Verifying the LLVM

Steve Zdancewic

DeepSpec Summer School 2017

Substitution.v

SUBSTITUTION

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:

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

where P : Program $\longrightarrow$ Locals $\longrightarrow$ 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.

Verified "Micro" Transformations

- Redundant instruction insertion
  - add new (unused) definitions
  - add a redundant store:

```
%x = load t* %p   ⇒  %x = load t* %p
                     store %p, %x
```

- Dead-instruction elimination (DIE)
- Dead store elimination (DSE)
  - if there are no loads from an address after a store, eliminate the store
- Dead alloca elimination (DAE)
  - if there are no loads or stores to an allocated address, remove it

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 some LLVM transformations

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.

Strategy for Proving Transformations

- 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

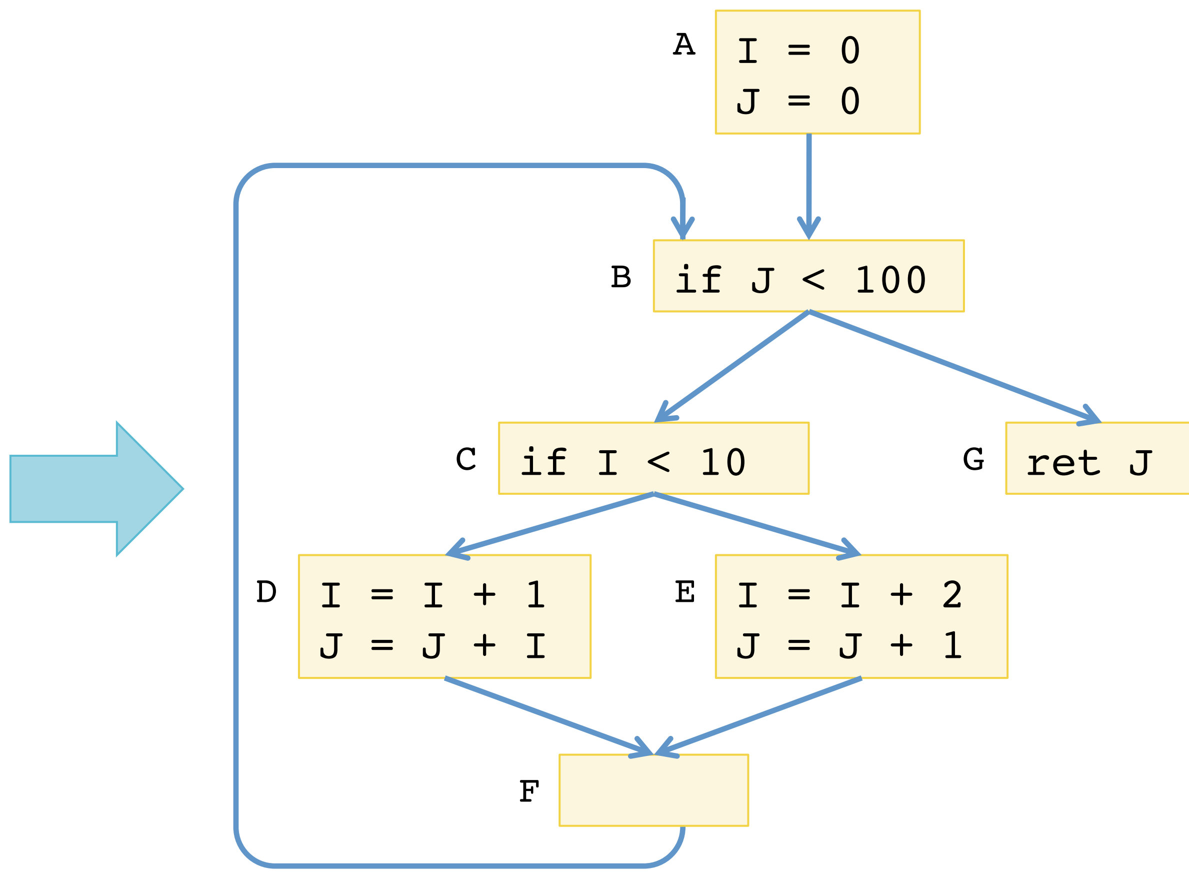
SSA CONSTRUCTION

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

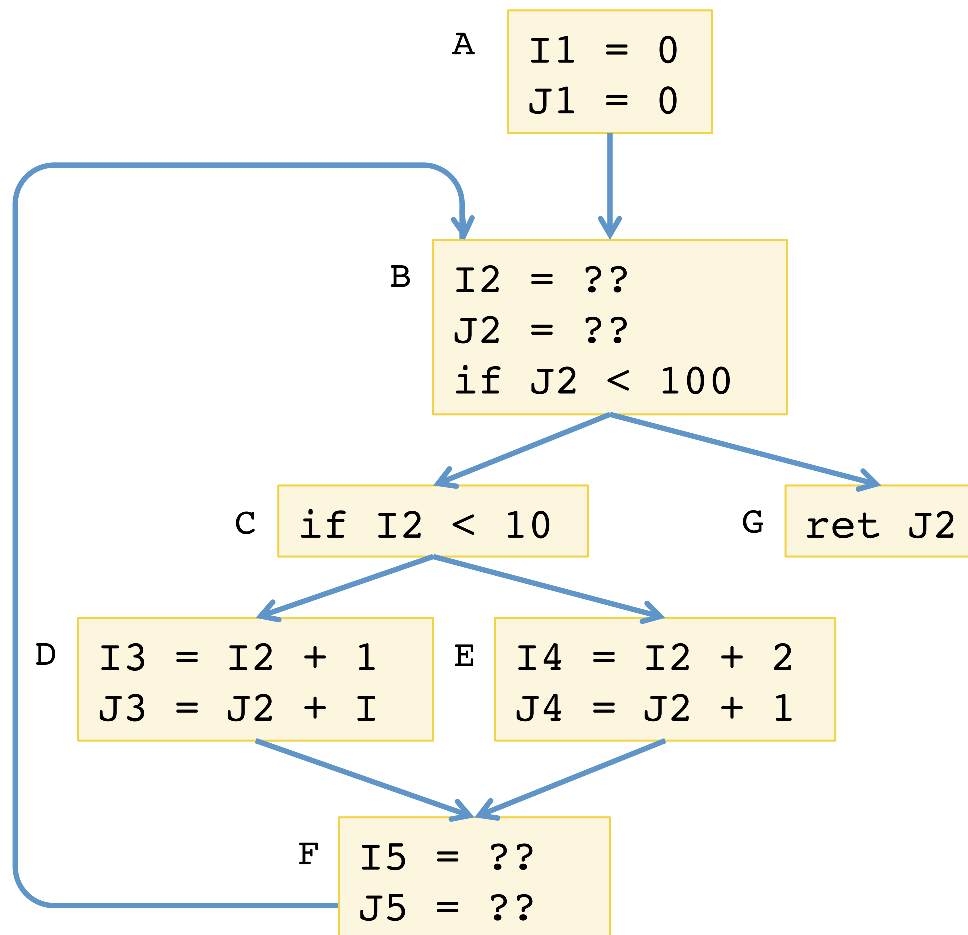
```
I := 0;;
J := 0;;
WHILE J < 100 DO
   IF I < 10 THEN
      I := I + 1;;
      J := J + I
   ELSE
      I := I + 2;;
      J := J + 1;
   FI
END;;
RETURN J
```



Step 1: Convert to a control-flow graph.

SSA Construction by Example

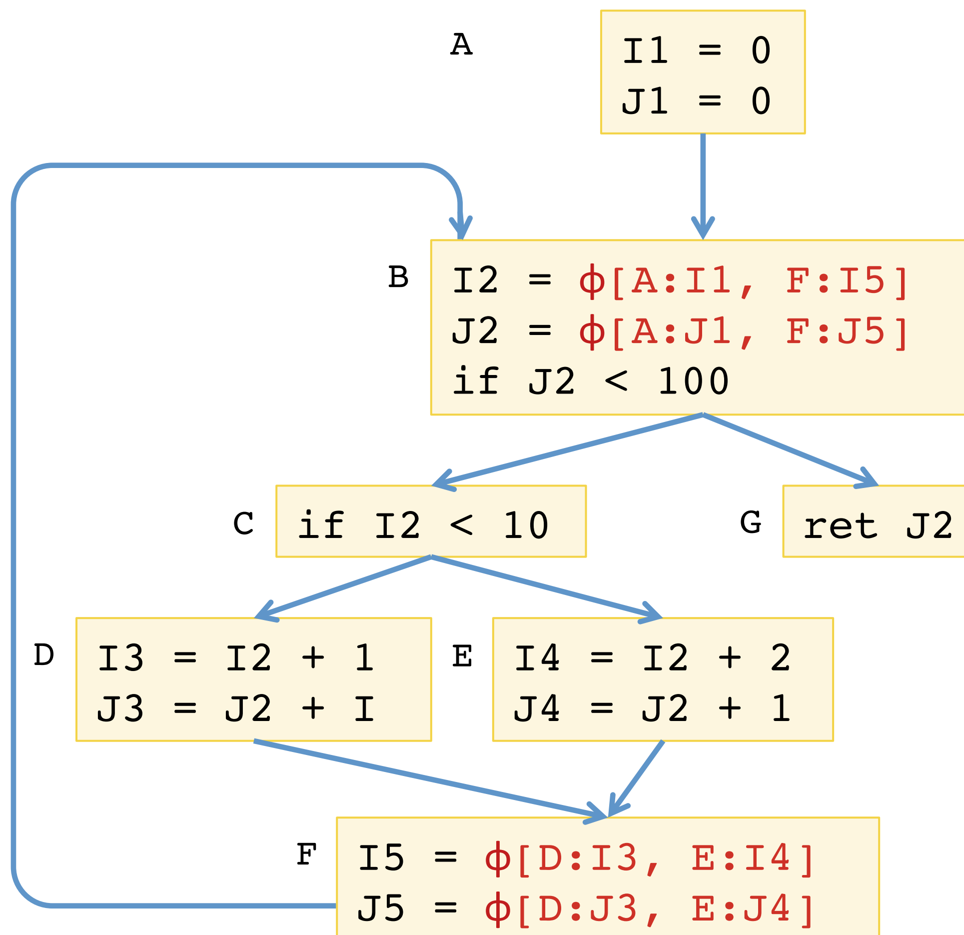
```
I := 0;;
J := 0;;
WHILE J < 100 DO
   IF I < 10 THEN
      I := I + 1;;
      J := J + I
   ELSE
      I := I + 2;;
      J := J + 1;
   FI
END;;
RETURN J
```



Step 2: Rename variables to satisfy single assignment.

SSA Construction by Example

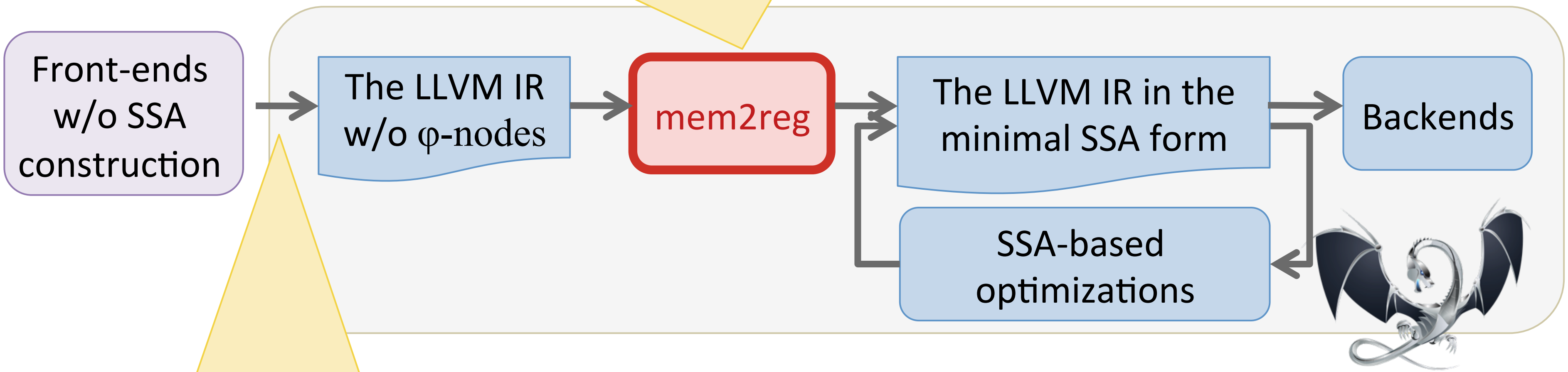
```
I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
```



Step 3: Insert "ϕ" functions that capture control dependence.

