Verifying LLVM Optimizations in Coq

Steve Zdancewic

Oregon PL Summer School 2013

Thanks To

- Dmitri Garbuzov
    - developed the Vminus & hands-on part of the lectures

- Jianzhou Zhao
    - developed the Vellvm Coq framework

- Santosh Nagarakatte
- Milo Martin

- Xavier Leroy
    - some of the slides are modeled after his

Motivation: SoftBound/CETS

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

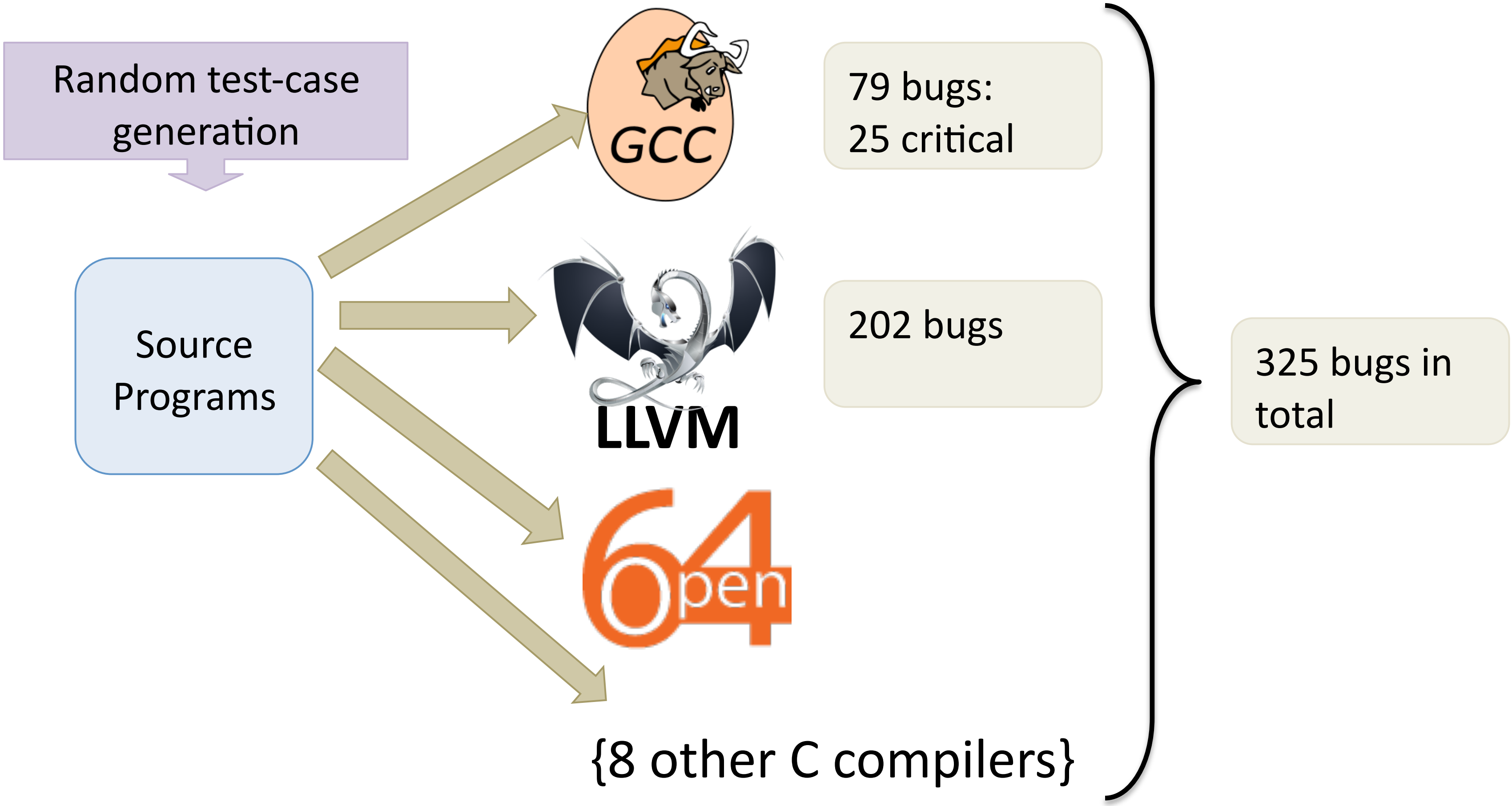
SoftBound
CETS

- 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]



Random test-case generation

Source Programs

79 bugs:
25 critical

202 bugs

{8 other C compilers}

325 bugs in total

Motivation: Semantics

Are these two C programs equivalent?

```
int Sum = (N & (N % 2 ? 0 : ~0)
        | ( ((N & 2)>>1) ^ (N & 1) ) );
```

```
int Sum = 0;
for (int i = 1; i < N; ++i)
{
   Sum = Sum ^ i;
}
```

(Yes!)

Motivation: OPLSS

- Demonstrate some applications of techniques from the summer school:
  - Formal Modeling in Coq
  - Operational Semantics
  - Preservation & Progress-style safety proofs
  - Simulation arguments

- Introduction to LLVM IR
  - Potentially useful target for PL implementations

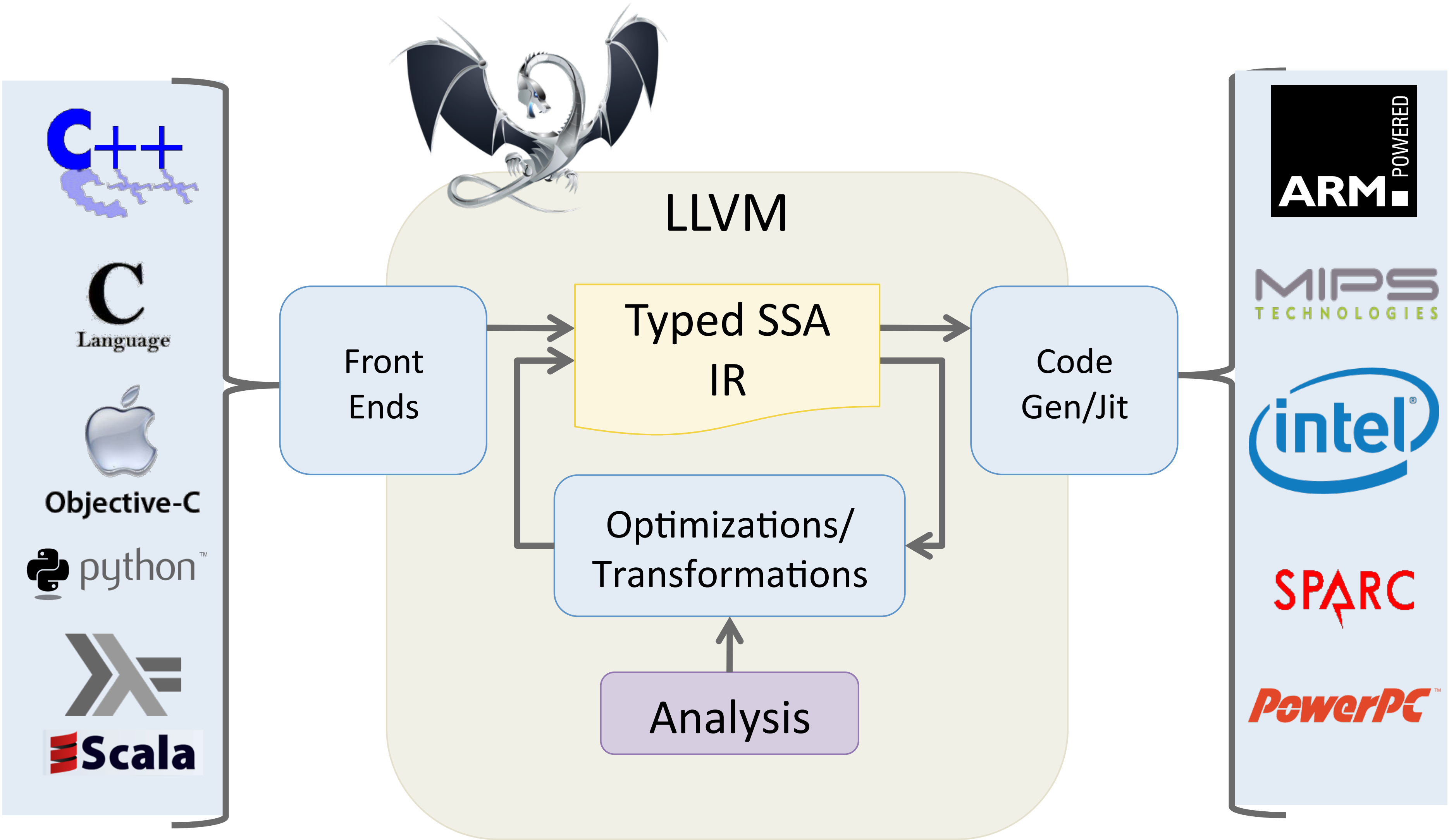
Low-level Virtual Machine (LLVM)

[Lattner et al.]

- Began in 2002 as Chris Lattner's Masters Thesis
- Has since evolved into an industrial-strength compiler intermediate language
  - open source
  - used widely in academia
  - used extensively by Apple
  - very active community
- Key features:
  - Simple design: one IR for many analyses/optimizations
  - Single Static Assignment
  - Typed IR
- See: http://llvm.org

LLVM Compiler Infrastructure

[Lattner et al.]



LLVM

Front Ends

Typed SSA IR

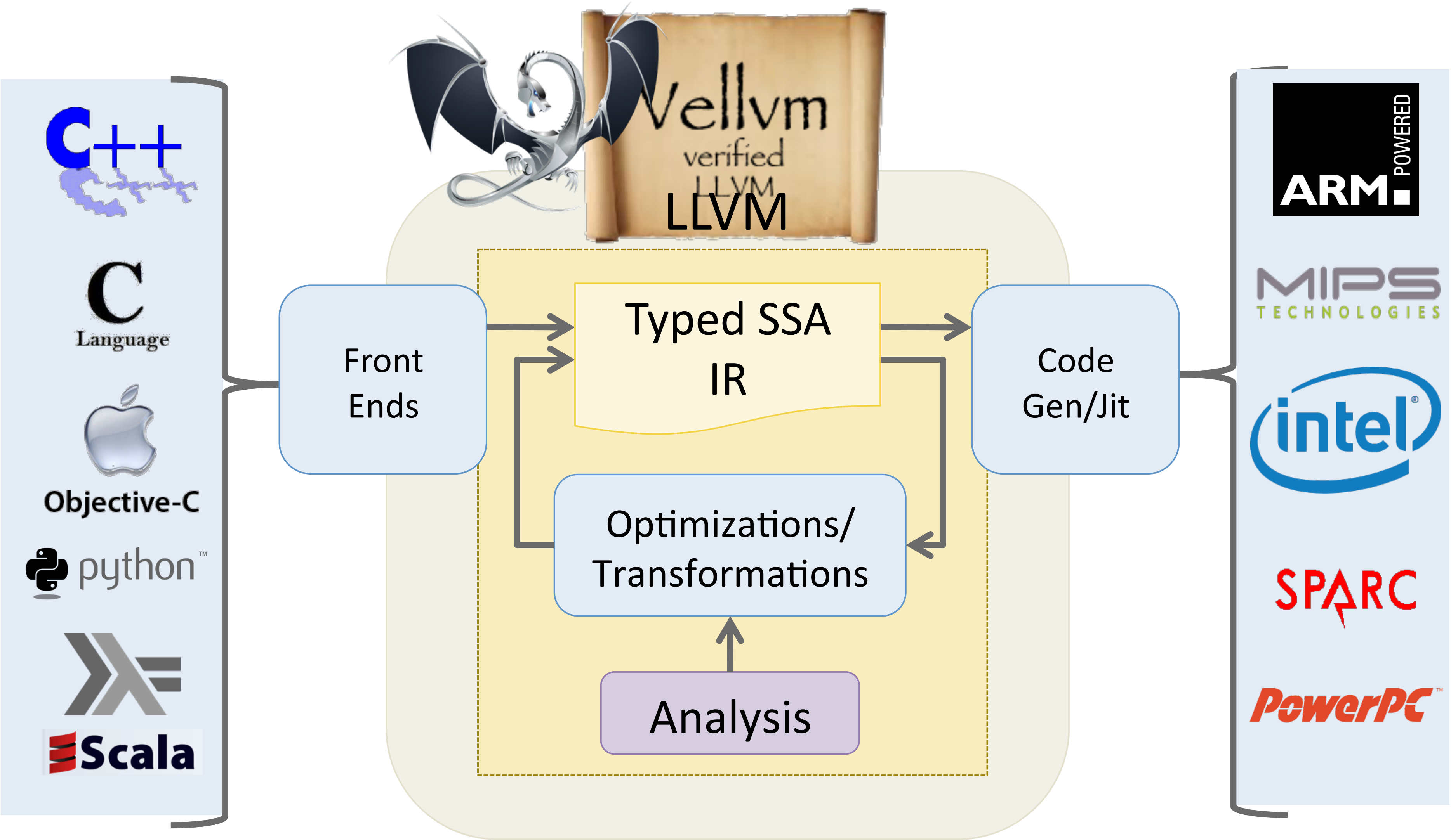
Code Gen/Jit

Optimizations/ Transformations

Analysis

LLVM Compiler Infrastructure

[Lattner et al.]



LLVM

Front Ends

Typed SSA IR

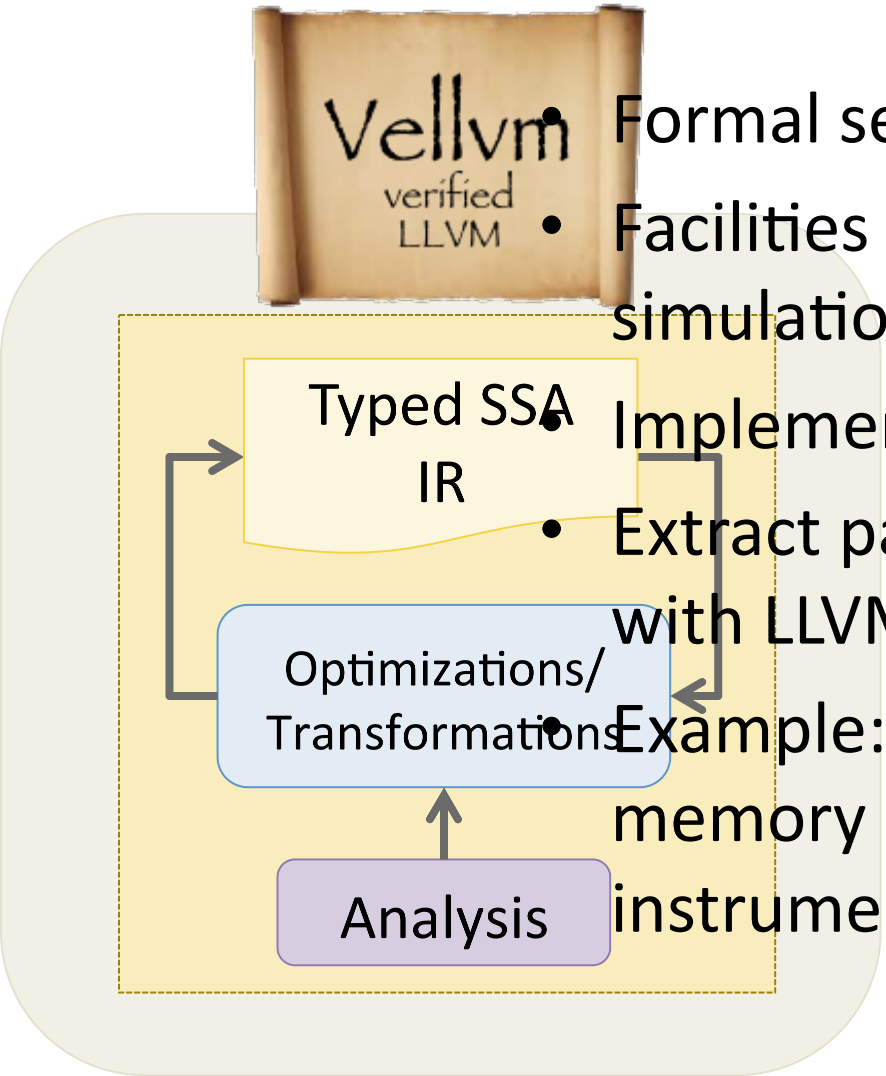
Optimizations/ Transformations

Analysis

Code Gen/Jit

The Vellvm Project

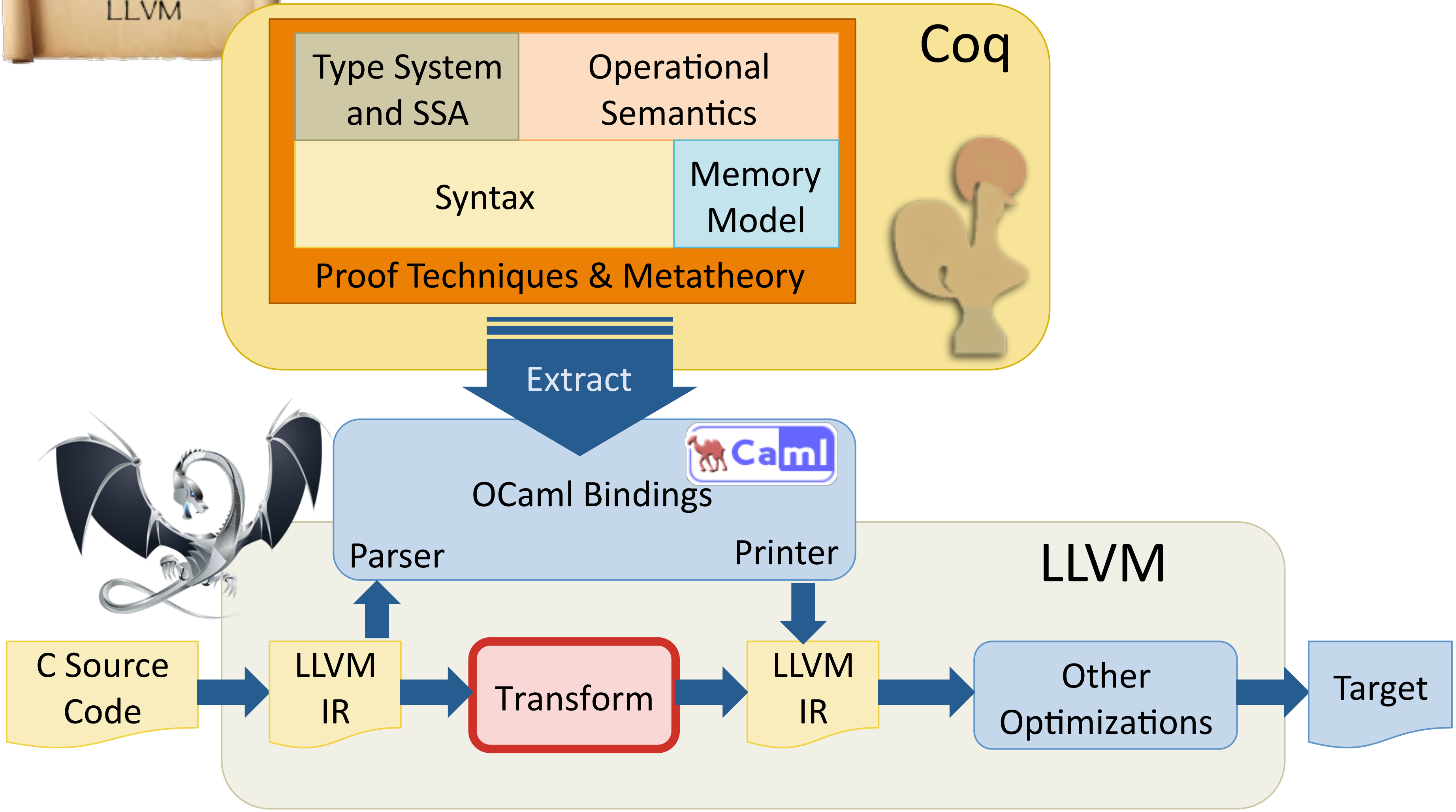
[Zhao et al. POPL 2012, CPP 2012, PLDI 2013]



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

Vellvm
verified
LLVM

Vellvm Framework



Coq

Type System and SSA

Operational Semantics

Syntax

Memory Model

Proof Techniques & Metatheory

Extract

Caml

OCaml Bindings

Parser

Printer

LLVM

C Source Code

LLVM IR

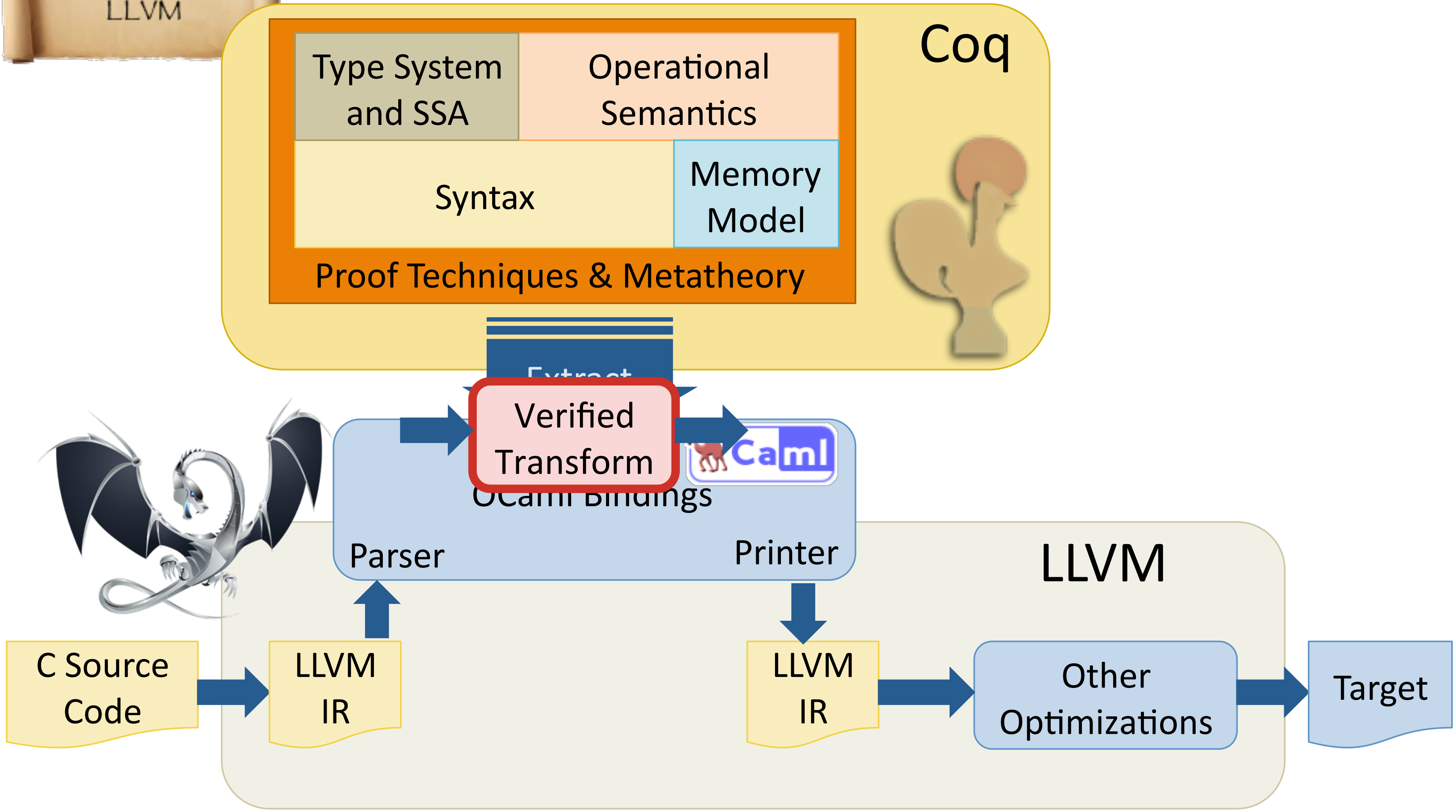
Transform

LLVM IR

Other Optimizations

Target

Vellvm Framework



Coq

Type System and SSA

Operational Semantics

Syntax

Memory Model

Proof Techniques & Metatheory

Extract

Verified Transform

OCaml Bindings

Parser

Printer

LLVM

C Source Code

LLVM IR

LLVM IR

Other Optimizations

Target

Plan

- Vminus: a highly simplified SSA IR based on LLVM
  - What is SSA?
- Verified Compilation of Imp to Vminus
  - What does it mean to “verify compilation”?
- Scaling up: Vellvm
  - Taste of the full LLVM IR
  - Operational Semantics
  - Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
  - mem2reg
- Conclusion:
  - challenges & research directions

(Unoptimized) LLVM IR Code

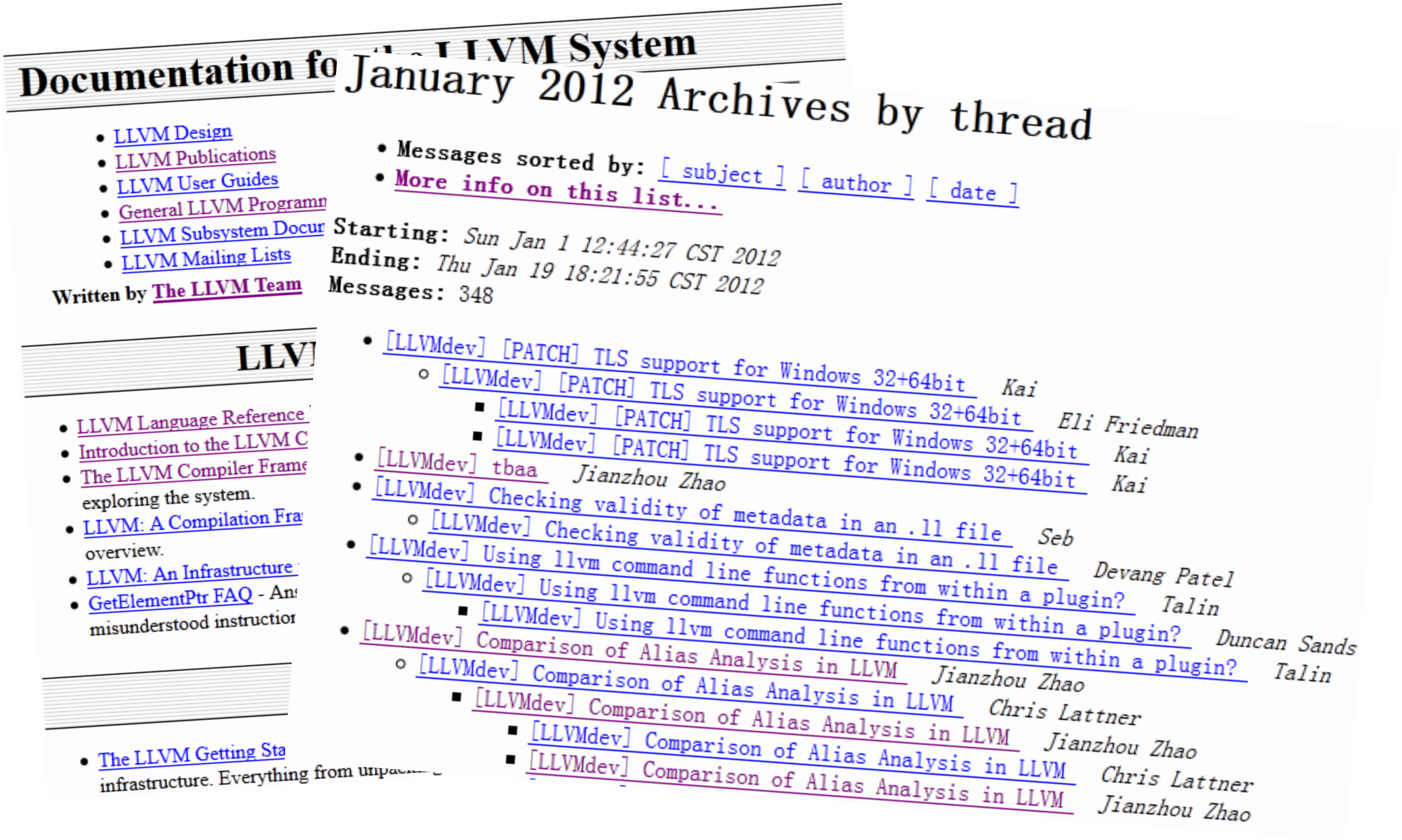
example.ll

```
define i32 @factorial(i32 %n) nounwind uwtable ssp {
entry:
  %1 = alloca i32, align 4
  %acc = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %acc, align 4
  br label %start

start:           ; preds = %entry, %else
  %3 = load i32* %1, align 4
  %4 = icmp ugt i32 %3, 0
  br i1 %4, label %then, label %else

then:            ; preds = %start
  %6 = load i32* %acc, align 4
  %7 = load i32* %1, align 4
  %8 = mul i32 %6, %7
  store i32 %8, i32* %acc, align 4
  %9 = load i32* %1, align 4
  %10 = sub i32 %9, 1
  store i32 %10, i32* %1, align 4
  br label %start

else:            ; preds = %start
  %12 = load i32* %acc, align 4
  ret i32 %12
}
```

example.c

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

Distilling the LLVM



Documentation fo LLVM System

- LLVM Design
- LLVM Publications
- LLVM User Guides
- General LLVM Programm
- LLVM Subsystem Docur
- LLVM Mailing Lists

Written by **The LLVM Team**

LLVI

- LLVM Language Reference
- Introduction to the LLVM C
- The LLVM Compiler Frame exploring the system.
- LLVM: A Compilation Fra overview.
- LLVM: An Infrastructure
- GetElementPtr FAQ - Ans misunderstood instructior

- The LLVM Getting Sta infrastructure. Everything from unpac

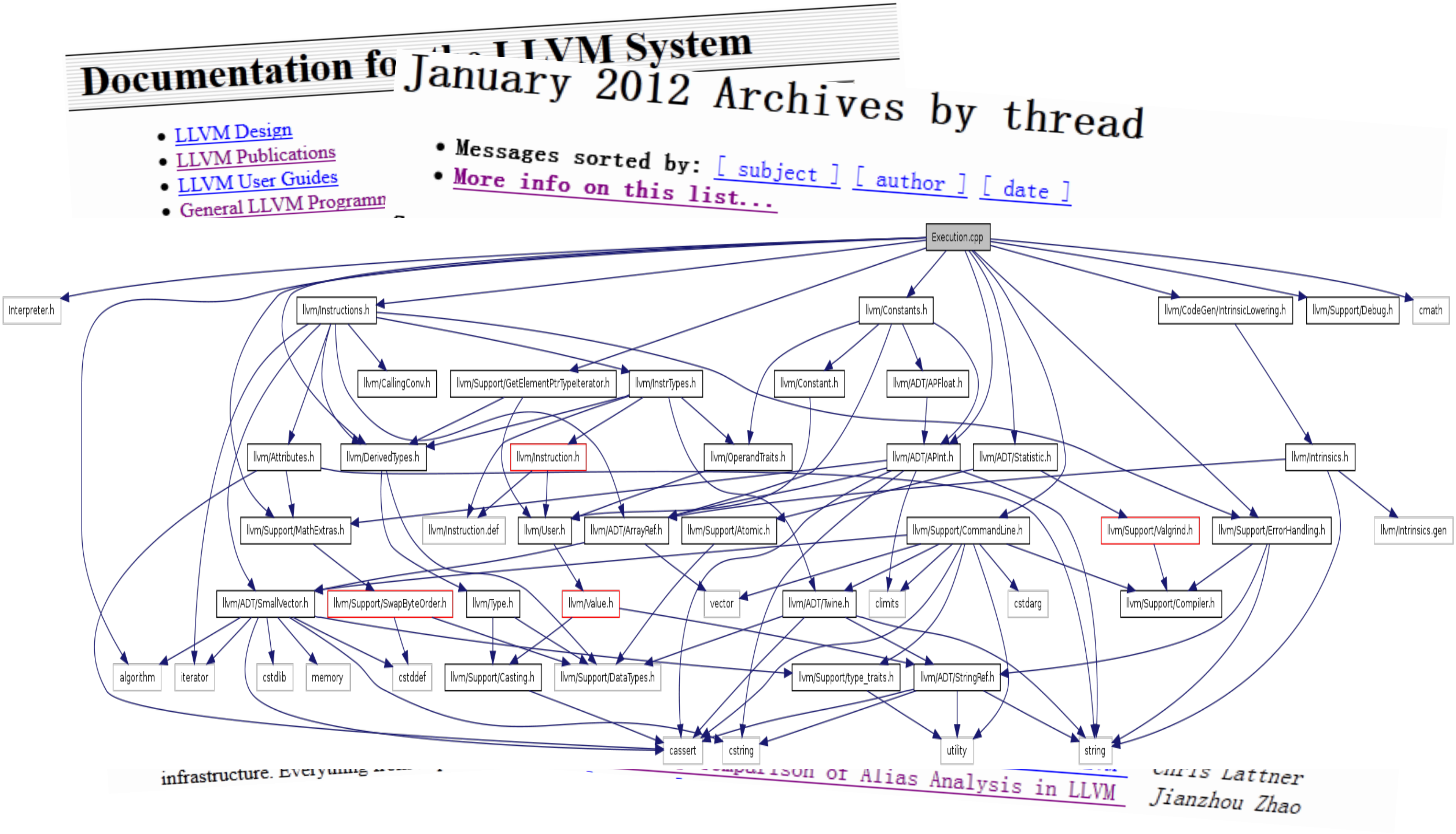
January 2012 Archives by thread

- **Messages sorted by:** [subject] [author] [date]
- **More info on this list...**

Starting: *Sun Jan 1 12:44:27 CST 2012*
Ending: *Thu Jan 19 18:21:55 CST 2012*
Messages: 348

- [LLVMdev] [PATCH] TLS support for Windows 32+64bit *Kai*
  - [LLVMdev] [PATCH] TLS support for Windows 32+64bit *Eli Friedman*
    - [LLVMdev] [PATCH] TLS support for Windows 32+64bit *Kai*
    - [LLVMdev] [PATCH] TLS support for Windows 32+64bit *Kai*
- [LLVMdev] tbaa *Jianzhou Zhao*
- [LLVMdev] Checking validity of metadata in an .ll file *Seb*
  - [LLVMdev] Checking validity of metadata in an .ll file *Devang Patel*
- [LLVMdev] Using llvm command line functions from within a plugin? *Talin*
  - [LLVMdev] Using llvm command line functions from within a plugin? *Duncan Sands*
    - [LLVMdev] Using llvm command line functions from within a plugin? *Talin*
- [LLVMdev] Comparison of Alias Analysis in LLVM *Jianzhou Zhao*
  - [LLVMdev] Comparison of Alias Analysis in LLVM *Chris Lattner*
    - [LLVMdev] Comparison of Alias Analysis in LLVM *Jianzhou Zhao*
      - [LLVMdev] Comparison of Alias Analysis in LLVM *Chris Lattner*
      - [LLVMdev] Comparison of Alias Analysis in LLVM *Jianzhou Zhao*

Distilling the LLVM



Documentation fo

LLVM System

- LLVM Design
- LLVM Publications
- LLVM User Guides
- General LLVM Programm

January 2012 Archives by thread

- Messages sorted by: [subject] [author] [date]
- More info on this list...

infrastructure.

