Lecture 25

CIS 4521/5521: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: Wednesday, April 30th at 10:00pm
  - Posted test case due Tuesday, April 29th at 10:00pm

- Lecture Cancelled this Thursday - 4/24
  - Dr. Zdancewic will be out of town

- Final Exam:
  - According to registrar: Thursday, May 8th noon - 2:00pm
  - Coverage: emphasizes material since the midterm
  - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

LOOPS AND DOMINATORS

Loops in Control-flow Graphs

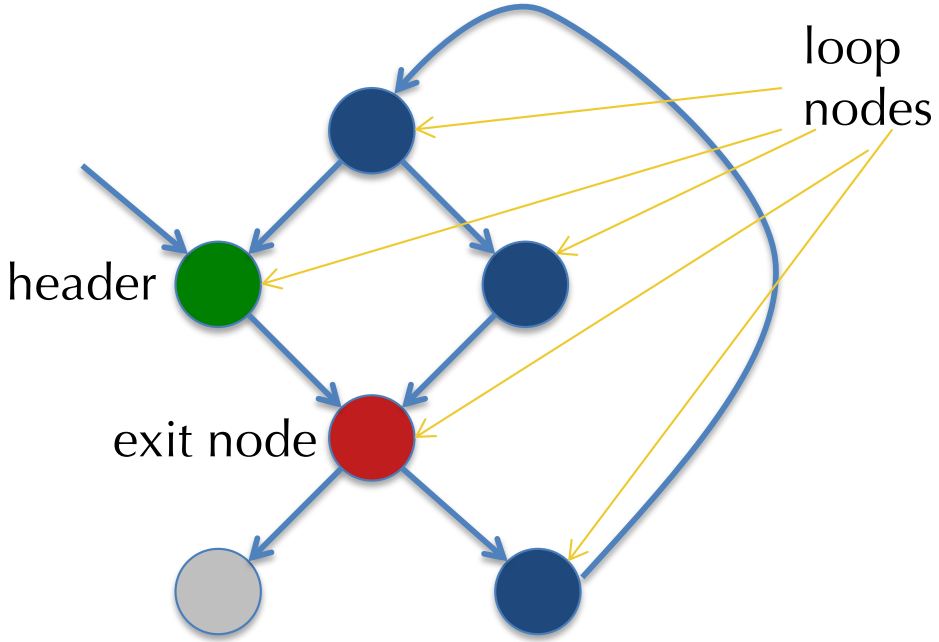
- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important
- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.
- Loops may be hard to recognize at the quadruple / LLVM IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…
- Problem: *How do we identify loops in the control-flow graph?*

Definition of a Loop

- A *loop* is a set of nodes in the control flow graph.
  - One distinguished entry point called the *header*

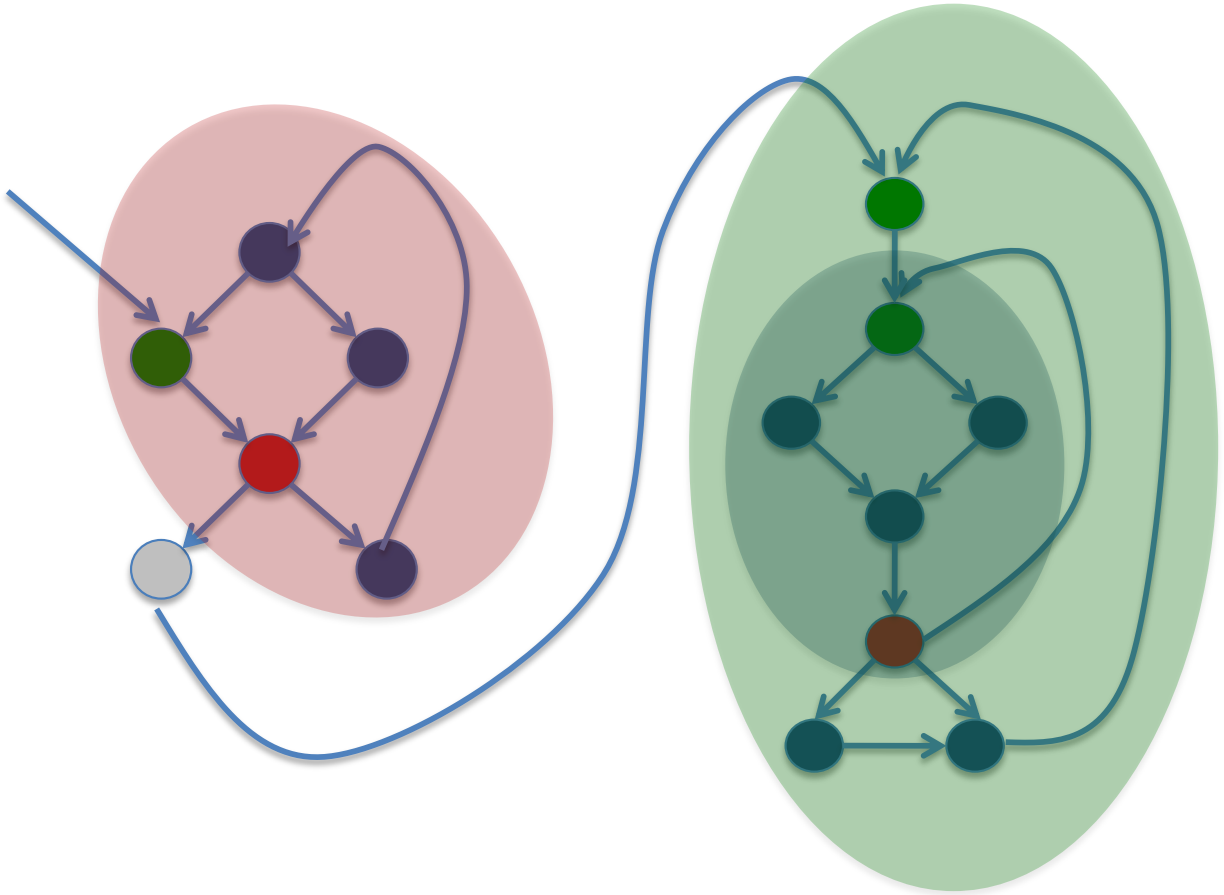
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*

- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes

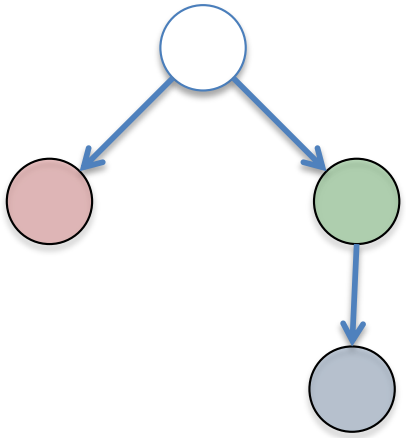


Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:



Control Tree:



The control tree depicts the nesting structure of the program.

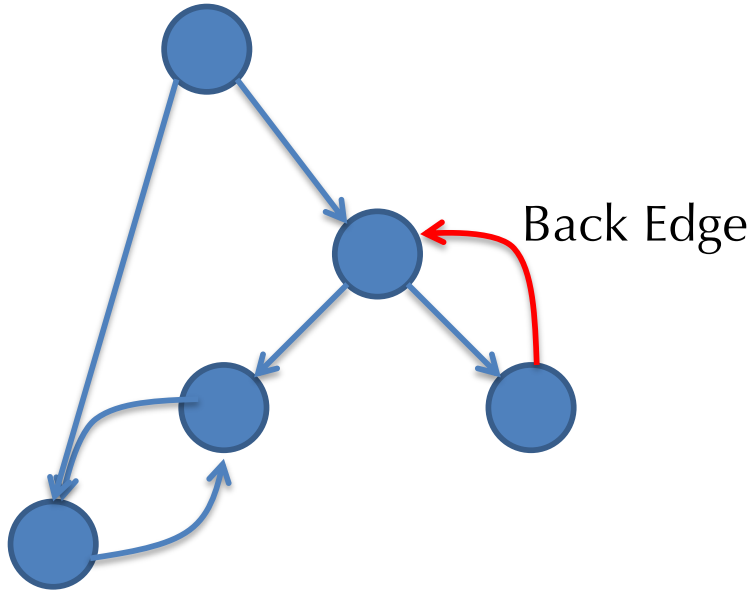
Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.

- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.