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;
```

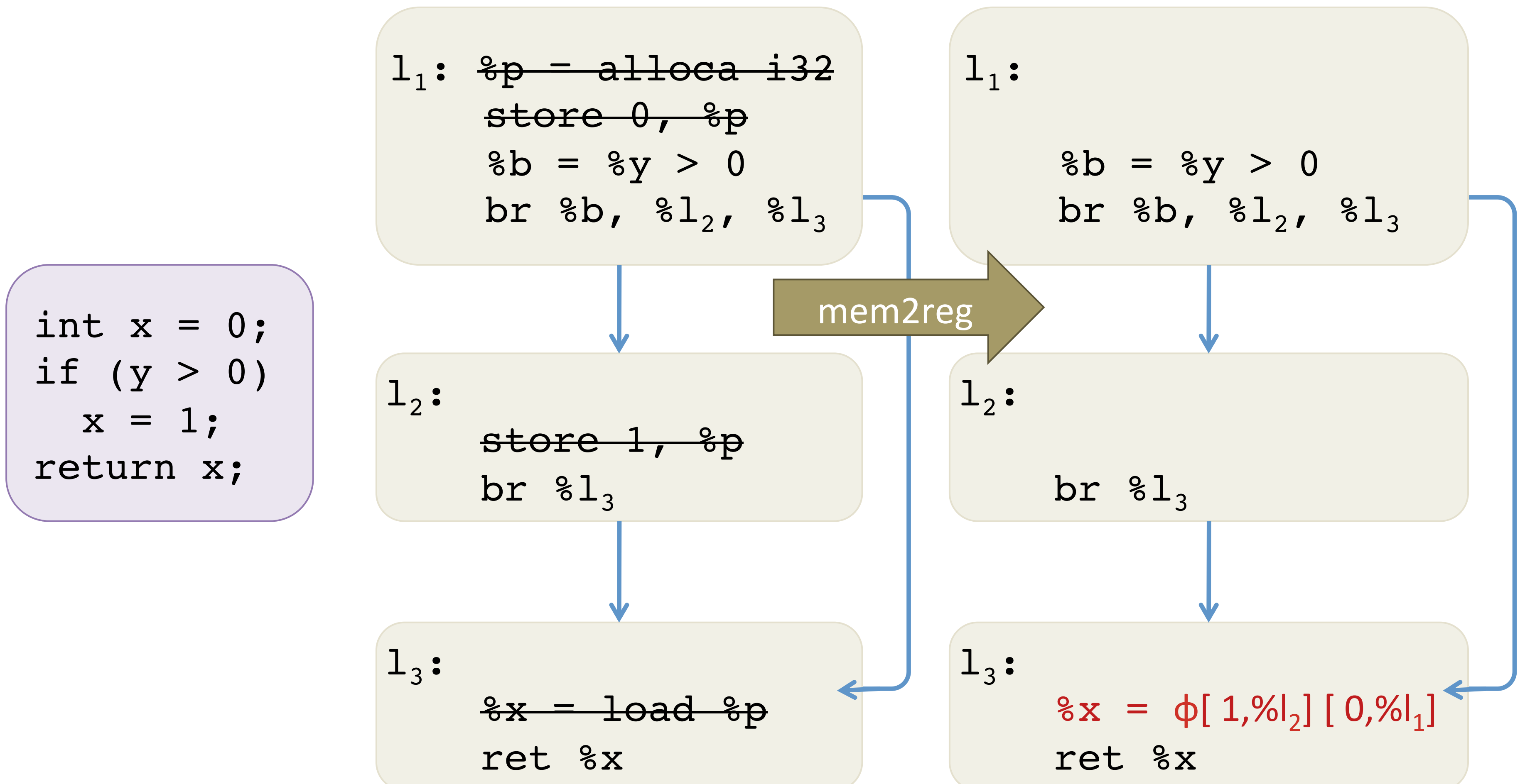
```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3
```

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

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

The LLVM IR in the trivial SSA form

mem2reg Example



The LLVM IR in the trivial SSA form

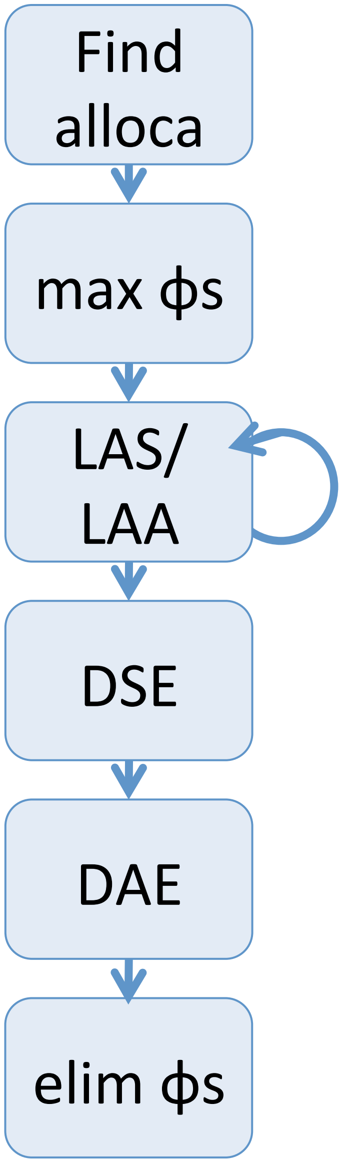
Minimal SSA after mem2reg

mem2reg Algorithm

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

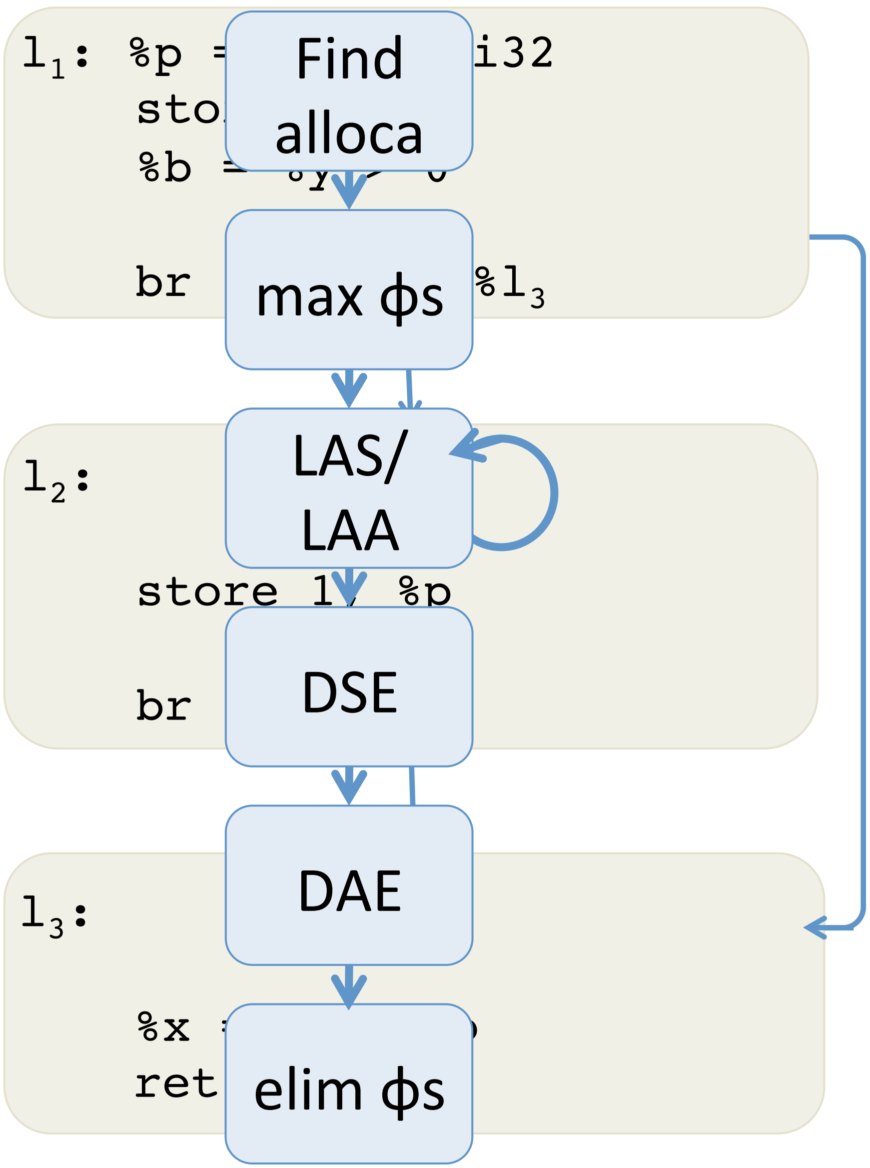
- Intermediate stage breaks SSA invariant
  - Defining semantics & well formedness non-trivial

vmem2reg Algorithm



- Incremental algorithm
- Pipeline of micro-transformations
  - Preserves SSA semantics
  - Preserves well-formedness
- Inspired by Aycock & Horspool 2002.

Example of vmem2reg Algorithm

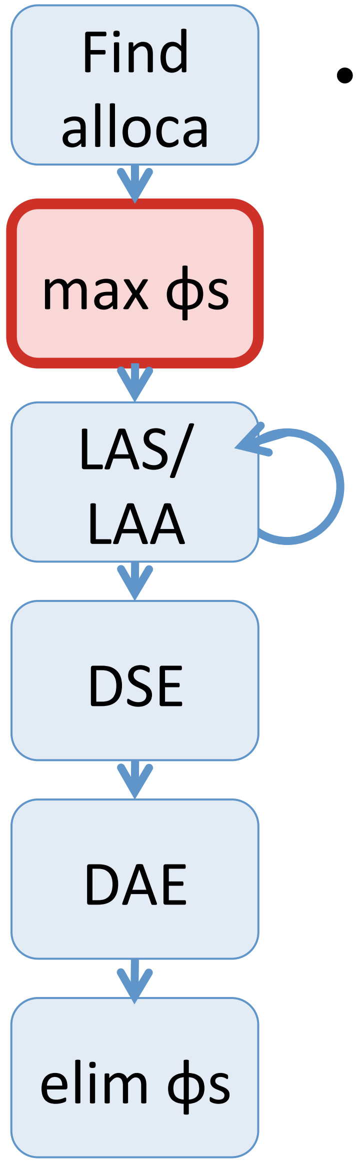


Example of vmem2reg Algorithm