Alias Analysis in LLVM

Chris Lattner
Jianzhou Zhao

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 (a la Imp)

Vminus by Example

```
entry:
```

```
loop:
```

```
exit:
```

Control-flow Graphs:
+ Labeled blocks

Vminus by Example

entry:

r_0 = ...
r_1 = ...
r_2 = ...

loop:

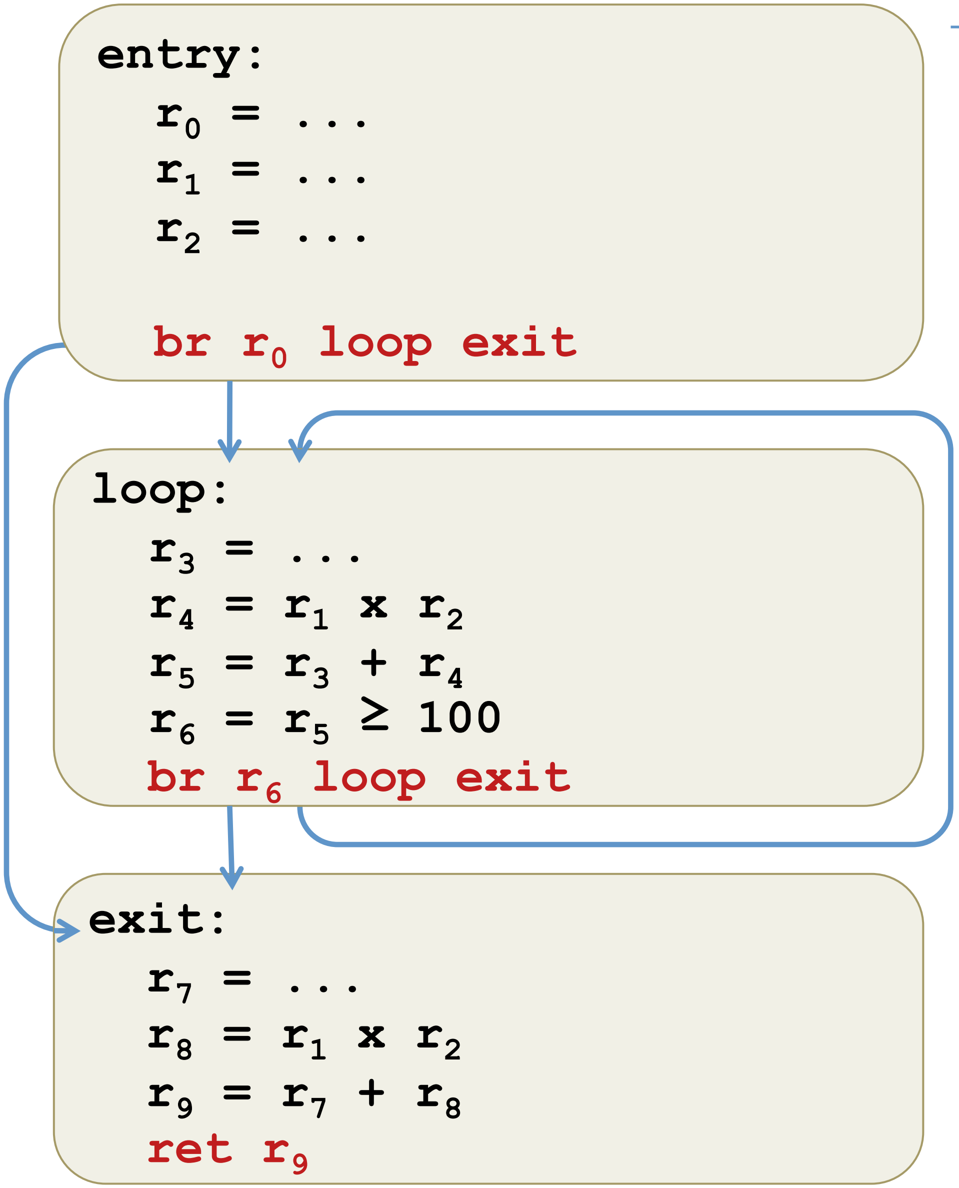
r_3 = ...
$r_4 = r_1 \times r_2$
$r_5 = r_3 + r_4$
$r_6 = r_5 \geq 100$

exit:

r_7 = ...
$r_8 = r_1 \times r_2$
$r_9 = r_7 + r_8$

Control-flow Graphs:
+ Labeled blocks
+ Binary Operations

Vminus by Example



entry:

r_0 = ...

r_1 = ...

r_2 = ...

br r_0 loop exit

loop:

r_3 = ...

$r_4 = r_1 \times r_2$

$r_5 = r_3 + r_4$

$r_6 = r_5 \geq 100$

br r_6 loop exit

exit:

r_7 = ...

$r_8 = r_1 \times r_2$

$r_9 = r_7 + r_8$

ret r_9

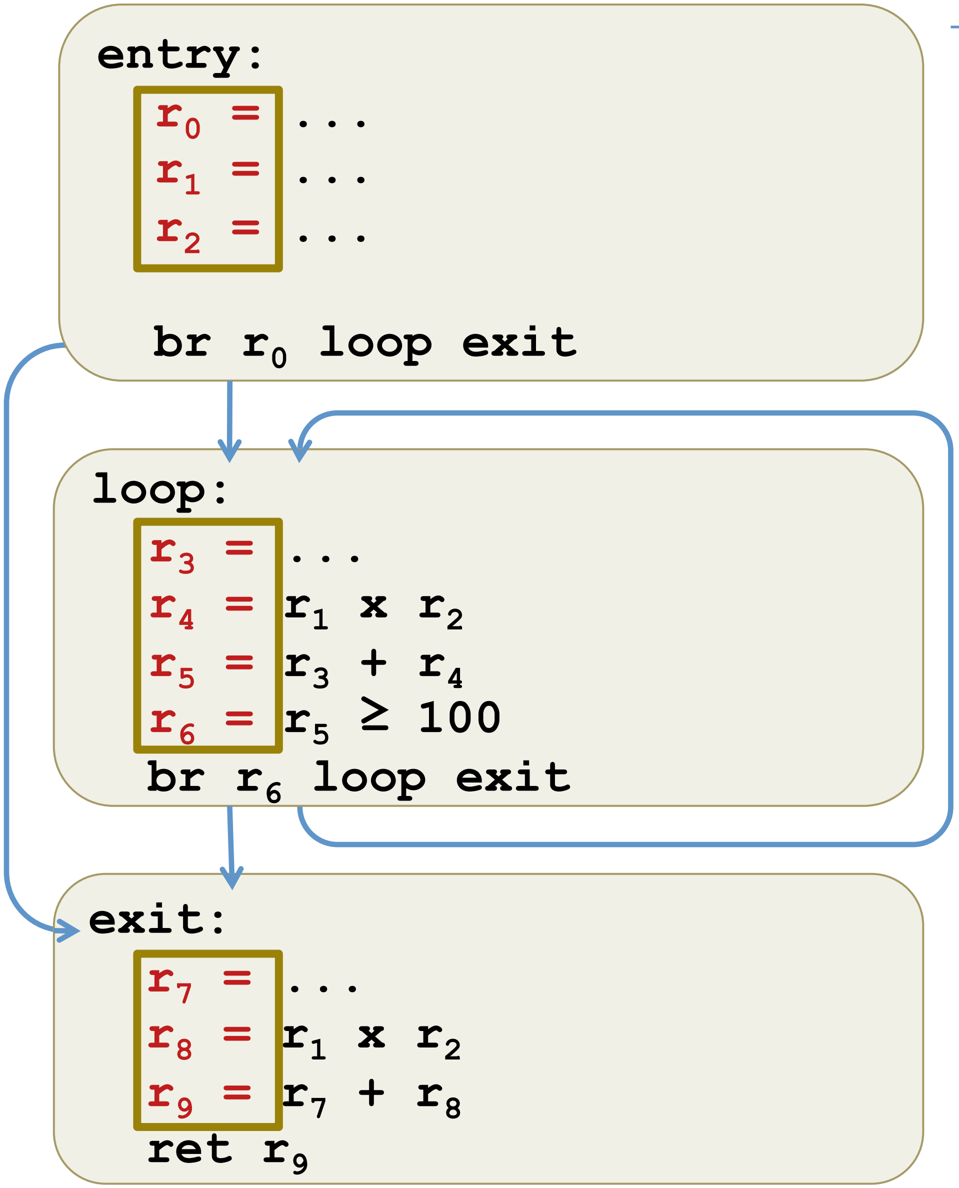
Control-flow Graphs:

+ Labeled blocks

+ Binary Operations

+ Branches/Return

Vminus by Example



entry:
$r_0 = \ldots$
$r_1 = \ldots$
$r_2 = \ldots$
br r_0 loop exit

loop:
$r_3 = \ldots$
$r_4 = r_1 \times r_2$
$r_5 = r_3 + r_4$
$r_6 = r_5 \geq 100$
br r_6 loop exit

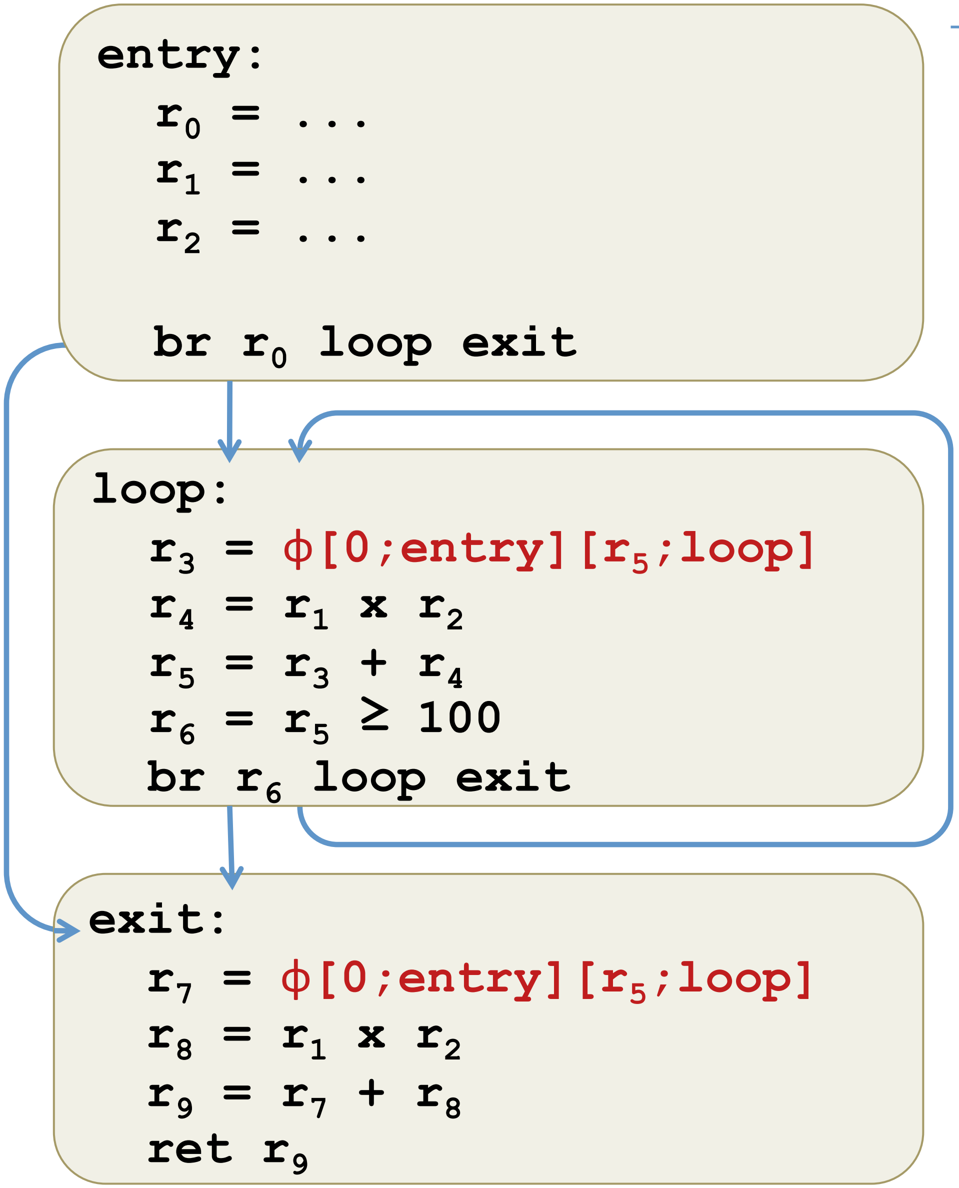
exit:
$r_7 = \ldots$
$r_8 = r_1 \times r_2$
$r_9 = r_7 + r_8$
ret r_9

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



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

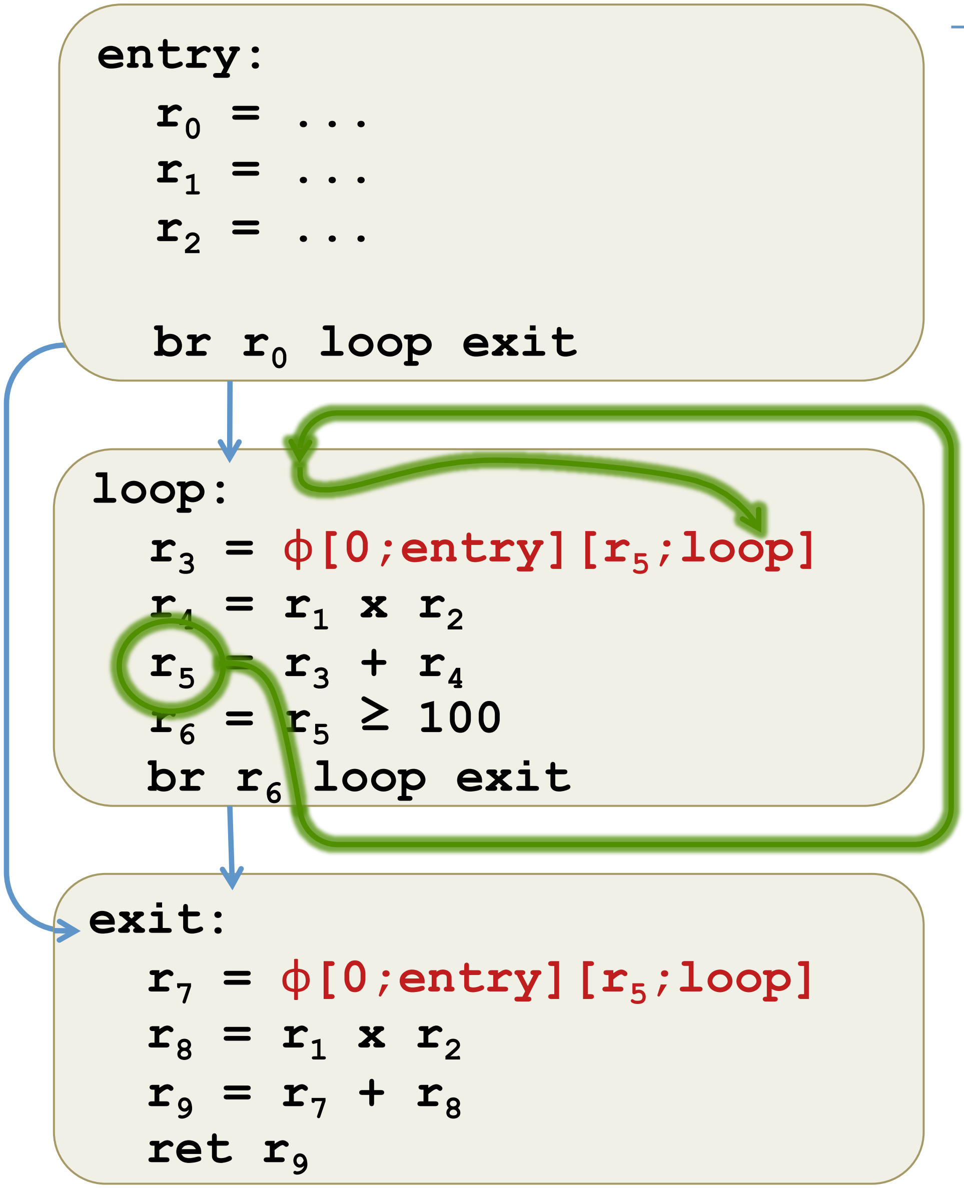
br r_0 **loop exit**

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

Vminus 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)

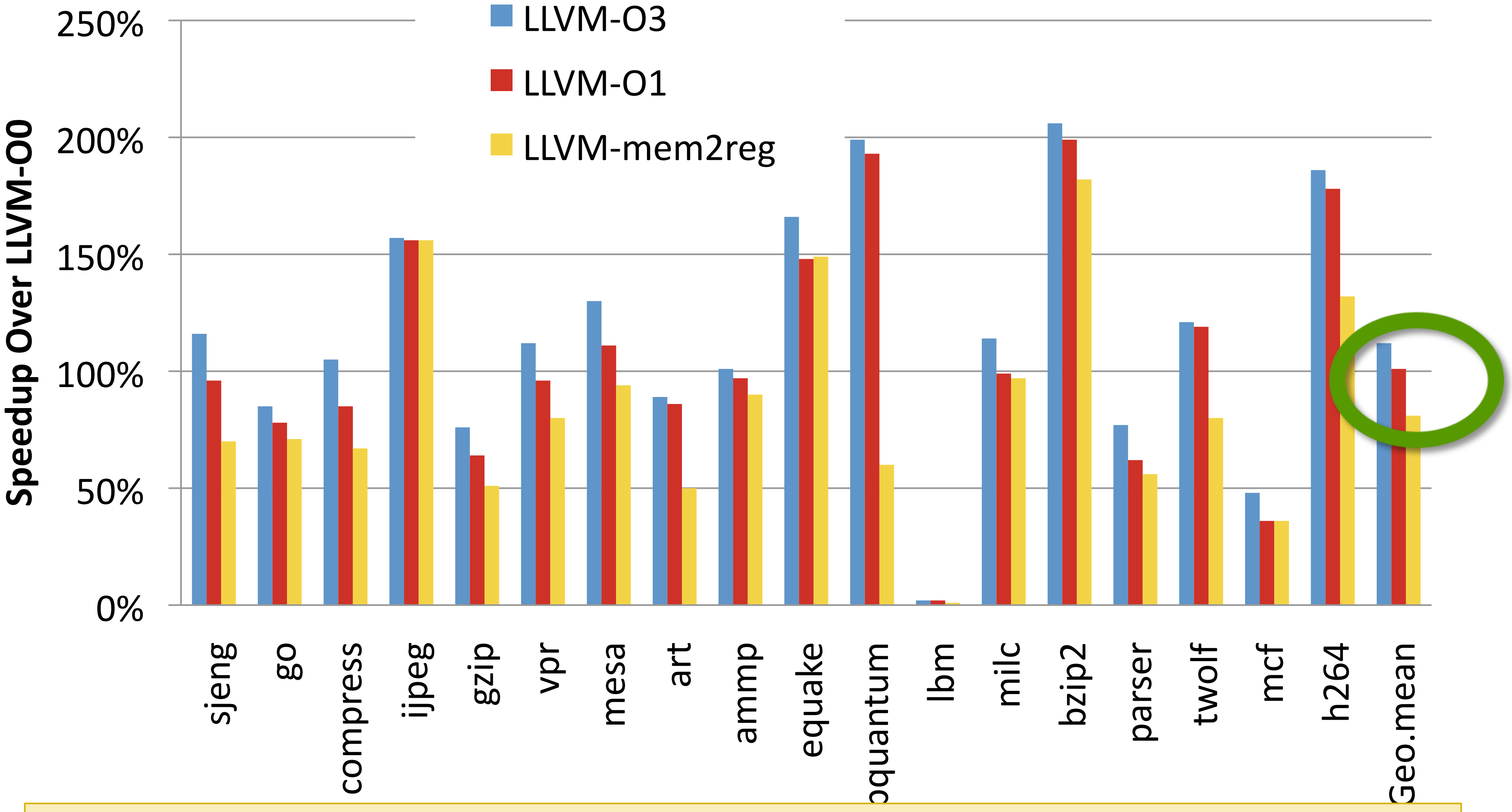
Static Single Assignment (SSA)

- 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.
  - Used in gcc, clang, intel, Jikes, HotSpot, Open64, …

SSA IR’s in Practice

- SSA simplifies register allocation:
    - The left-hand sides of SSA assignments can be thought of as “registers”
    - Renaming corresponds to “live range splitting” (decouples false dependencies)
    - register allocation is (arguably) the most important optimization for performance on modern processors

Critical Optimization in LLVM



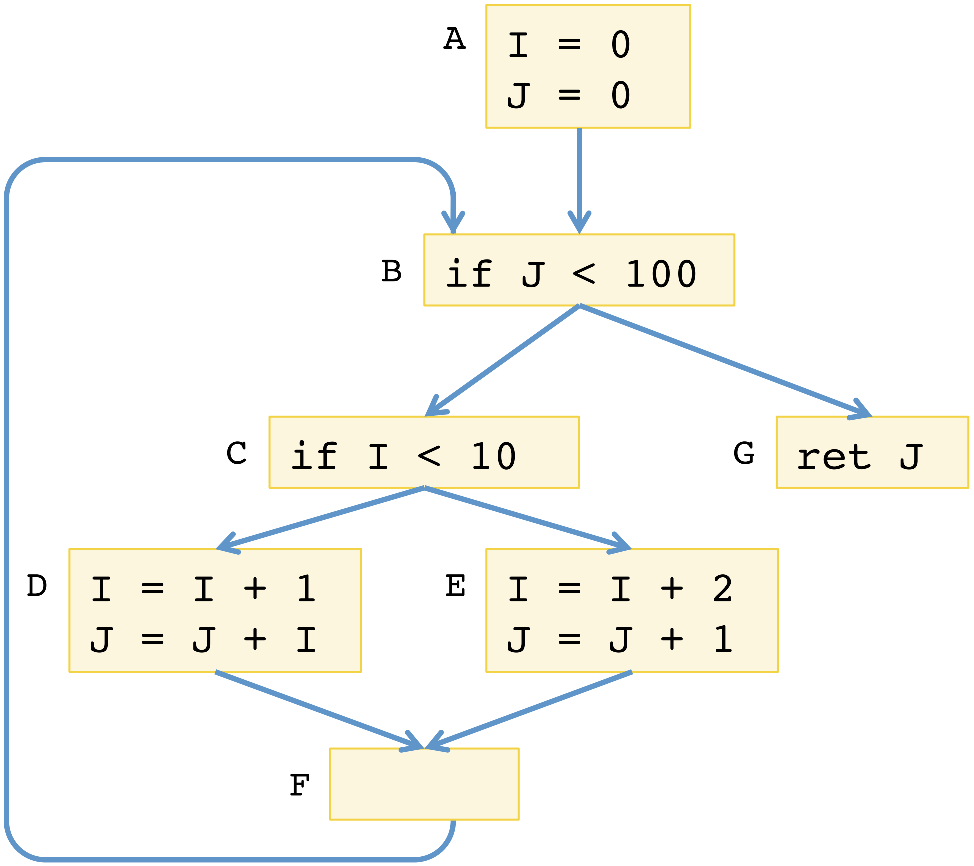
O1 speeds up the program by 101%.
mem2reg speeds it up by 81%

SSA Construction by Example

```
I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
```

SSA Construction by Example

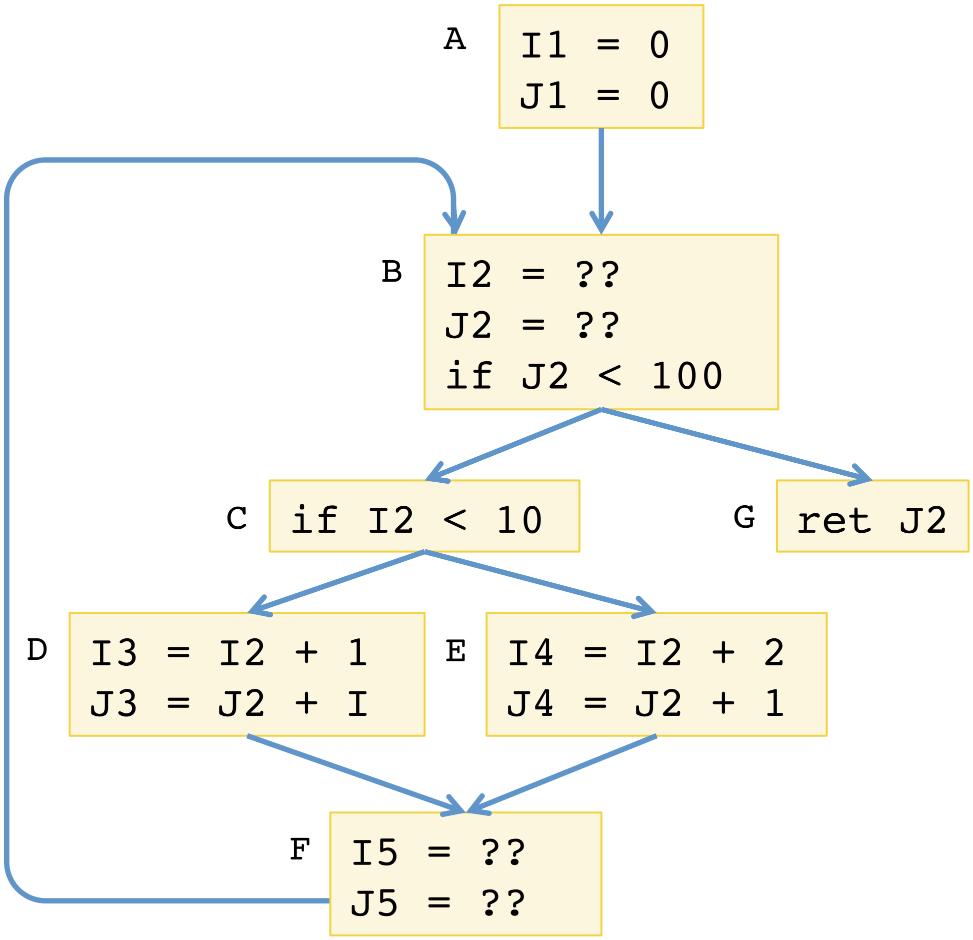
```
I := 0;;
J := 0;;
WHILE J < 100 DO
   IF I < 10 THEN
      I := I + 1;;
      J := J + I
   ELSE
      I := I + 2;;
      J := J + 1;
   FI
END;;
RETURN J
```



Step 1: Convert to a control-flow graph.

SSA Construction by Example

```
I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
```



```
A
I1 = 0
J1 = 0
```

```
B
I2 = ??
J2 = ??
if J2 < 100
```

```
C
if I2 < 10
```

```
G
ret J2
```

```
D
I3 = I2 + 1
J3 = J2 + I
```

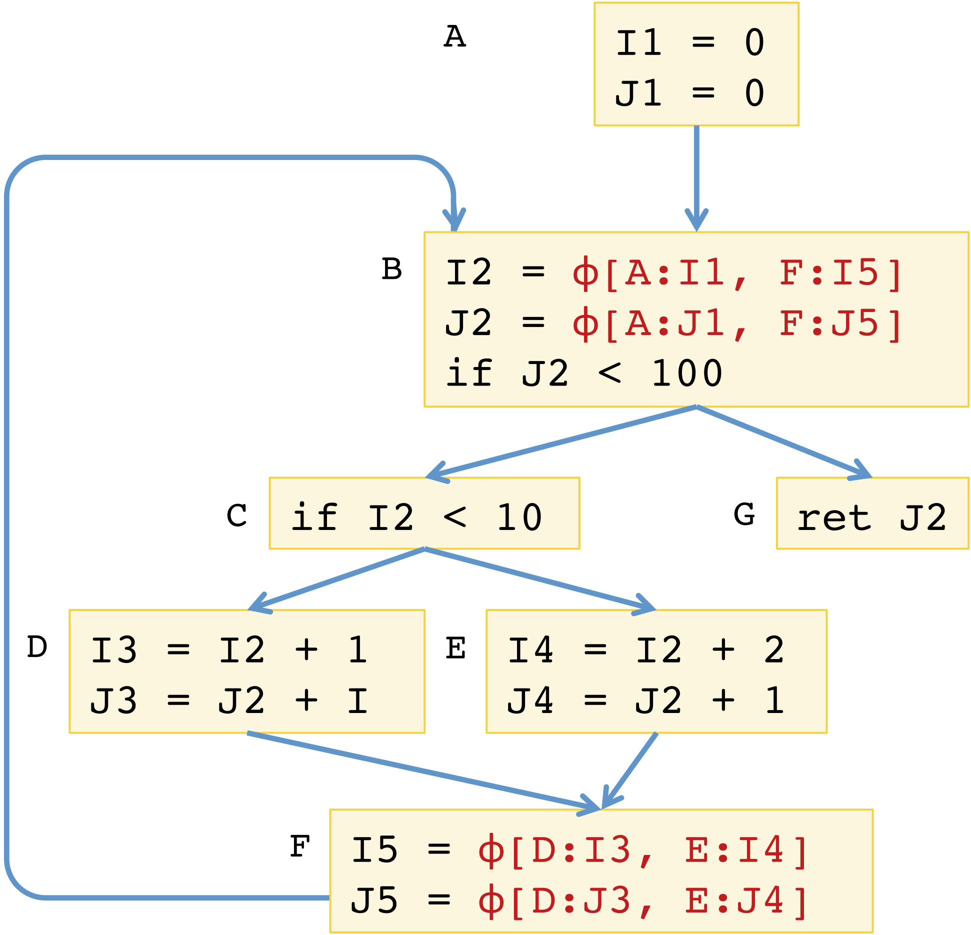
```
E
I4 = I2 + 2
J4 = J2 + 1
```

```
F
I5 = ??
J5 = ??
```

Step 2: Rename variables to satisfy single assignment.

SSA Construction by Example

```
I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
```



```
A
I1 = 0
J1 = 0

B
I2 = ϕ[A:I1, F:I5]
J2 = ϕ[A:J1, F:J5]
if J2 < 100

C
if I2 < 10

G
ret J2

D
I3 = I2 + 1
J3 = J2 + I

E
I4 = I2 + 2
J4 = J2 + 1

F
I5 = ϕ[D:I3, E:I4]
J5 = ϕ[D:J3, E:J4]
```

Step 3: Insert “ϕ” functions that capture control dependence.