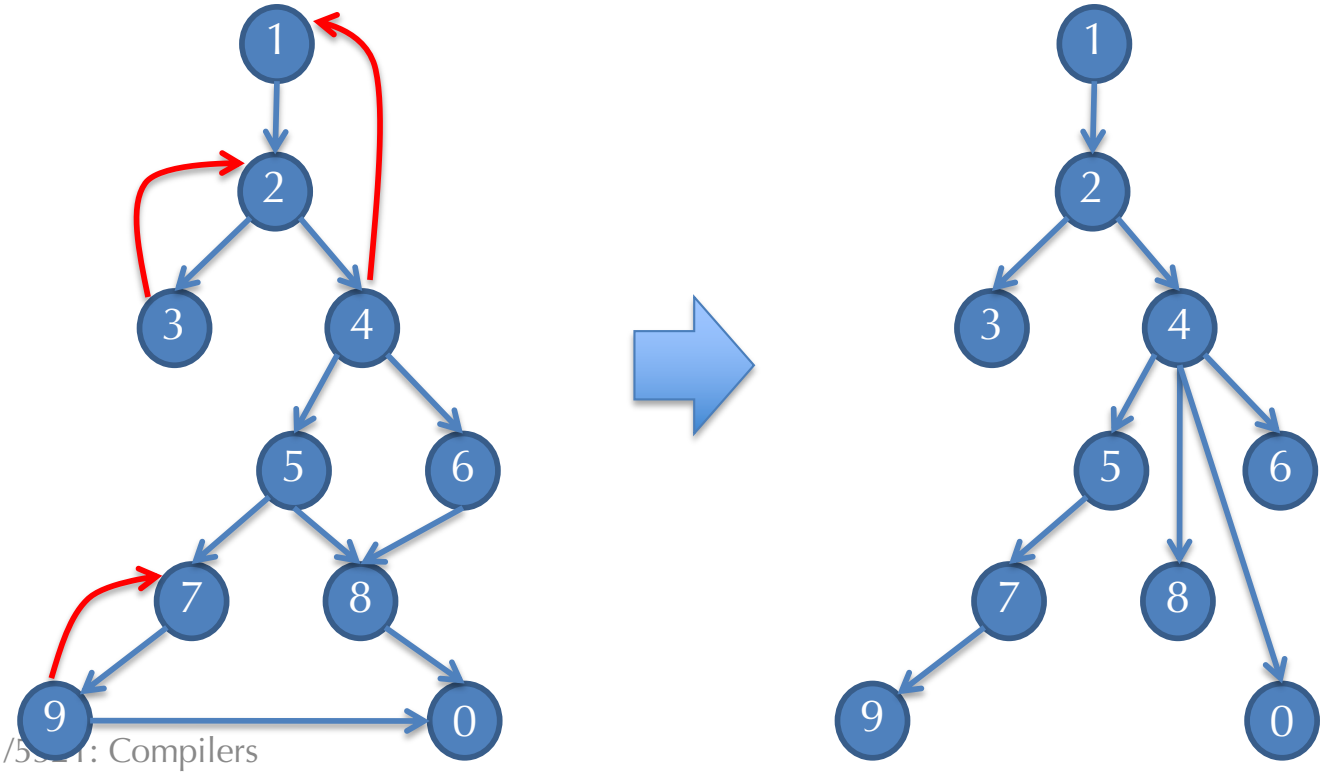
- An edge in the graph is a *back edge* if the target node dominates the source node.

- A loop contains at least one back edge.



Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation

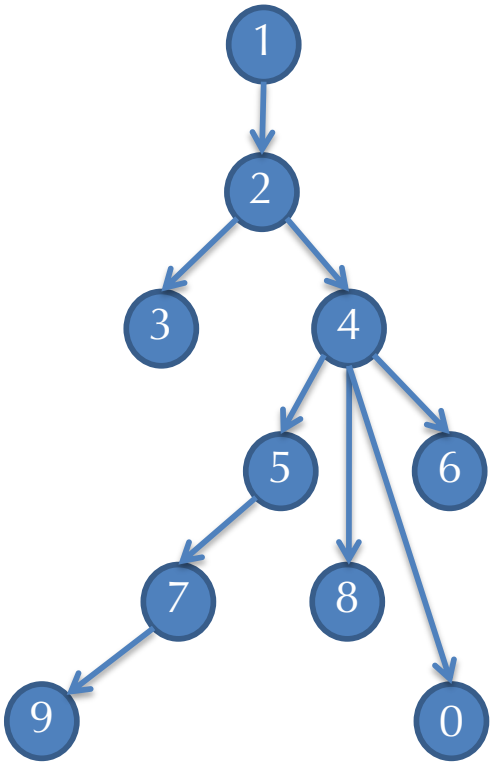


Dominator Dataflow Analysis

- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- "Every node dominates itself."
  - $out[n] := in[n] \cup \{n\}$
- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $\top = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ is $\cap$
- Easy to show monotonicity and that F_n distributes over meet.
  - So algorithm terminates and is MOP

Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - *e.g.*, Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes

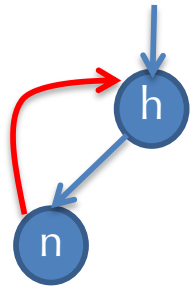


- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:
  - Dom[8] ∩Dom[7] = Dom[4]

- See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

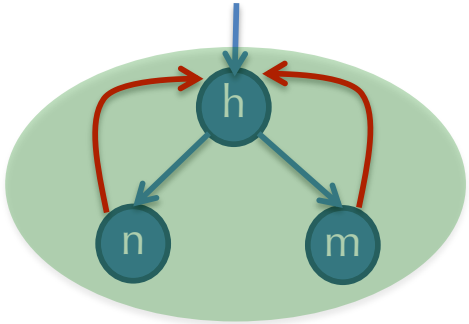
Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
  - Edge n → h where h dominates n
- Each back edge has a *natural loop*:
  - h is the header
  - All nodes reachable from h that also reach n without going through h



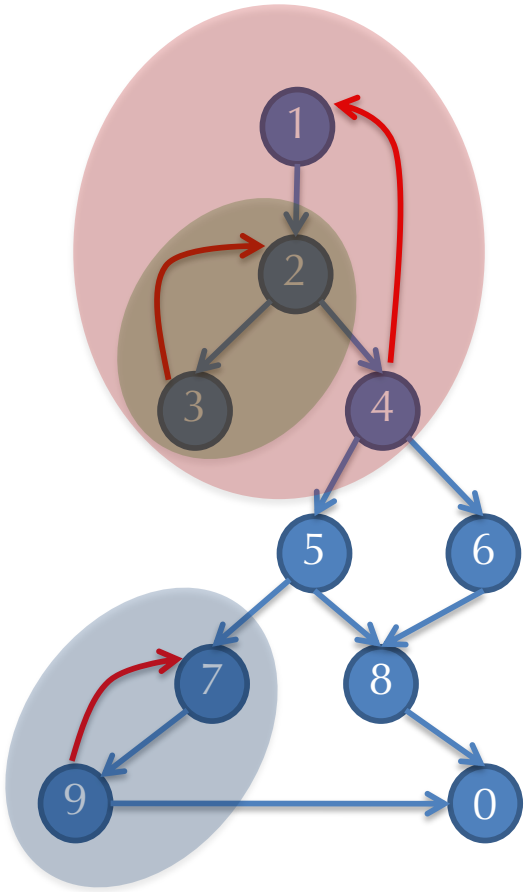
- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}

- Two loops may share the same header: merge them



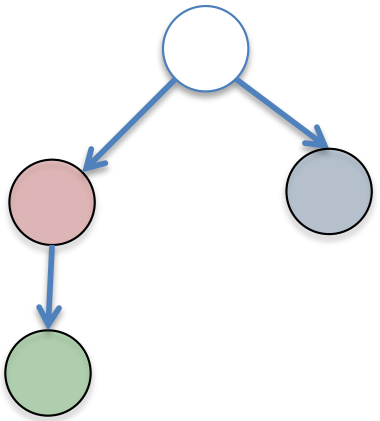
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

Example Natural Loops



Natural Loops

Control Tree:



The control tree depicts the nesting structure of the program.

Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
  - Deeply nested loops pay off the most for optimization.

- Need to know loop headers / back edges for doing
  - loop invariant code motion
  - loop unrolling

- Dominance information also plays a role in converting to SSA form
  - Used internally by LLVM to do register allocation.

Phi nodes

Alloc “promotion”

Register allocation

REVISITING SSA

Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation of backend: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.

Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

- Does this always work?