```
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
```



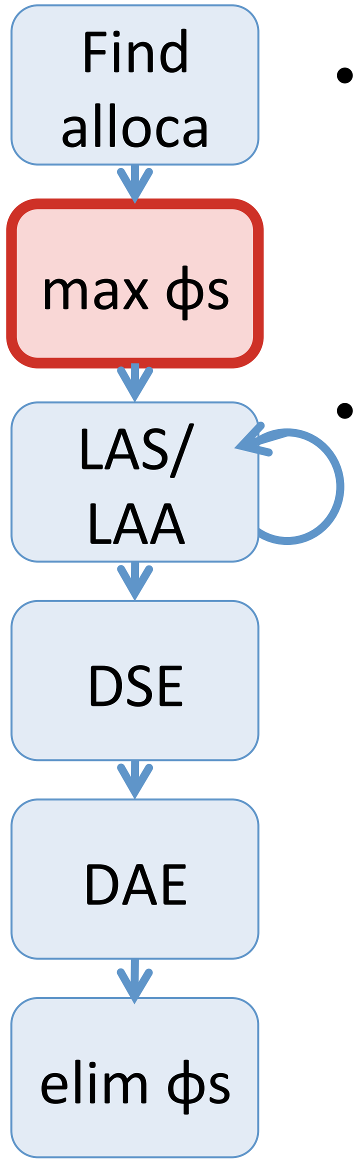
- How to place phi nodes without breaking SSA?

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
```



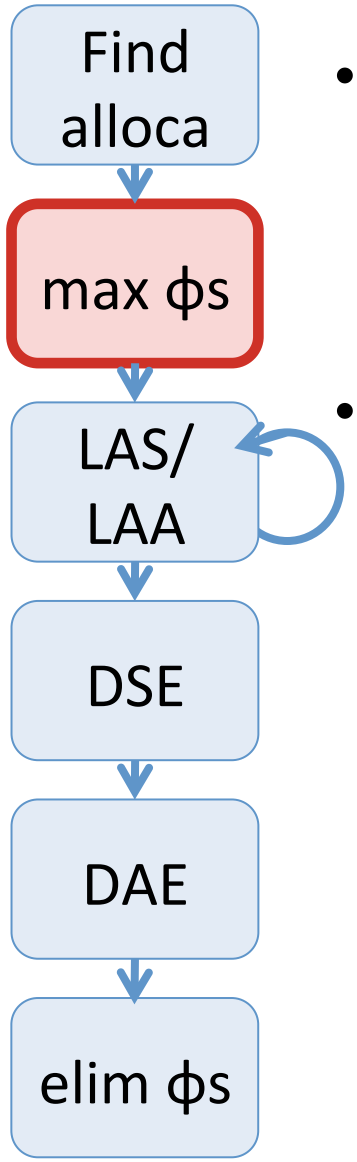
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

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
```



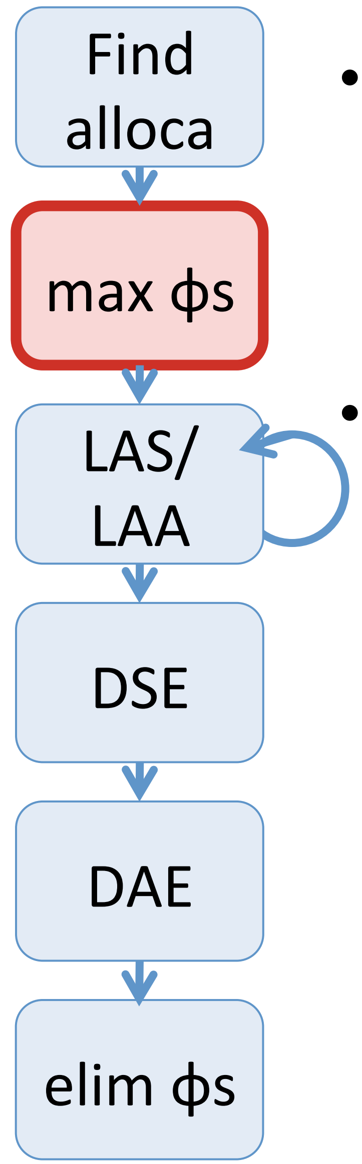
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block

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
```



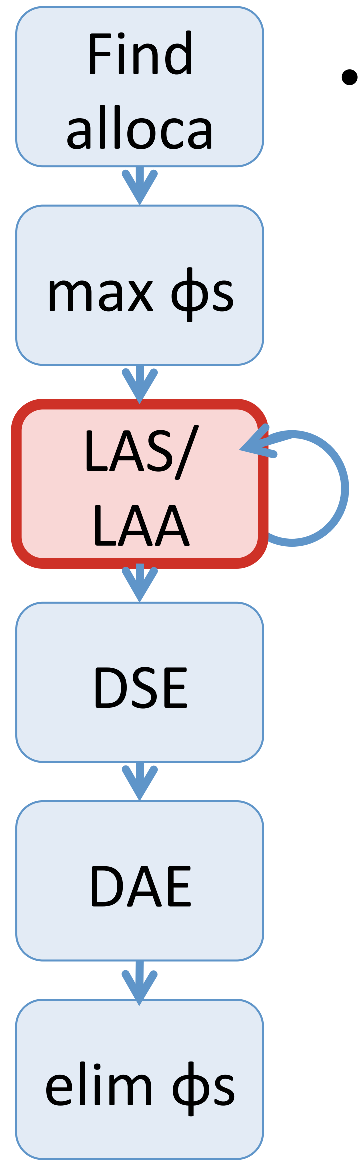
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block
  - Insert stores after ϕ-nodes

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
```

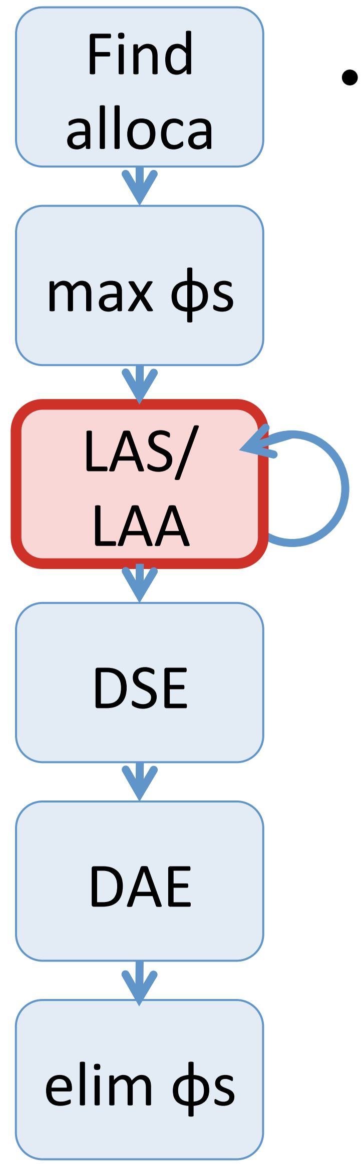
```
l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



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

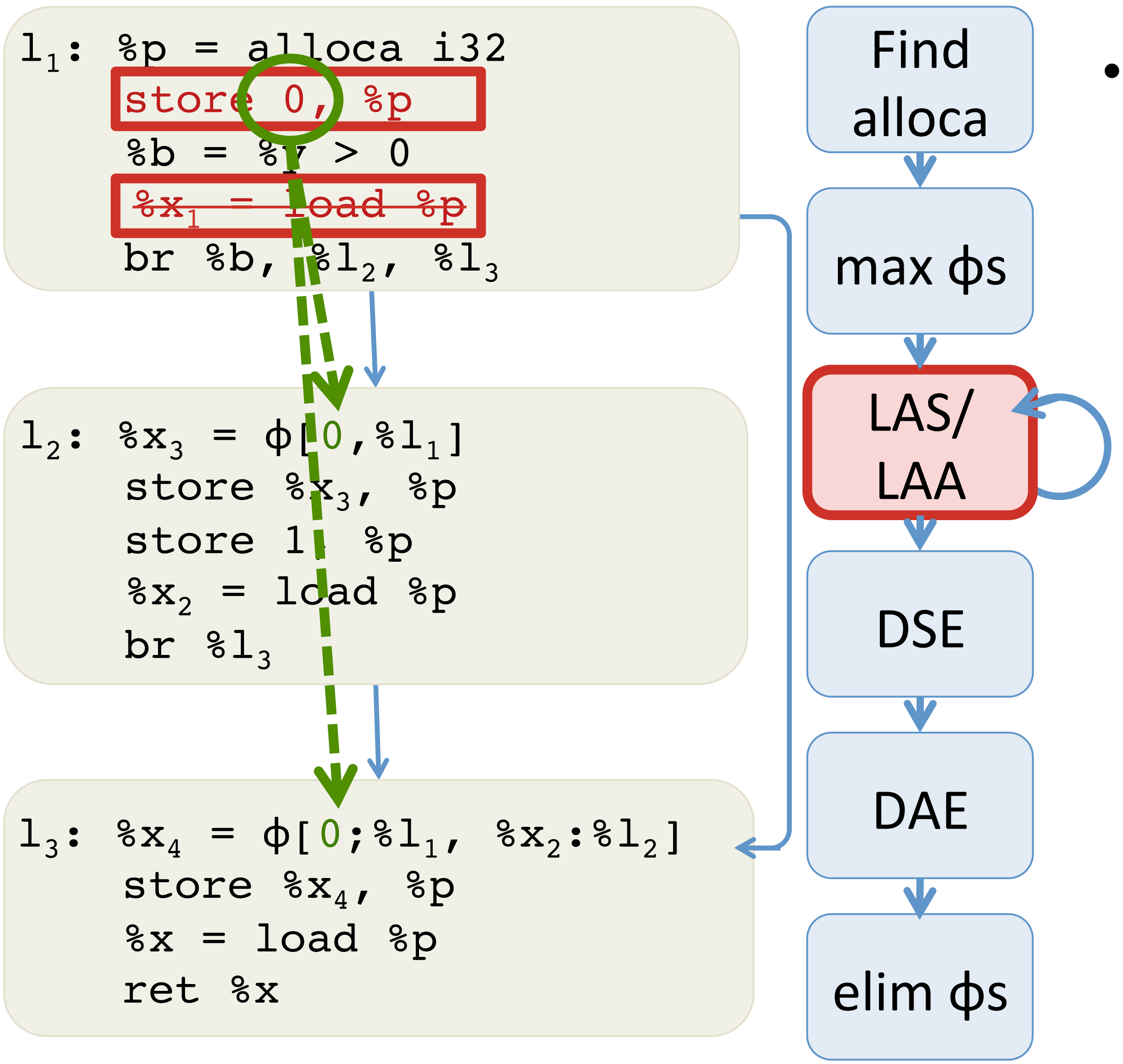
Example of vmem2reg Algorithm

```
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
```



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

Example of vmem2reg Algorithm



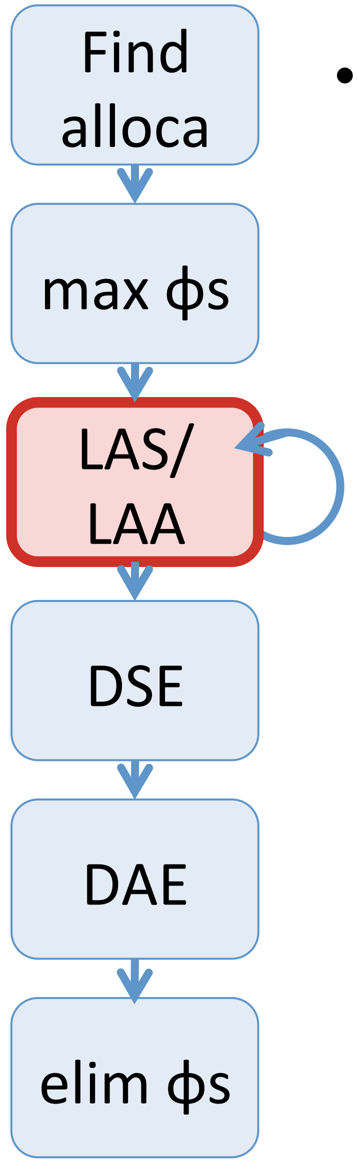
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm

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

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

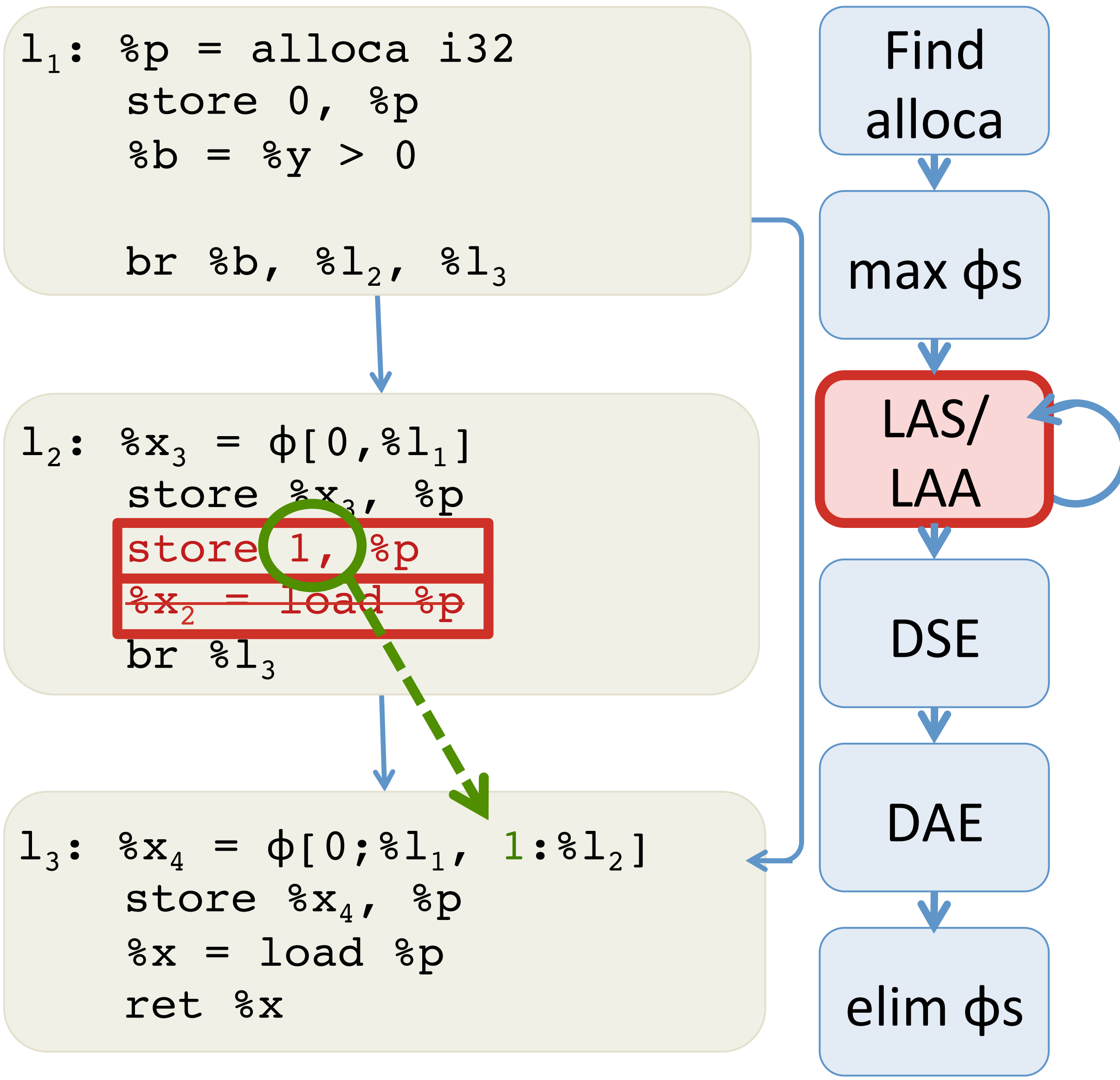
```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
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



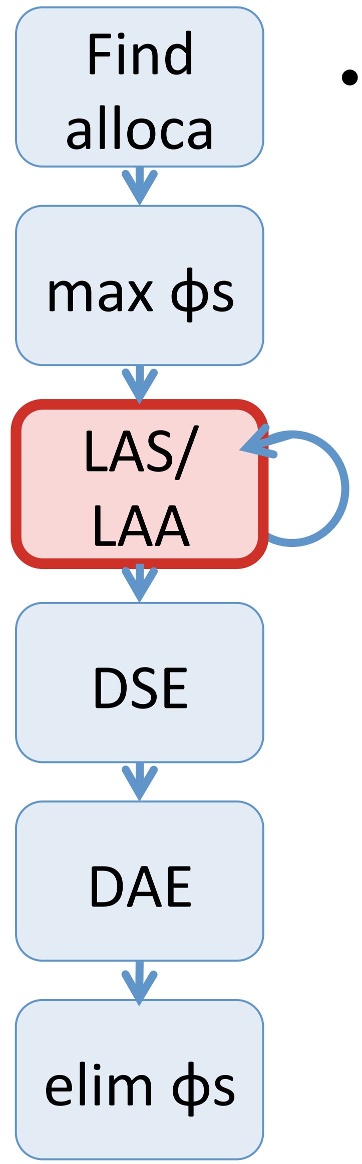
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

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



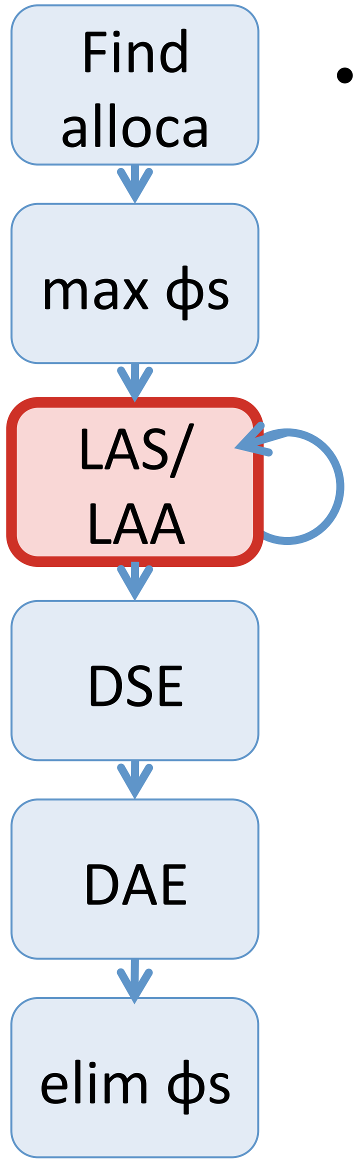
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example of vmem2reg Algorithm

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

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

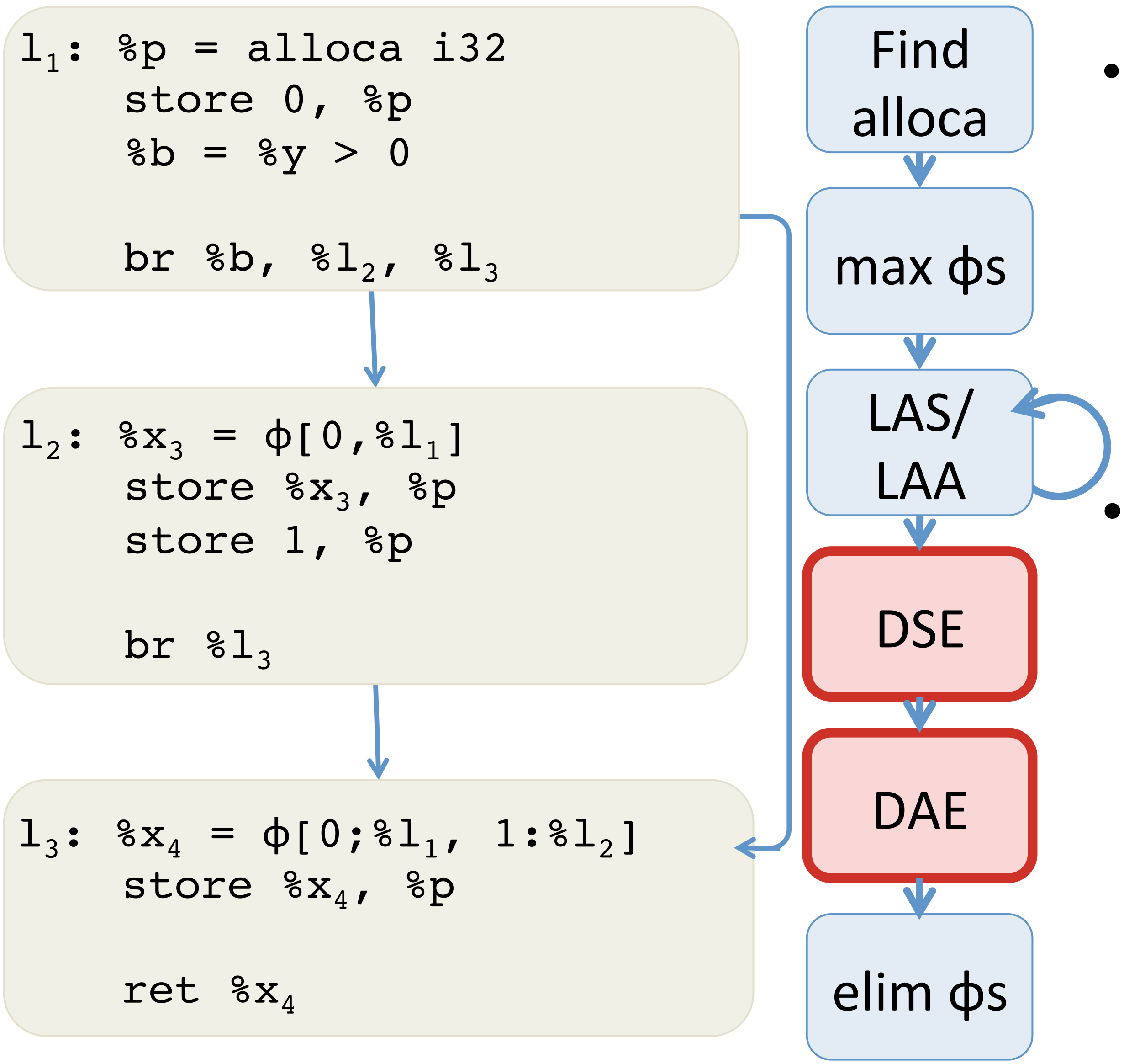
```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