SSA IR’s in Practice (2)

- SSA yields an efficient representation
  - Simplifies Def-Use information needed in dataflow analysis
  - Imperative data structure to map a definition to its uses
- However: Real SSA IRs still retain mutable state
  - SSA uid’s don’t have addresses…
  - memory operations: explicit pointer manipulation, allocation
  - example (in C):

```
int foo() {
  int x;
  init(&x);     // pointer escapes
  return x;
}
```

  - suggests the idea of “promoting” some imperative variables to SSA-style (those whose addresses don’t “escape”)

Vminus.Vminus.v

Up to the CFG module

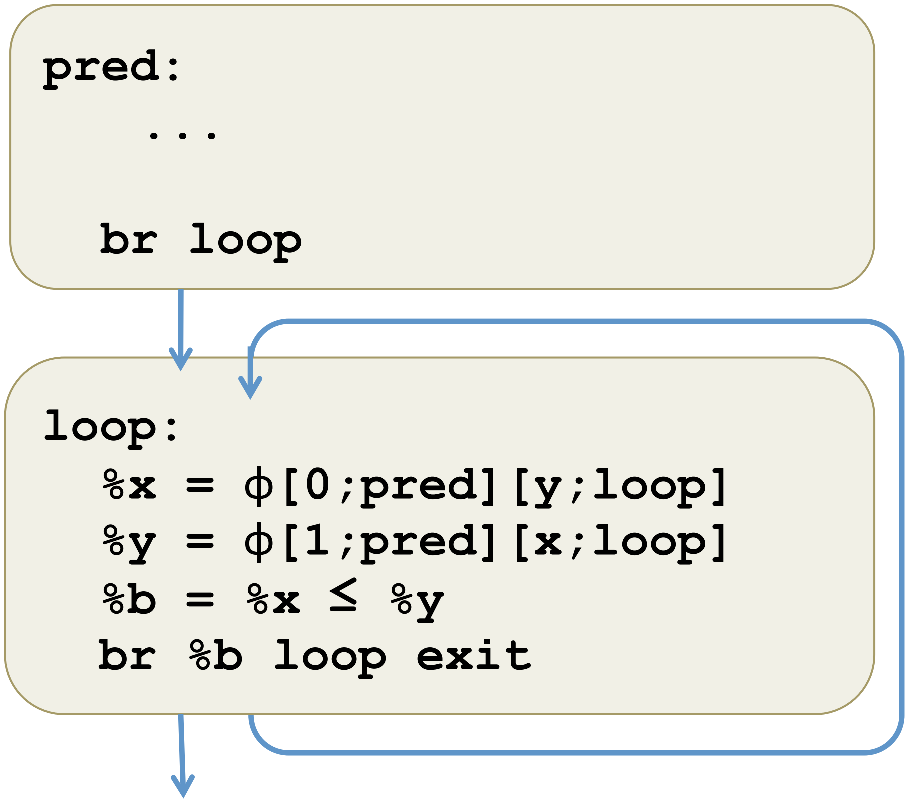
Vminus Operational Semantics

- 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:



```
pred:
     ...

   br loop
```

```
loop:
   %x = ϕ[0;pred][y;loop]
   %y = ϕ[1;pred][x;loop]
   %b = %x ≤ %y
   br %b loop exit
```

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)

Vminus.Vminus.v

Opsem module

End of Part I

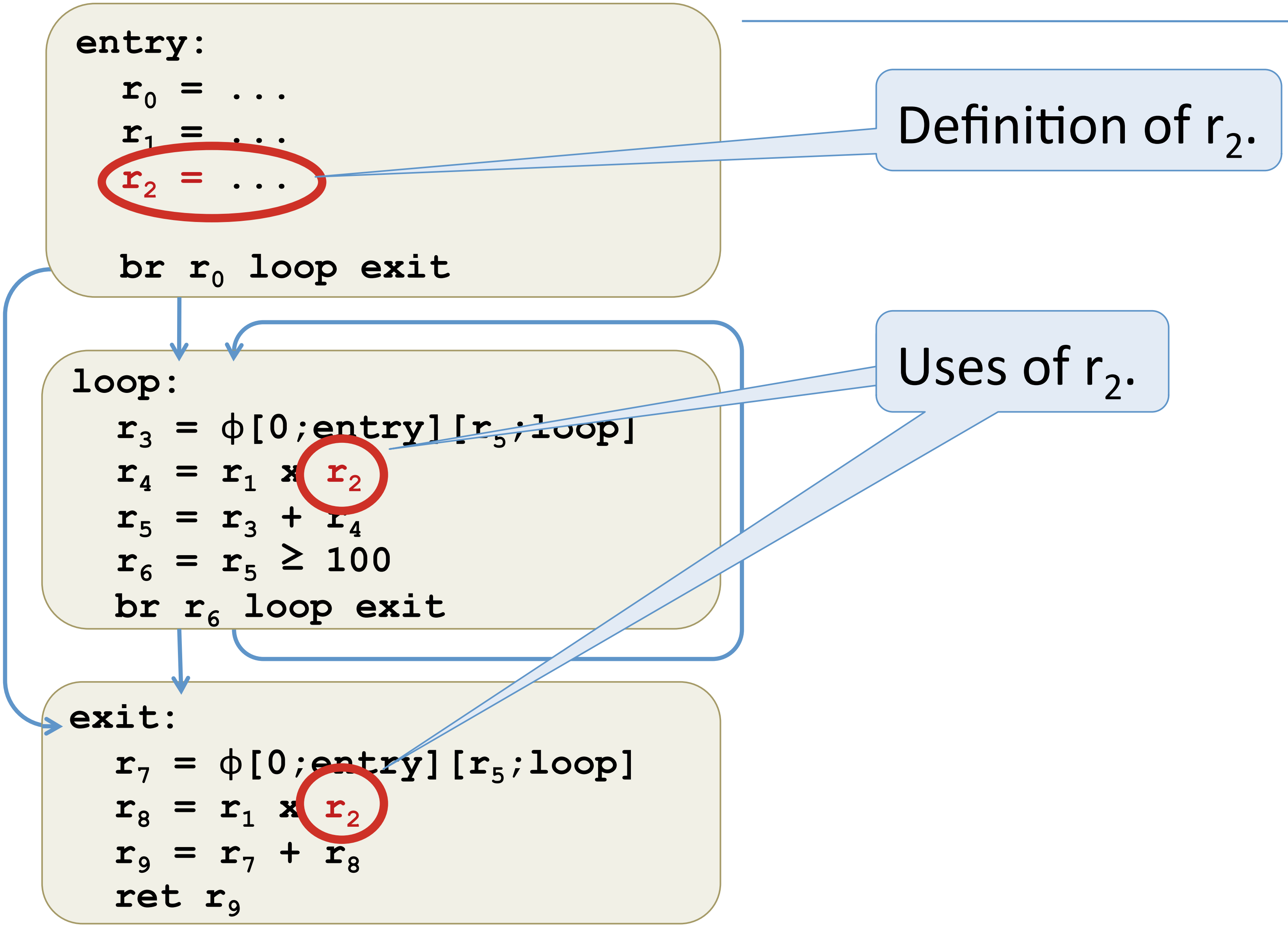
1

Recap

- Yesterday:
  - Defined a simple language called Vminus.
  - Five types of instructions:
    - binary arithmetic / load / store / phi nodes / terminators
  - Static Single Assignment
  - Operational semantics
    - Small step, relational

- Today: Static Semantics for Vminus
  - Scoping for SSA variables

Key SSA Invariant



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

br r_0 loop exit

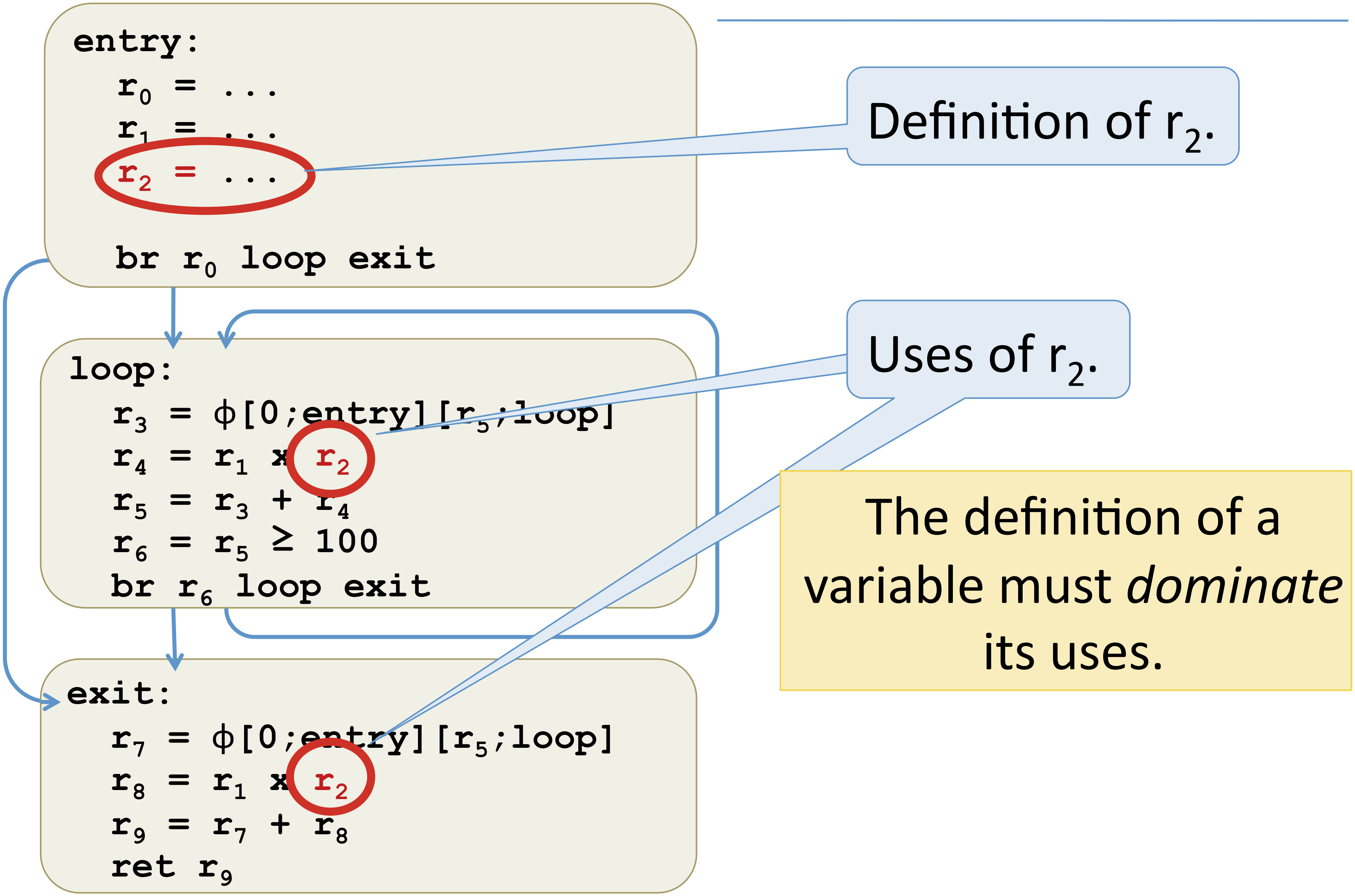
loop:
$r_3 = \phi$[0;entry][r_5;loop]
$r_4 = r_1$ x r_2
$r_5 = r_3 + r_4$
$r_6 = r_5 \geq 100$
br r_6 loop exit

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

Definition of r_2.

Uses of r_2.

Key SSA Invariant



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 ≥ 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

Definition of r_2.

Uses of r_2.

The definition of a variable must *dominate* its uses.

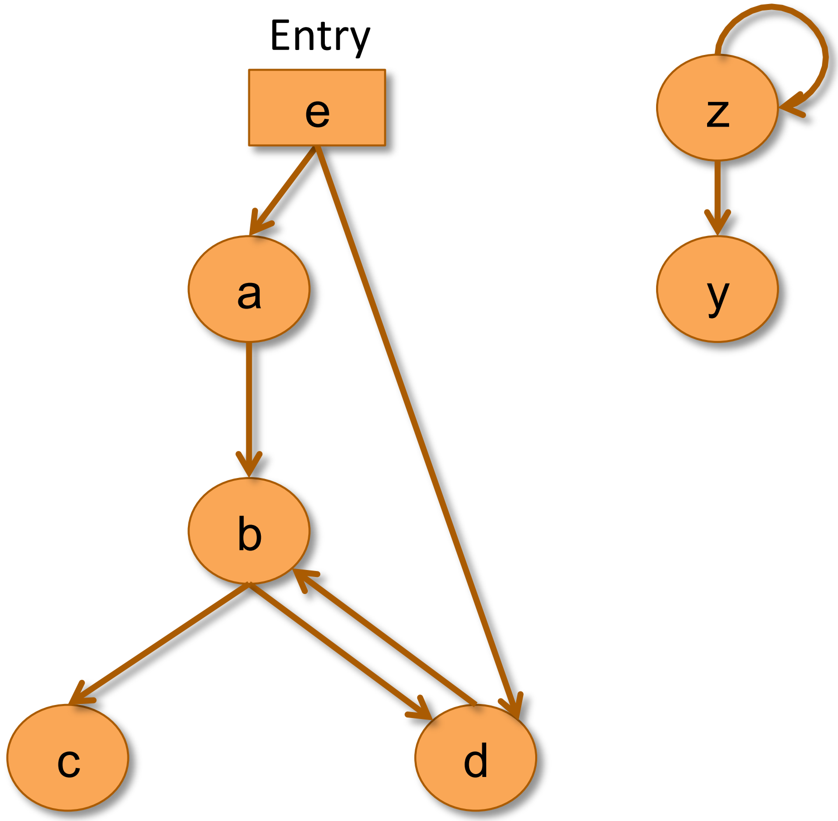
Defining SSA Variable Scope

Graph: g corresponds to a “fine grained” CFG

Nodes: program points
(maybe more than one per block)

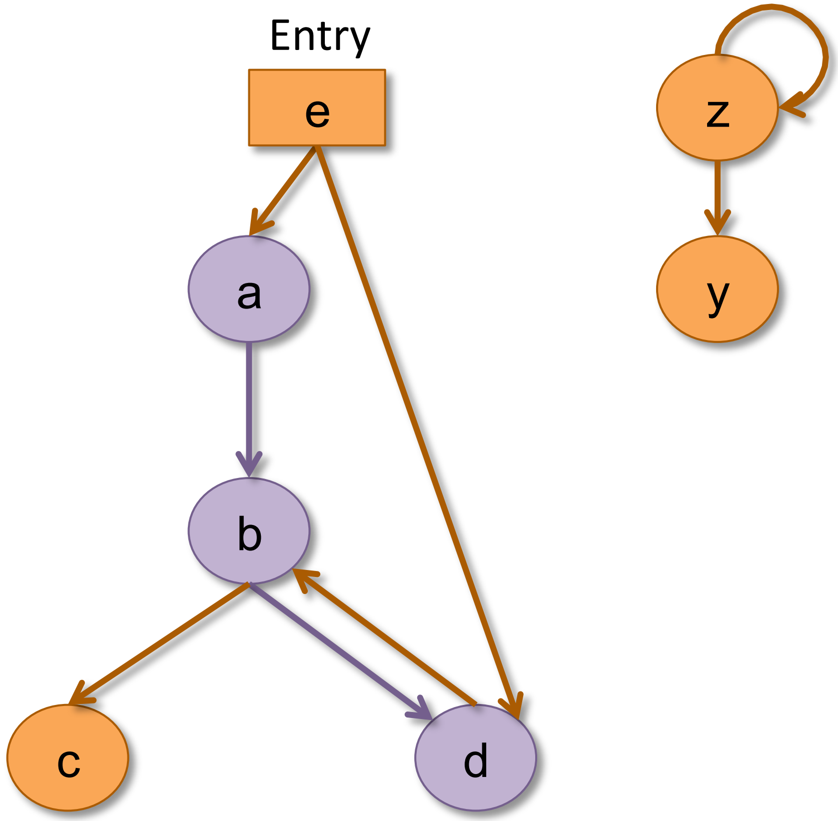
Edges: “fallthroughs”, jump and branch instructions

Distinguished entry



Paths

- Paths:
  `Path g a d [a;b;d]`



Reachability

- Paths:

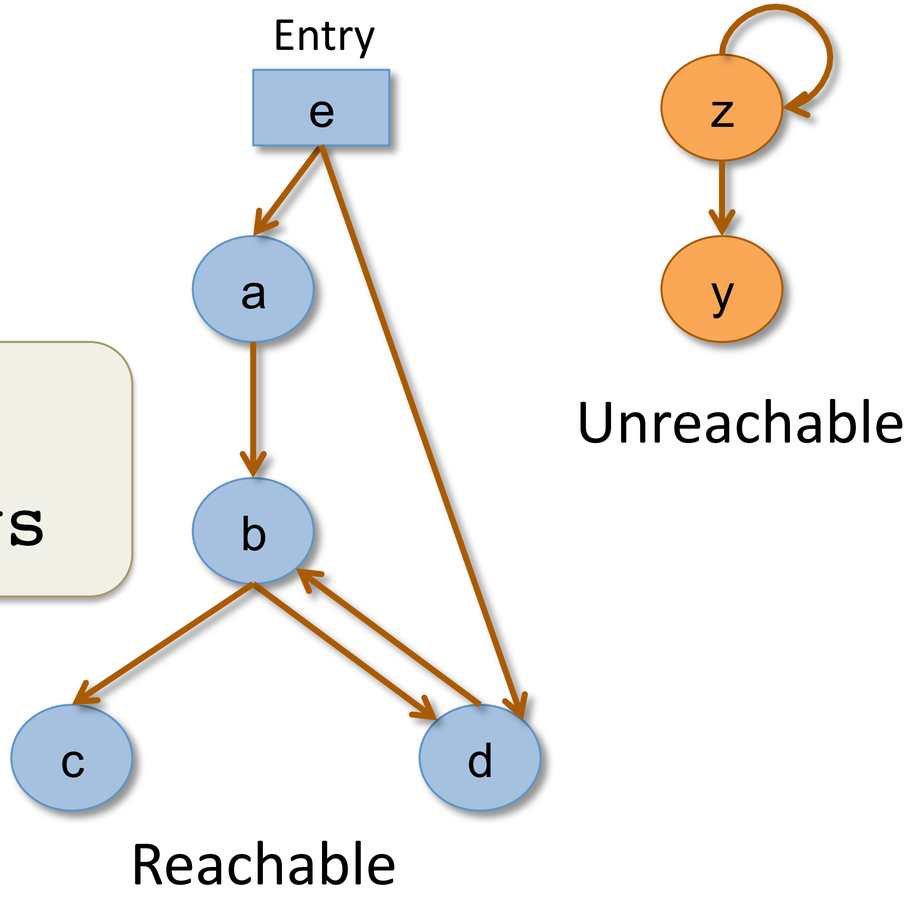
  `Path g a d [a;b;d]`

- Reachability:

  `Reachable g x`

iff

`∃vs. Path g e x vs`



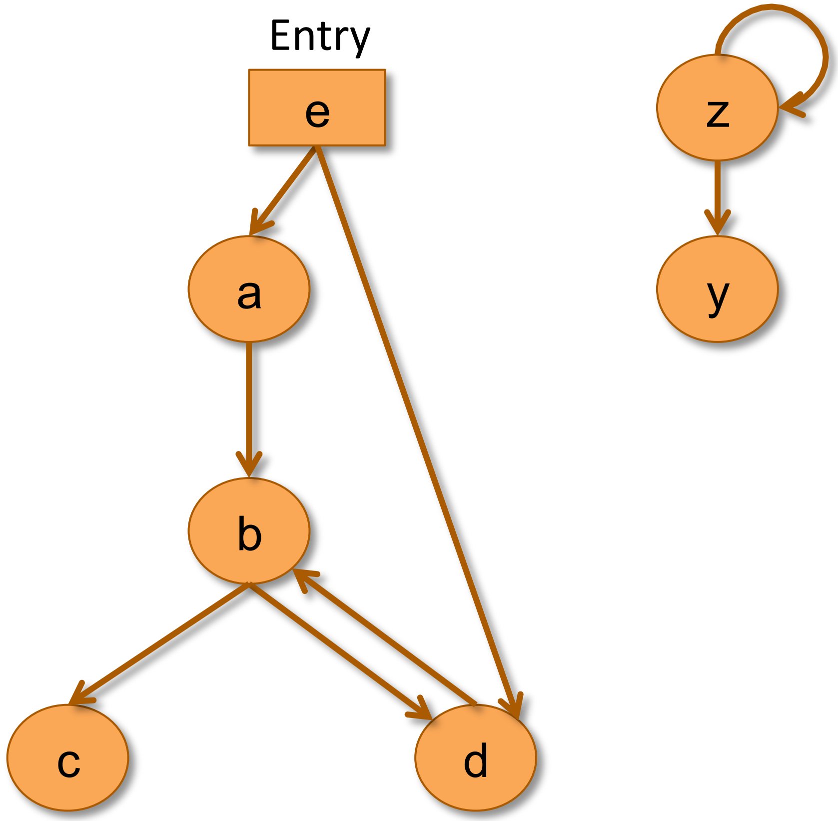
Domination

- Paths:

  `Path g a d [a;b;d]`
- Reachability:

  `Reachable g x`
- Domination:

  `Dom g b c`



iff every path from e to c goes through b.

Domination

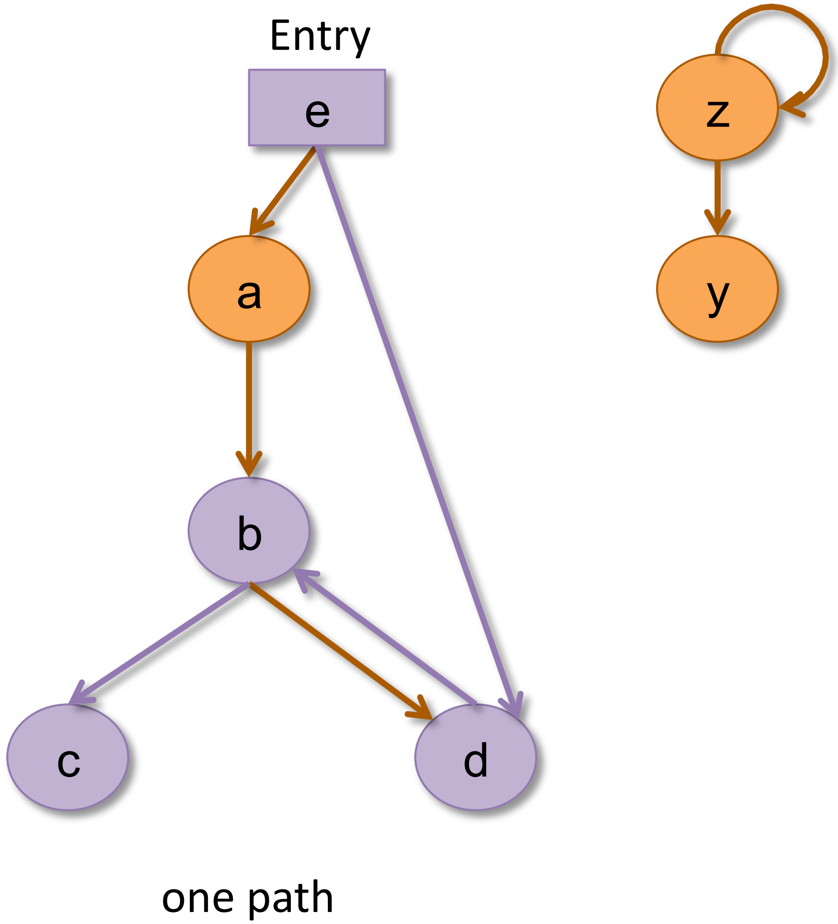
- Paths:

  `Path g a d [a;b;d]`
- Reachability:

  `Reachable g x`
- Domination:

  `Dom g b c`

iff every path from e to c goes through b.



Domination

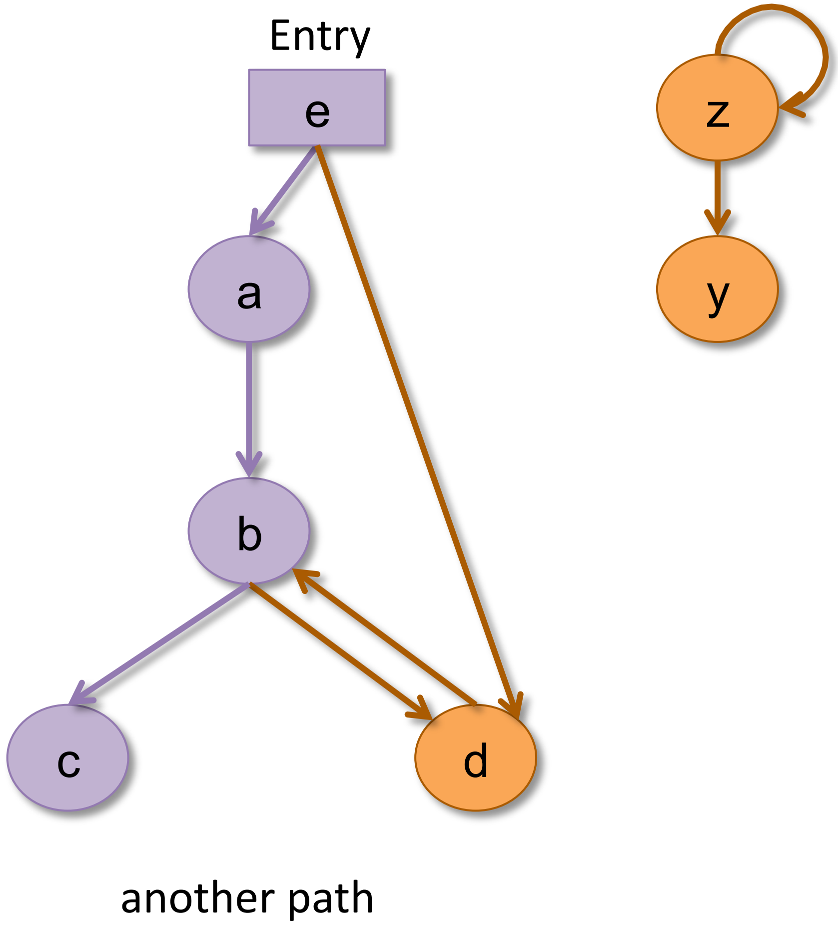
- Paths:

  `Path g a d [a;b;d]`
- Reachability:

  `Reachable g x`
- Domination:

  `Dom g b c`

iff every path from e to c goes through b.



Domination

- Paths:

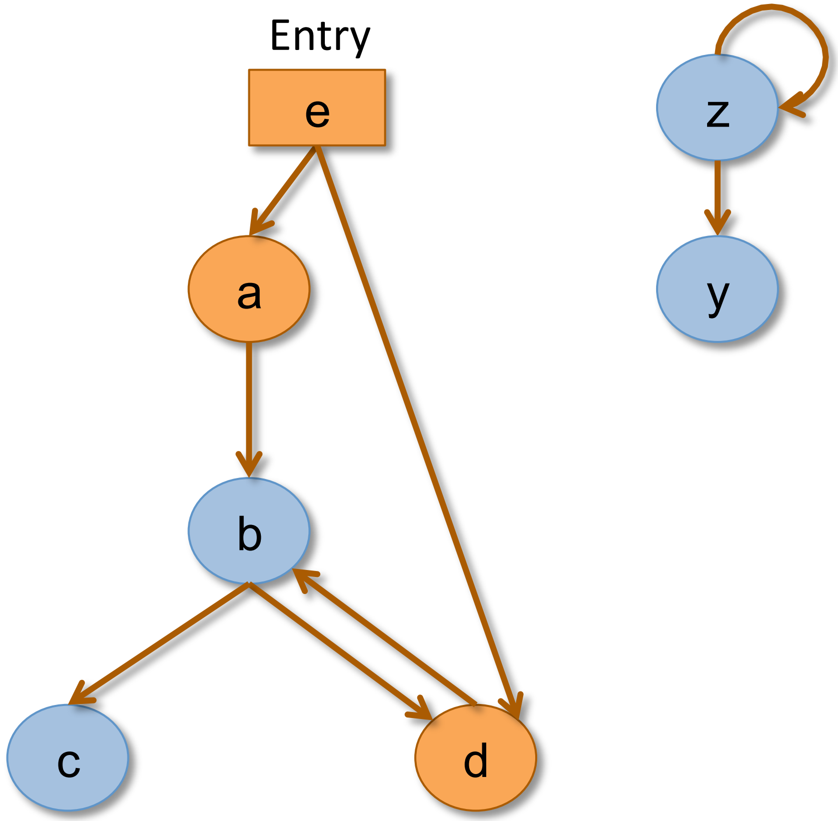
  `Path g a d [a;b;d]`

- Reachability:

  `Reachable g x`

- Domination:

  `Dom g b c`



Nodes dominated by b.

Strict Domination

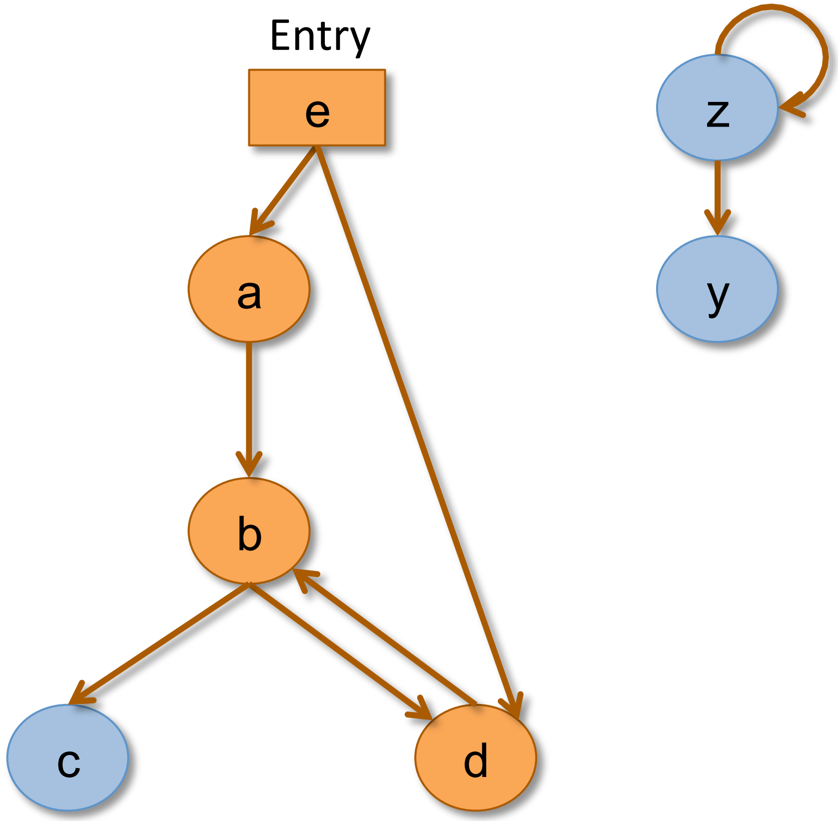
- Paths:

  `Path g a d [a;b;d]`
- Reachability:

  `Reachable g x`
- Domination:

  `Dom g b c`
- Strict Domination:

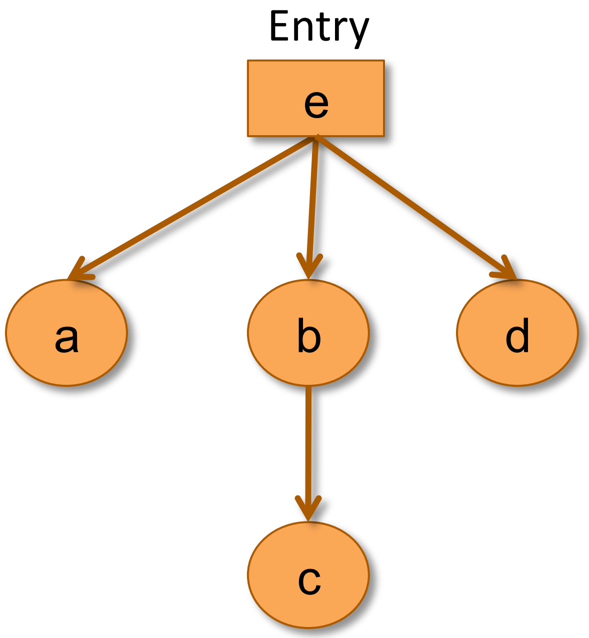
  `SDom g b c`



Nodes strictly
dominated by b.