What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %z2 = %add i64 ???, 3
```

- What do we put for ???

Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.
    `%uid = phi <ty>` v_1, `<label`$_1$`>`, … , v_n, `<label`$_n$`>`

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```

Phi Nodes and Loops

- Importantly, the `%uids` on the right-hand side of a phi node can be defined "later" in the control-flow graph.
  - Means that `%uids` can hold values "around a loop"
  - Scope of `%uids` is defined by *dominance*

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```

Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i64           // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x      // store the pointer into the heap
```

```
entry:
  %x = alloca i64         // %x cannot be promoted
  %y = call foo(i64* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated

Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"

- Replace `loads/stores` to `alloc`'ed variables with freshly-generated `%uids`

- Eliminate the now unneeded `load/store/alloca` instructions.

Where to Place ϕ functions?

- Need to calculate the "Dominance Frontier"

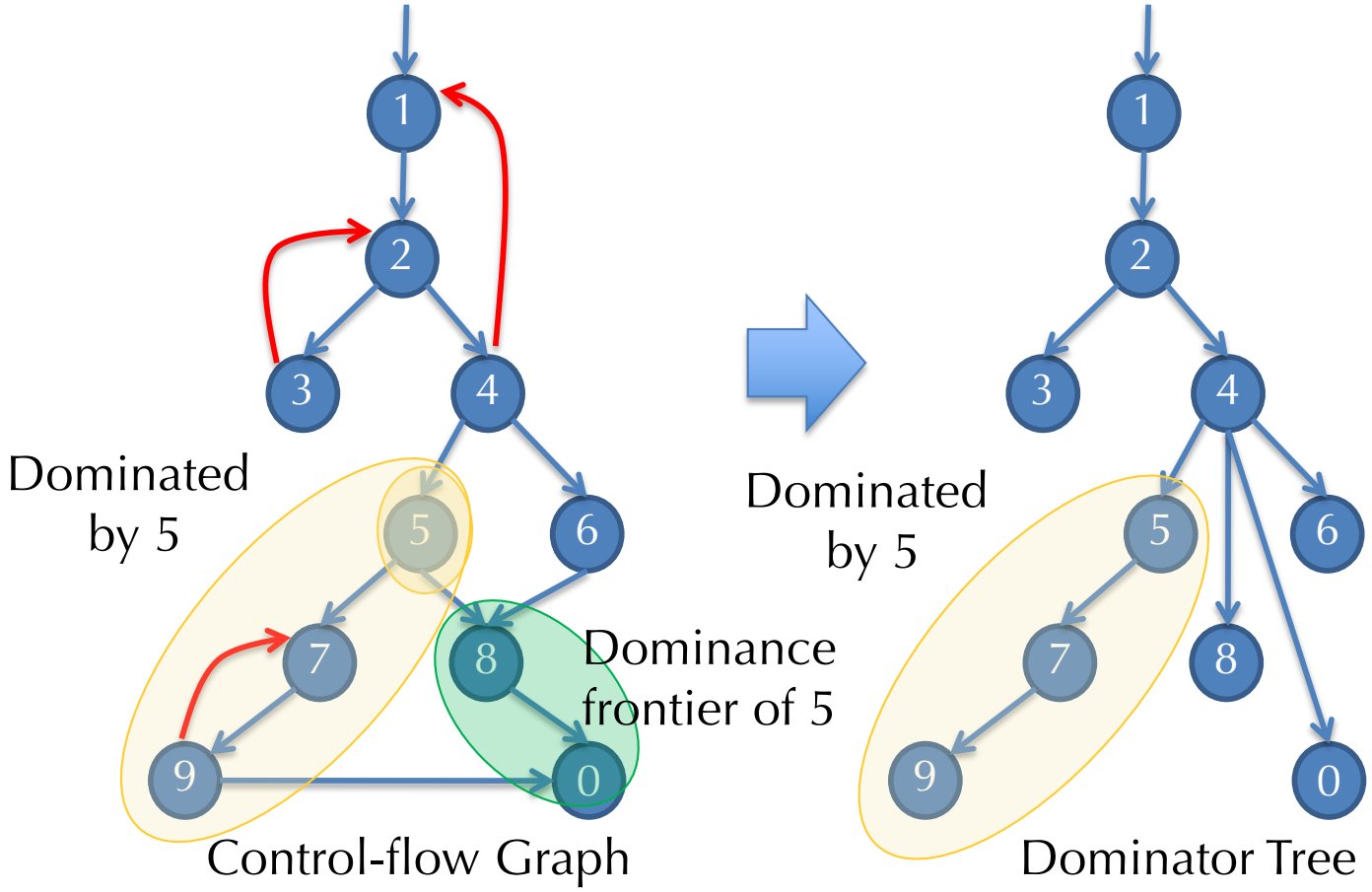
- Node A ***strictly dominates*** node B if A dominates B and A ≠ B.
  - Note: A does not strictly dominate B if A does not dominate B or A = B.

- The ***dominance frontier*** of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
  - Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

- Write DF[n] for the dominance frontier of node n.

Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {1}, DF[2] = {1,2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {7,0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}



Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)

- Adds each B to the DF sets to which it belongs