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



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

Example of vmem2reg Algorithm



- 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

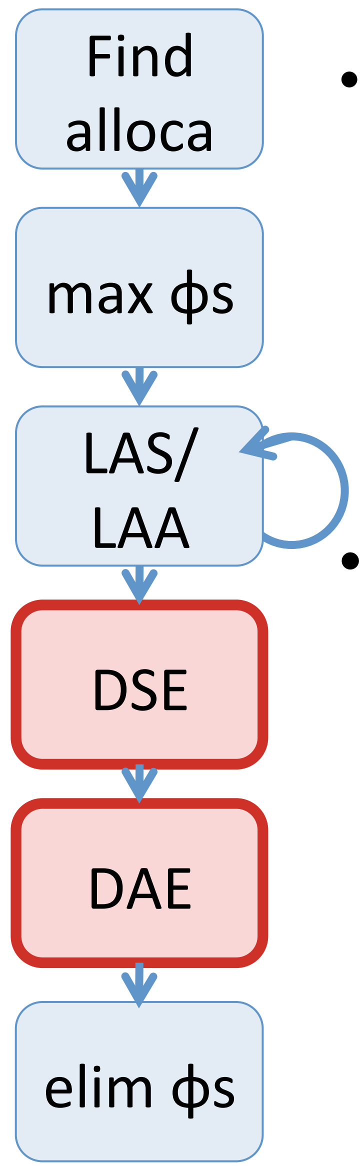
br %b, $\%l_2$, $\%l_3$

l_2: $\%x_3 = \phi[0,\%l_1]$
~~store $\%x_3$, %p~~
~~store 1, %p~~

br $\%l_3$

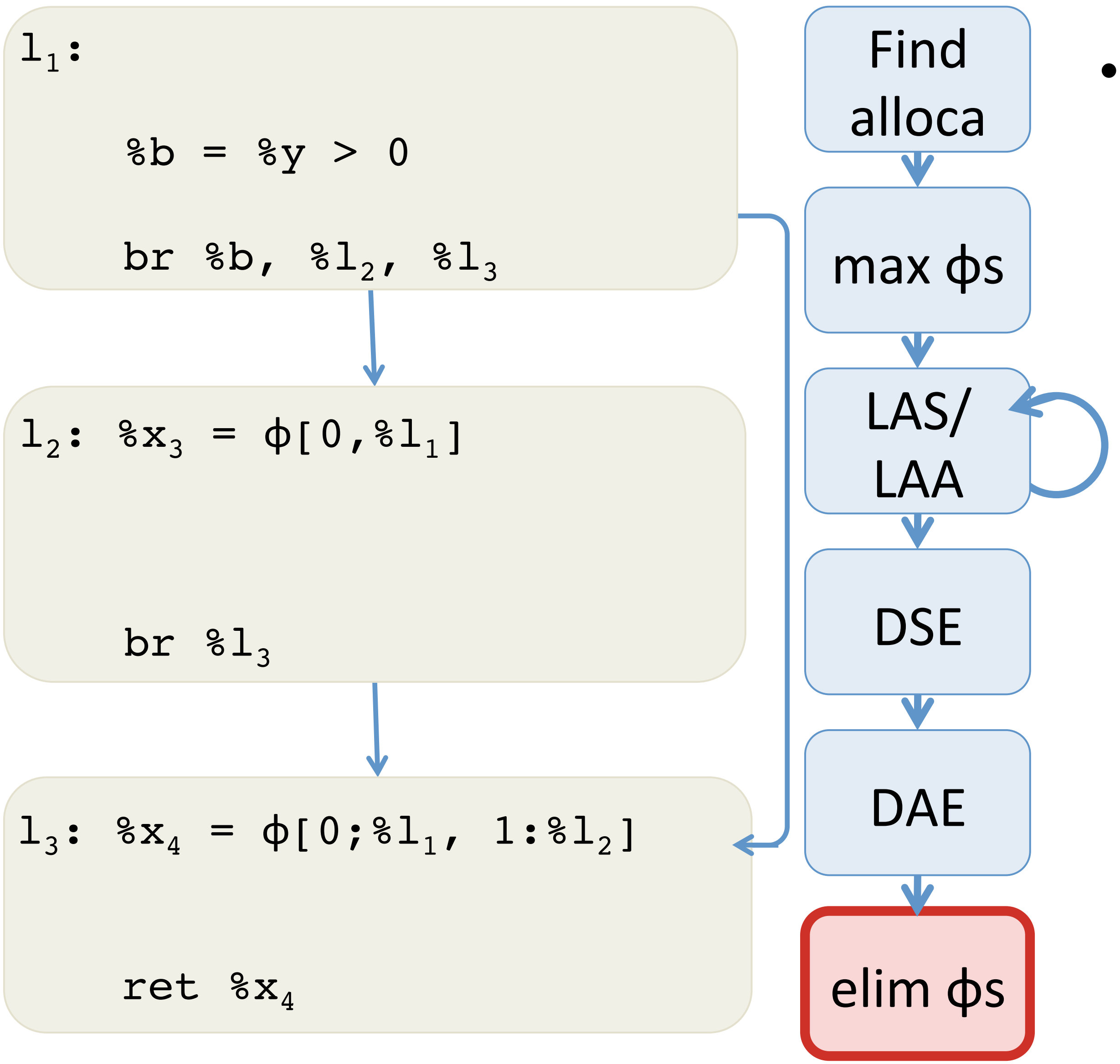
l_3: $\%x_4 = \phi[0;\%l_1,\ 1:\%l_2]$
~~store $\%x_4$, %p~~

ret $\%x_4$



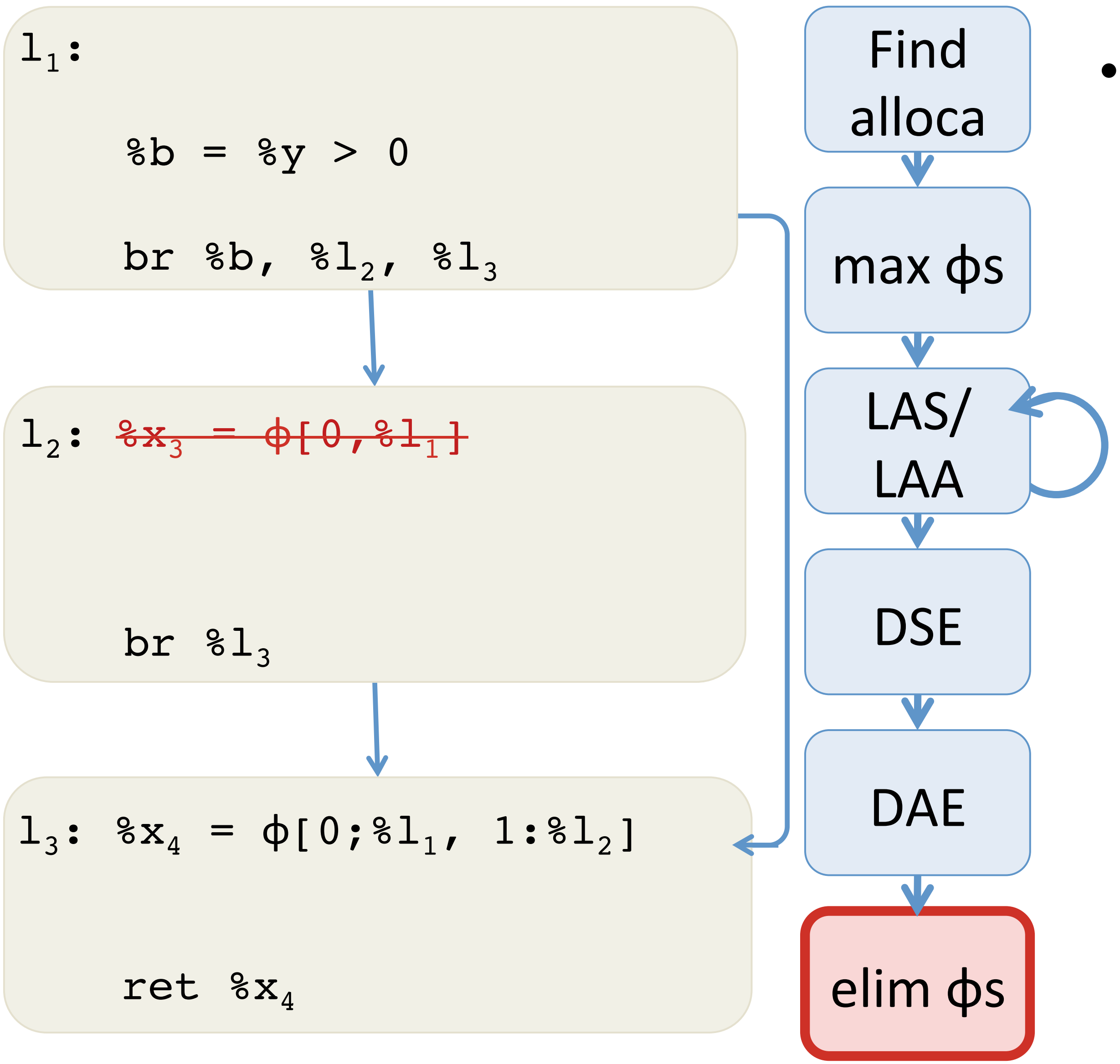
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/ stores.