Domination Tree

- Order the reachable nodes by (immediate) dominators, and you get a tree:

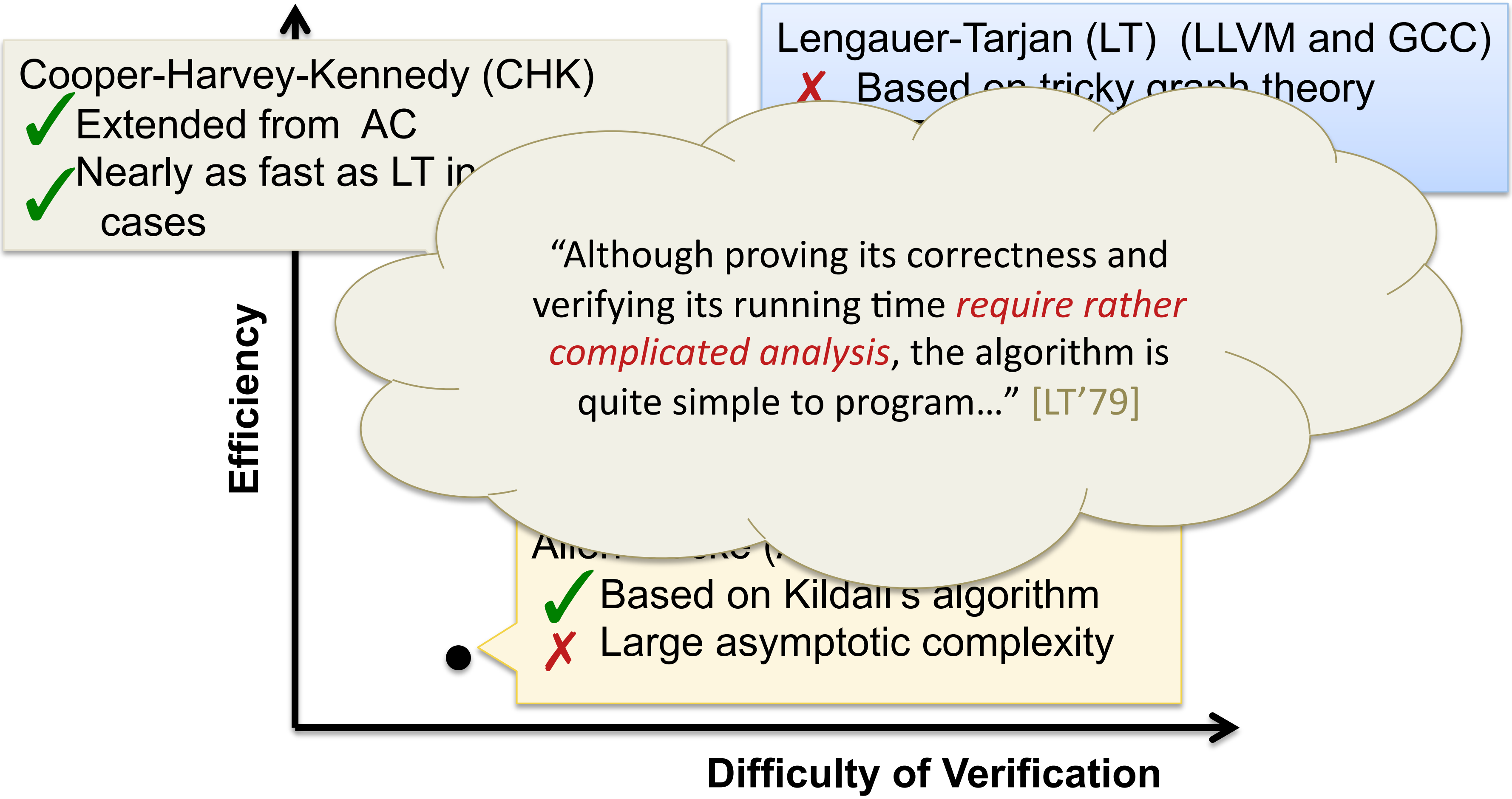


- This is an inductive data structure (unlike CFG) ⇒ better for certain proofs. (e.g. those that have to do with scoping).

Vminus.Dom.v

Coq

Dominator Algorithm Tradeoffs



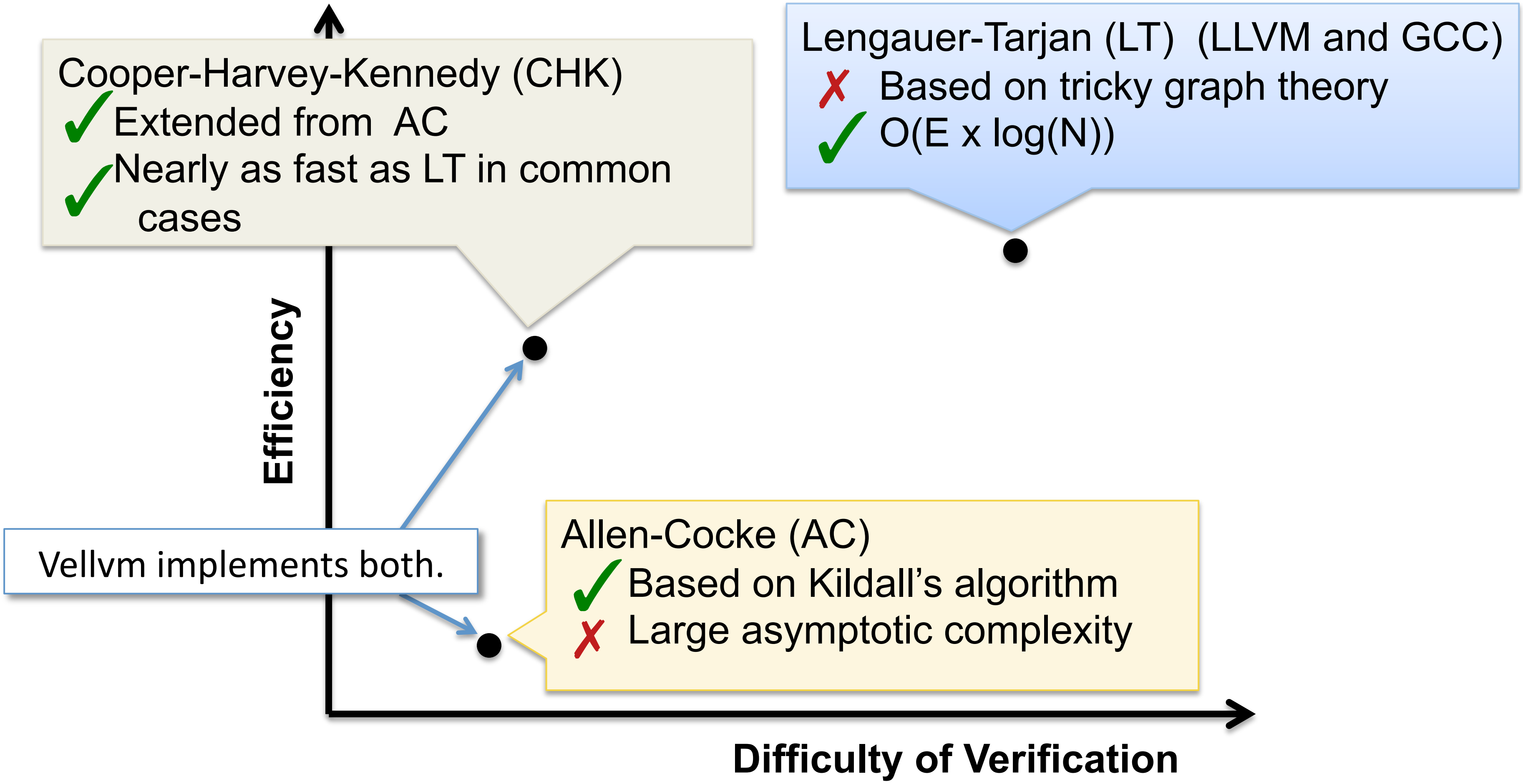
Cooper-Harvey-Kennedy (CHK)
- ✓ Extended from AC
- ✓ Nearly as fast as LT in cases

Lengauer-Tarjan (LT) (LLVM and GCC)
- ✗ Based on tricky graph theory

Allen-Cocke (A
- ✓ Based on Kildall's algorithm
- ✗ Large asymptotic complexity

"Although proving its correctness and verifying its running time *require rather complicated analysis*, the algorithm is quite simple to program…" [LT'79]

Dominator Algorithm Tradeoffs



Cooper-Harvey-Kennedy (CHK)
- ✓ Extended from AC
- ✓ Nearly as fast as LT in common cases

Lengauer-Tarjan (LT) (LLVM and GCC)
- ✗ Based on tricky graph theory
- ✓ O(E x log(N))

Allen-Cocke (AC)
- ✓ Based on Kildall's algorithm
- ✗ Large asymptotic complexity

Vellvm implements both.

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 $\mathrm{wf}(f, \sigma_0)$.
- *Preservation*:

  If $\vdash f$ and $f \vdash \sigma \longmapsto \sigma'$ and $\mathrm{wf}(f, \sigma)$ then $\mathrm{wf}(f, \sigma')$
- *Progress*:

  If $\vdash f$ and $\mathrm{wf}(f, \sigma)$ 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

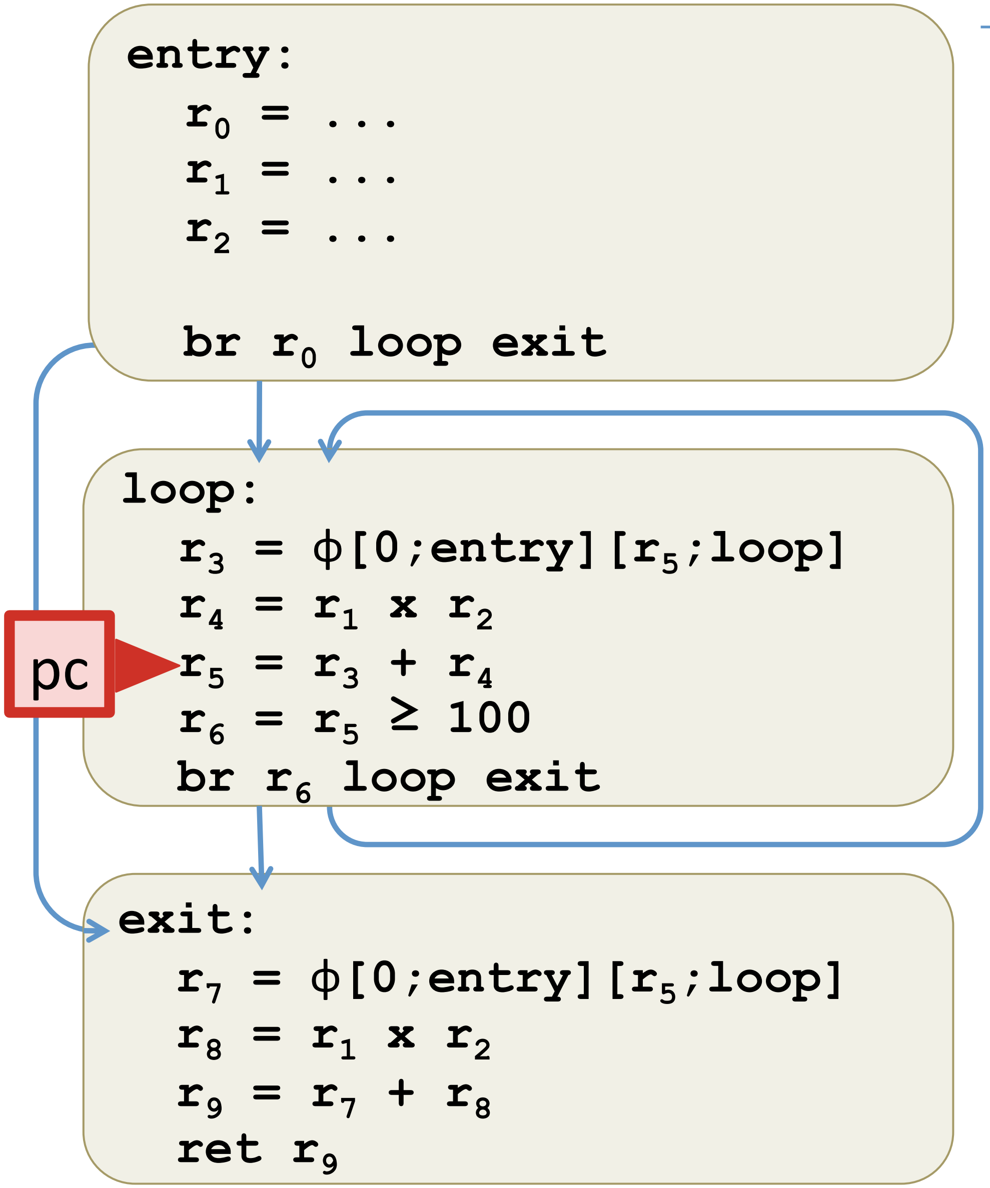
- *Initialization*: If $\vdash f$ then $\mathrm{wf}(f, \sigma_0)$.
- *Preservation*:

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

- *Progress*:

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

Well-formed States



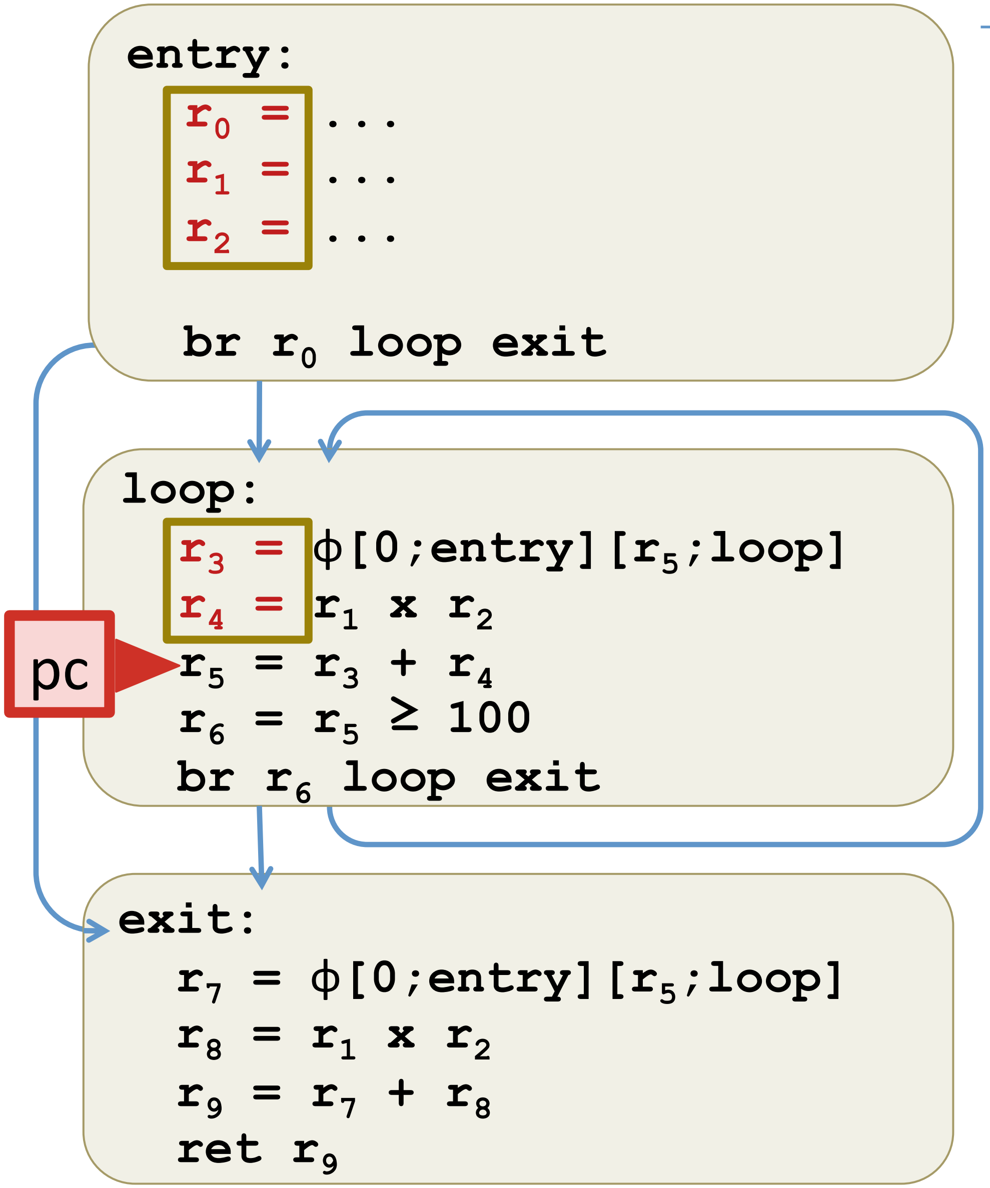
```
entry:
  r0 = ...
  r1 = ...
  r2 = ...

  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
```

State σ is:

pc = program counter

δ = local values

Well-formed States (Roughly)



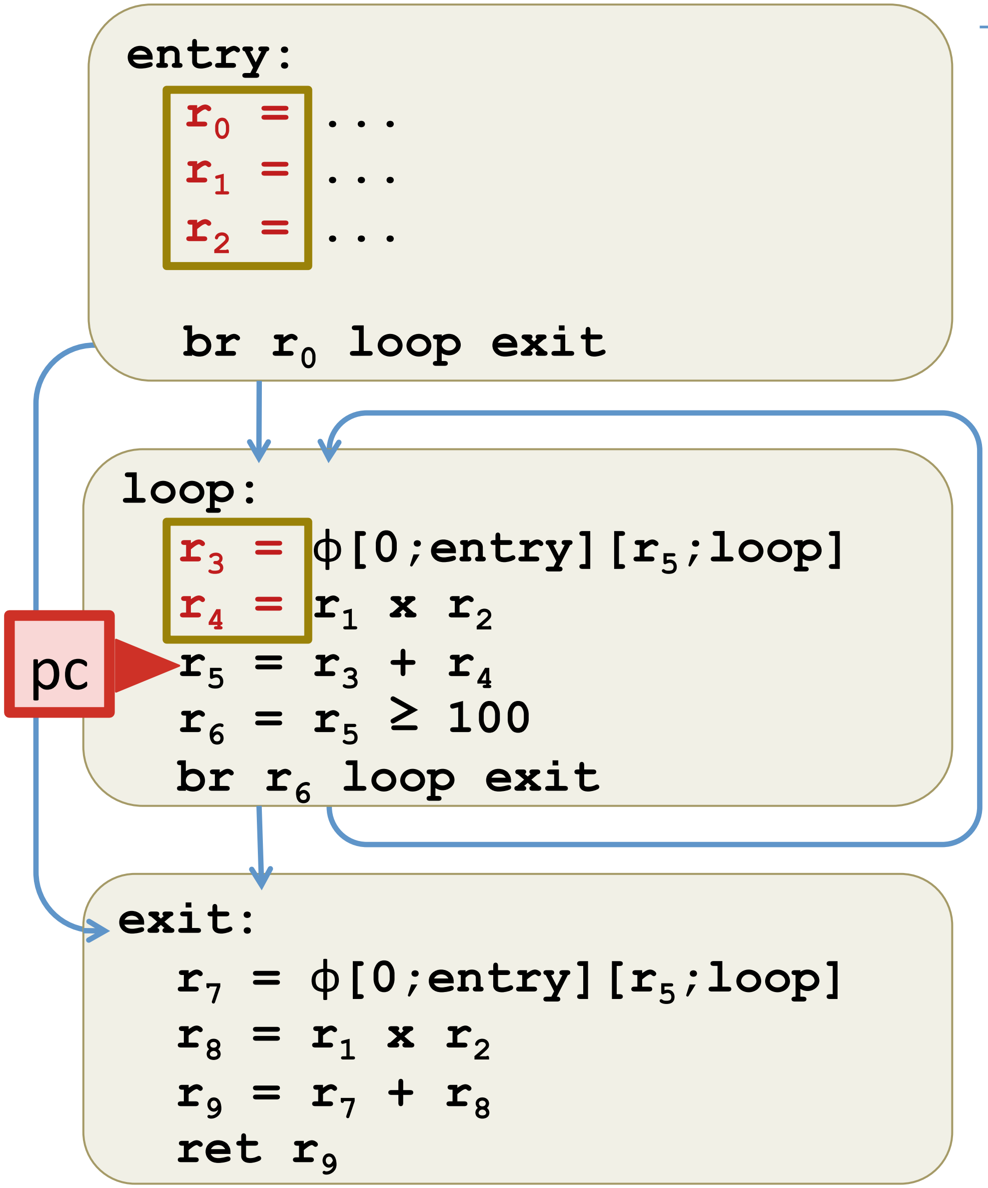
```
entry:
  r0 = ...
  r1 = ...
  r2 = ...

  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
```

State σ is:

pc = program counter
δ = local values

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

Well-formed States (Roughly)



```
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
```

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."

Vminus.Vminus.v

Typing

Compiler Verification

- 1967: Correctness of a Compiler for Arithmetic Expressions
  [McCarthy, Painter]
- 1972: Proving Compiler Correctness in a Mechanized Logic
  [Milner, Weyhrauch]

- … many interesting developments

See: Compiler Verification, A Bibliography [Dave, 2003]

- 2006-present: CompCert [Leroy, et al.]
  - (Nearly!) fully verified compiler from C to Power PC, ARM, etc.
  - Randomized compiler testing found no bugs (in the verified components – the original, unverified parser had a bug)
- Others: Verified Software Toolchain [Appel, et al.]

Vminus.Imp.v

Coq

Execution Models

- Interpretation:
  - program represented by abstract syntax
  - tree traversed by interpreter
- Compilation to native code:
  - program translated to machine instructions
  - executed by hardware
- Compilation to virtual machine code:
  - program translated to “virtual machine” instructions
  - interpreted (efficiently)
  - further translated to machine code
  - just-in-time compiled to machine code

Correct Execution?

- What does it mean for an Imp program to be executed correctly?

- Even at the interpreter level we could show *equivalence* between the small-step and the large-step operational semantics:

$$\text{cmd} \,/\, \text{st} \longmapsto^{*} \texttt{SKIP} \,/\, \text{st}'$$

iff

$$\text{cmd} \,/\, \text{st} \Downarrow \text{st}'$$

Compiler Correctness?

- We have to relate the source and target language semantics across the compilation function $\mathfrak{C}[\text{-}] : \text{source} \rightarrow \text{target}$.

$$\text{cmd} / \text{st} \;\; {}_{S}\!\longmapsto^{*} \;\; \texttt{SKIP} / \text{st}'$$

iff

$$\mathfrak{C}[\text{cmd}] / \mathfrak{C}[\text{st}] \;\; {}_{T}\!\longmapsto^{*} \;\; \mathfrak{C}[\text{st}']$$

- Is this enough?
- What if cmd goes into an infinite loop?

Comparing Behaviors

- Consider two programs P and P’ possibly in different languages.
    - e.g. P is an Imp program, P’ is its compilation to Vminus
- The semantics of the languages associate to each program a set of observable behaviors:

$\mathfrak{B}(P)$ and $\mathfrak{B}(P')$

- Note: $|\mathfrak{B}(P)| = 1$ if P is deterministic, > 1 otherwise

What is Observable?

- For Imp-like languages:

```
observable behavior ::=
    | terminates(st)        (i.e. observe the final state)
    | diverges
    | goeswrong
```

- For pure functional languages:

```
observable behavior ::=
    | terminates(v)         (i.e. observe the final value)
    | diverges
    | goeswrong
```

What about I/O?

- Add a *trace* of input-output events performed:

t ::= [] | e :: t (finite traces)
coind. T ::= [] | e :: T (finite and infinite traces)

observable behavior ::=
| terminates(t, st) (end in state st after trace t)
| diverges(T) (loop, producing trace T)
| goeswrong(t)

Examples

- P1:
  `print(1);` / st ⇒ terminates(out(1)::[],st)

- P2:
  `print(1); print(2);` / st
  ⇒ terminates(out(1)::out(2)::[],st)

- P3:
  `WHILE true DO print(1) END` / st
  ⇒ diverges(out(1)::out(1)::…)

- So $\mathfrak{B}(P1) \neq \mathfrak{B}(P2) \neq \mathfrak{B}(P3)$

Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

$$\mathfrak{B}(\mathrm{P1}) = \mathfrak{B}(\mathrm{P2})$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions   f() + g()

- Concurrent programs often permit nondetermism
  - Classic optimizations can reduce this nondterminism
  - Example:

```
a := x + 1; b := x + 1     ||          x := x+1

                      vs.

a := x + 1; b := a         ||          x := x+1
```

- As we'll see, LLVM explicitly allows nondeterminism.

Backward Simulation

- Program P2 can exhibit fewer behaviors than P1:

$$\mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
- Also called *refinement*.

What about goeswrong?

- Compilers often translate away bad behaviors.

x := 1/y ; x := 42 vs. x := 42
(divide by 0 error) (always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"

Safe Backwards Simulation

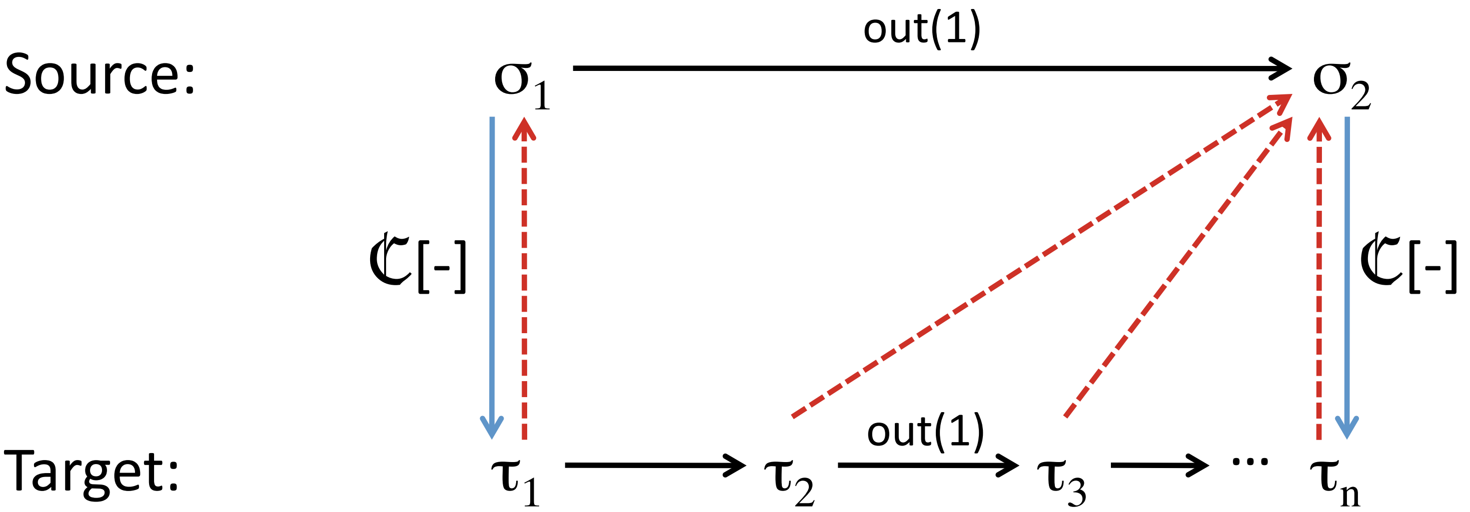
- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- Idea: let S be the functional specification of the program: A set of behaviors not containing goeswrong(t).
  - A program P satsifies the spec if $\mathfrak{B}(P) \subseteq S$

- Lemma: If P2 is a safe backwards simulation of P1 and P1 satisfies the spec, then P2 does too.

Building Backward Simulations



$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[rrrr, "\text{out(1)}"] \arrow[d, "\mathfrak{C}[-]"'] & & & & \sigma_2 \arrow[d, "\mathfrak{C}[-]"] \\
\text{Target:} & \tau_1 \arrow[r] \arrow[u, dashed] & \tau_2 \arrow[r, "\text{out(1)}"] \arrow[urrr, dashed] & \tau_3 \arrow[r] \arrow[urr, dashed] & \cdots & \tau_n \arrow[u, dashed]
\end{tikzcd}$$

Idea: The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

Tricky parts:

- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have invent some way of relating the intermediate states to the source.
- the compilation function goes the wrong way to help!

End of Part 2

2

Safe Backwards Simulation

- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong(t)} \notin \mathfrak{B}(\text{P1}) \quad \Rightarrow \quad \mathfrak{B}(\text{P1}) \supseteq \mathfrak{B}(\text{P2})$$

- Idea: let S be the functional specification of the program: A set of behaviors not containing goeswrong(t).
  - A program P satsifies the spec if $\mathfrak{B}(\text{P}) \subseteq \text{S}$

- Lemma: If P2 is a safe backwards simulation of P1 and P1 satisfies the spec, then P2 does too.

Safe Forwards Simulation

- Source program’s behaviors are a subset of the target’s:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \subseteq \mathfrak{B}(P2)$$

- P2 captures all the good behaviors of P1, but could exhibit more (possibly bad) behaviors.

- But: Forward simulation is significantly easier to prove:
  - Only need to show the existence of a compatible target trace.

Determinism!

- Lemma: If P2 is deterministic then forward simulation implies backward simulation.
- Proof: $\varnothing \subset \mathfrak{B}(P1) \subseteq \mathfrak{B}(P2) = \{b\}$ so $\mathfrak{B}(P1) = \{b\}$.
- Corollary: safe forward simulation implies safe backward simulation if P2 is deterministic.

Forward Simulations

$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[rrrrr] \arrow[d, dash, "\sim"'] & & & & & \sigma_2 \arrow[d, dash, dashed, "\sim"] \\
\text{Target:} & \mathfrak{C}[\sigma_1] \arrow[r, dashed] & \tau_2 \arrow[r, dashed] & \tau_3 \arrow[r, dashed] & \cdots & & \mathfrak{C}[\sigma_2]
\end{tikzcd}$$

Idea: Show that every transition in the source program:

- is simulated by some sequence of transitions in the target
- while preserving a relation ~ between the states

Imp: A Refresher

```
id     := X|Y|Z|…                             Variables

aexp   := n | id | aexp + aexp |              Arithmetic Expressions
          aexp — aexp | aexp * aexp

bexp   := true | false | aexp = aexp          Boolean Expressions
          !bexp | bexp && bexp

cmd :=
  | SKIP                                     Do nothing
  | id ::= aexp                              Assignment
  | cmd ;; cmd                               Sequence
  | IFB bexp THEN cmd ELSE cmd FI            Conditional
  | WHILE bexp DO cmd END                    Loop
```

See Vminus/Imp.v for the Coq formalism

Vminus.CompilImp.v

Coq

Lock-step Forward Simulation

$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[r] \arrow[d, dash, "\sim"'] & \sigma_2 \arrow[d, dash, dashed, "\sim"] \\
\text{Target:} & \mathfrak{C}[\sigma_1] \arrow[r, dashed] & \mathfrak{C}[\sigma_2]
\end{tikzcd}$$

A single source-program step is simulated by a single target step.

(Solid = assumptions, Dashed = must be shown)

“Plus”-step Forward Simulation

$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[rrrr] \arrow[d, dash, "\sim"'] & & & & \sigma_2 \arrow[d, dash, dashed, "\sim"] \\
\text{Target:} & \mathfrak{C}[\sigma_1] \arrow[r, dashed] & \tau_0 \arrow[r, dashed] & \tau_1 \arrow[r, dashed] & \ldots & \tau_n
\end{tikzcd}$$

A single source-program step is simulated by *one or more* target steps. (But only finitely many!)

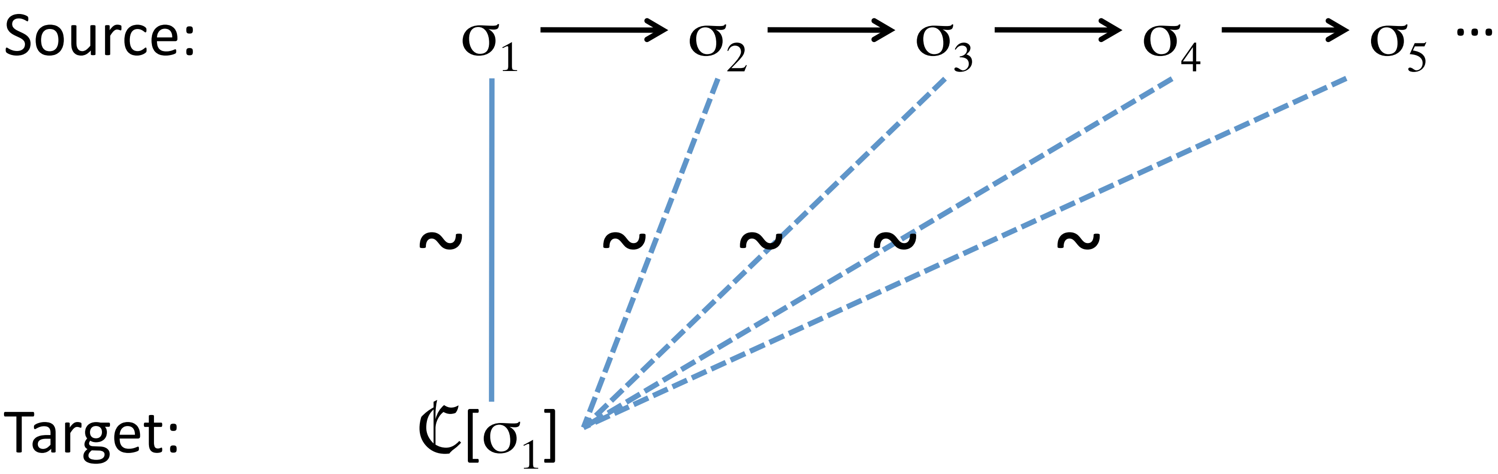
(Solid = assumptions, Dashed = must be shown)

Optional Forward Simulation

$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[rr] \arrow[d, dash, "\sim"'] & & \sigma_2 \arrow[dll, dash, dashed, "\sim"] \\
\text{Target:} & \mathfrak{C}[\sigma_1] & &
\end{tikzcd}$$

A single source-program step is simulated by zero steps in the target.