```
for all nodes B
    if #(pred[B]) ≥ 2                    // (just an optimization)
     for each p ∈pred[B] {
          runner := p                    // start at the predecessor of B
          while (runner ≠ doms[B])       // walk up the tree adding B
               DF[runner] := DF[runner] ∪ {B}
               runner := doms[runner]
    }
```

Insert ϕ at Join Points

- Lift the DF[n] to a set of nodes N in the obvious way:

$$DF[N] = \bigcup_{n \in N} DF[n]$$

- Suppose that at variable x is defined at a set of nodes N.

$$DF_0[N] = DF[N]$$
$$DF_{i+1}[N] = DF[DF_i[N] \cup N]$$

Let J[N] be the *least fixed point* of the sequence:

$$DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$$

That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$

  - J[N] is called the "join points" for the set N
- We insert ϕ functions for the variable x at each node in J[N].
  - x = ϕ(x, x, …, x); (one "x" argument for each predecessor of the node)
  - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver.

- Intuition:
  - If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too…

Example Join-point Calculation

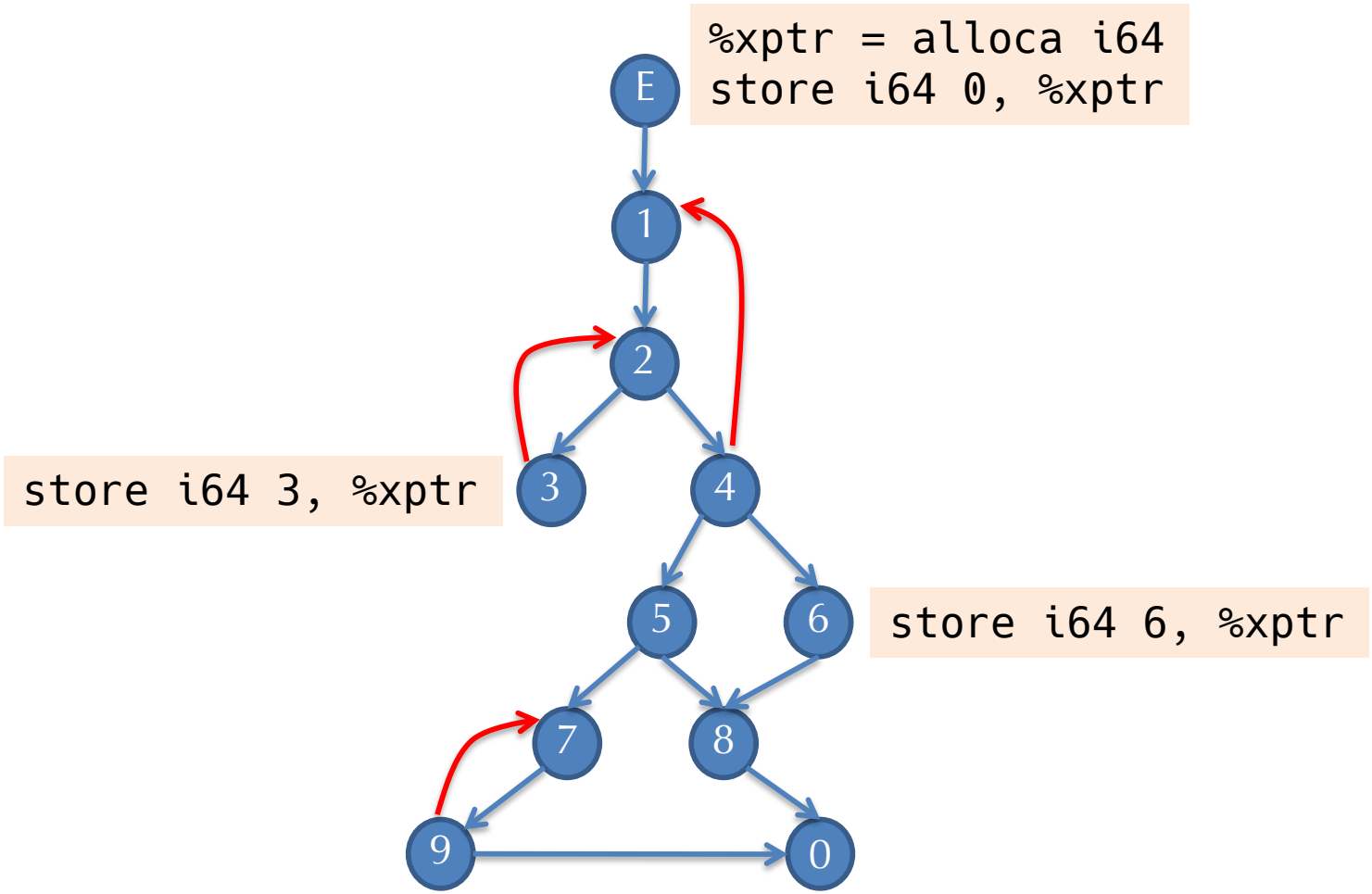
- Suppose the variable x is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- $DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\}$
- $DF_1[\{3,6\}]$
  $= DF[DF_0\{3,6\} \cup \{3,6\}]$
  $= DF[\{2,3,6,8\}]$
  $= DF[2] \cup DF[3] \cup DF[6] \cup DF[8]$
  $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $DF_2[\{3,6\}]$
  $= \ldots$
  $= \{1,2,8,0\}$

- So $J[\{3,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.

