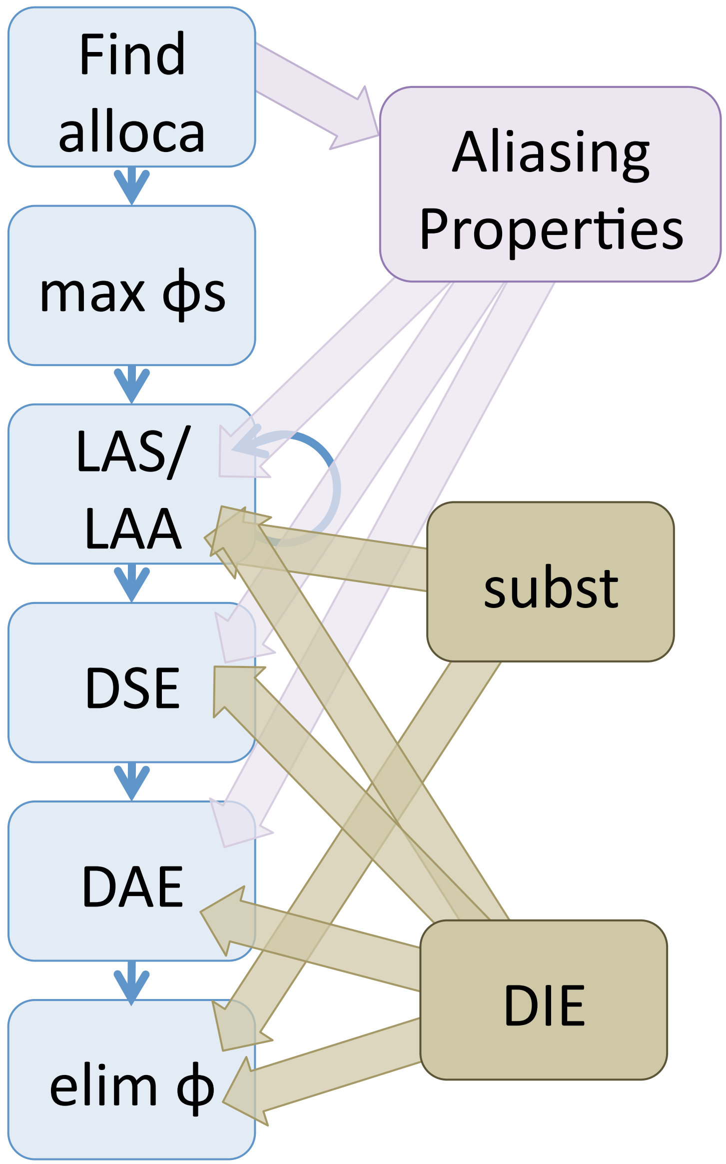
- Incremental algorithm
- Pipeline of micro-transformations
  - Preserves SSA semantics
  - Preserves well-formedness

- Inspired by Aycock & Horspool 2002.

How to Establish Correctness?



How to Establish Correctness?



1. Simple aliasing properties
   (e.g. to determine promotability)
2. Instantiate proof technique for
   - Substitution
   - Dead Instruction Elimination

   P_{DIE} = …
   Initialize(P_{DIE})
   Preservation(P_{DIE})
   Progress(P_{DIE})
4. Put it all together to prove composition of "pipeline" correct.