Problem with “Infinite Stuttering”



Source: $\sigma_1 \longrightarrow \sigma_2 \longrightarrow \sigma_3 \longrightarrow \sigma_4 \longrightarrow \sigma_5 \cdots$

$\sim \quad \sim \quad \sim \quad \sim \quad \sim$

Target: $\mathfrak{C}[\sigma_1]$

An infinite sequence of source transitions can be “simulated” by 0 transitions in the target!

(This simulation doesn’t preserve nontermination.)

Solution: Disallow such “trivial” simulations

$$\begin{tikzcd}
\text{Source:} & \sigma_1 \arrow[rr] \arrow[d, dash, "\sim"'] & & \sigma_2 \arrow[dll, dash, dashed, "\sim"] \\
\text{Target:} & \mathfrak{C}[\sigma_1] & & \boxed{|\sigma_2| < |\sigma_1|}
\end{tikzcd}$$

Equip the source language with a measure $|\sigma|$ and require that $|\sigma_2| < |\sigma_1|$.

The measure can’t decrease indefinitely, so the target program must either take a step or the source must terminate.

The target diverges if the source program does.

Vminus.CompilImp.v

Coq

Is Backward Simulation Hopeless?

- Suppose the source & target languages are the same.
  - So they share the same definition of program state.
- Further suppose that the steps are very “small”.
  - Abstract machine (i.e. no “complex” instructions).
- Further suppose that “compilation” is only a very minor change.
  - add or remove a single instruction
  - substitute a value for a variable

- Then: backward simulation is more achievable
  - it’s easier to invent the “decompilation” function because the “compilation” function is close to trivial

- Happily: This is the situation for LLVM optimizations

Lock-Step Backward Simulation

$$\begin{tikzcd}
S_1 \arrow[r, dashed, "o"] \arrow[d, dash, "\sim"'] & S_2 \arrow[d, dash, dashed, "\sim"] \\
T_1 \arrow[r, "o"] & T_2
\end{tikzcd}$$

o is either an “observable event” or a “silent event”

$o ::= e \mid \varepsilon$

Example use: proving variable subsitution correct.

Right-Option Backward Simulation

$$\begin{tikzcd}
S_1 \arrow[r, dashed, "o"] \arrow[d, dash, "\sim"'] & S_2 \arrow[d, dash, dashed, "\sim"] \\
T_1 \arrow[r, "o"] & T_2
\end{tikzcd}$$

OR

$$\begin{tikzcd}
S_1 \arrow[r, dashed, "\varepsilon"] \arrow[d, dash, "\sim"'] & S_2 \arrow[dl, dash, dashed, "\sim"] \\
T_1 &
\end{tikzcd}$$

$$\boxed{|S_2| < |S_1|}$$

- Either:
  - the source and target are in lock-step simulation.

  Or
  - the source takes a silent transition to a smaller state

Example use: removing an instruction in the target.

Right-Option Backward Simulation

$$\begin{tikzcd}
S_1 \arrow[r, dashed, "o"] \arrow[d, dash, "\sim"'] & S_2 \arrow[d, dash, dashed, "\sim"] \\
T_1 \arrow[r, "o"] & T_2
\end{tikzcd}$$

OR

$$\begin{tikzcd}
S_1 \arrow[r, dashed, "\varepsilon"] \arrow[d, dash, "\sim"'] & S_2 \arrow[dl, dash, dashed, "\sim"] \\
T_1 &
\end{tikzcd}$$

$$\boxed{|S_2| < |S_1|}$$

- Either:
  - the source and target are in lock-step simulation.

  Or
  - the source takes a silent transition to a smaller state

Example use: removing an instruction in the target.

Left-Option Backward Simulation

$$\begin{tikzcd} S_1 \arrow[r, dashed, "o"] \arrow[d, dash, "\sim"'] & S_2 \arrow[d, dash, dashed, "\sim"] \\ T_1 \arrow[r, "o"] & T_2 \end{tikzcd}$$

OR

$$\begin{tikzcd} S_1 \arrow[d, dash, "\sim"'] \arrow[dr, dash, dashed, "\sim"] & \\ T_1 \arrow[r, "\varepsilon"] & T_2 \end{tikzcd}$$

$\boxed{|T_2| < |T_1|}$

- Either:
  - the source and target are in lock-step simulation.

Or

  - the target takes a silent transition to a smaller state

Example use: adding an instruction to the target.

Generalizing Safety

- Definition of wf:

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

- Generalize like this:

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

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

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

- Methodology: for a given P prove t

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

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 creating the simulation relation for correctness of:
  - code motion, dead variable elimination, common expression elimination, etc.

End of Part 3

3

Strategy for Proving Optimizations

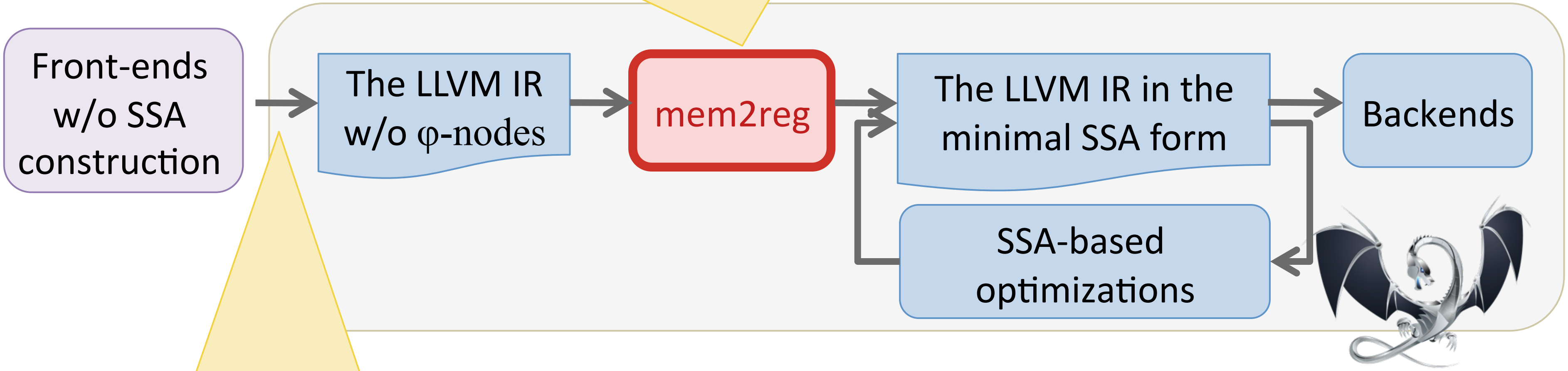
- Decompose the program transformation into a sequence of "micro" transformations
  - e.g. code motion =
    1. insert "redundant" instruction
    2. substitute equivalent definitions
    3. remove the "dead" instruction

- Use the backward simulations to show each "micro" transformation correct.
  - Often uses a generalization of the Vminus safety property

- Compose the individual proofs of correctness

mem2reg in LLVM

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



Front-ends w/o SSA construction

The LLVM IR w/o φ-nodes

mem2reg

The LLVM IR in the minimal SSA form

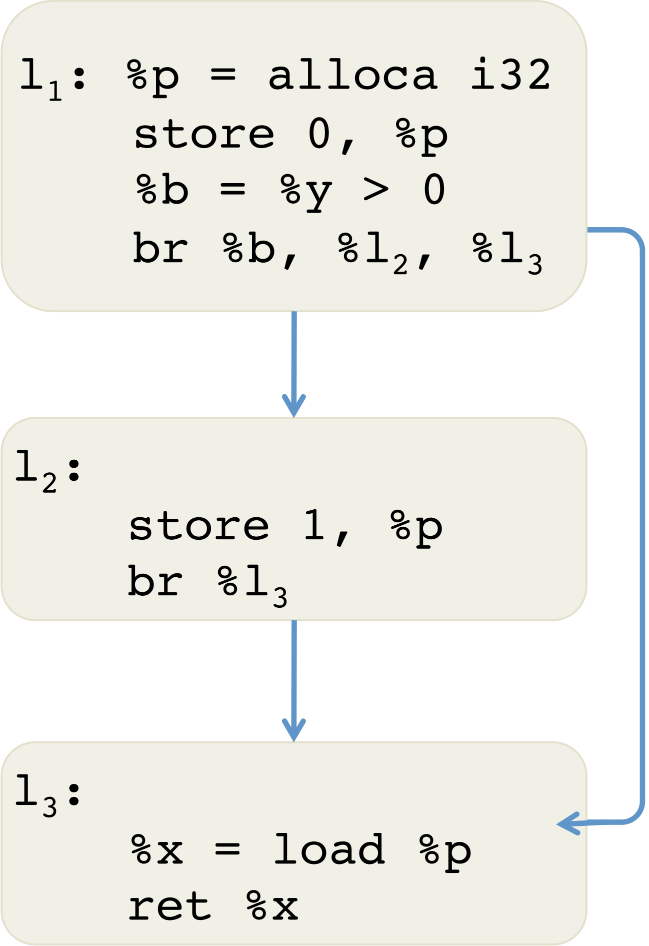
Backends

SSA-based optimizations

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

mem2reg Example

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



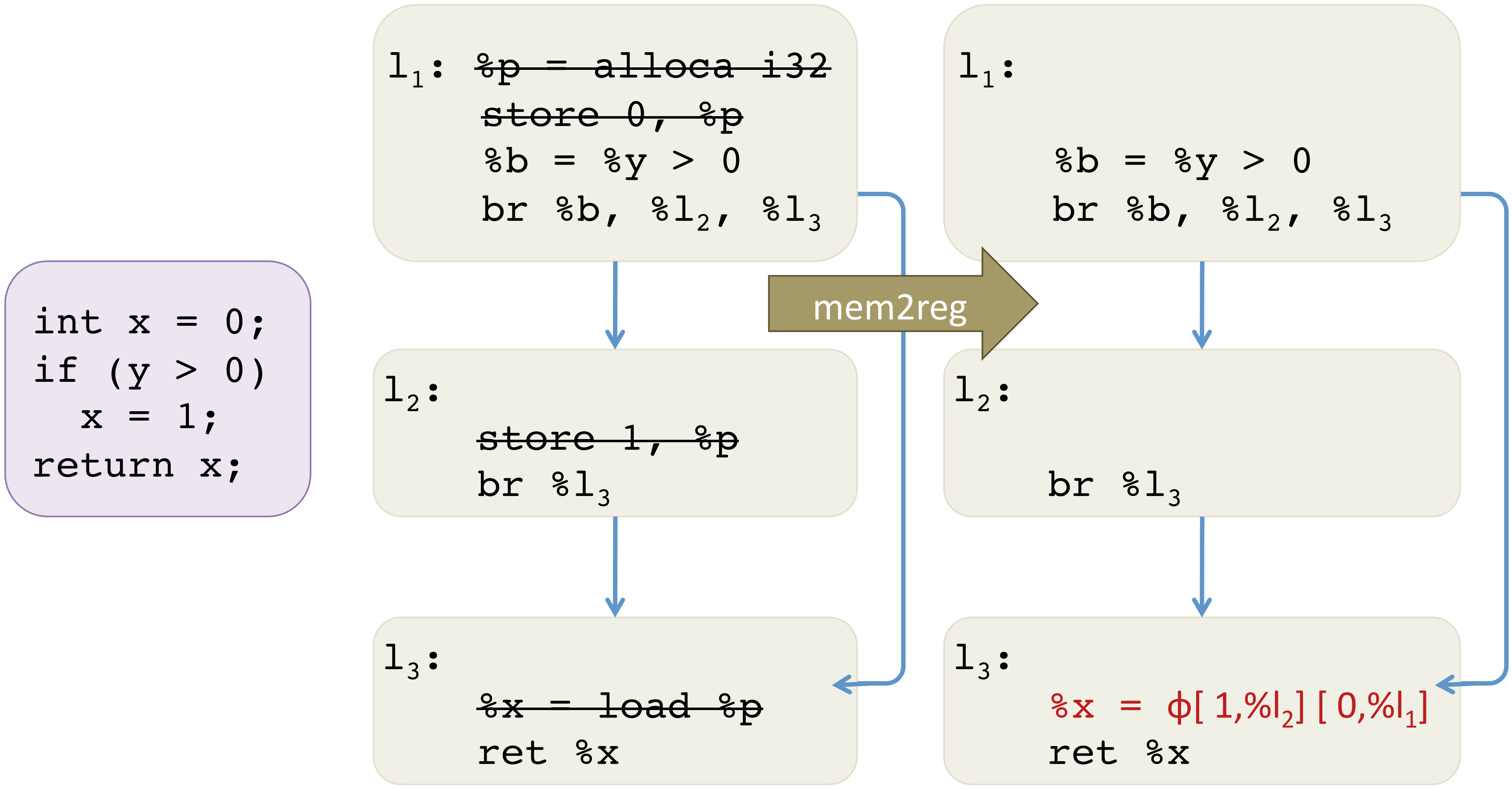
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3
```

```
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;
```

l_1: ~~%p = alloca i32~~
~~store 0, %p~~
%b = %y > 0
br %b, %l_2, %l_3

l_2:
~~store 1, %p~~
br %l_3

l_3:
~~%x = load %p~~
ret %x

mem2reg

l_1:
%b = %y > 0
br %b, %l_2, %l_3

l_2:
br %l_3

l_3:
%x = ϕ[1,%l_2] [0,%l_1]
ret %x

The LLVM IR in the trivial SSA form

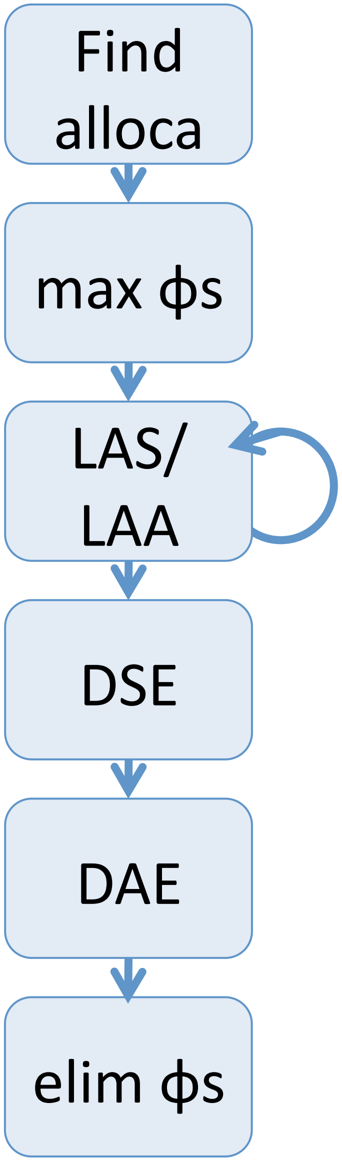
Minimal SSA after mem2reg

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


Find alloca

max ϕs

LAS/ LAA

DSE

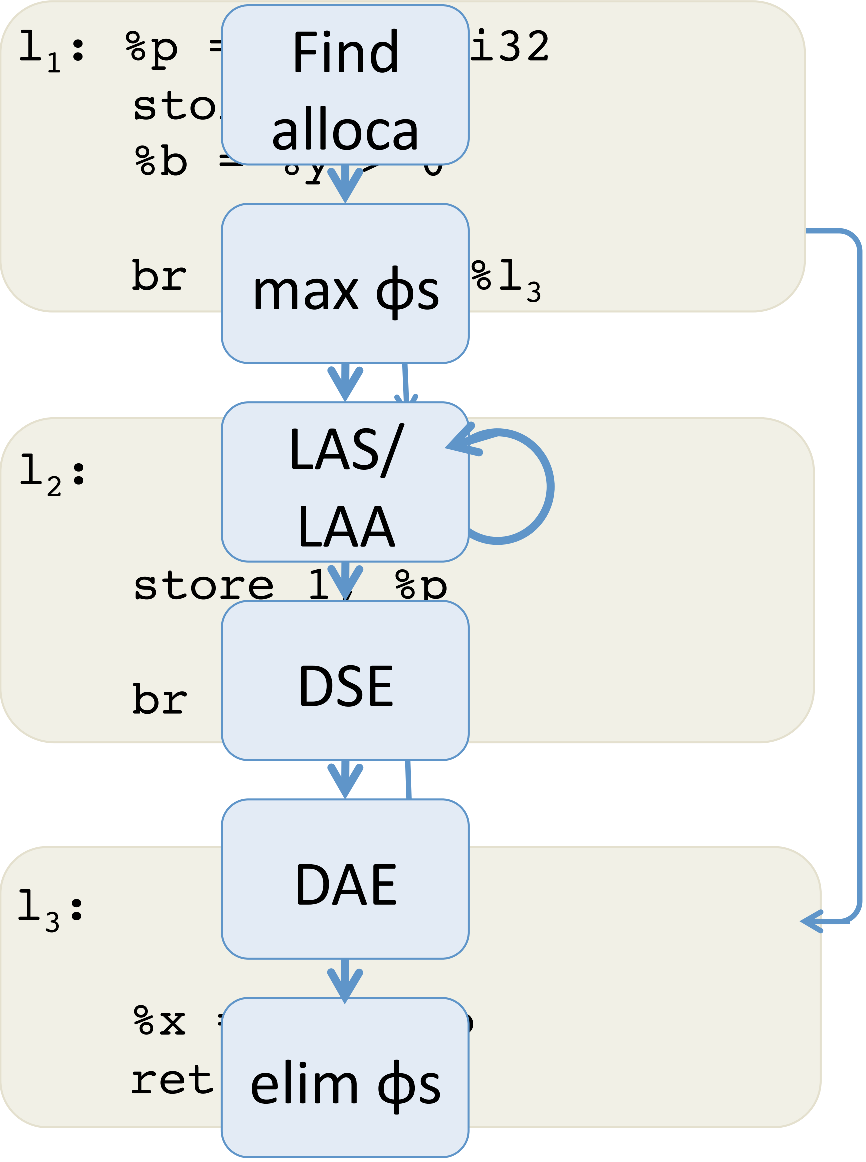
DAE

elim ϕs

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

- Inspired by Aycock & Horspool 2002.

Example of vmem2reg Algorithm



Find alloca

max ϕs

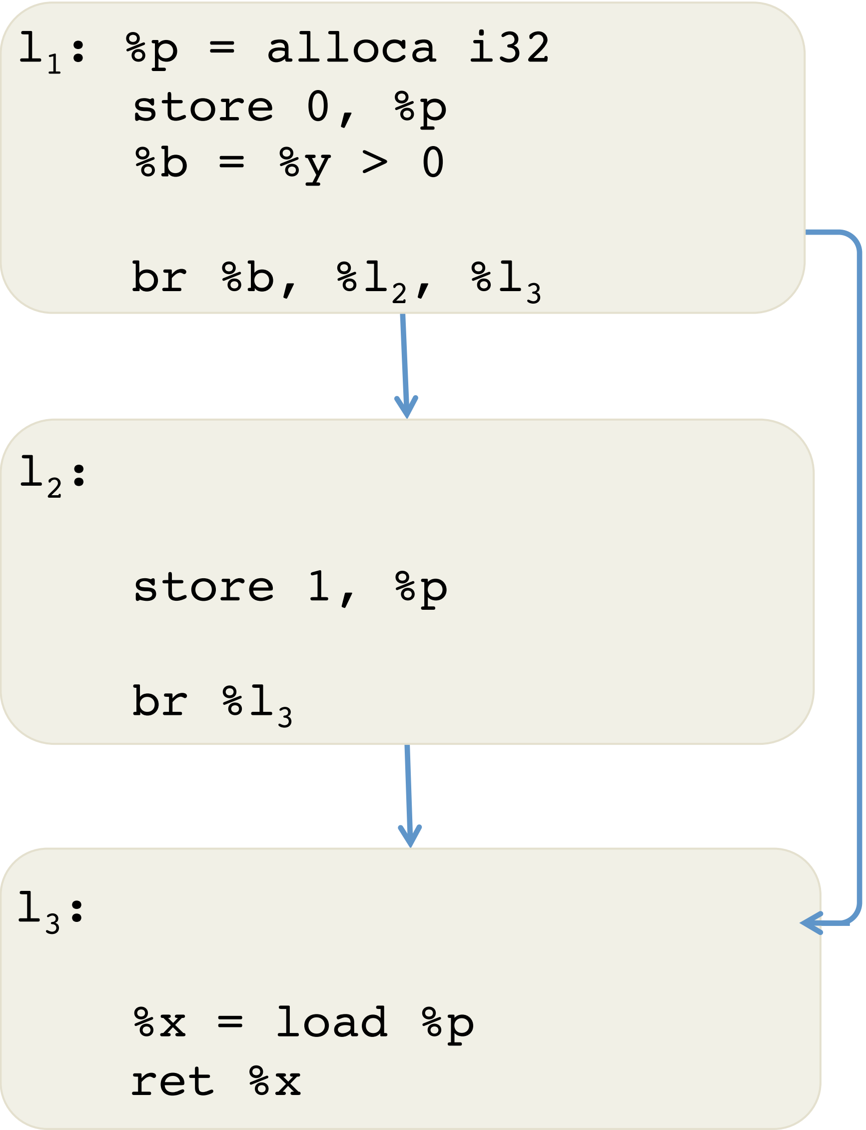
LAS/LAA

DSE

DAE

elim ϕs

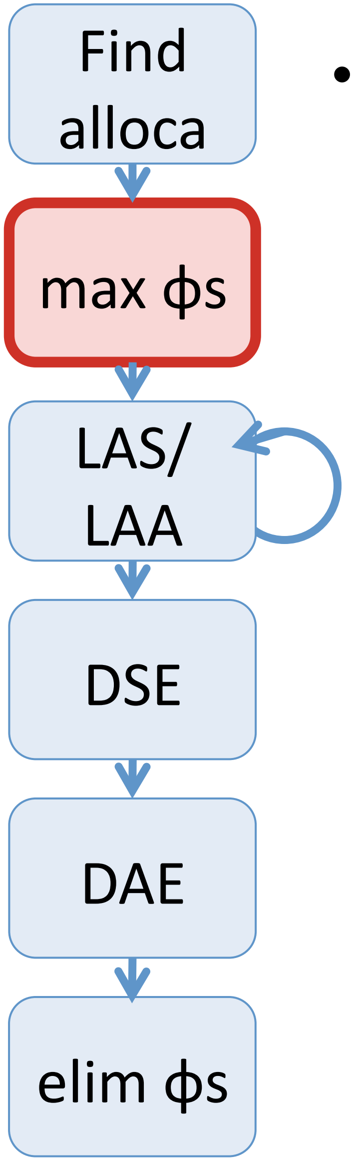
Example of vmem2reg Algorithm



```
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
```



Find alloca

max ϕs

LAS/ LAA

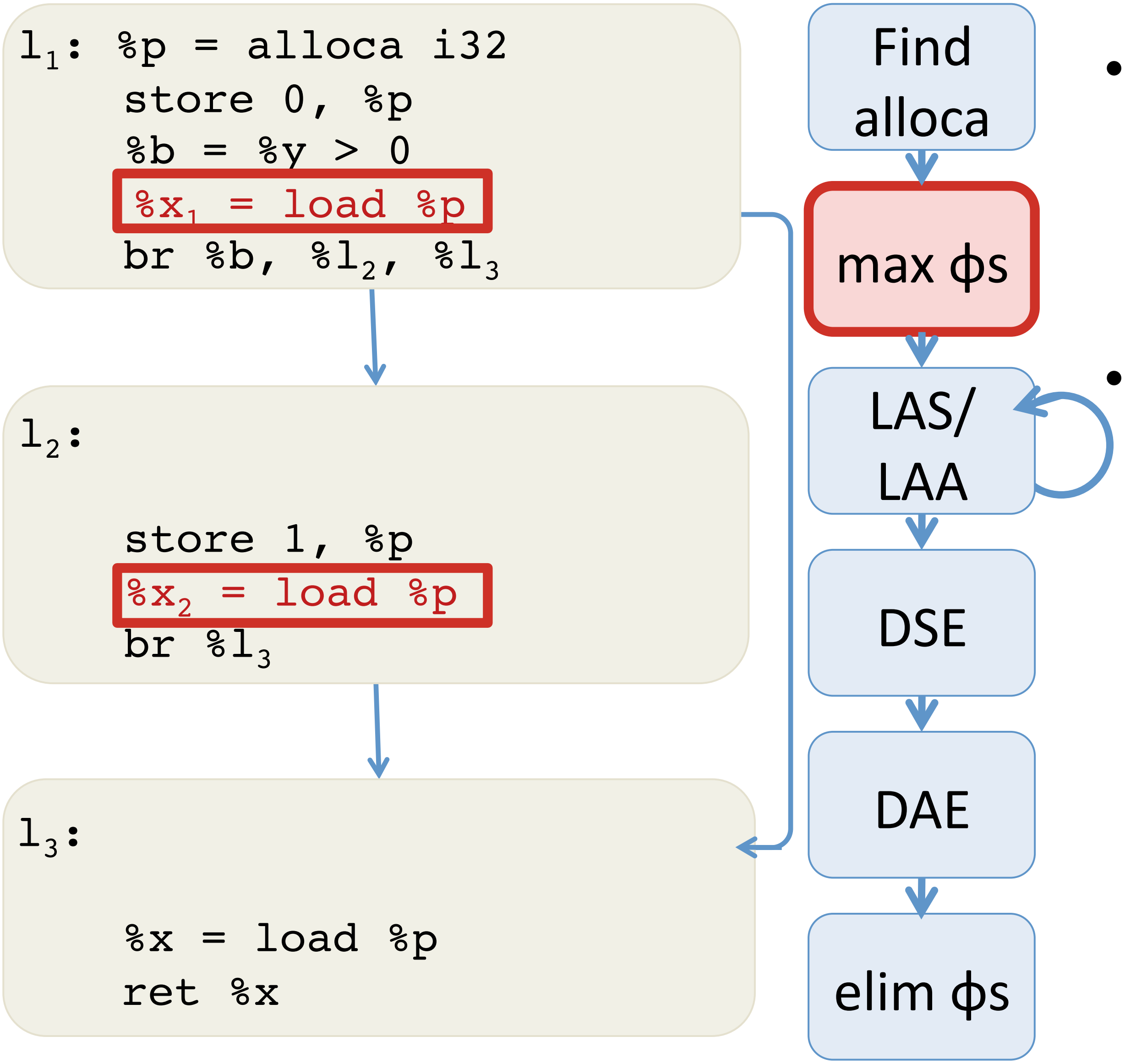
DSE

DAE

elim ϕs

- How to place phi nodes without breaking SSA?

Example of vmem2reg Algorithm



```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2:

    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3:

    %x = load %p
    ret %x
```

Find alloca

max ϕs

LAS/ LAA

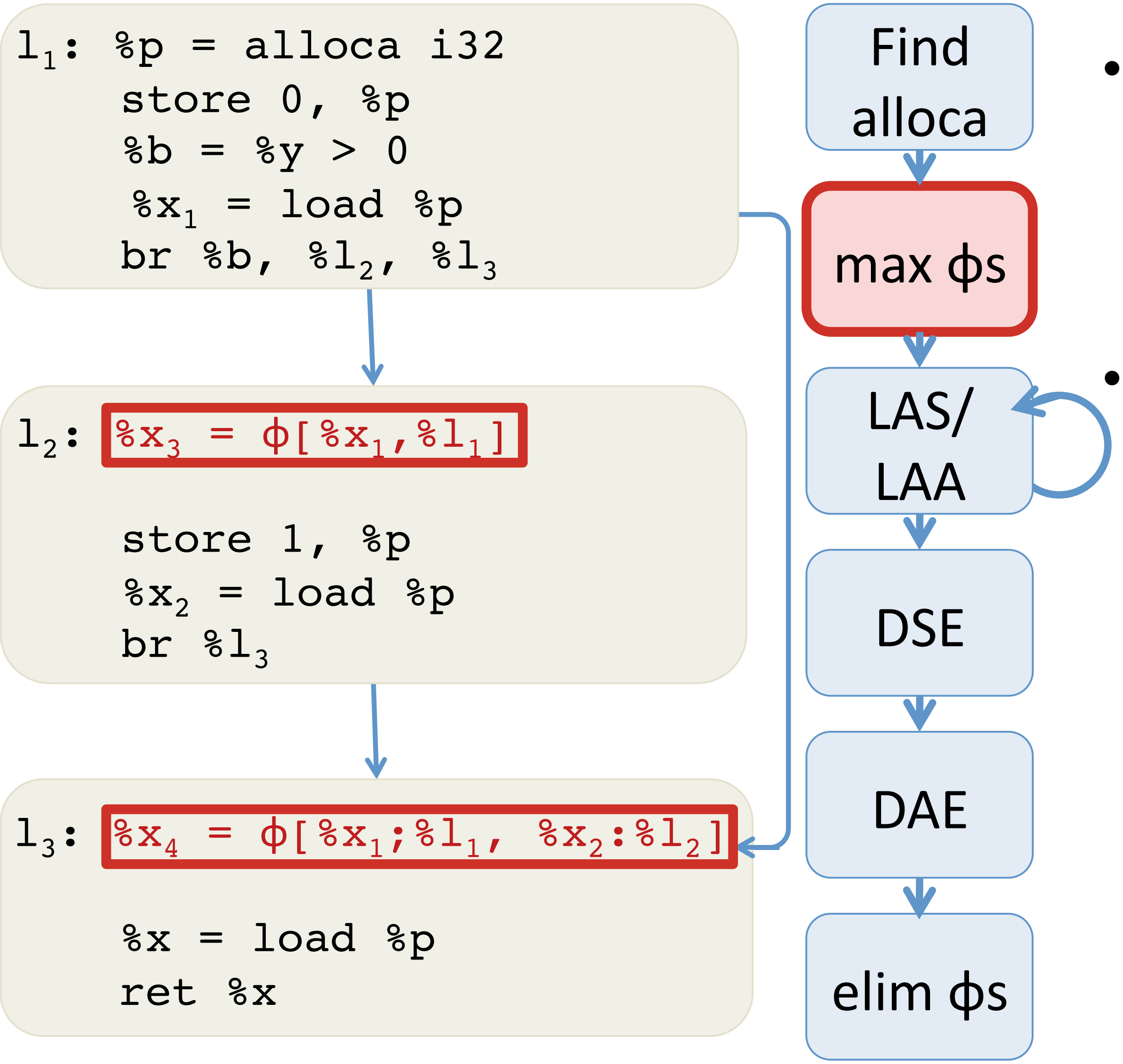
DSE

DAE

elim ϕs

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example of vmem2reg Algorithm



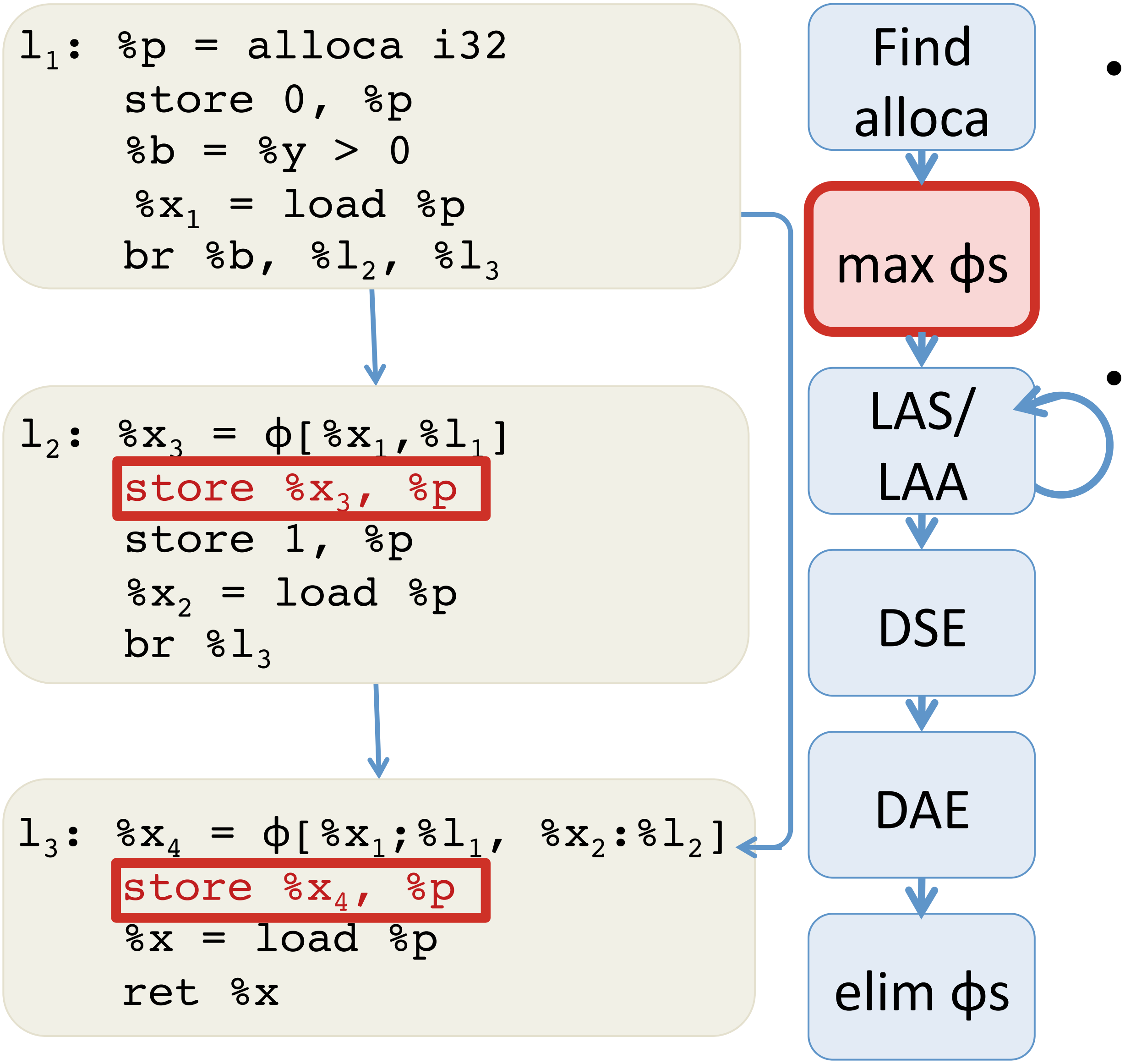
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
     %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]

    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = ϕ[%x1;%l1, %x2:%l2]

    %x = load %p
    ret %x
```