Join Points Pictorially

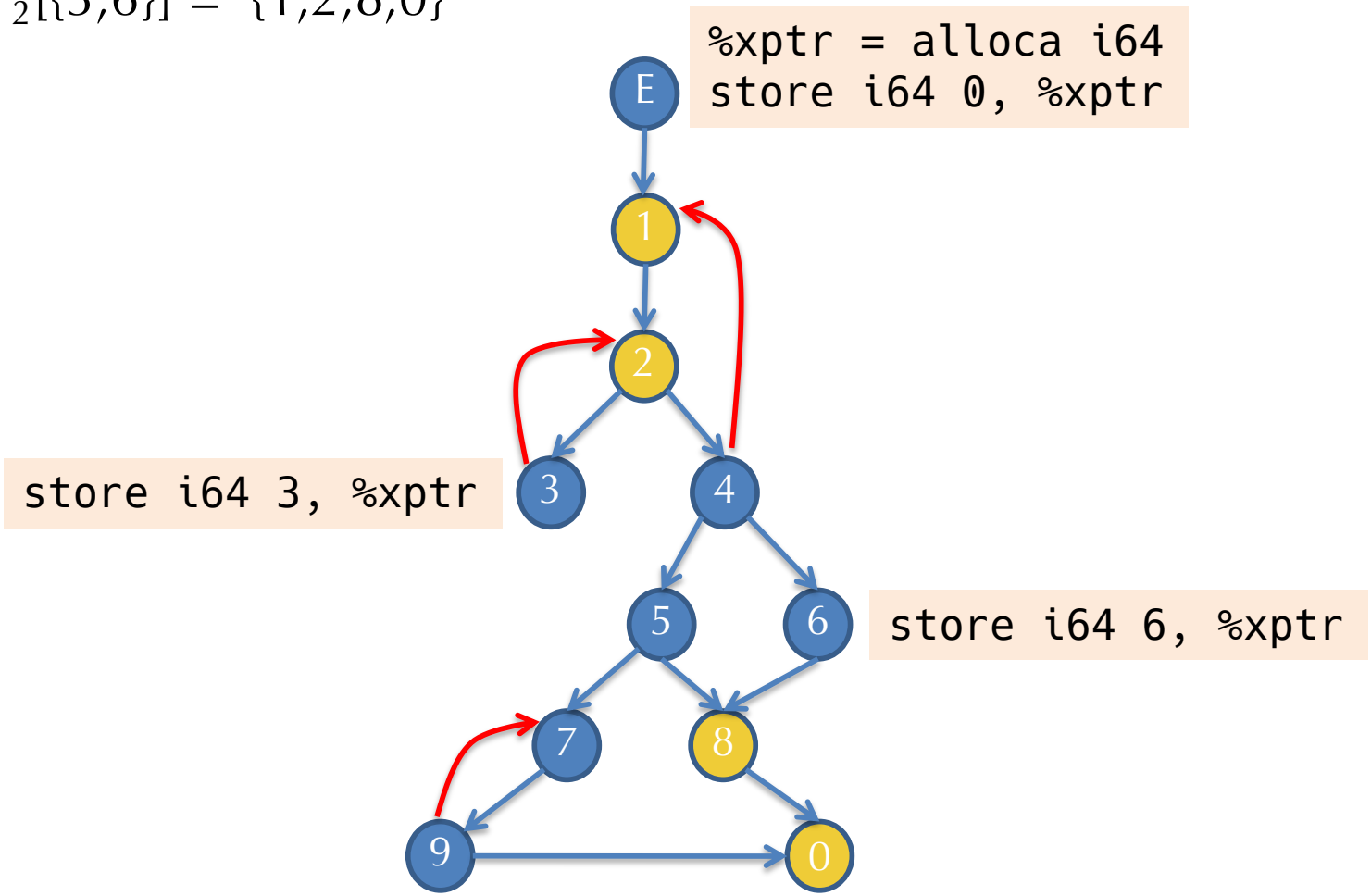
- Suppose variable x is modified at nodes {3, 6}
  - frontend compiles variable x to "xptr" and uses alloca (in entry node E)



Control Flow Graph

Join Points Pictorially

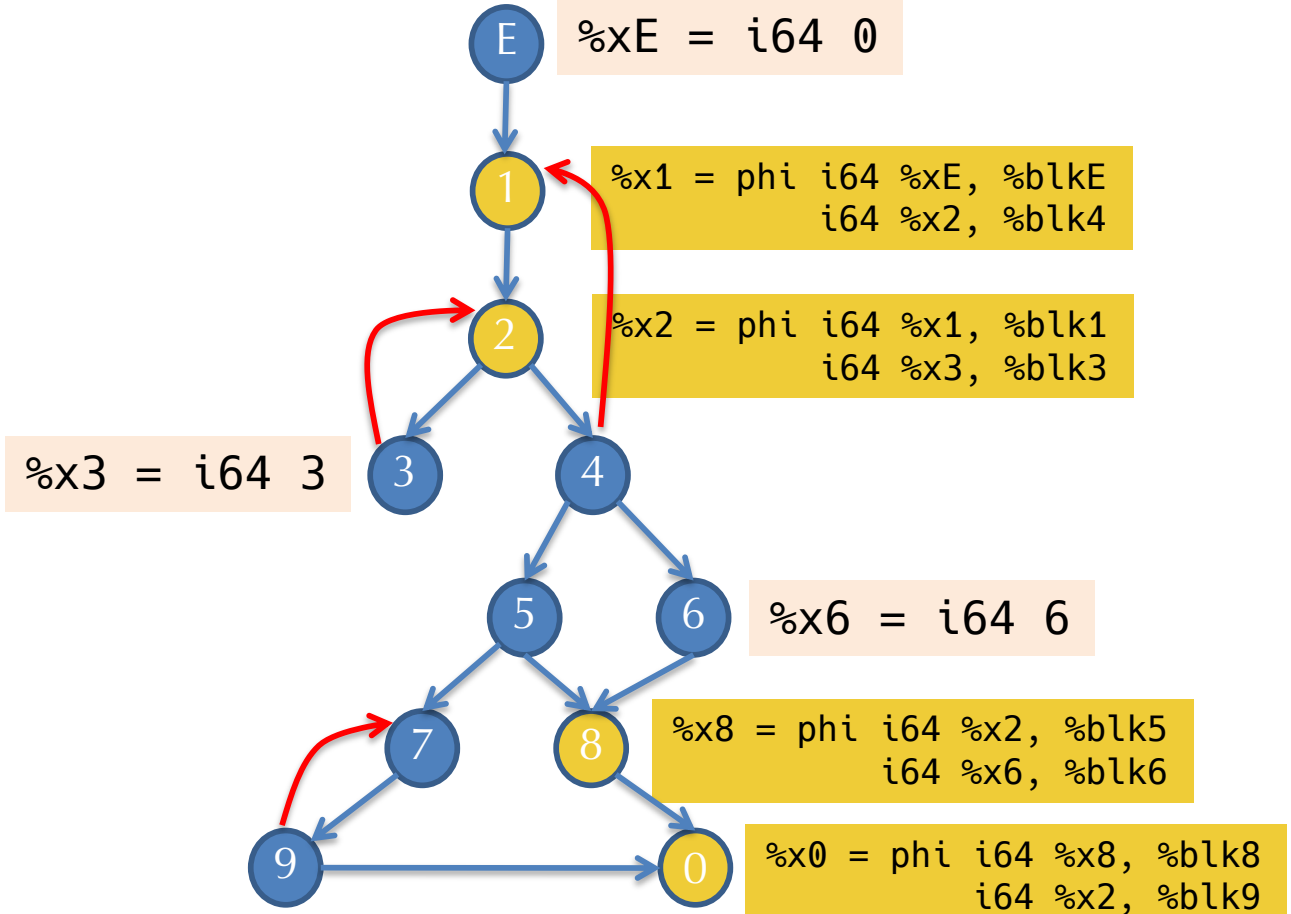
- Suppose variable x is modified at nodes {3, 6}
  - the frontend compiles variable x to "xptr" and uses alloca (in node 0)
- $DF_2[\{3,6\}] = \{1,2,8,0\}$



Control Flow Graph

Rename & Insert Phi Nodes

- Suppose variable x is modified at nodes {3, 6}
  - the frontend compiles variable x to "xptr" and uses alloca (in node 0)
- $DF_2[\{3,6\}] = \{1,2,8,0\}$

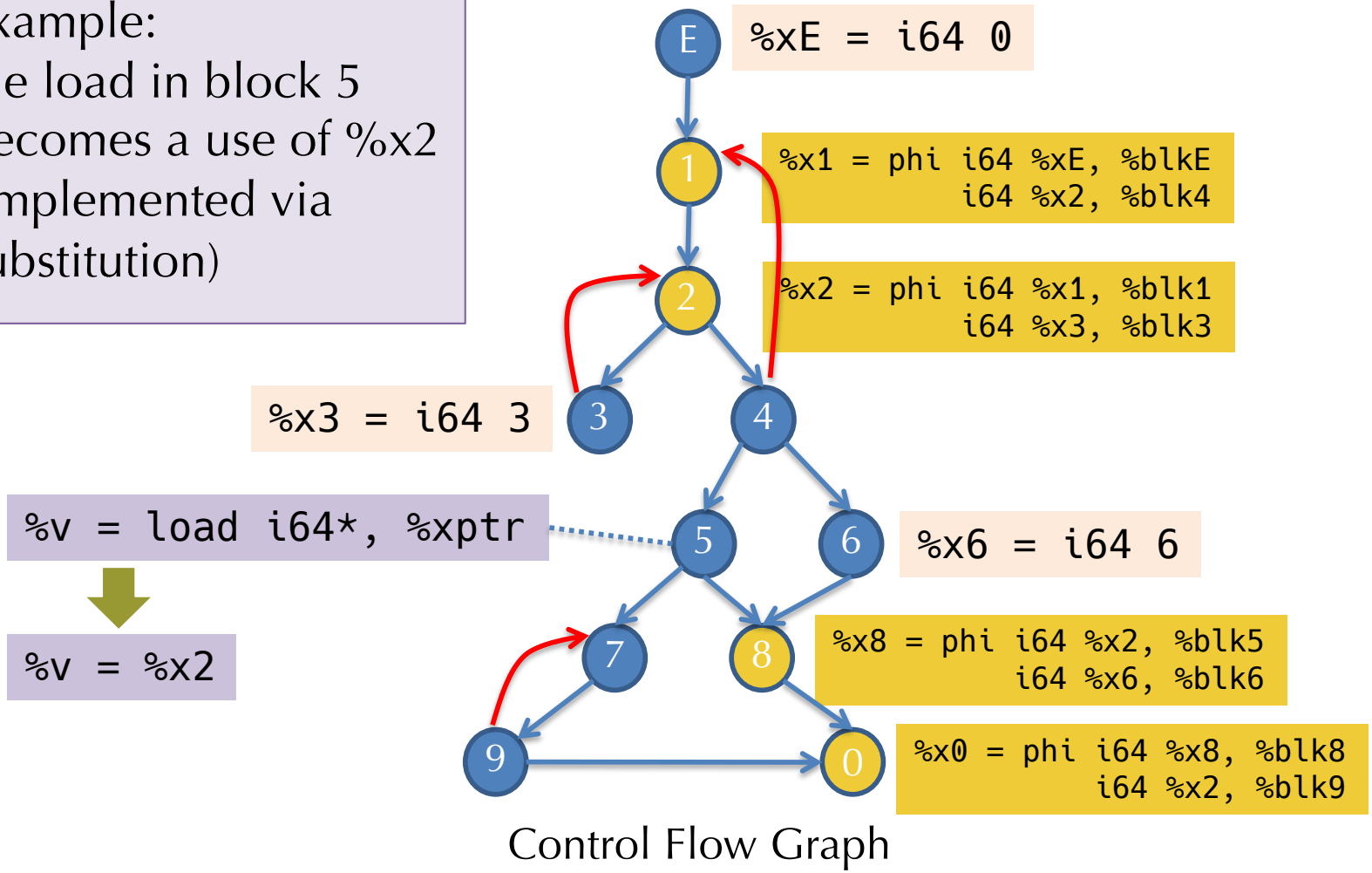


Control Flow Graph

Rename & Insert Phi Nodes