vmem2reg is Correct

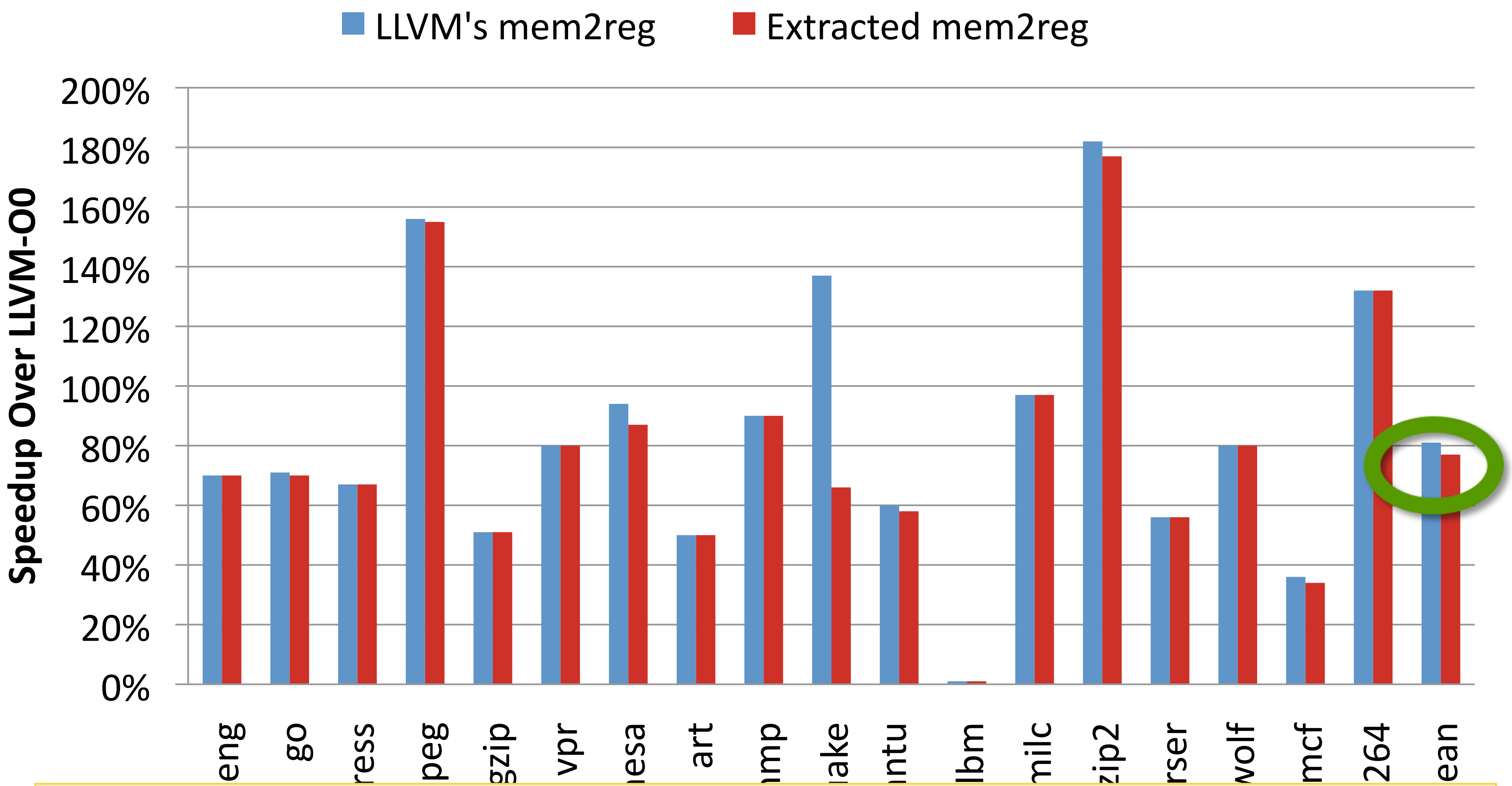
Theorem: The vmem2reg algorithm preserves the semantics of the source program.

Proof:

Composition of simulation relations from the "mini" transformations, each built using instances of the sdom proof technique.

(See Coq Vellvm development.) ☐

Runtime overhead of verified mem2reg



Vmem2reg: 77% LLVM's mem2reg: 81%

(LLVM's mem2reg promotes allocas used by intrinsics)

Plan

- Tour of the LLVM IR
- Vellvm infrastructure
  - Operational Semantics
  - SSA Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion

Related Work

- CompCert [Leroy et al.]
- CompCertSSA [Barthe, Demange et al. ESOP 2012]
  - Translation validate the SSA construction
- Verified Software Toolchain [Appel et. al]
- Verifiable SSA Representation [Menon et al. POPL 2006]
  - Identify the well-formedness safety predicate for SSA
- Specification of SSA
  - Temporal checking & model checking for proving SSA transforms [Mansky et al, ITP 2010]
  - Matrix representation of ϕ nodes [Yakobowski, INRIA]
  - Type system equivalent to SSA [Matsuno et al]

Conclusions

- Proof techniques for verifying SSA transformations
  - Generalize the SSA scoping predicate
  - Preservation/progress + simulations.
- Verified:
  - Softbound & vmem2reg
  - Similar performance to native implementations
- See the papers/coq sources for details!
- Future:
  - Clean up + make more accessible
  - Tutorial for Oregon PL Summer School
  - Alias analysis? Concurrency?
  - Applications to more LLVM-SSA optimizations

http://www.cis.upenn.edu/~stevez/vellvm/

Vellvm

verified

LLVM

http://www.cis.upenn.edu/~stevez/vellvm/