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block

Example of vmem2reg Algorithm



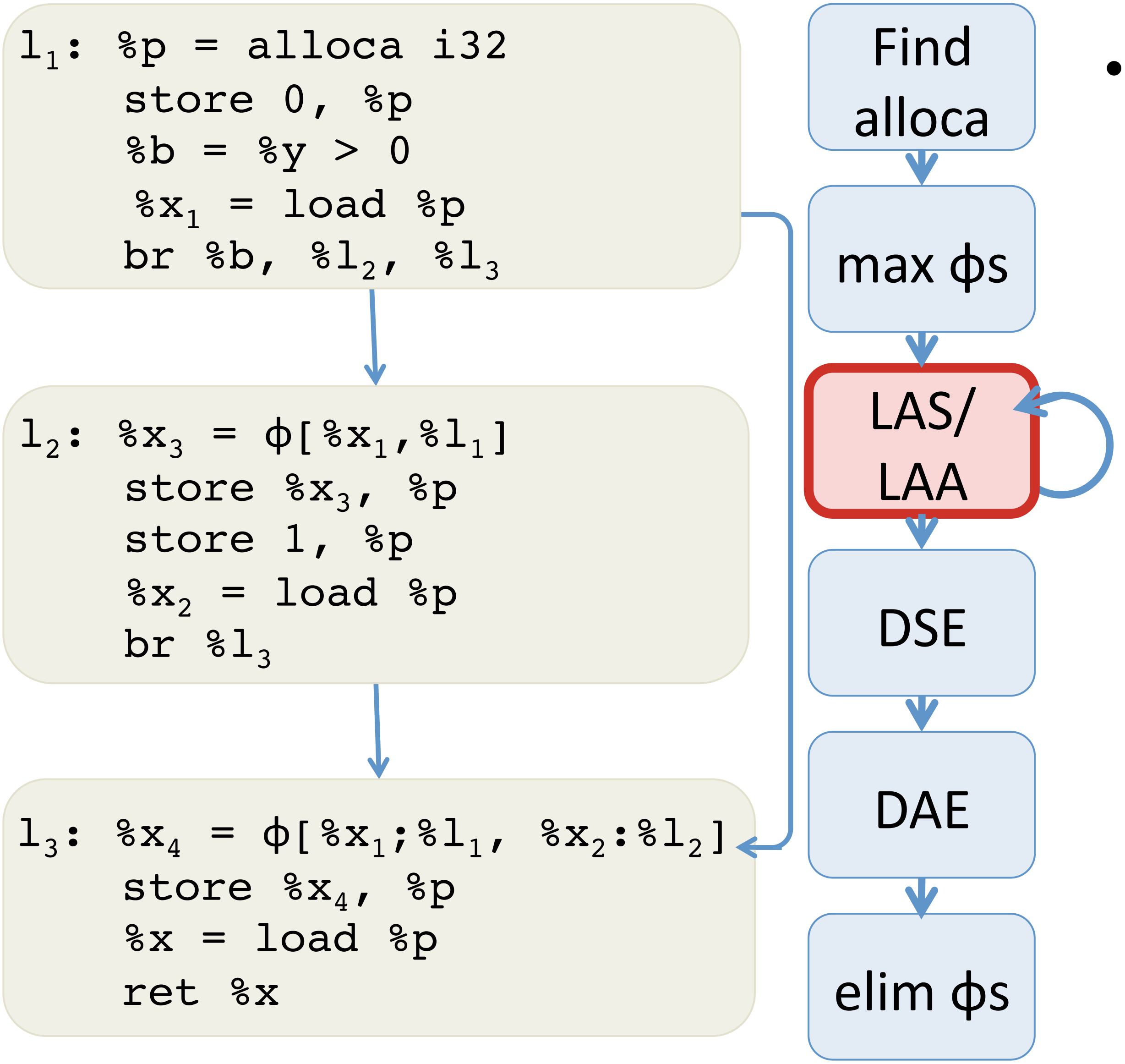
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
     %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = ϕ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block
  - Insert stores after ϕ-nodes

Example of vmem2reg Algorithm



```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = ϕ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

Find alloca

max ϕs

LAS/ LAA

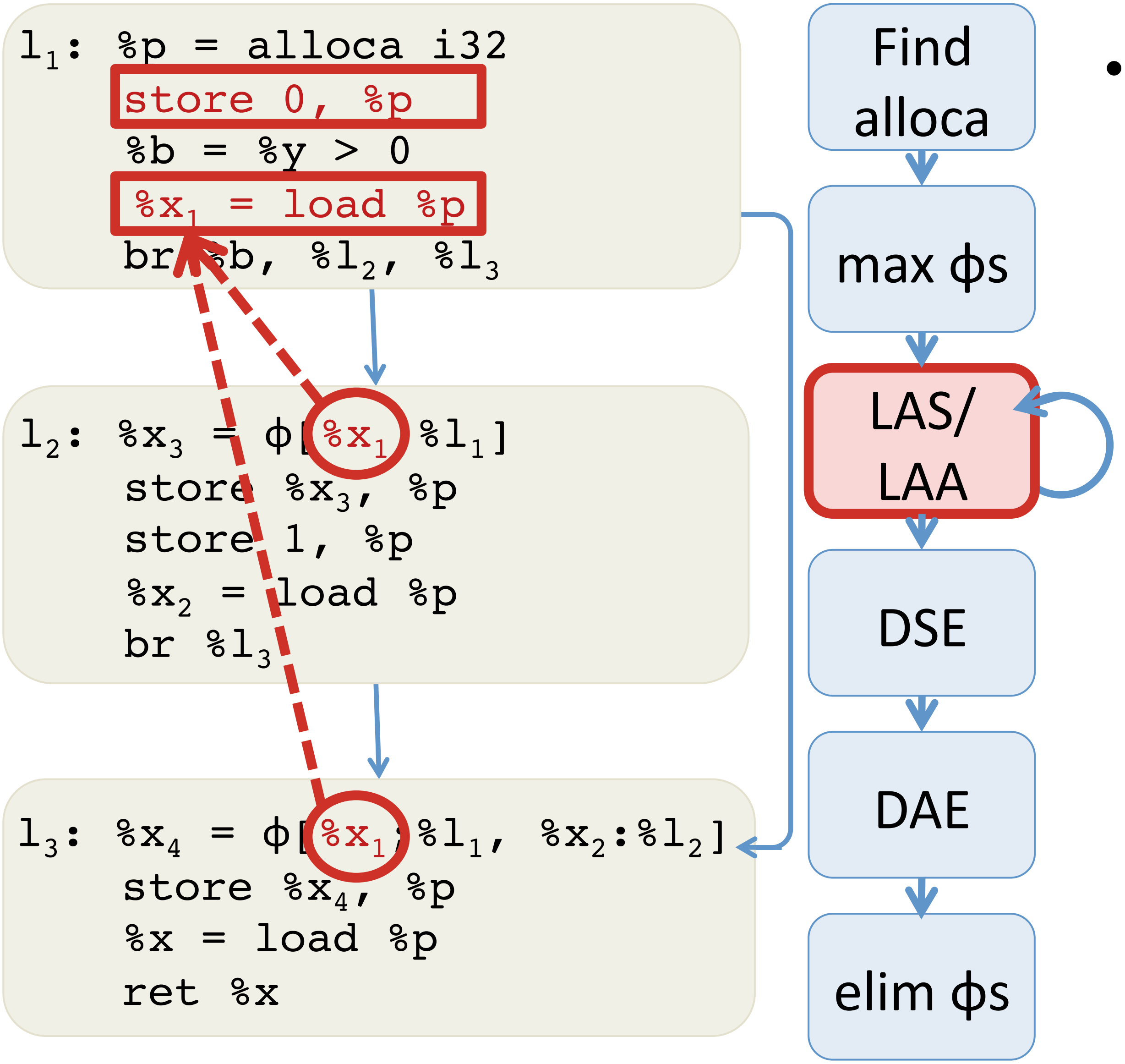
DSE

DAE

elim ϕs

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1:%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = ϕ[%x1:%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

Find alloca

max ϕs

LAS/ LAA

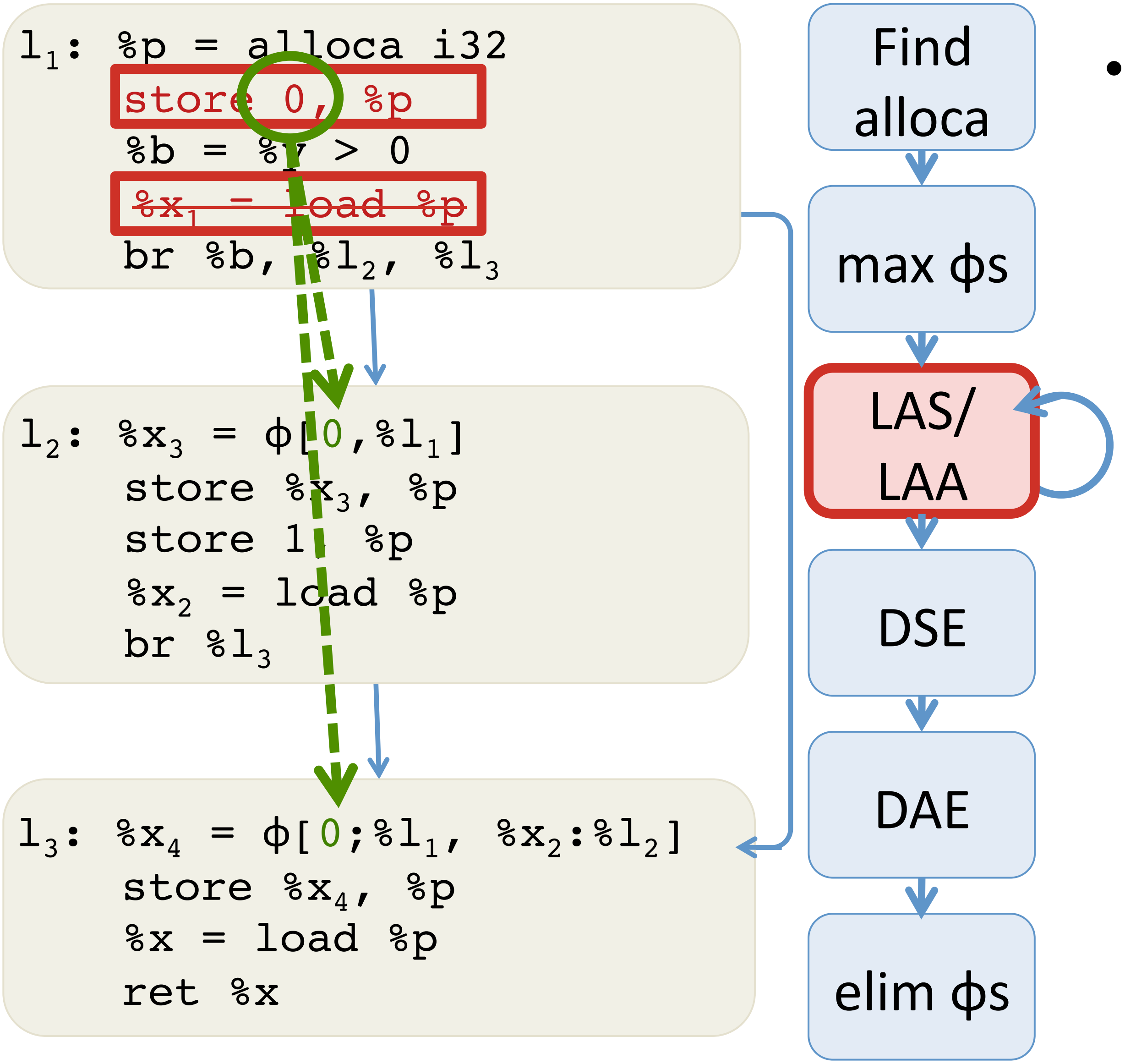
DSE

DAE

elim ϕs

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



l_1: %p = alloca i32
store 0, %p
%b = %y > 0
~~%x_1 = load %p~~
br %b, %l_2, %l_3

l_2: %x_3 = ϕ[0,%l_1]
store %x_3, %p
store 1, %p
%x_2 = load %p
br %l_3

l_3: %x_4 = ϕ[0;%l_1, %x_2:%l_2]
store %x_4, %p
%x = load %p
ret %x

Find alloca

max ϕs

LAS/ LAA

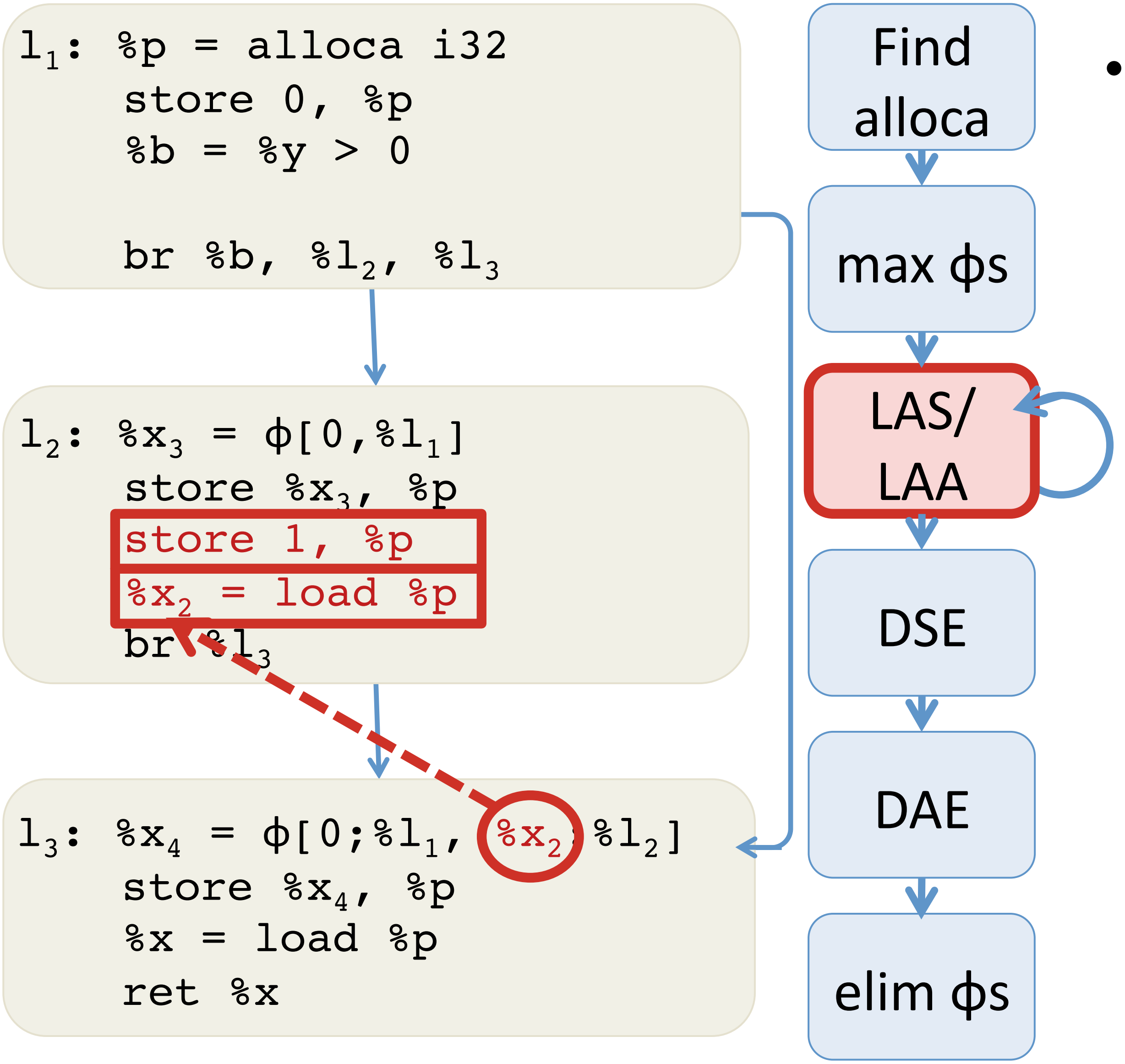
DSE

DAE

elim ϕs

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



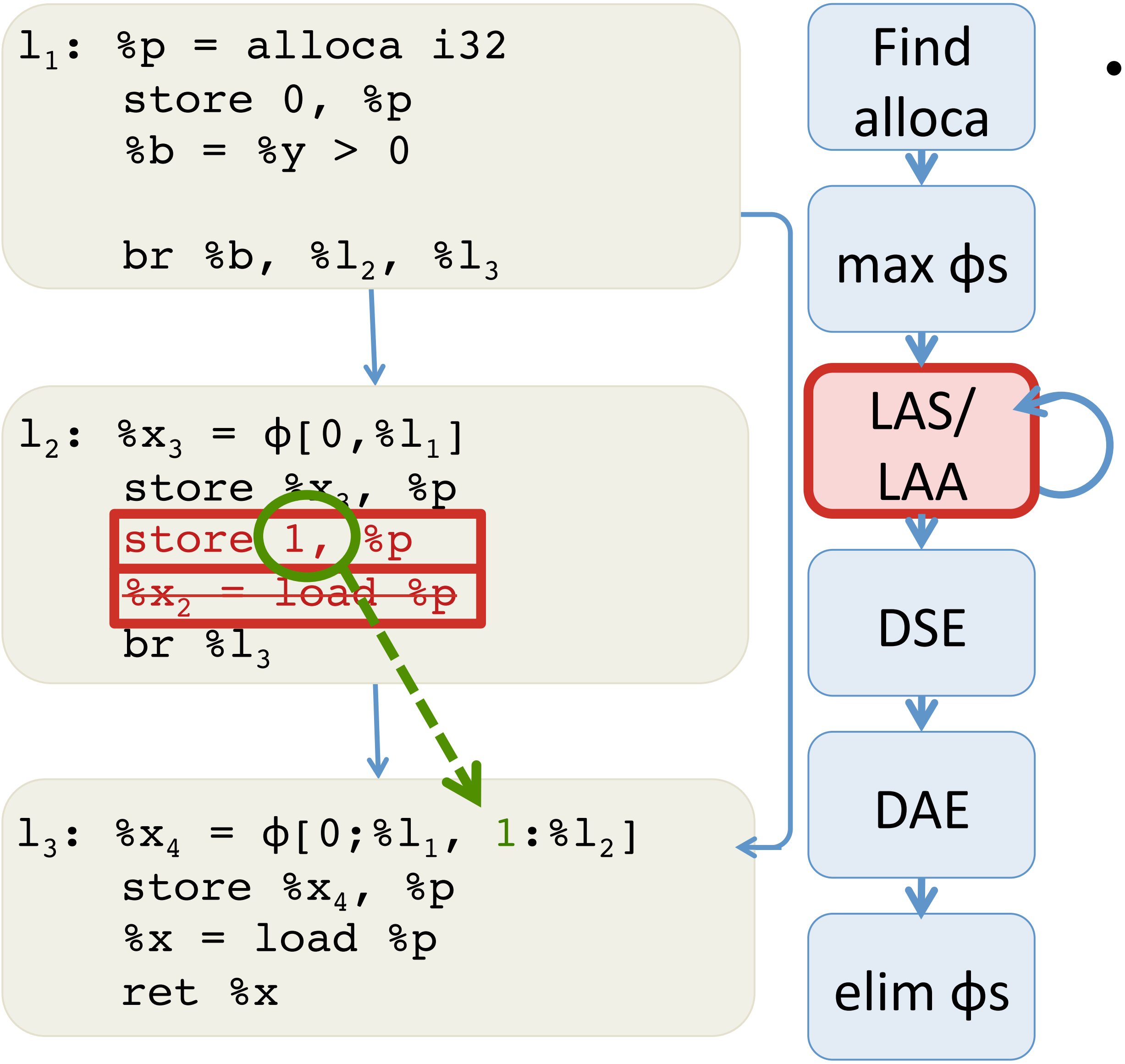
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = ϕ[0;%l1, %x2, %l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



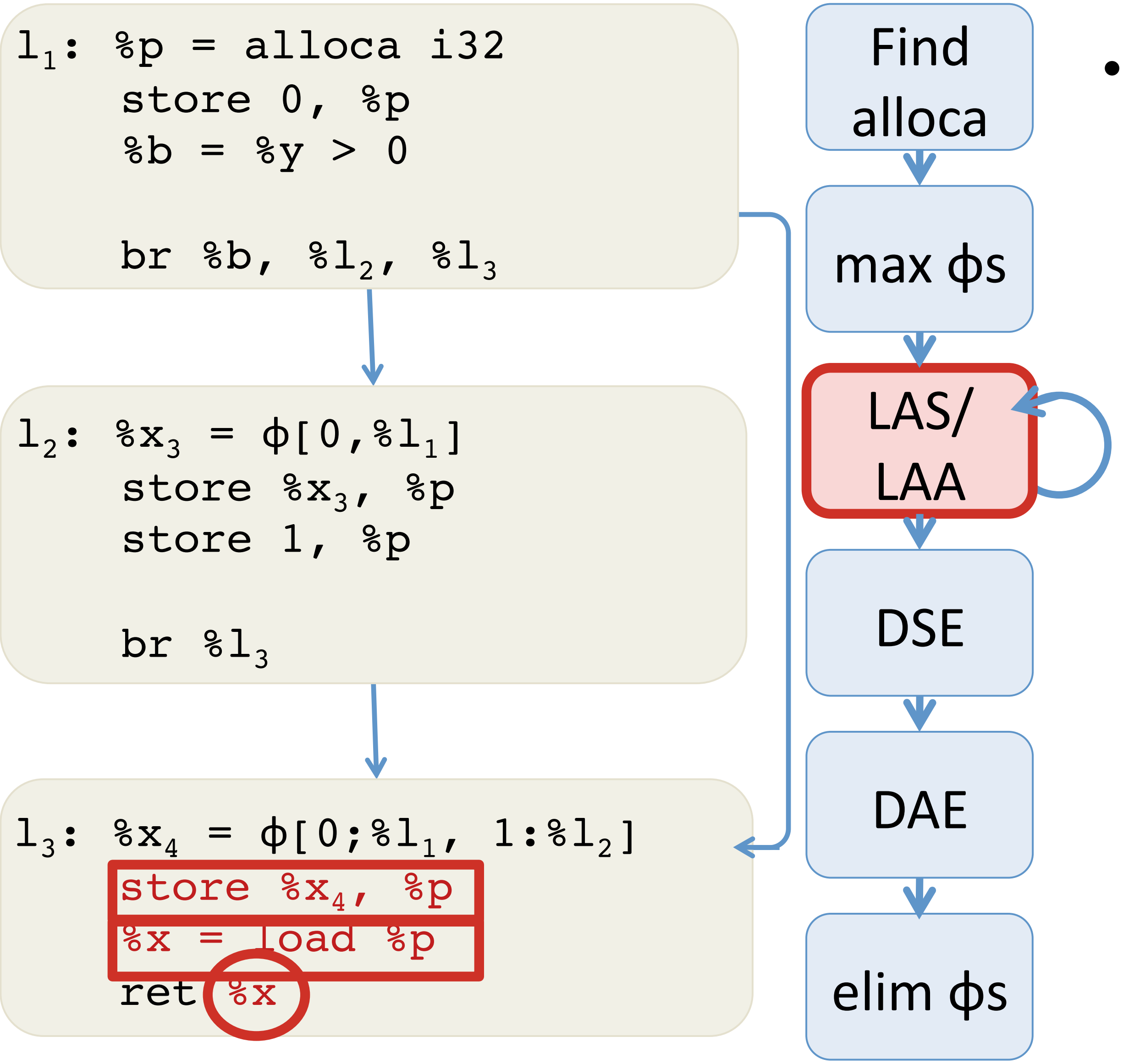
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    ~~%x2 = load %p~~
    br %l3
```

```
l3: %x4 = ϕ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



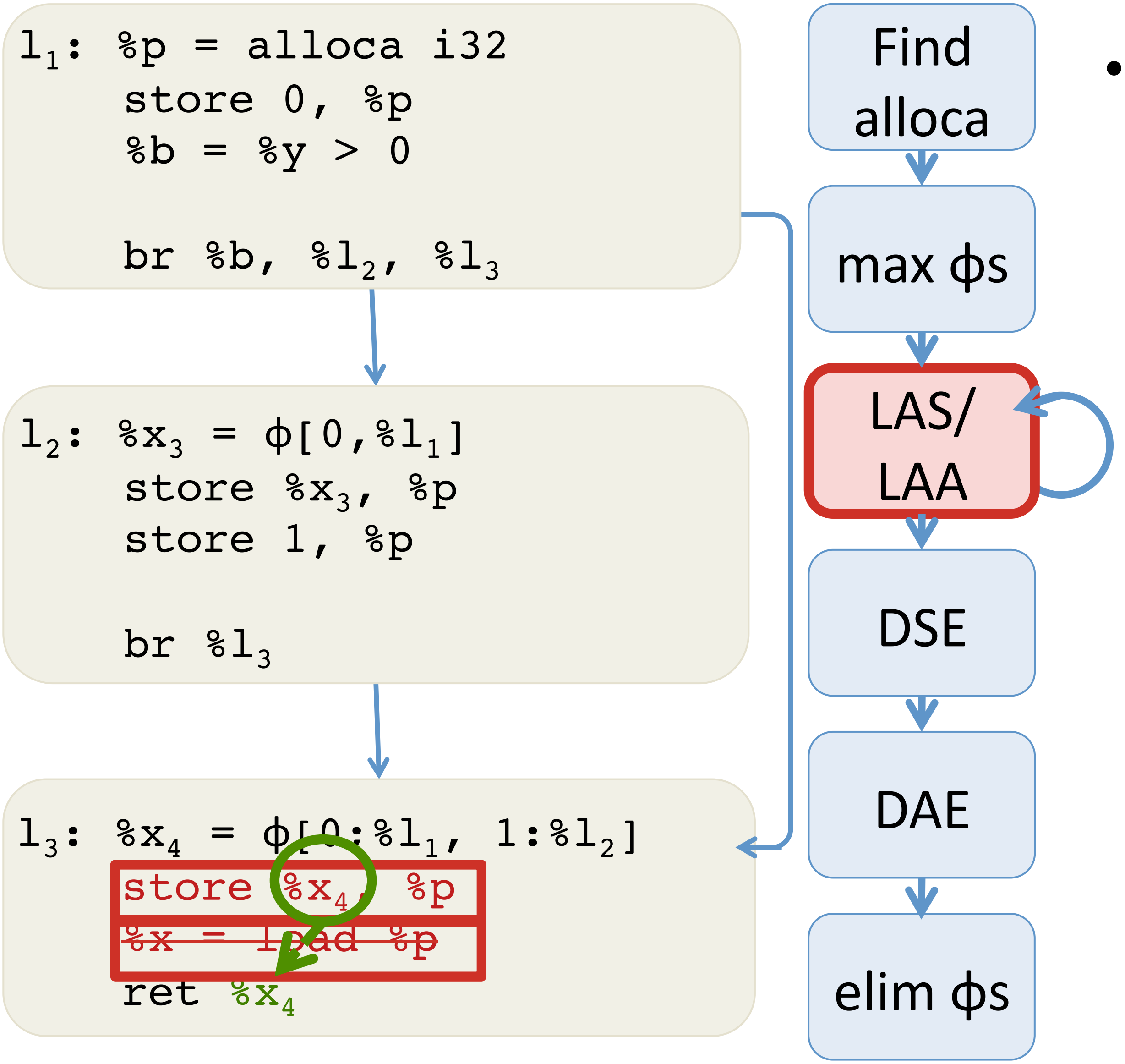
```
l_1: %p = alloca i32
     store 0, %p
     %b = %y > 0

     br %b, %l_2, %l_3
```

```
l_2: %x_3 = ϕ[0,%l_1]
     store %x_3, %p
     store 1, %p

     br %l_3
```

```
l_3: %x_4 = ϕ[0;%l_1, 1:%l_2]
     store %x_4, %p
     %x = load %p
     ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