- Loads of the original x variable become uses of the renamed version in scope (i.e. the definition that dominates the use)

Example:
the load in block 5 becomes a use of %x2 (implemented via substitution)



Control Flow Graph

Phi Placement Alternative

- Less efficient, but easier to understand:
- Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)
- If all values flowing into phi node are the same, then eliminate it:

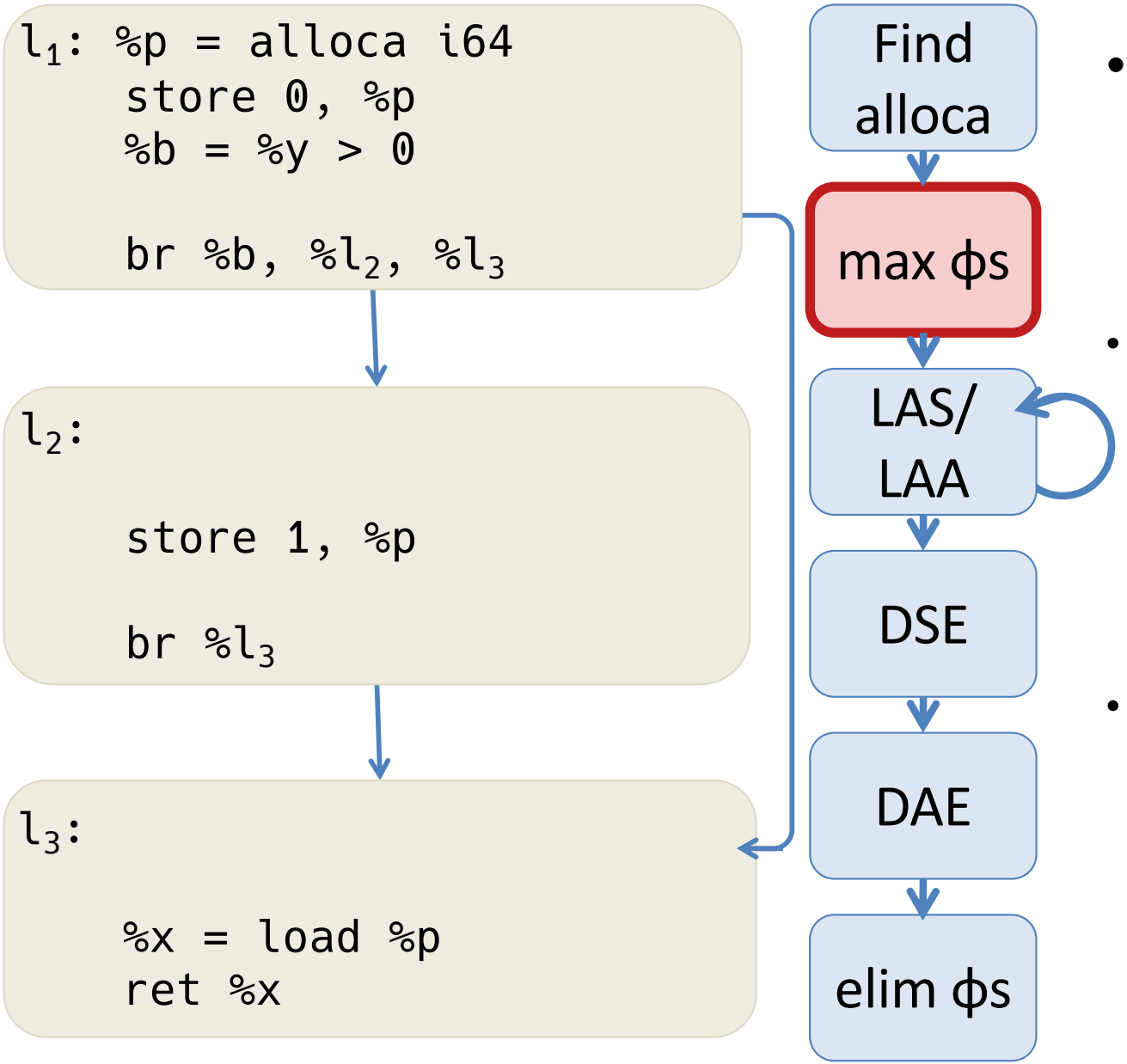
```
%x = phi   t %y, %pred1   t %y  %pred2  … t %y %predK
// code that uses %x
⇒
// code with %x replaced by %y
```

- Interleave with other optimizations
  - copy propagation
  - constant propagation
  - etc.

Legend of "simple" optimizations*:
LAS = load after store
LAA = load after alloca
DSE = dead store elimination
DAE = dead alloca elimination