Example of vmem2reg Algorithm



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

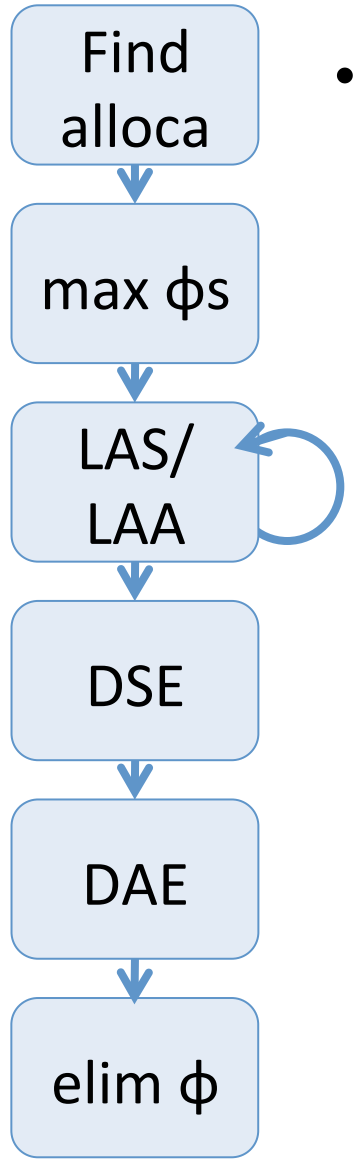
Example of vmem2reg Algorithm



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

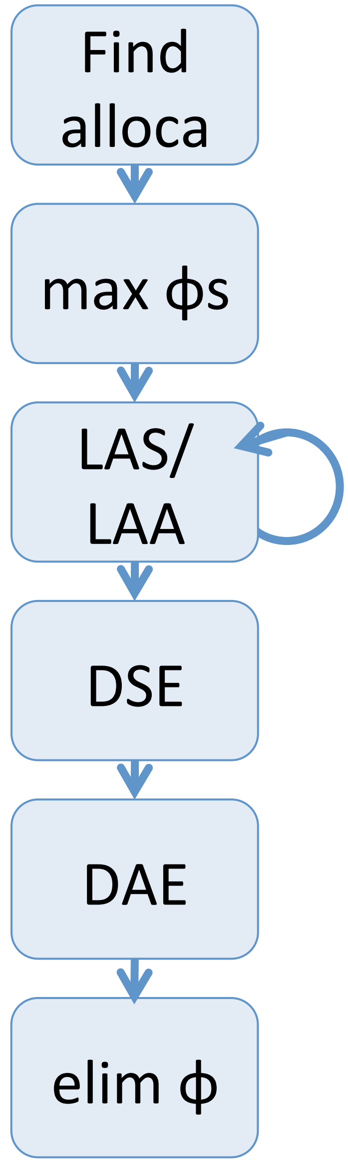
Example of vmem2reg Algorithm

```
l1:
    %b = %y > 0
    br %b, %l2, %l3

l2:
    br %l3

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

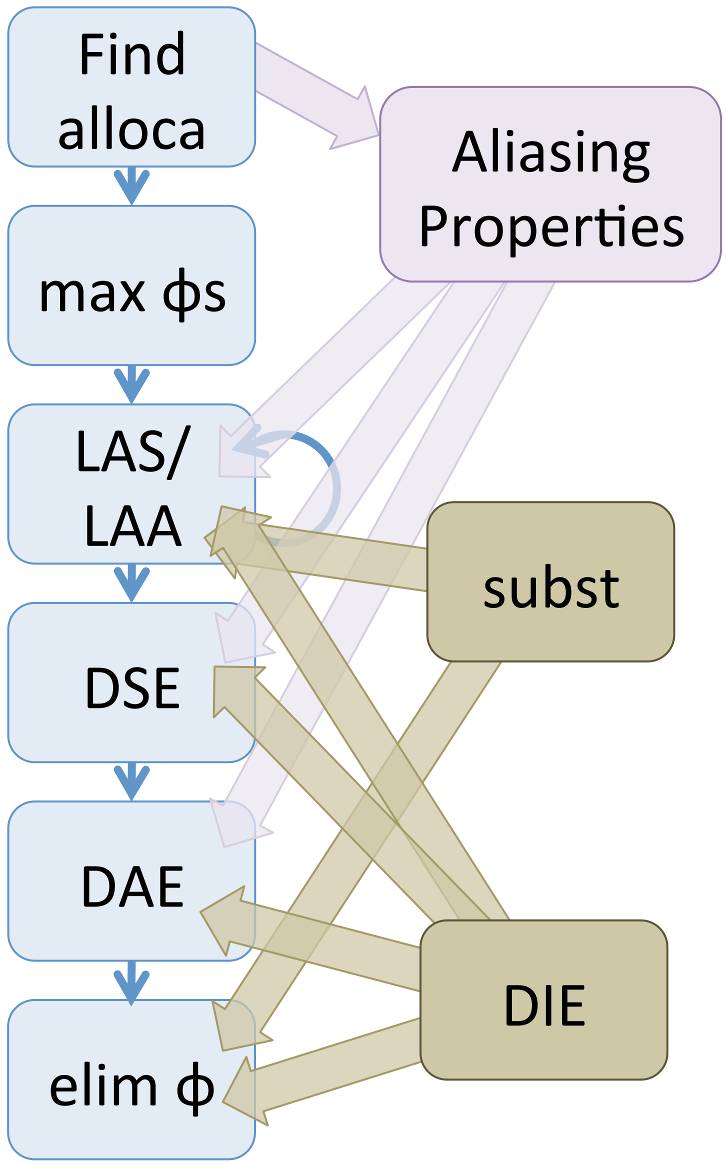


- Done!

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.) □

SCALING UP: LLVM

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

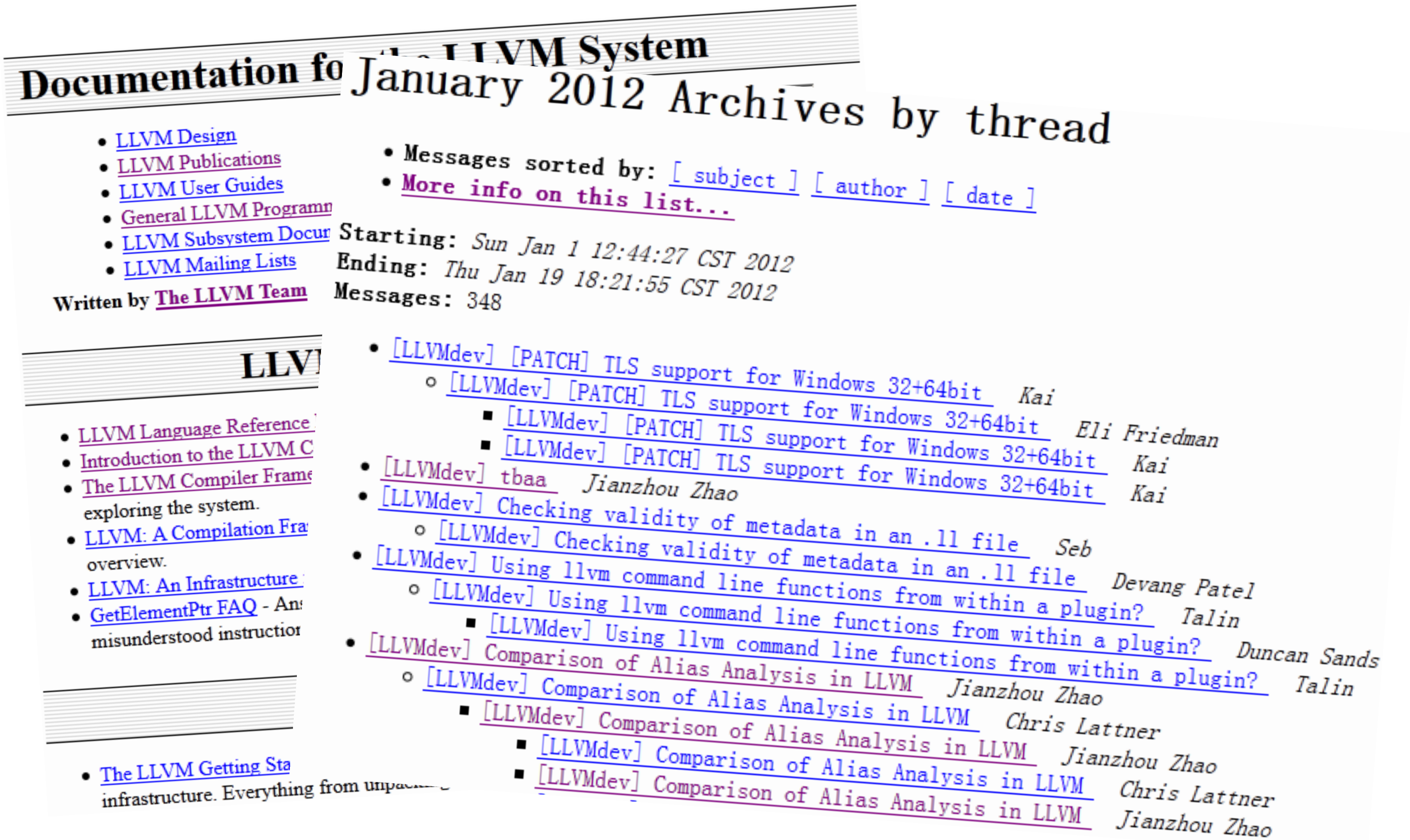
- LLVM's IR is uses types to describe the structure of data.

```
ty ::=
  | i1 | i8 | i32 |…              N-bit integers
  | [<#elts> x t]                 arrays
  | r (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
```

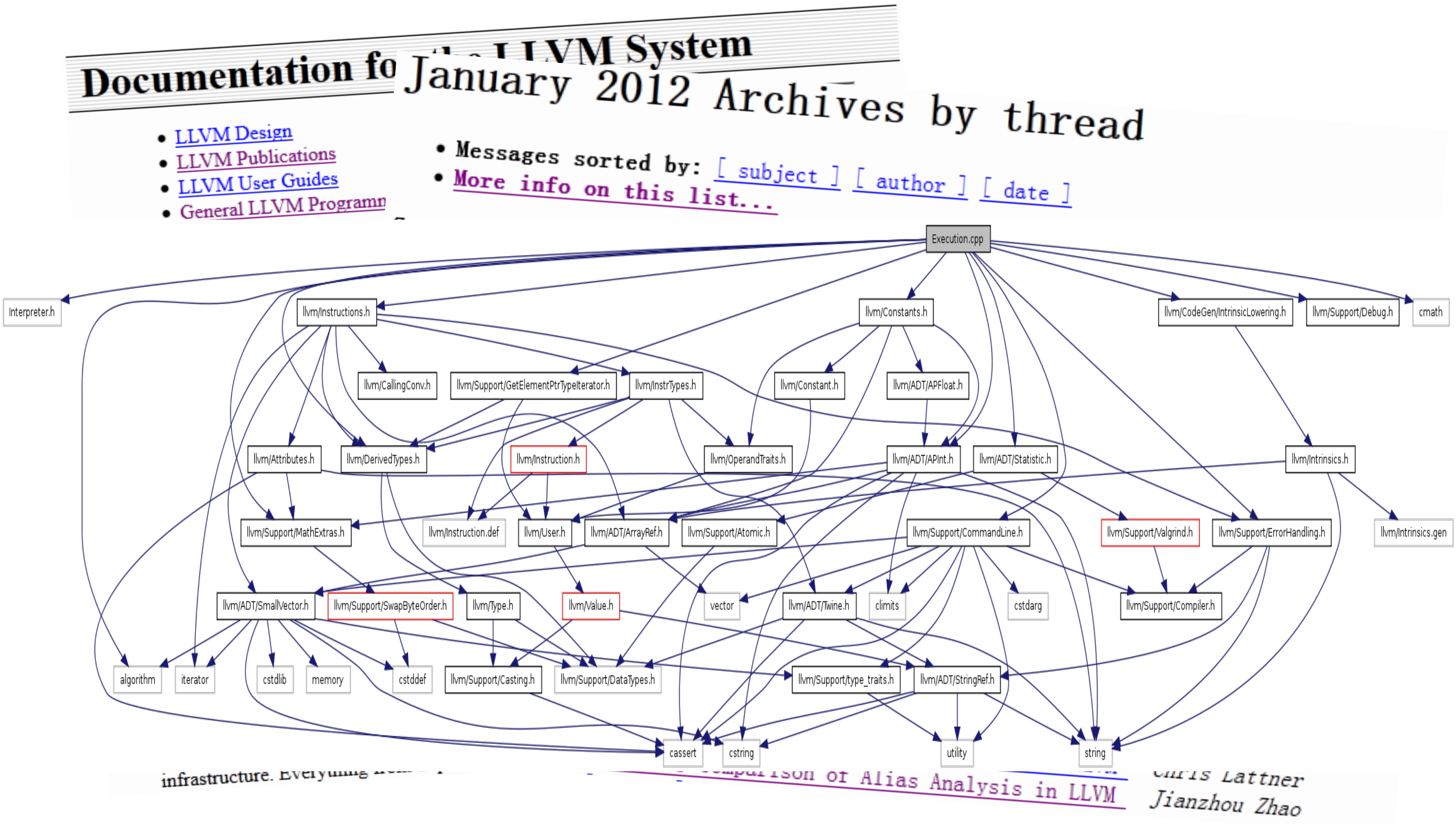
- <#elts> is an integer constant >= 0
- (Recursive) Structure types can be named at the top level:

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

Distilling the LLVM



Distilling the LLVM



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
}
```

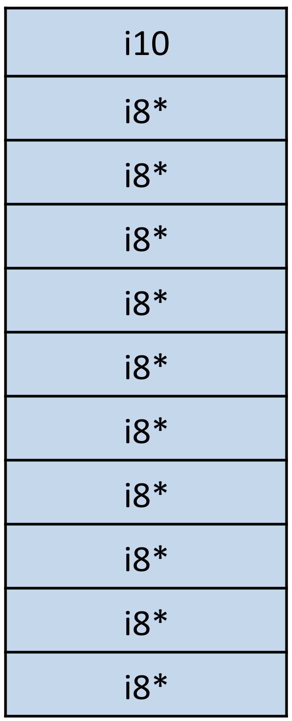
Final answer: ADDR + `sizeof(struct ST) + sizeof(struct RT) + sizeof(int) + sizeof(int) + 5*20*sizeof(int) + 13*sizeof(int)`

*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

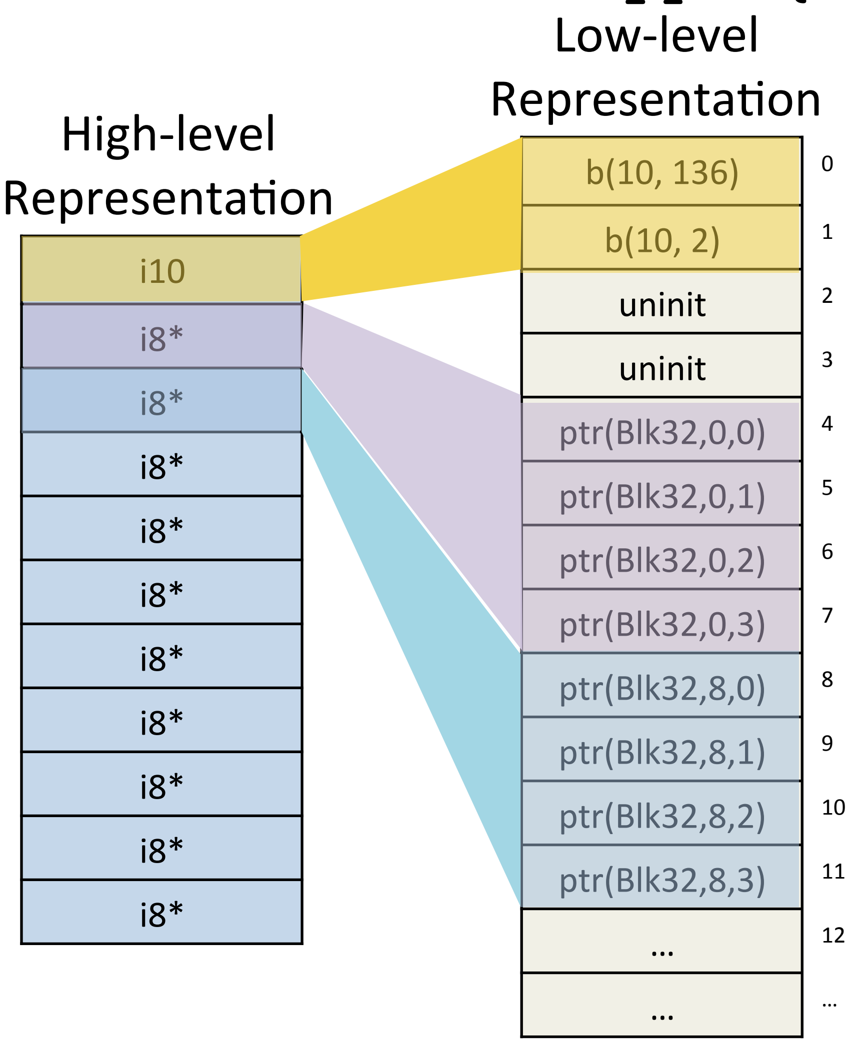


- Manipulate structured types.