l_1: %p = alloca i32
store 0, %p
%b = %y > 0

br %b, %l_2, %l_3

l_2: %x_3 = ϕ[0,%l_1]
store %x_3, %p
store 1, %p

br %l_3

l_3: %x_4 = ϕ[0:%l_1, 1:%l_2]
store %x_4, %p
~~%x = load %p~~
ret %x_4

Find alloca

max ϕs

LAS/ LAA

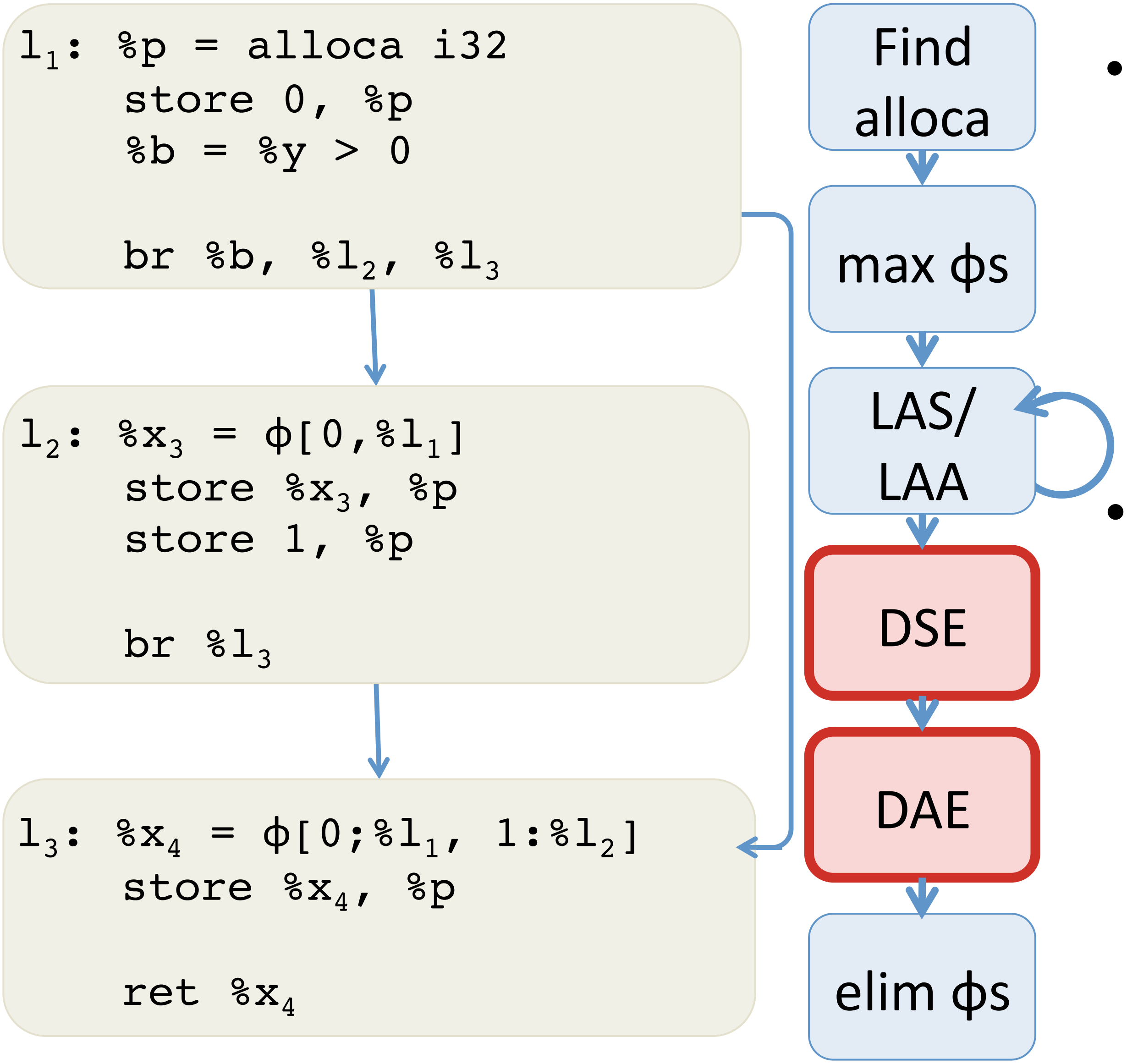
DSE

DAE

elim ϕs

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm



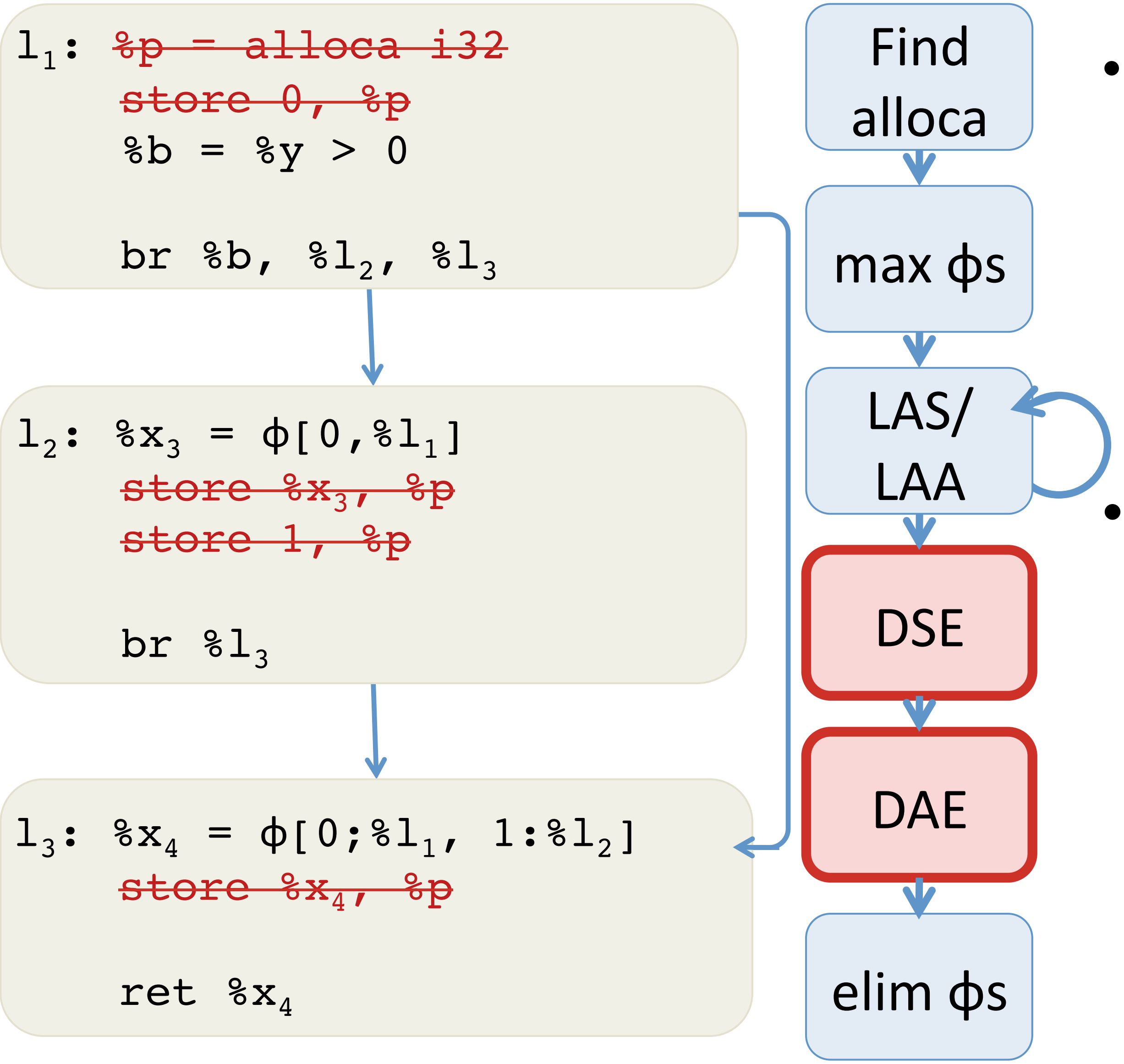
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = ϕ[0;%l1, 1:%l2]
    store %x4, %p

    ret %x4
```

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/ stores.

Example of vmem2reg Algorithm



l_1: ~~%p = alloca i32~~
~~store 0, %p~~
%b = %y > 0

br %b, %l_2, %l_3

l_2: %x_3 = ϕ[0,%l_1]
~~store %x_3, %p~~
~~store 1, %p~~

br %l_3

l_3: %x_4 = ϕ[0;%l_1, 1:%l_2]
~~store %x_4, %p~~

ret %x_4

Find alloca

max ϕs

LAS/ LAA

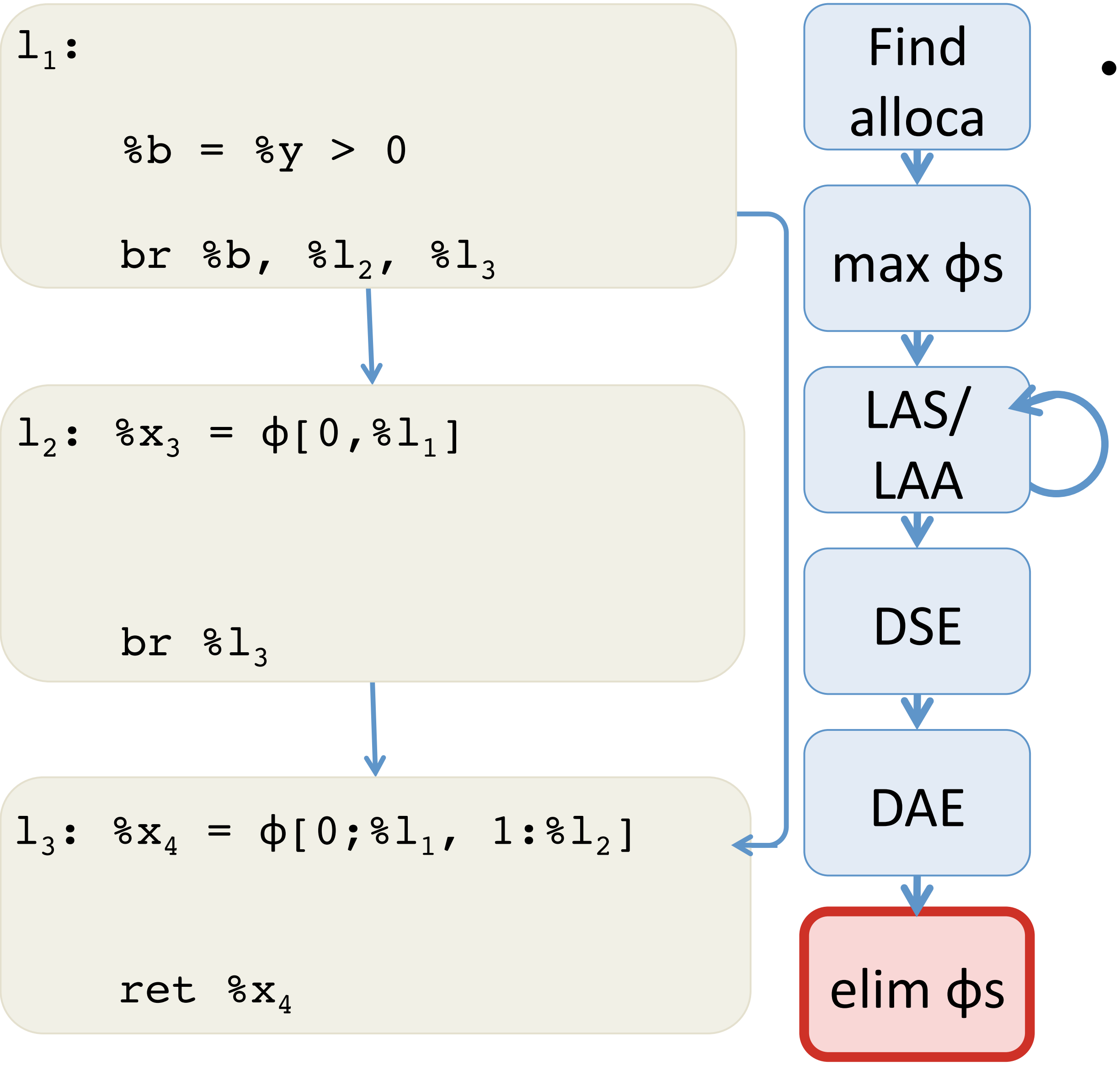
DSE

DAE

elim ϕs

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/ stores.

Example of vmem2reg Algorithm



l_1:

%b = %y > 0

br %b, %l_2, %l_3

l_2: %x_3 = ϕ[0,%l_1]

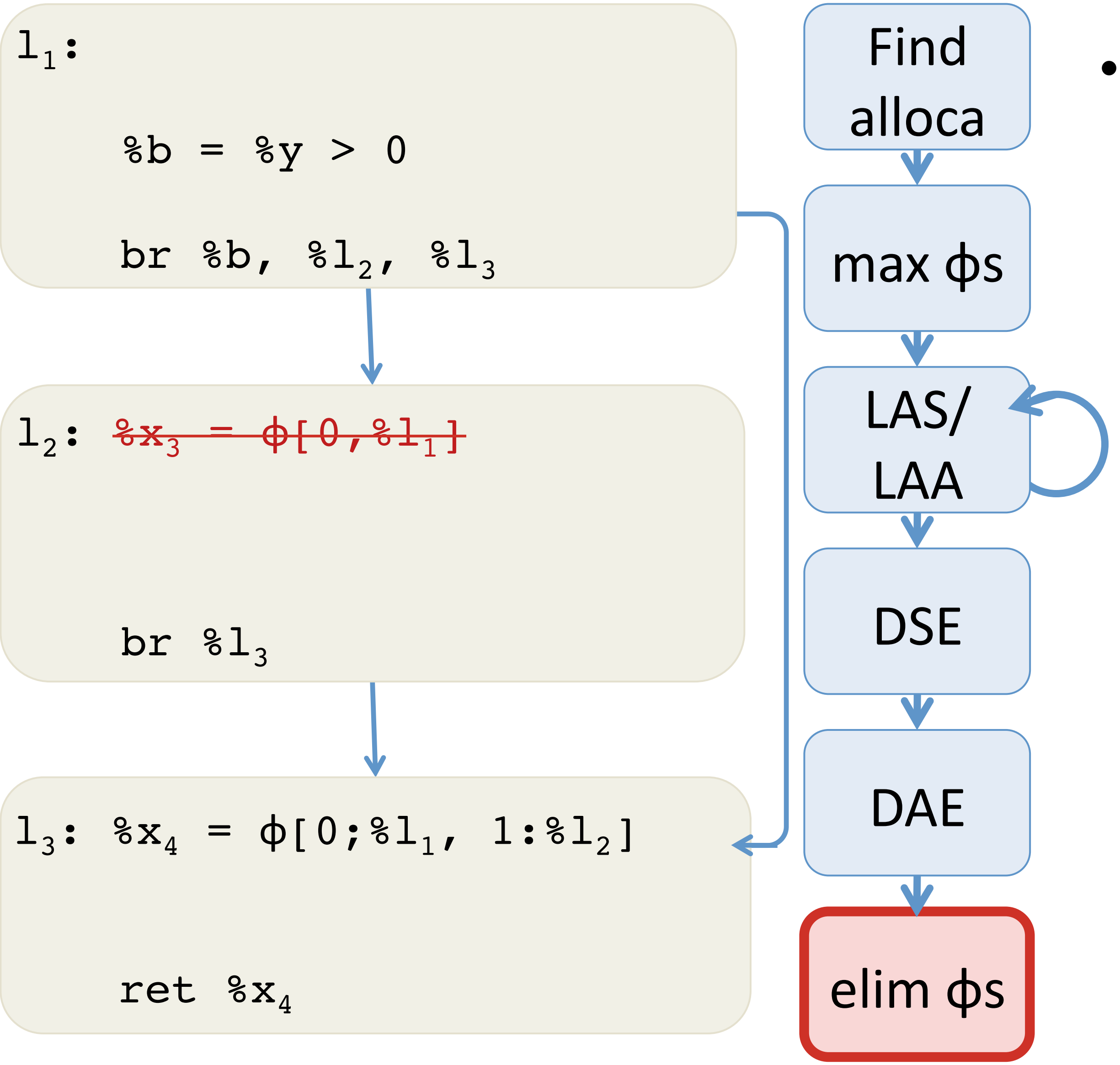
br %l_3

l_3: %x_4 = ϕ[0;%l_1, 1:%l_2]

ret %x_4

- Eliminate ϕ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Example of vmem2reg Algorithm



l_1:

%b = %y > 0

br %b, $\%l_2$, $\%l_3$

l_2: ~~$\%x_3 = \phi[0,\%l_1]$~~

br $\%l_3$

l_3: $\%x_4 = \phi[0;\%l_1,\ 1{:}\%l_2]$

ret $\%x_4$

Find alloca

max ϕs

LAS/ LAA

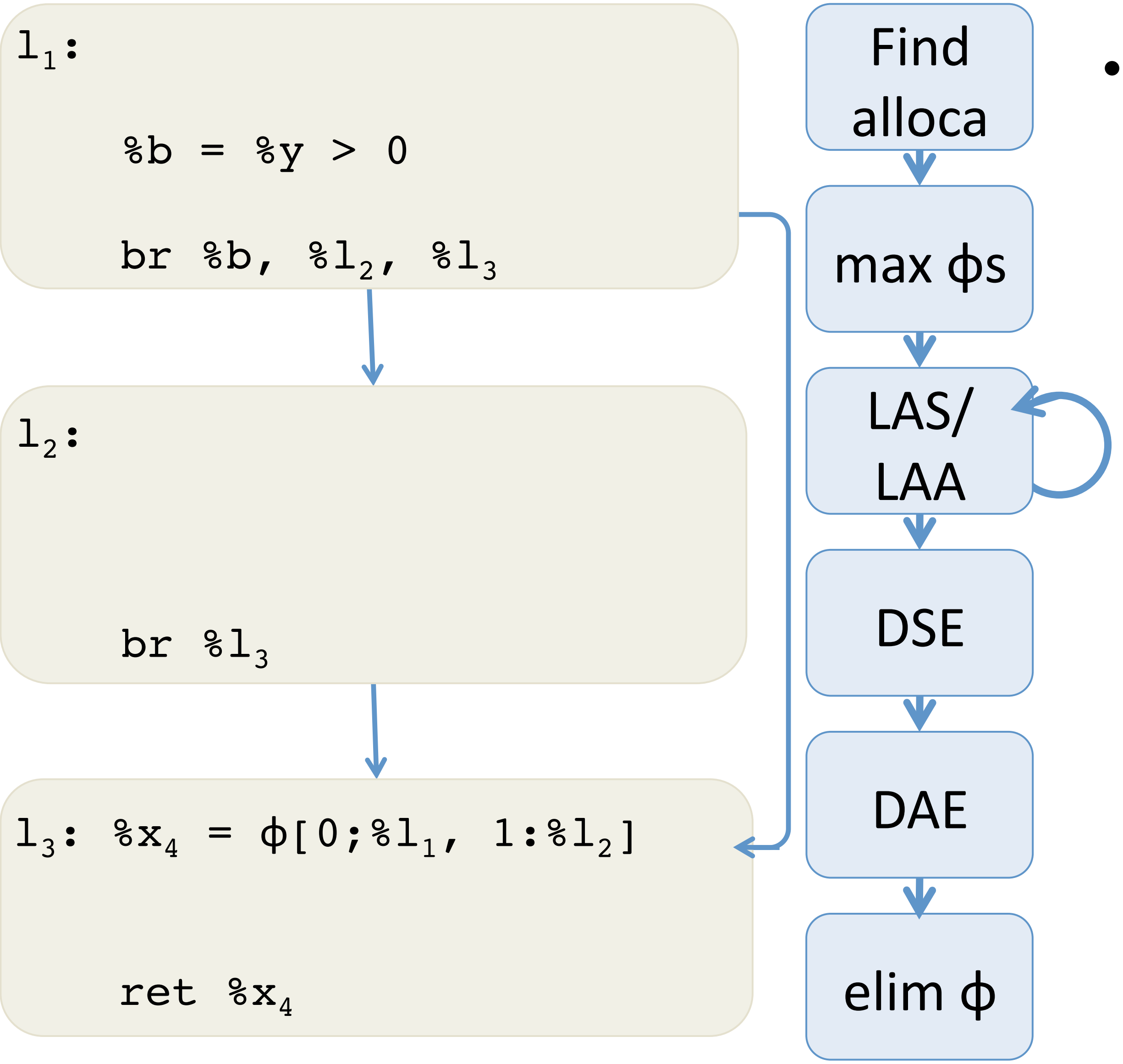
DSE

DAE

elim ϕs

- Eliminate ϕ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Example of vmem2reg Algorithm



```
l_1:
    %b = %y > 0
    br %b, %l_2, %l_3

l_2:
    br %l_3

l_3: %x_4 = ϕ[0;%l_1, 1:%l_2]
    ret %x_4
```

Find alloca

max ϕs

LAS/LAA

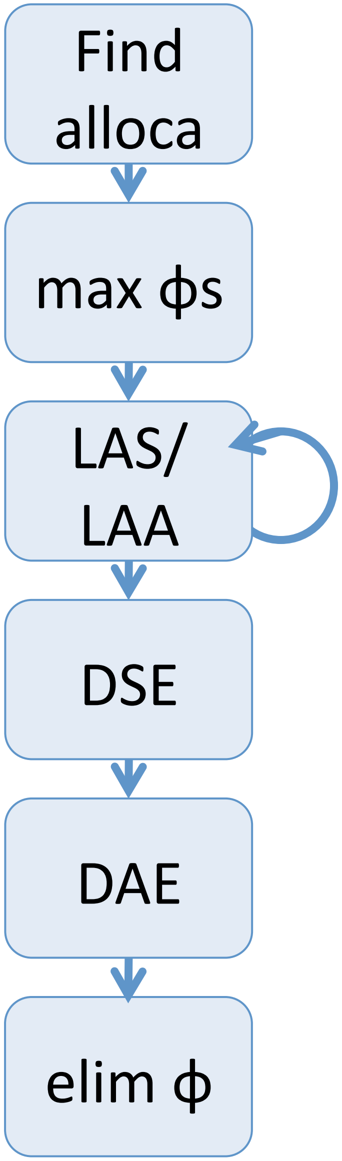
DSE

DAE

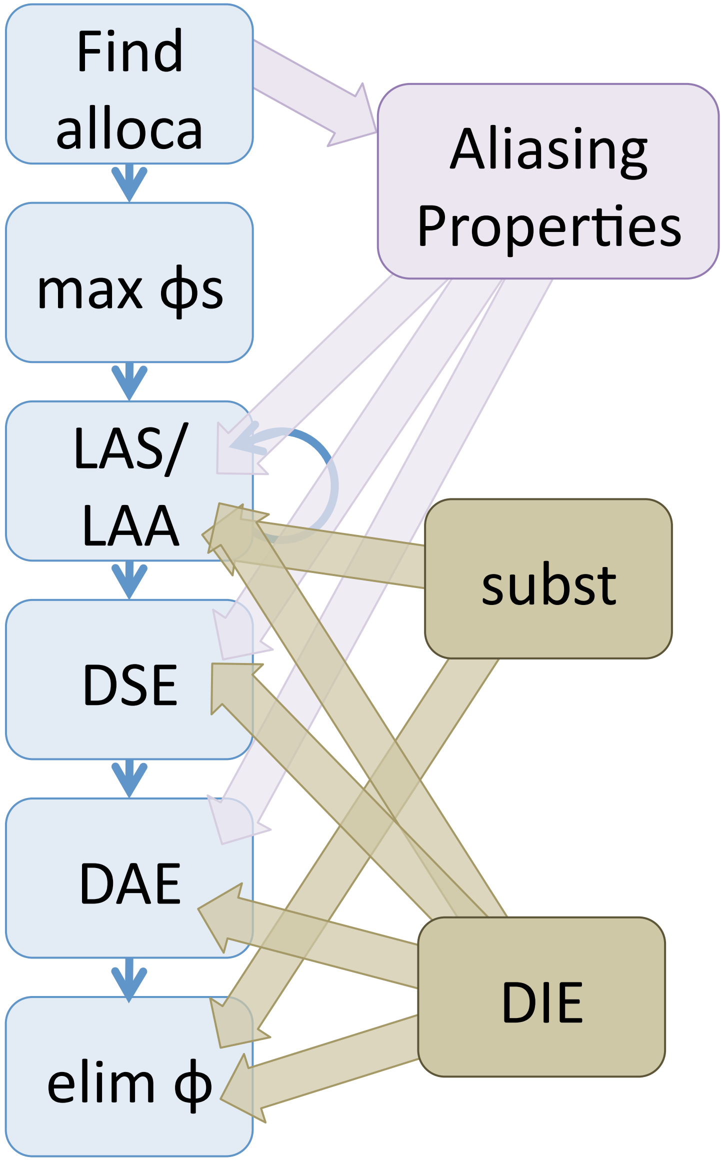
elim ϕ

- Done!

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

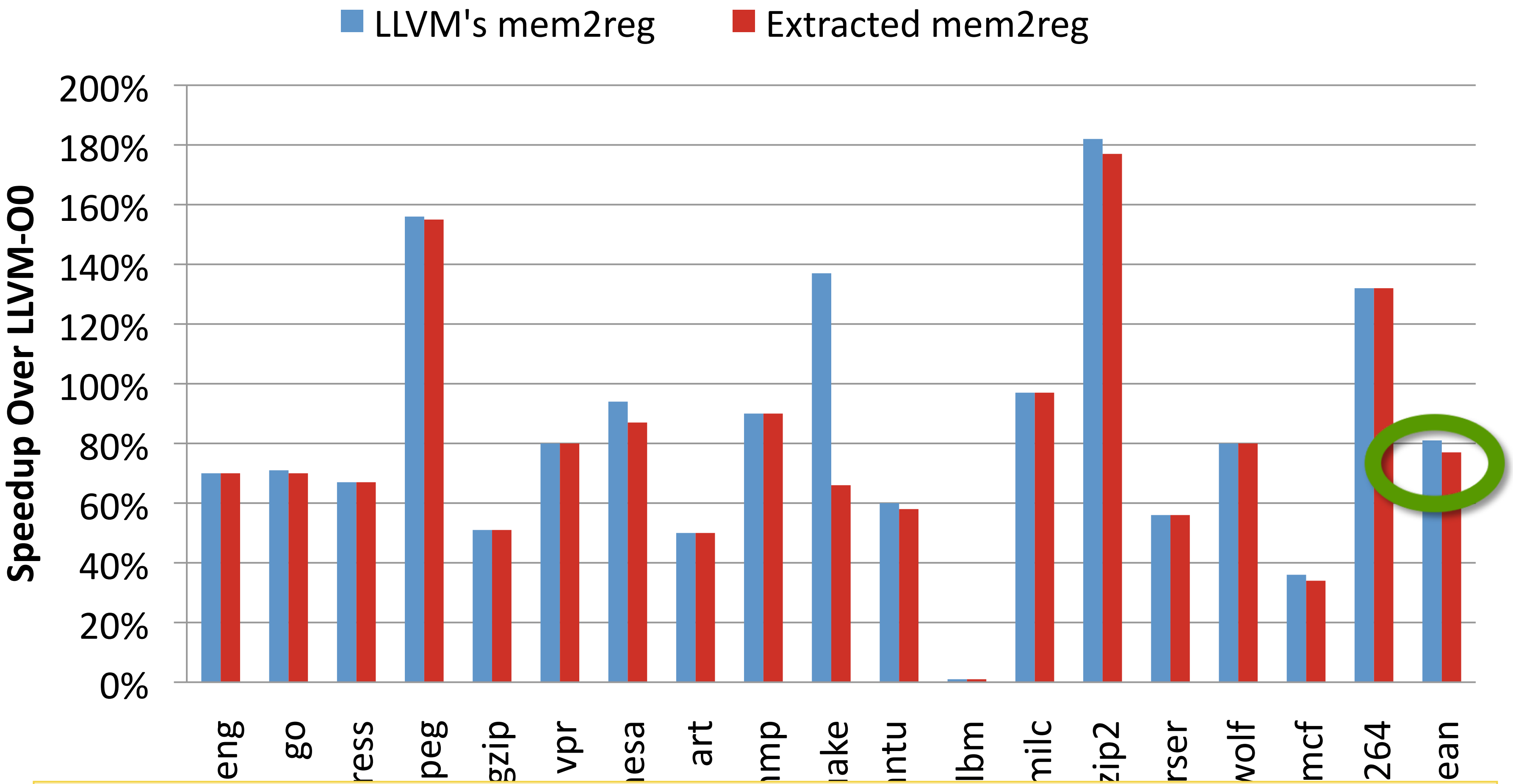
Theorem: The vmem2reg algorithm preserves the semantics of the source program.

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

- Vminus: a highly simplified SSA IR based on LLVM
  - What is SSA?
- Verified Compilation of Imp to Vminus
  - What does it mean to “verify compilation”?
- Scaling up: Vellvm
  - Taste of the full LLVM IR
  - Operational Semantics
  - Metatheory + Proof Techniques
- Case studies:
  - SoftBound memory safety
- Conclusion:
  - challenges & research directions

Other Parts of the LLVM IR

```
op       ::= %uid | constant | undef                Operands
bop      ::= add | sub | mul | shl | …              Operations
cmpop    ::= eq | ne | slt | sle | …                Comparison

insn ::=
  |  %uid = alloca ty                               Stack Allocation
  |  %uid = load ty op1                             Load
  |  store ty op1, op2                              Store
  |  %uid = getelementptr ty op1 …                  Address Calculation
  |  %uid = call rt fun(…args…)                     Function Calls
  |  …

phi ::=
  |  ϕ[op1;lbl1]...[opn;lbln]

terminator ::=
  |  ret %ty op
  |  br op label %lbl1, label %lbl2
  |  br label %lbl
```

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 (ty_1, ty_2, … , ty_n)        function types
  | {ty_1, ty_2, … , ty_n}          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 {ty_1, ty_2, … , ty_n}
```

Example LLVM Types

- An array of 341 integers: `[ 341 x i32 ]`
- A 2D array of integers: `[3 x [ 4 x i32 ]]`
- C-style linked lists:

```
%Node = type { i32, %Node*}
```

- Structs:

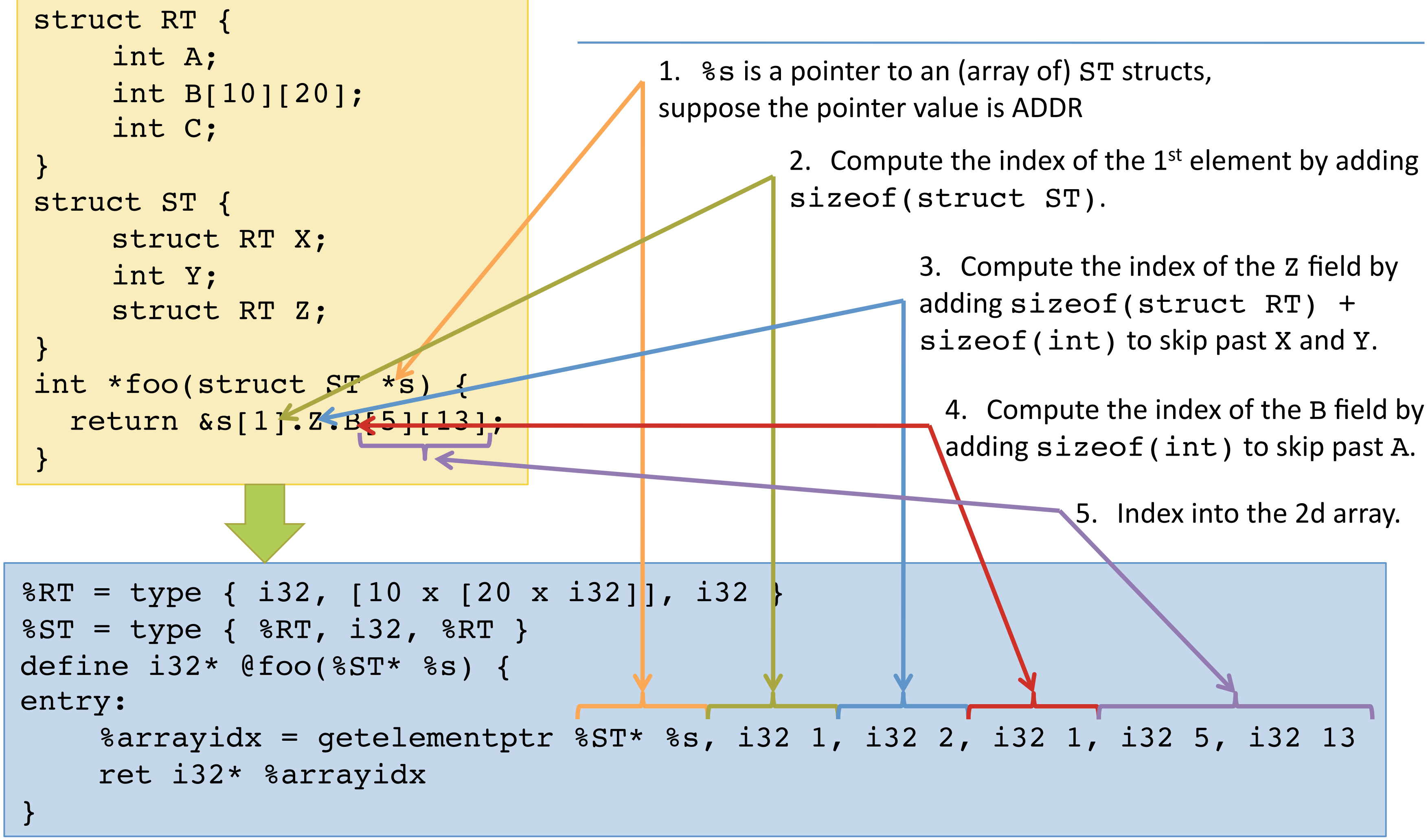
```
%Rect  = {  %Point, %Point,
            %Point, %Point }
%Point = { i32, i32 }
```

GetElementPtr

- LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a “path” through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
  - It is a “type indexed” operation, since the size computations involved depend on the type

```
insn ::= …
      |   %uid = getelementptr t*, %val, t1 idx1, t2 idx2 ,…
```

Example



```
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `sizeof(struct ST)`.
3. Compute the index of the `Z` field by adding `sizeof(struct RT) + sizeof(int)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `sizeof(int)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

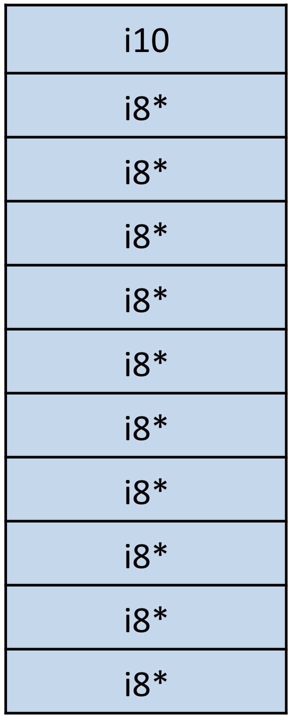
Final answer: ADDR + `sizeof(struct ST) + sizeof(struct RT) + sizeof(int)`
`+ sizeof(int) + 5*20*sizeof(int) + 13*sizeof(int)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

LLVM’s memory model

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

High-level
Representation

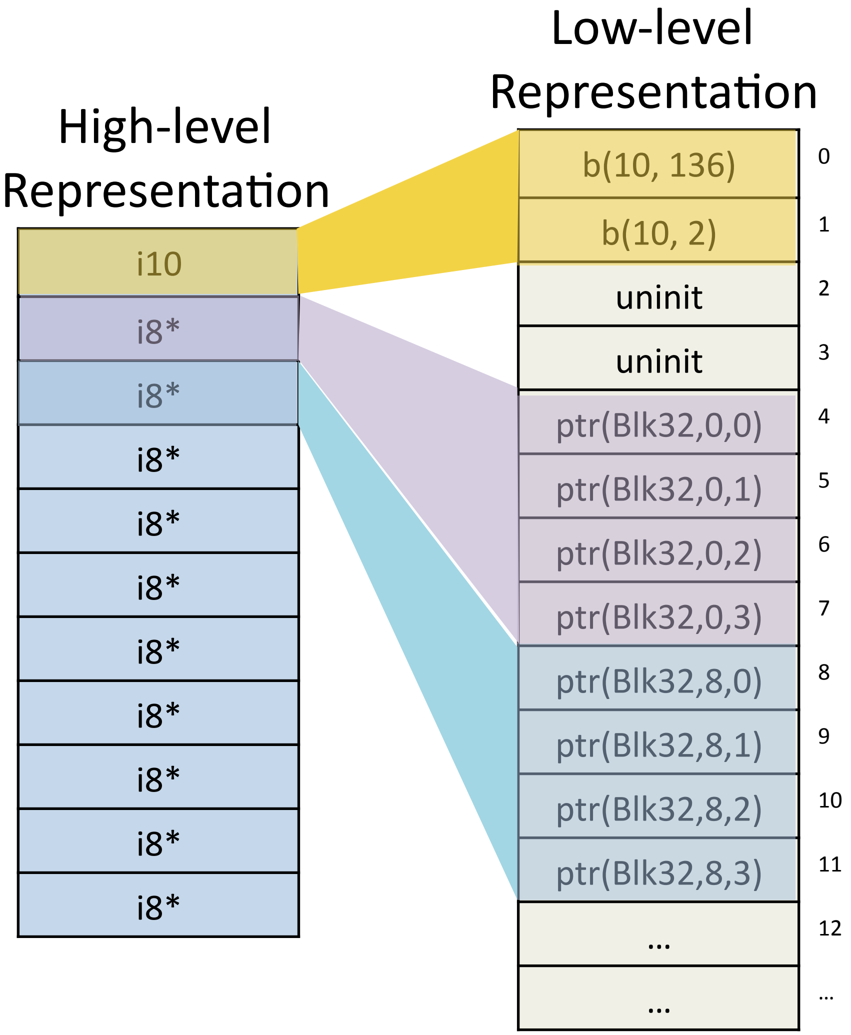


- Manipulate structured types.

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

LLVM's memory model

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

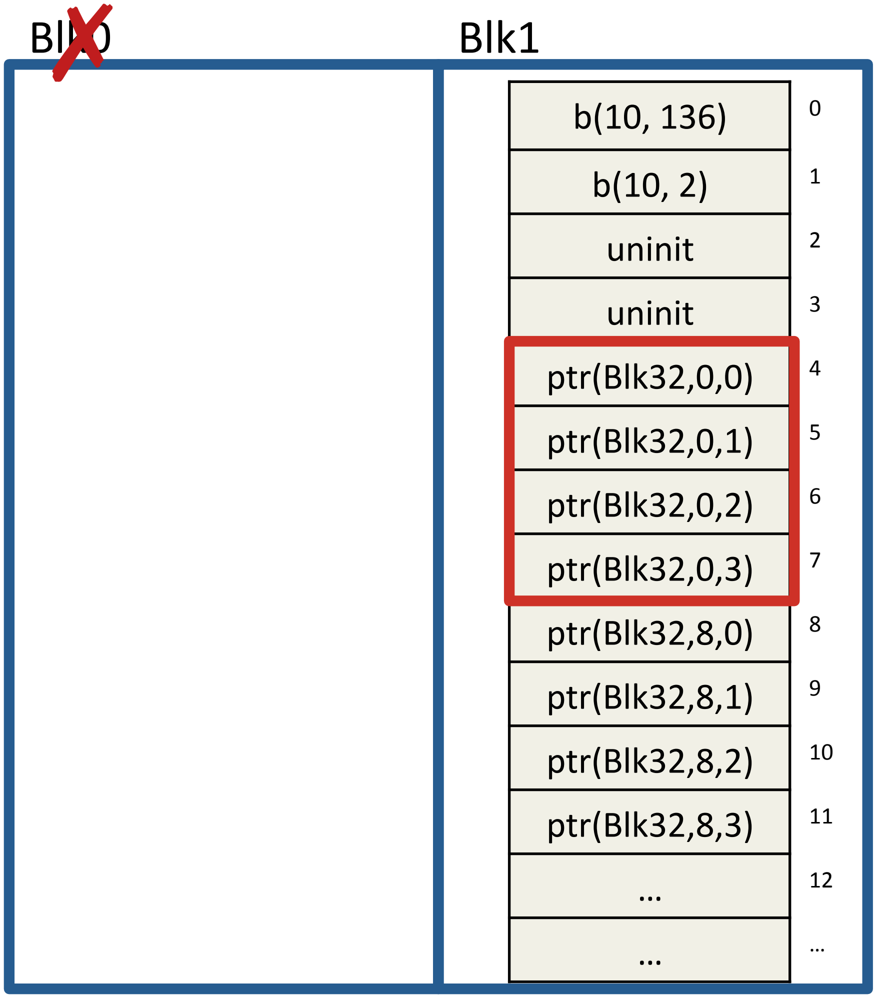


- Manipulate structured types.

```
%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

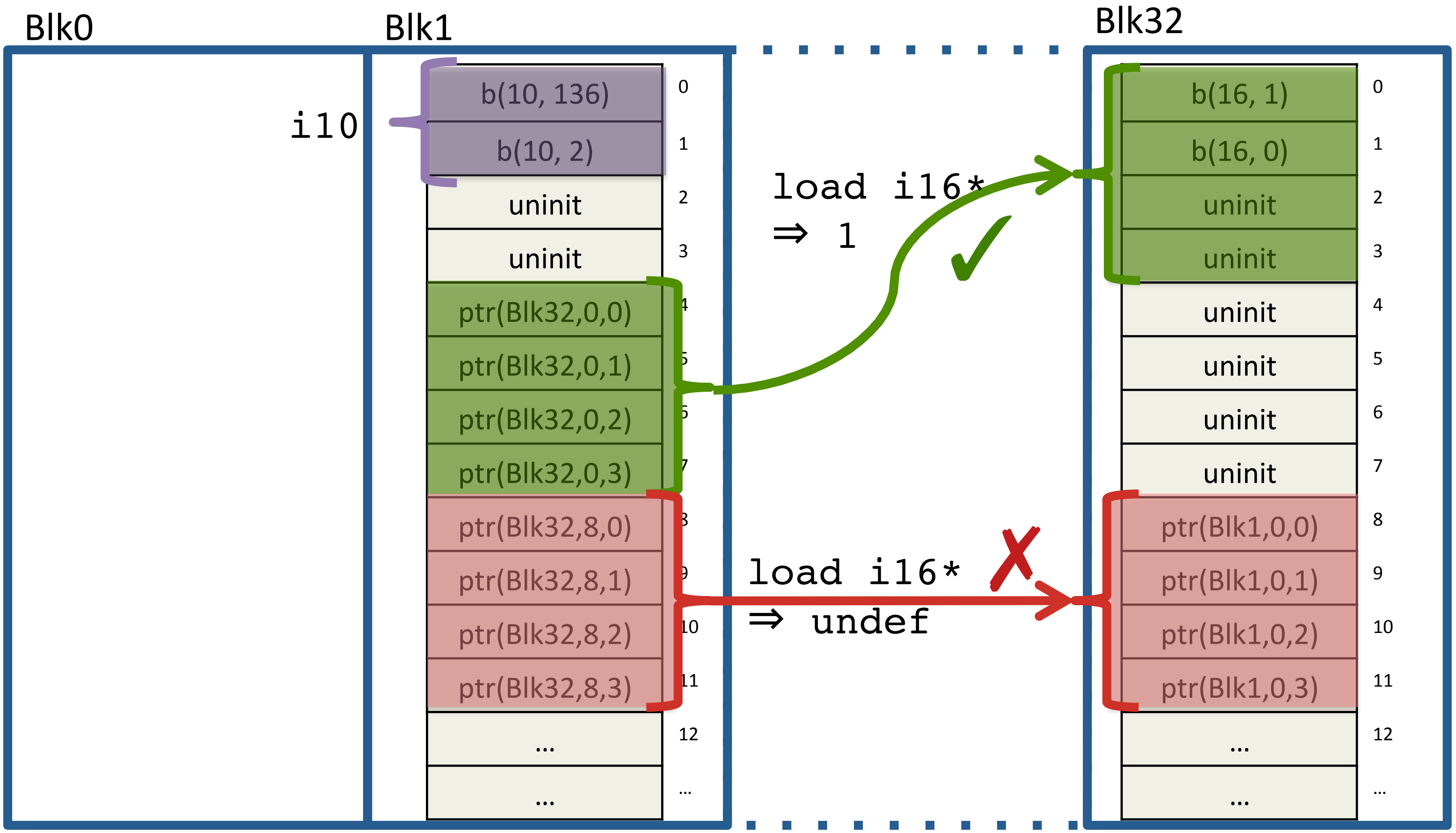


b(10, 136)	0
b(10, 2)	1
uninit	2
uninit	3
ptr(Blk32,0,0)	4
ptr(Blk32,0,1)	5
ptr(Blk32,0,2)	6
ptr(Blk32,0,3)	7
ptr(Blk32,8,0)	8
ptr(Blk32,8,1)	9
ptr(Blk32,8,2)	10
ptr(Blk32,8,3)	11
…	12
…	…

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

Dynamic Physical Subtyping

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



Blk0

Blk1

Blk32

i10

Blk1		Blk32	
b(10, 136)	0	b(16, 1)	0
b(10, 2)	1	b(16, 0)	1
uninit	2	uninit	2
uninit	3	uninit	3
ptr(Blk32,0,0)	4	uninit	4
ptr(Blk32,0,1)	5	uninit	5
ptr(Blk32,0,2)	6	uninit	6
ptr(Blk32,0,3)	7	uninit	7
ptr(Blk32,8,0)	8	ptr(Blk1,0,0)	8
ptr(Blk32,8,1)	9	ptr(Blk1,0,1)	9
ptr(Blk32,8,2)	10	ptr(Blk1,0,2)	10
ptr(Blk32,8,3)	11	ptr(Blk1,0,3)	11
...	12	...	12
...	...	...	...

`load i16*`
⇒ `1` ✓

`load i16*`
⇒ `undef` ✗

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.

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

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:

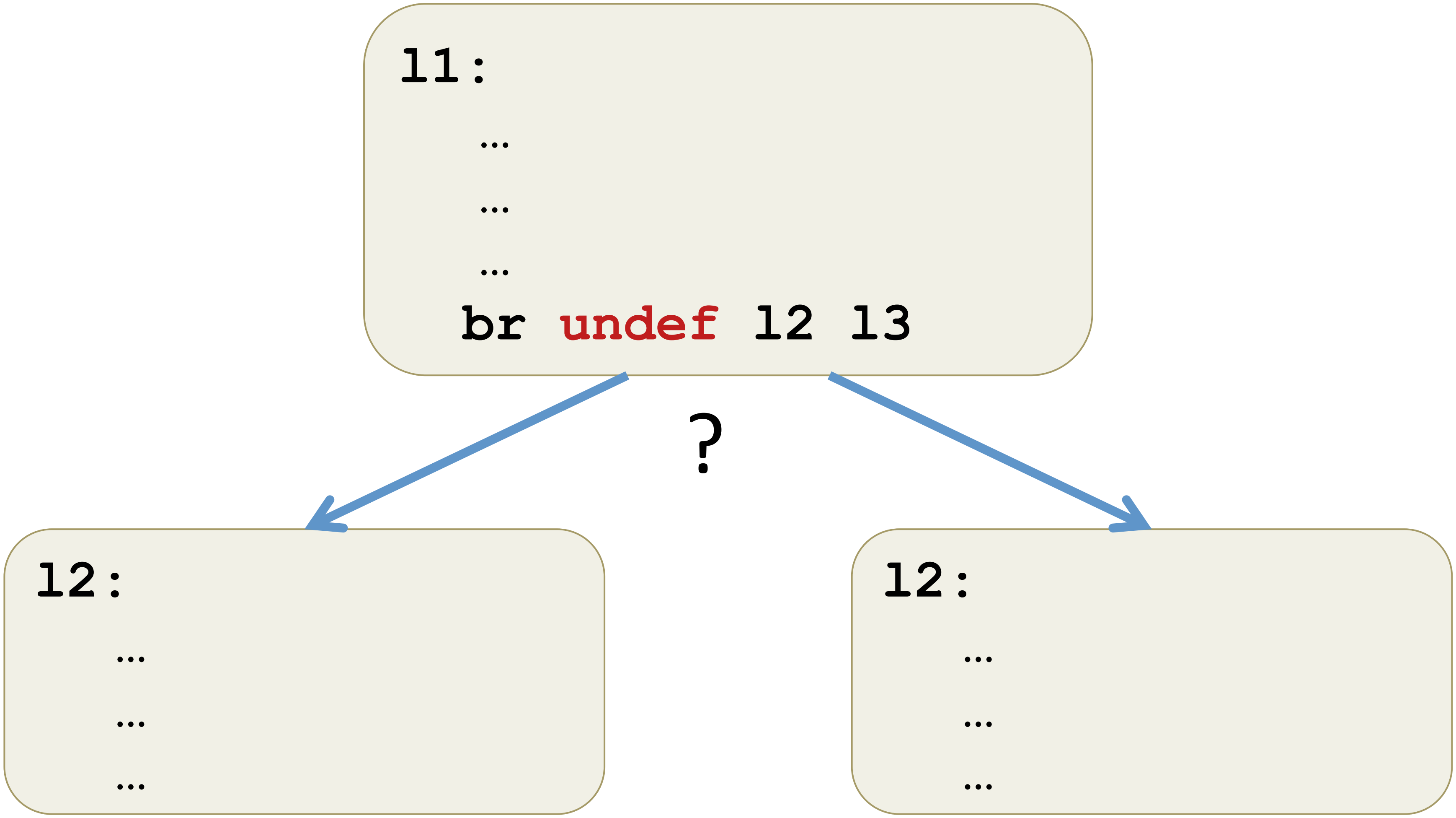
```
%x = or i8 undef, 1
%y = xor i8 %x %x
```

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

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

$$\begin{aligned}
[\![\texttt{\%x}]\!] &= \{a \texttt{ or } b \mid a \in [\![\texttt{i8 undef}]\!],\ b \in [\![1]\!]\} \\
&= \{1,3,5,\dots,255\} \\
[\![\texttt{\%y}]\!] &= \{a \texttt{ xor } b \mid a \in [\![\texttt{\%x}]\!],\ b \in [\![\texttt{\%x}]\!]\} \\
&= \boxed{\{0,2,4,\dots,254\}}
\end{aligned}$$

Nondeterministic Branches



```
l1:
   …
   …
   …
  br undef l2 l3
```

?

```
l2:
   …
   …
   …
```

```
l2:
   …
   …
   …
```

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

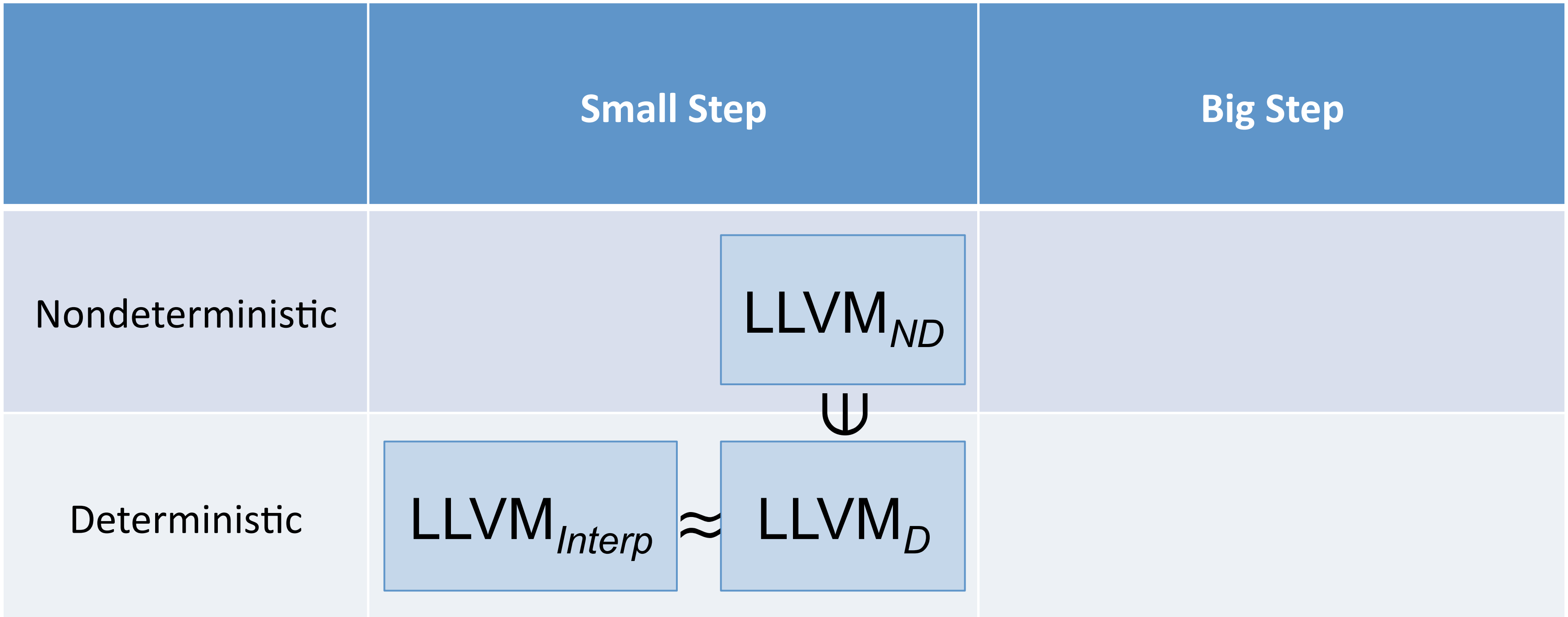
	Small Step	**Big Step**
Nondeterministic	LLVM_{ND}	
Deterministic		

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} ⩛	
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} ⟒	
Deterministic	$\mathrm{LLVM}_{Interp} \approx \mathrm{LLVM}_{D}$	$\gtrsim \mathrm{LLVM}^{*}_{DFn} \gtrsim \mathrm{LLVM}^{*}_{DB}$

Simulation up to "observable events":

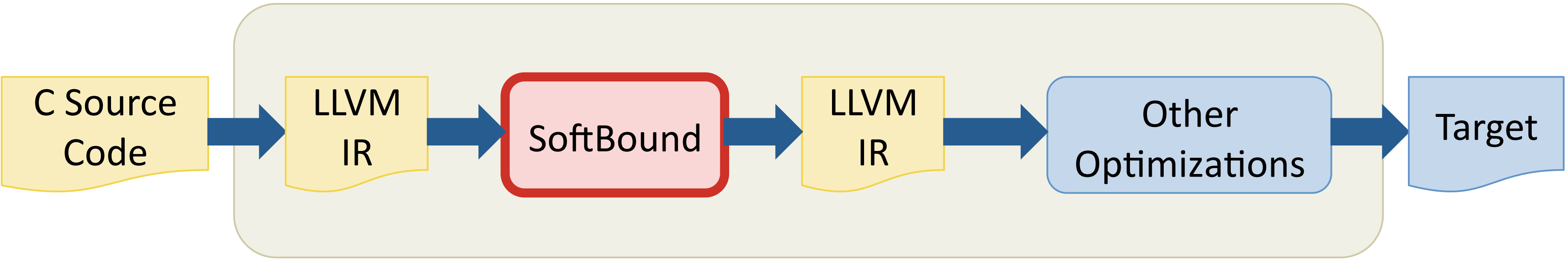
- useful for encapsulating behavior of function calls
- large step evaluation of basic blocks

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

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


C Source Code
LLVM IR
SoftBound
LLVM IR
Other Optimizations
Target

SoftBound

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

Maintain base and bound for all pointers

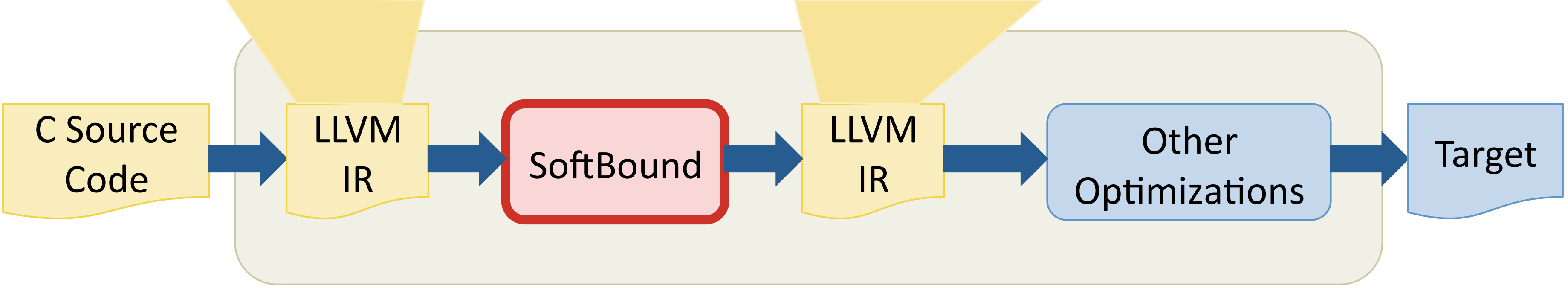
```
%q = gep %p, i32 0, i32 255
```

Propagate metadata on assignment

Check that a pointer is within its bounds when being accessed

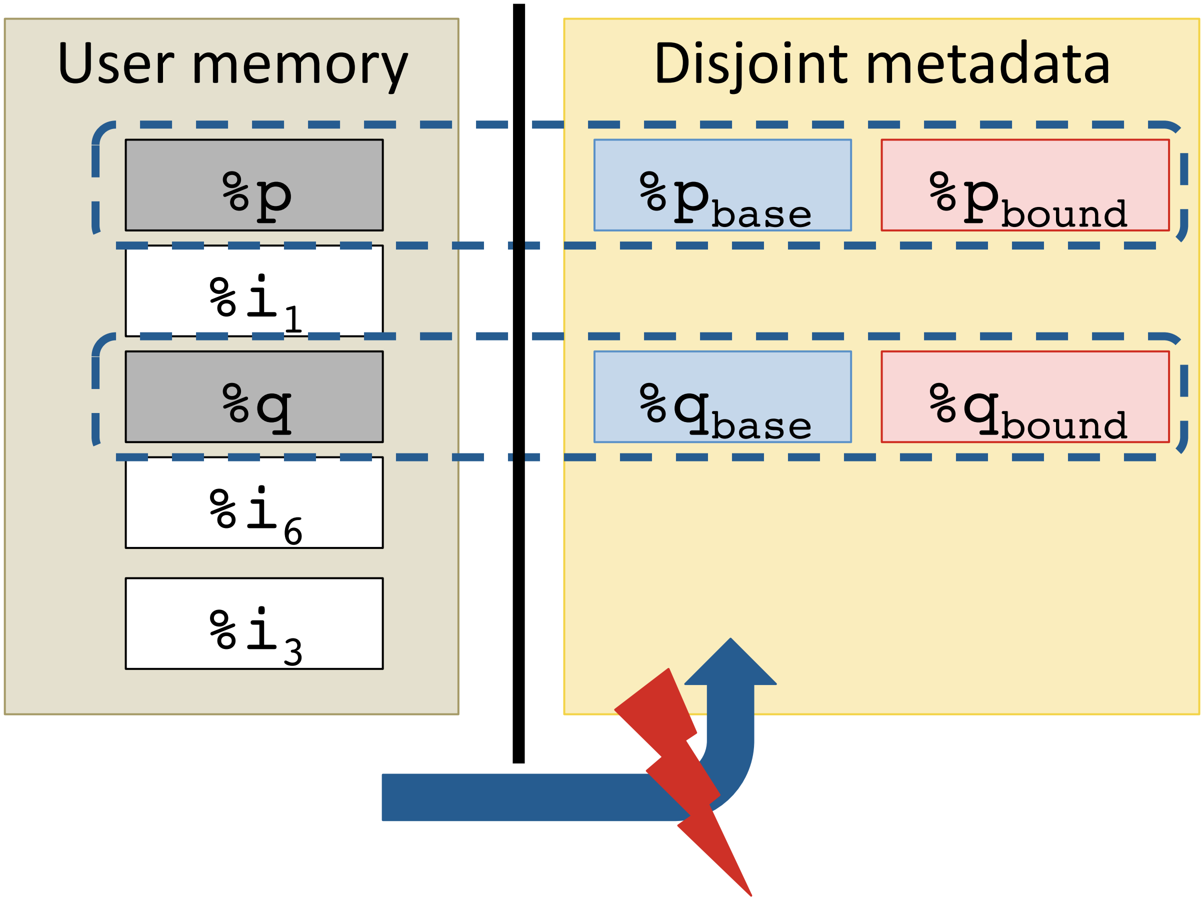
```
store i8 0, %q
```

```
%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
```



Disjoint Metadata

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



Proving SoftBound Correct

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

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

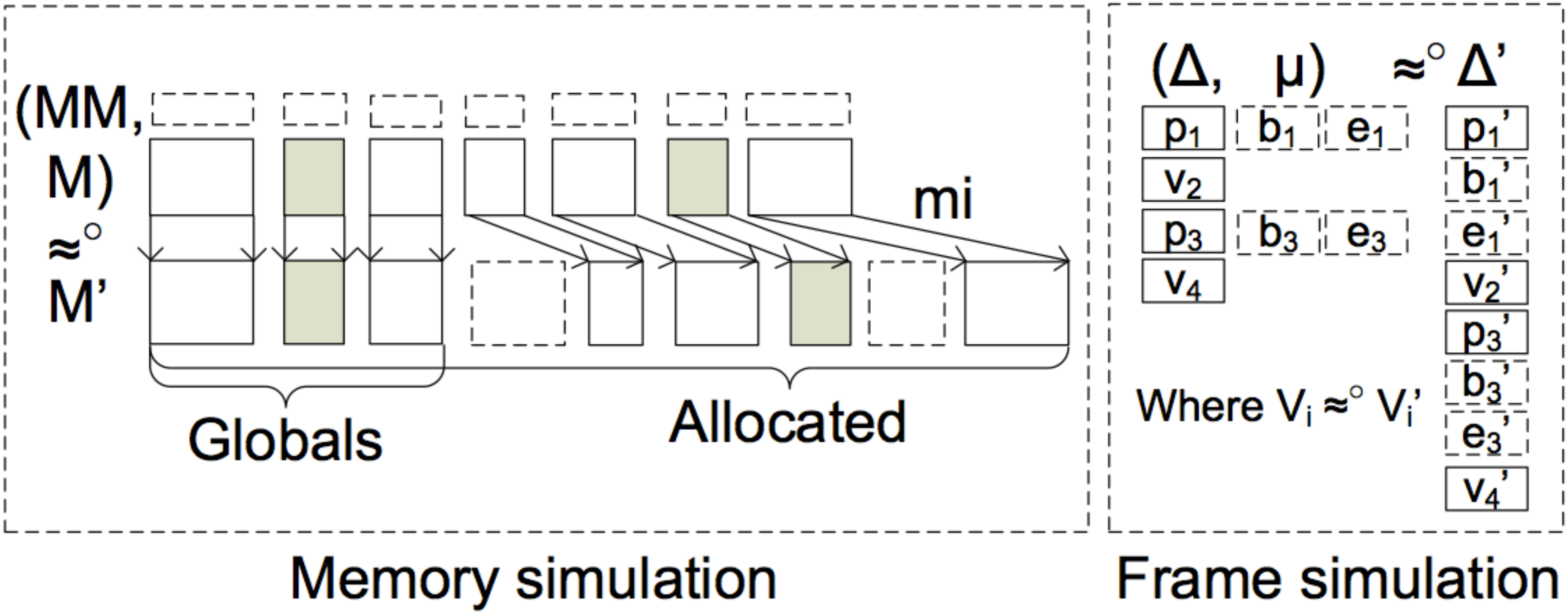
   - Builds in safety invariants "by construction"

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

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

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

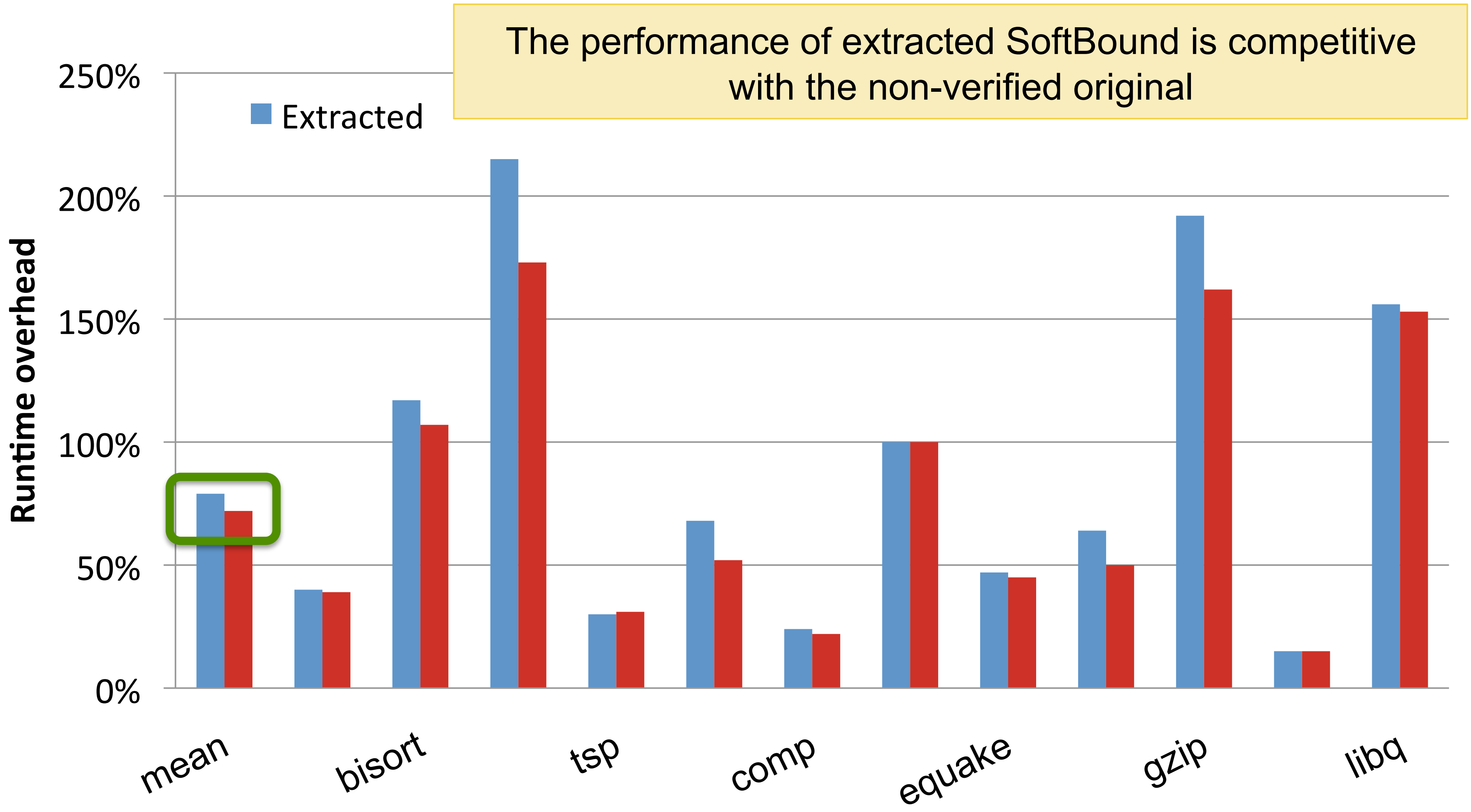
Memory Simulation Relation



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

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.
  - Simulation proofs
- Verified:
  - Softbound & vmem2reg
  - Similar performance to native implementations
- See the papers/coq sources for details!
- Future:
  - Clean up + make more accessible
  - Alias analysis? Concurrency?
  - Applications to more LLVM-SSA optimizations

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