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

LLVM’s memory model

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

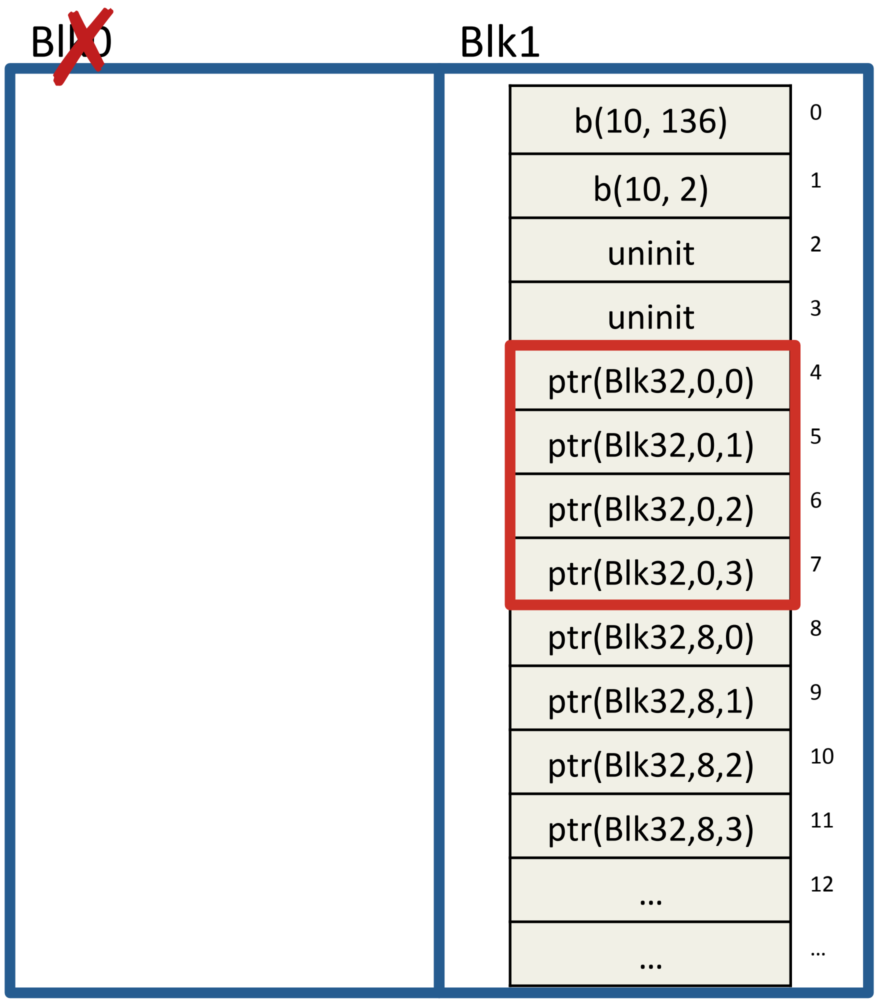


- Manipulate structured types.

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

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

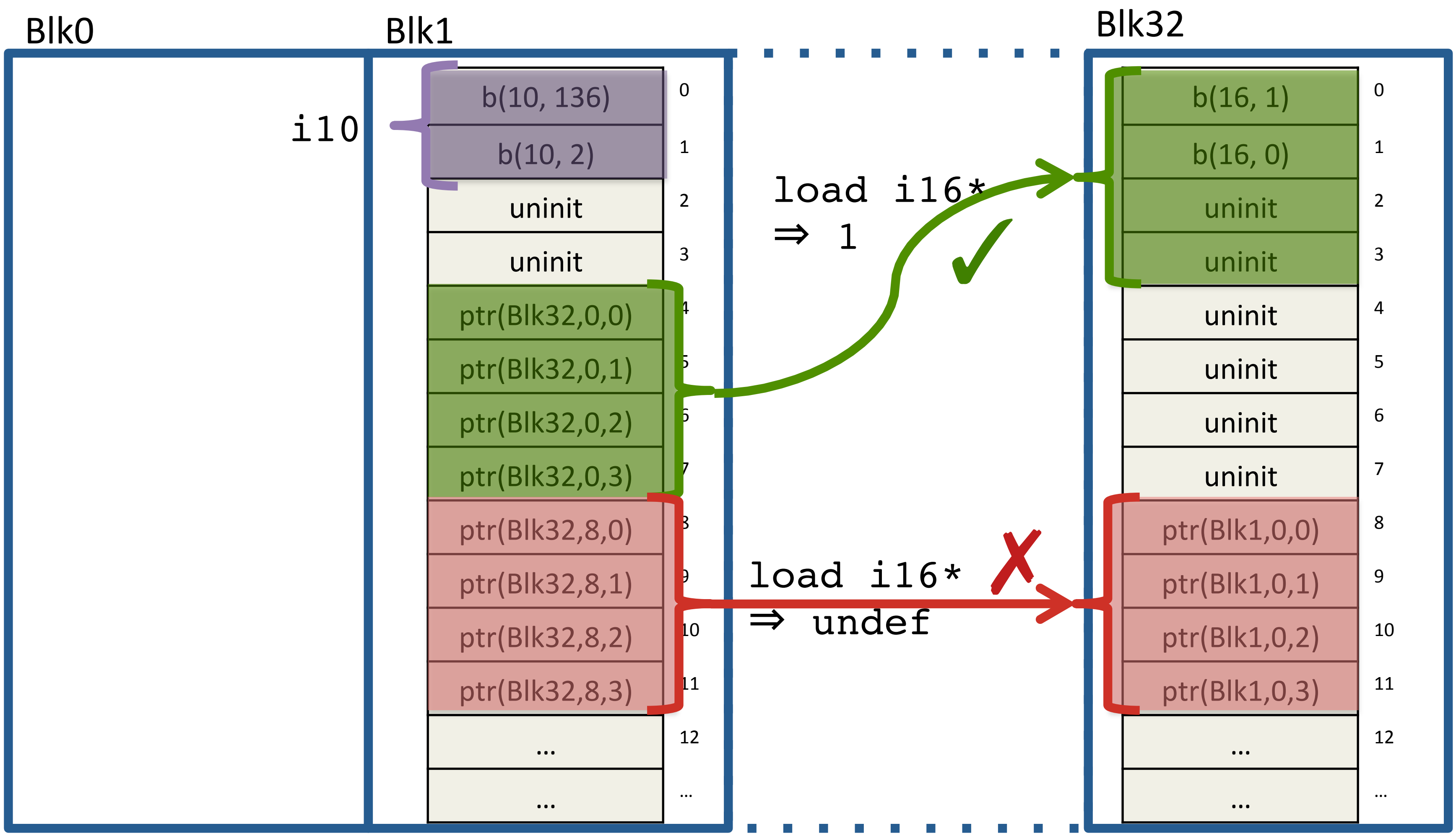
Adapting CompCert's Memory Model



- 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*]



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`

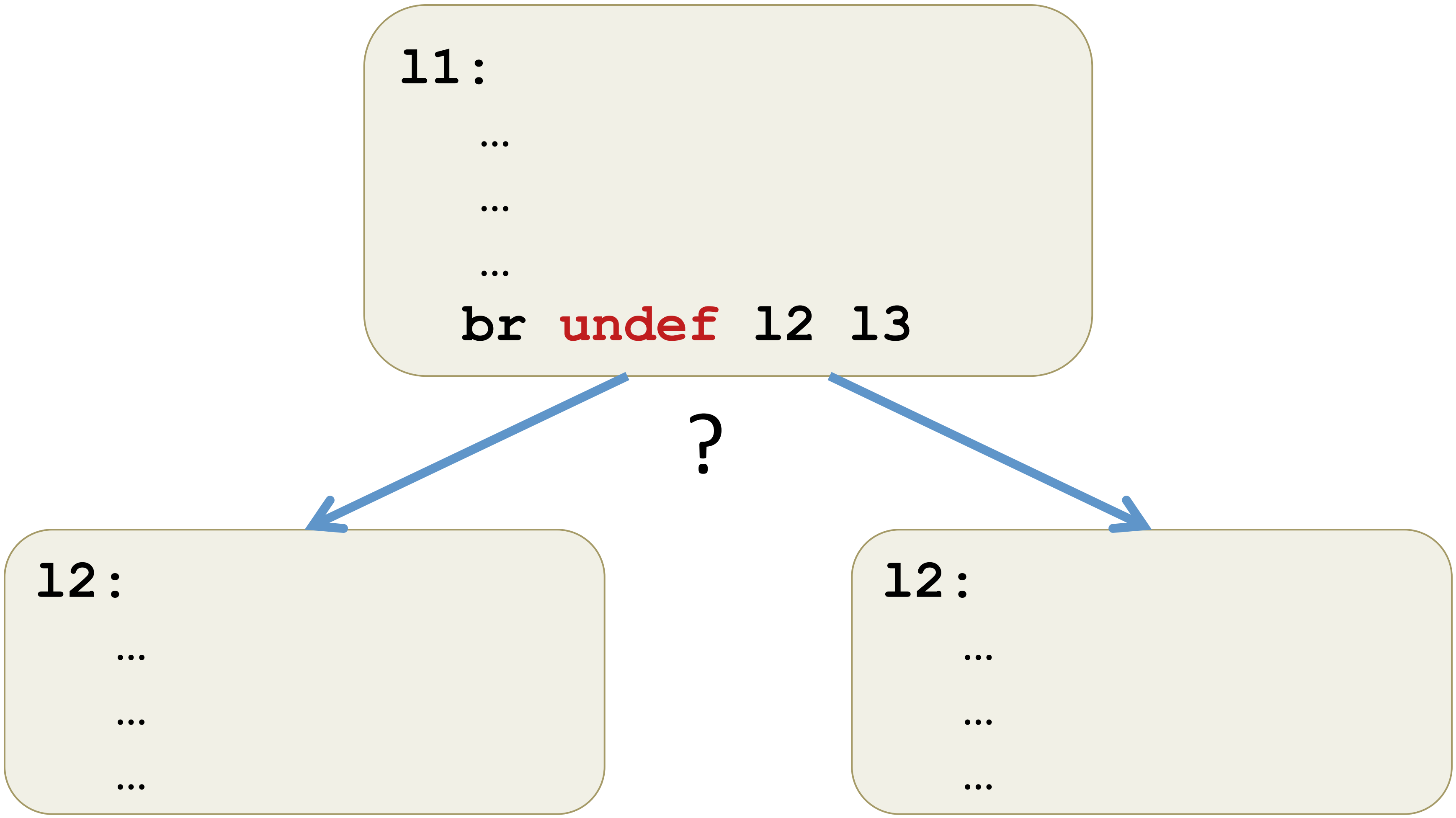
- Partially defined values are interpreted *nondeterministically* as sets of possible values:

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

$$[\![\texttt{i8 undef}]\!] = \{0,\ldots,255\}$$
$$[\![\texttt{i8 1}]\!] = \{1\}$$

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

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	$\boxed{\mathrm{LLVM}_{ND}}$	
Deterministic		

Deterministic Refinement

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

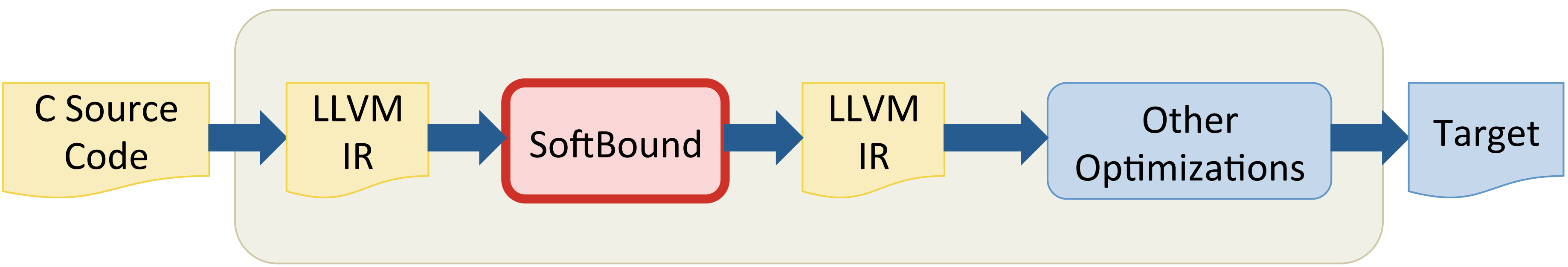
Simulation up to "observable events":

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

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

SoftBound

- 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



SoftBound

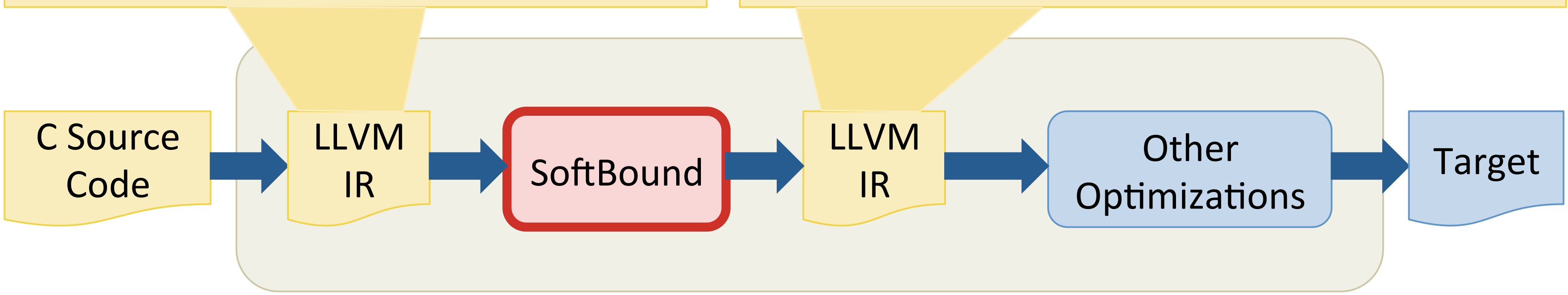
```
%p = call malloc [10 x i8]

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

store i8 0, %q
```

Maintain base and bound for all pointers

Propagate metadata on assignment

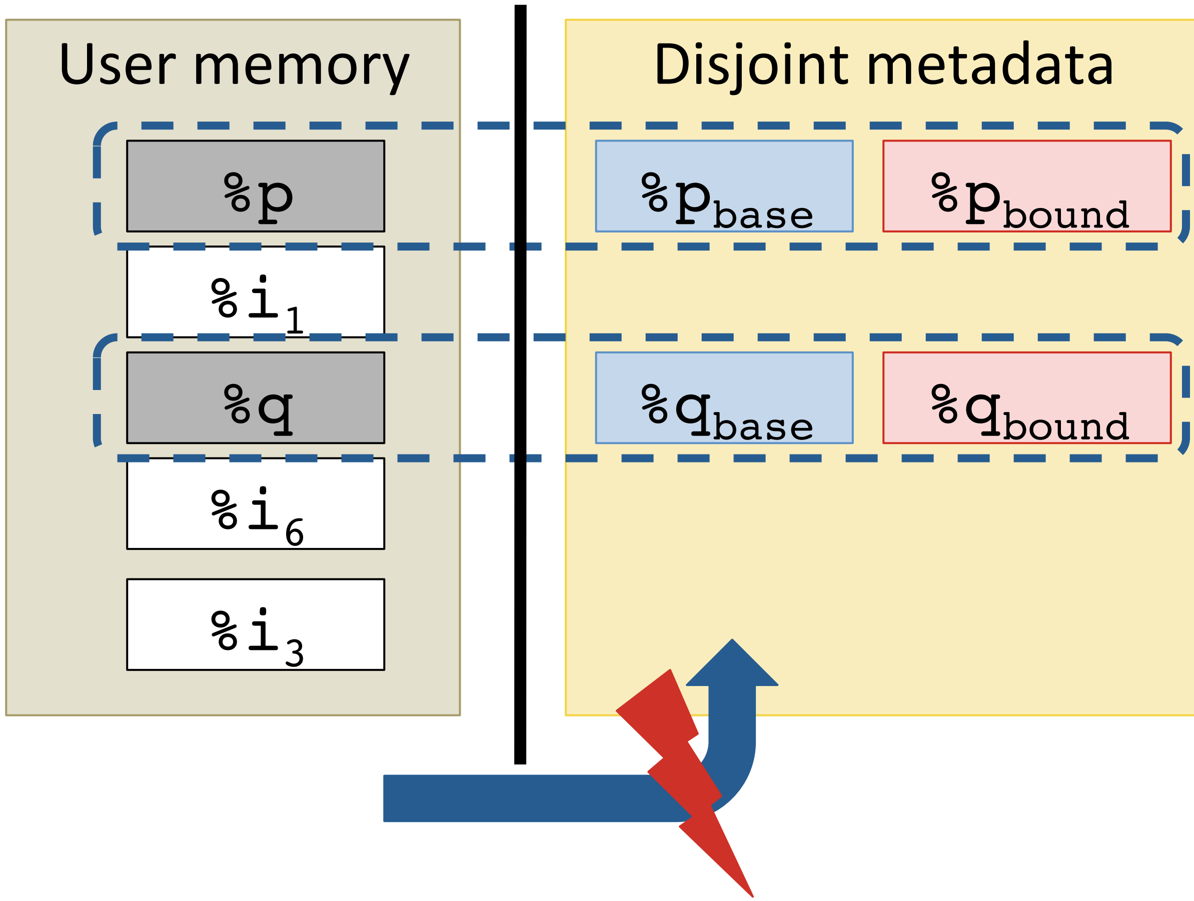
Check that a pointer is within its bounds when being accessed

```
%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
```



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:

$$f \vdash \sigma \overset{SB}{\longmapsto} \sigma'$$

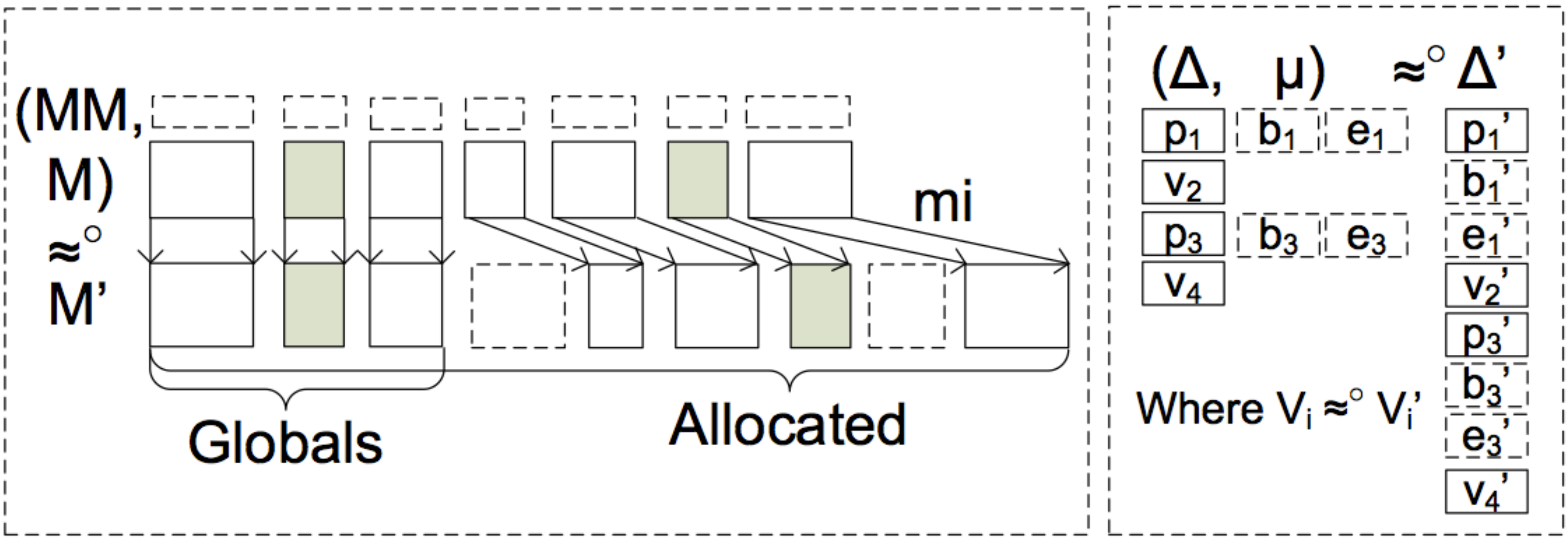
   – Builds in safety invariants "by construction"

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

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

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

Memory Simulation Relation



Memory simulation

Frame simulation

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)

VELLVM ⇒ VELLVM II

The Bad

- Large, monolithic code base
- Clunky proofs

Development	LOC (Defns. + Proofs)
syntax + semantics	~45K
mem2reg + optimizations	~60K
SoftBound	~15K

$\Rightarrow$ hard for others to adopt/adapt

The Bad

- Representation & semantics very syntactic
  - cfg ≈ lists of blocks
  - block ≈ lists of instructions
  - operational semantics uses this syntax

  ⇒ lots of boiler plate everywhere
- Tightly coupled to the memory model, design non-modular
- LLVM transformations not designed for verification
  - translating "informal" to "formal" proofs is difficult
- Limited use of proof automation

⇒ Hard to deal with change:

LLVM 2.6 ⇒ LLVM 3.0 ⇒ LLVM 3.6 ⇒ …

Coq 8.2 ⇒ Coq 8.3 ⇒ Coq 8.4 ⇒ Coq 8.5 ⇒…

CompCert's memory model evolution

Ongoing Work

- "Legacy Vellvm"
  - defunct
- Vellvm II
  - https://github.com/vellvm/vellvm

- Modernize & Refactor the development
  - more streamlined
  - experiment with LTS operational semantics

Partial Bibliography

- 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]
- LLVM Semantics
  - Taming Undefined Behavior in LLVM [Lee et al., PLDI17]

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/