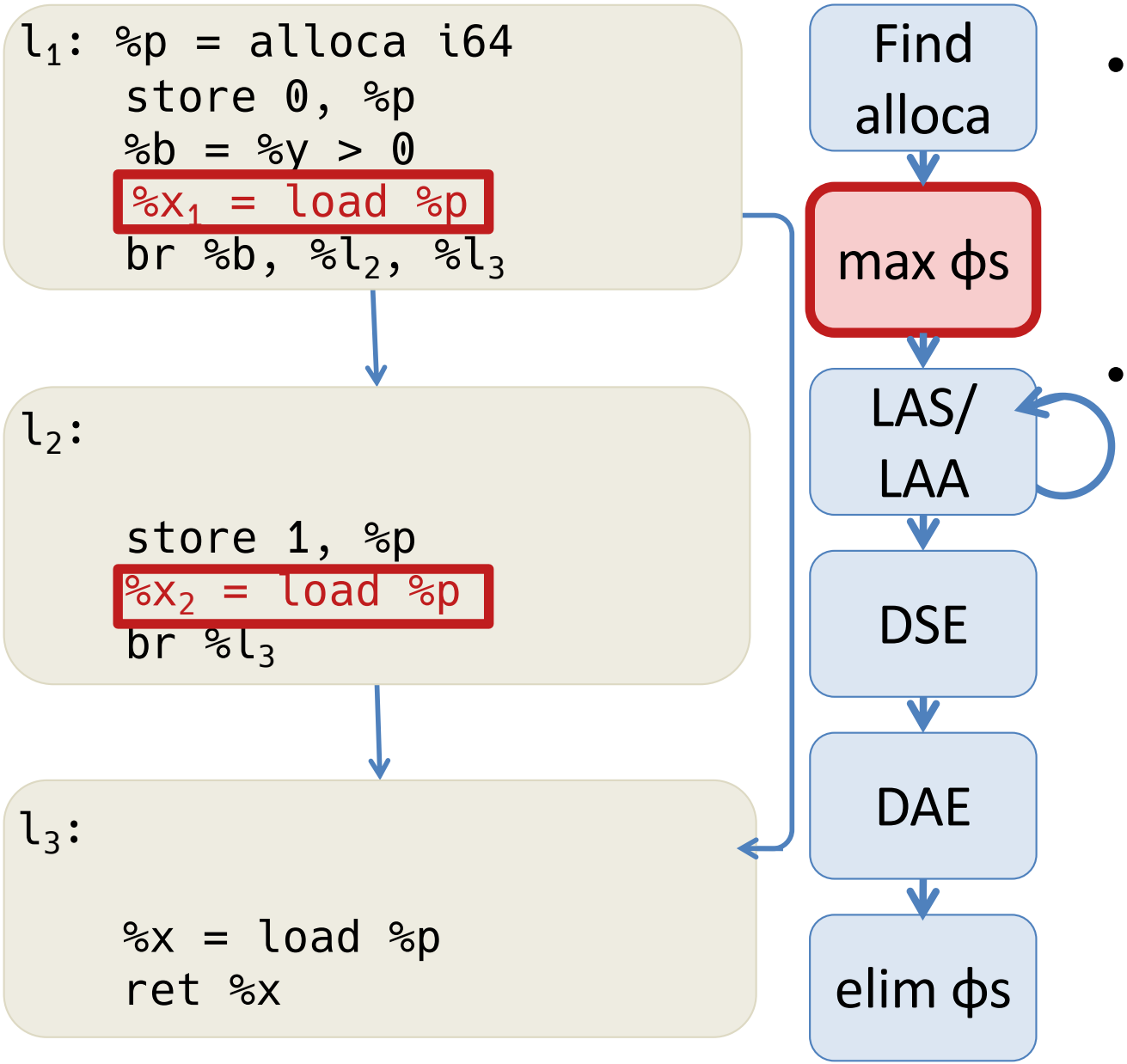
*nomenclature taken from LLVM IR passes

Example SSA Optimizations



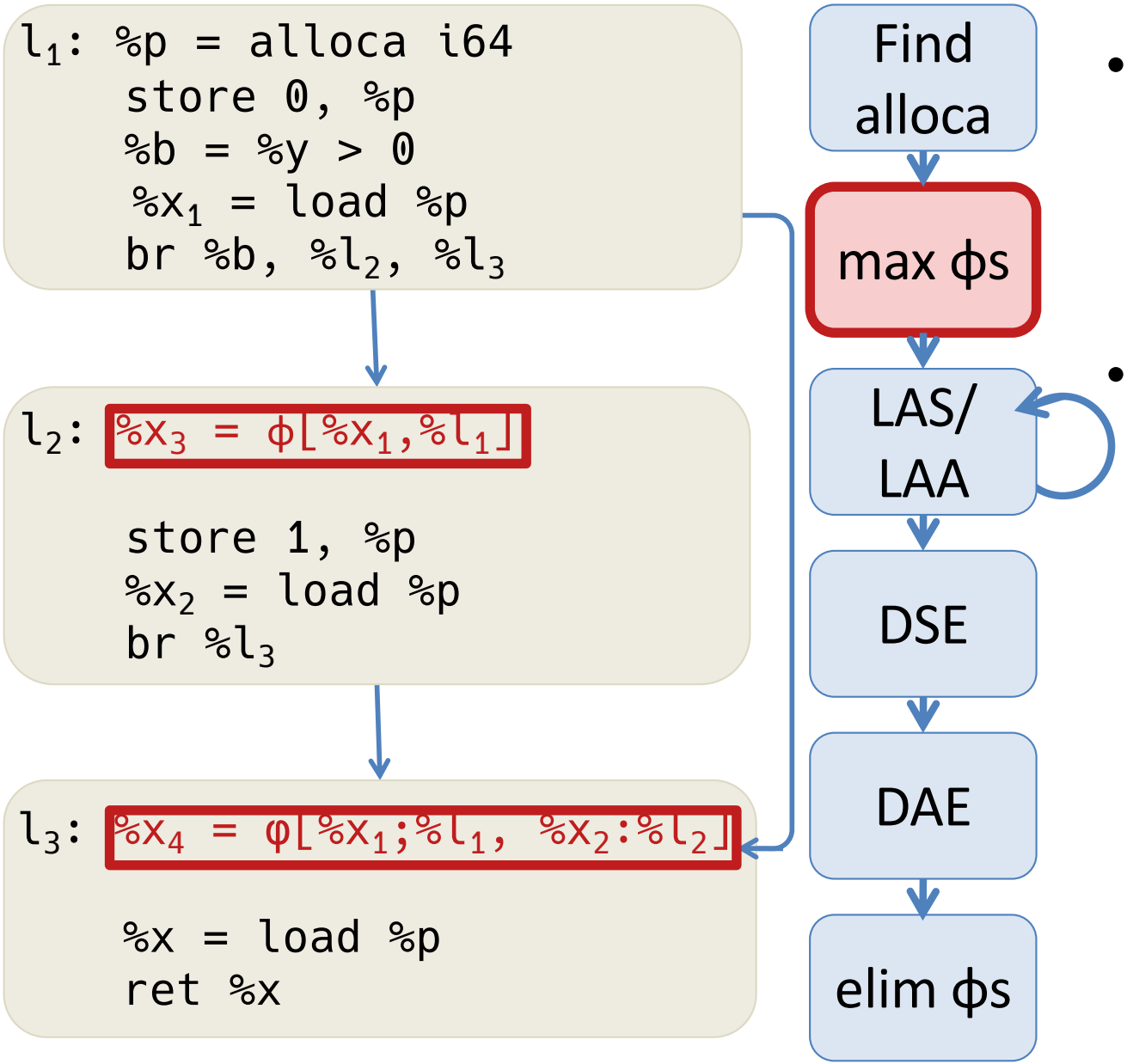
- How to place phi nodes without breaking SSA?
- Note: the "real" implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

Example SSA Optimizations



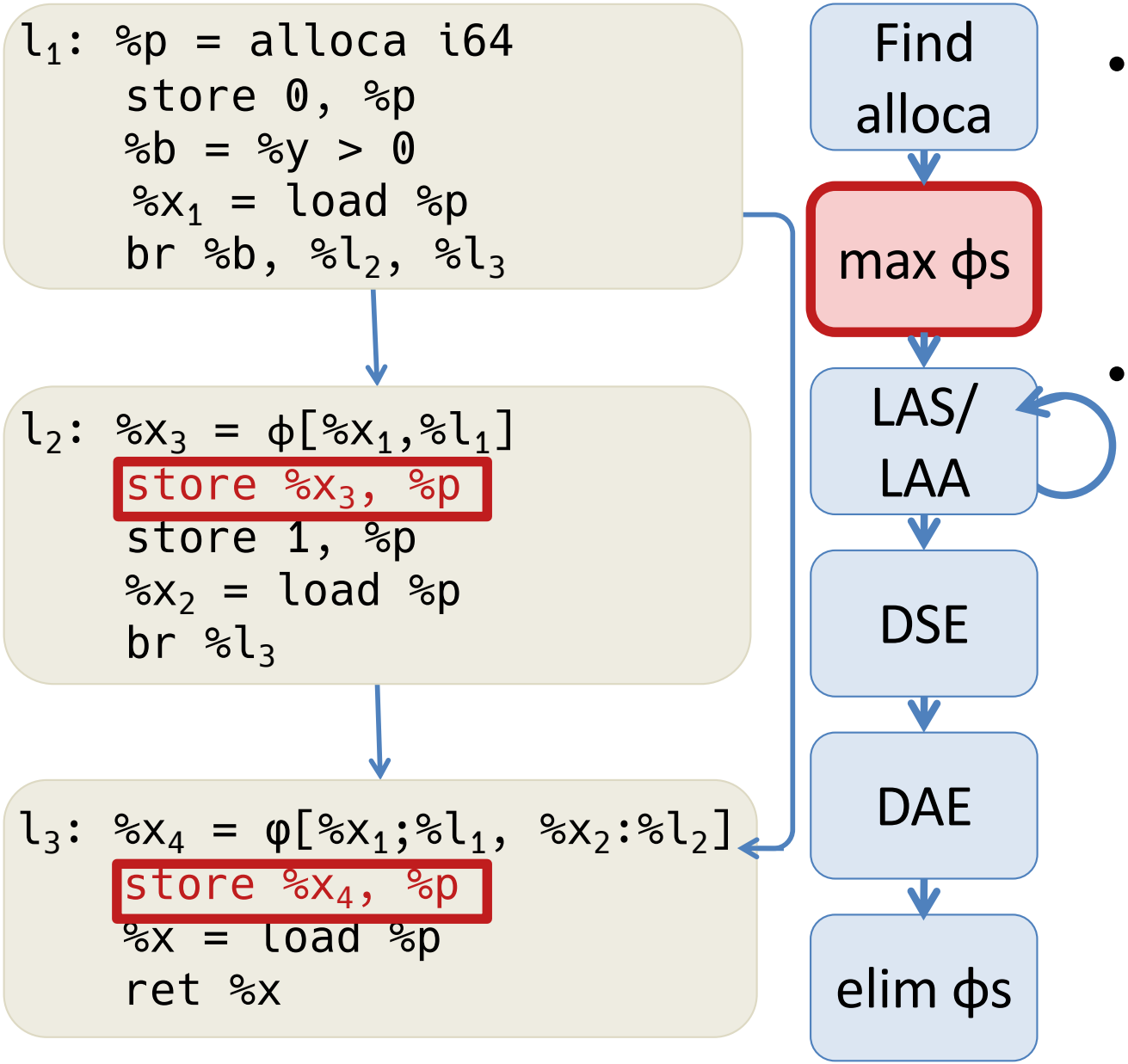
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example SSA Optimizations



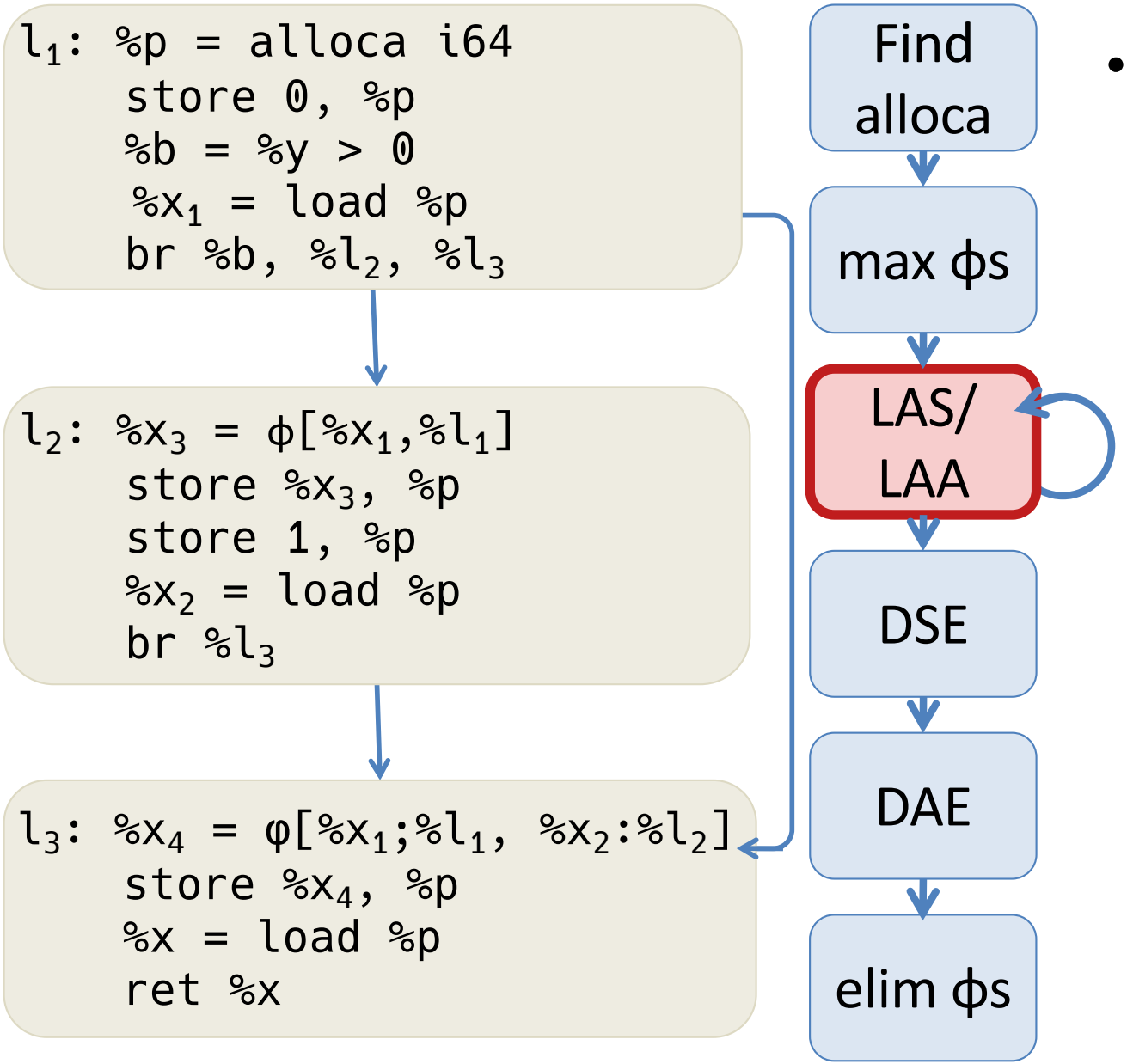
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block

Example SSA Optimizations



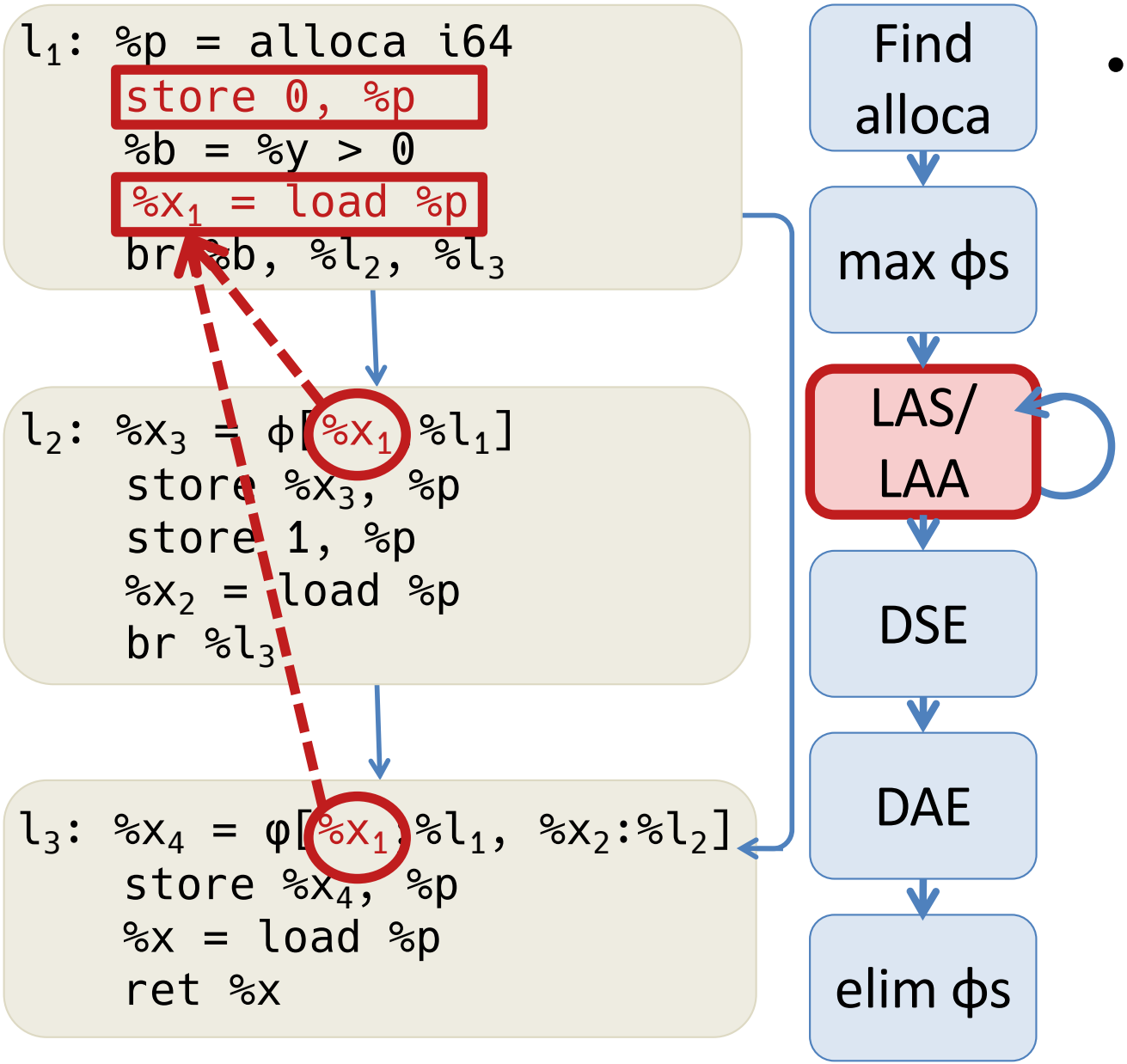
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

Example SSA Optimizations



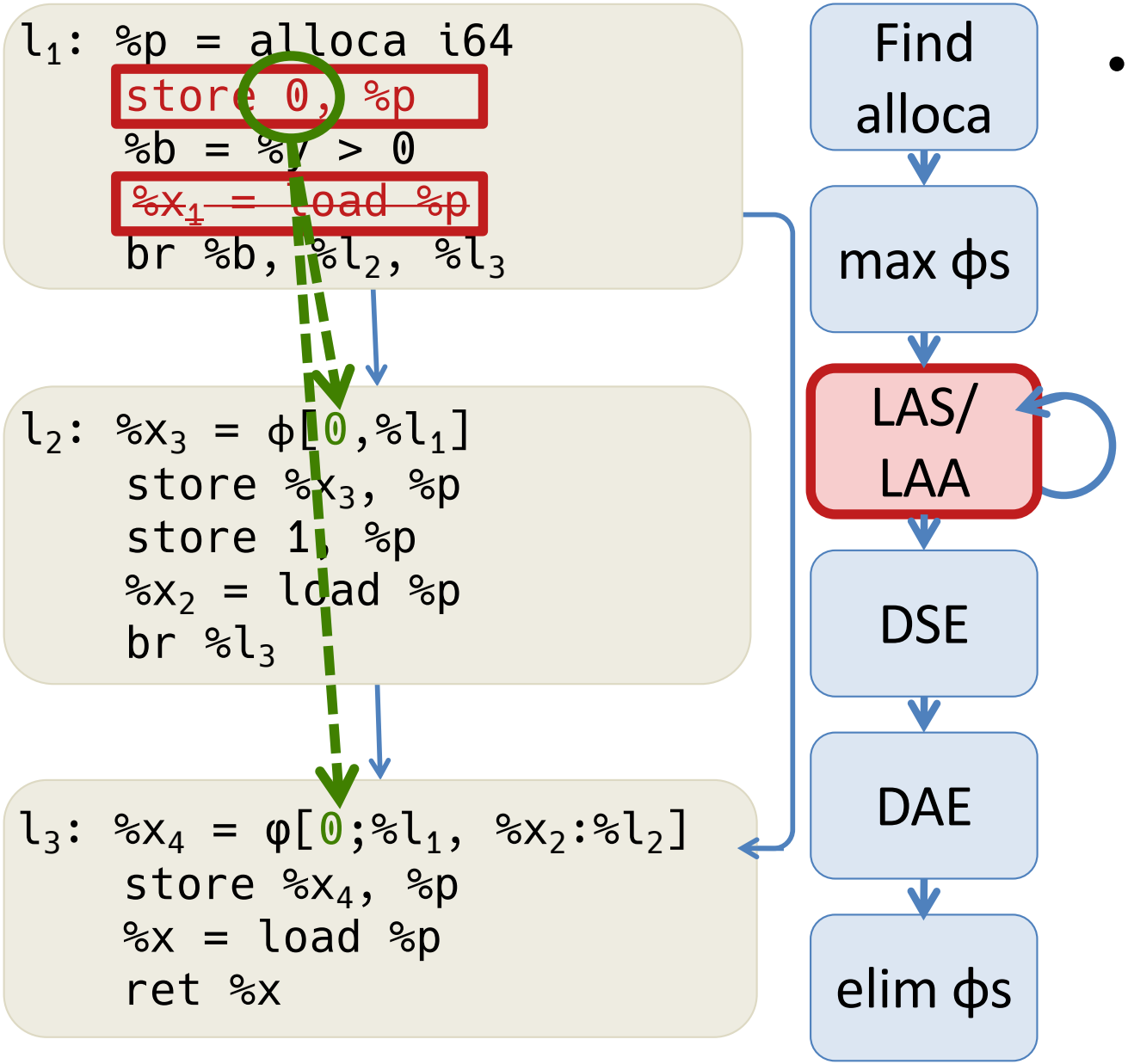
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



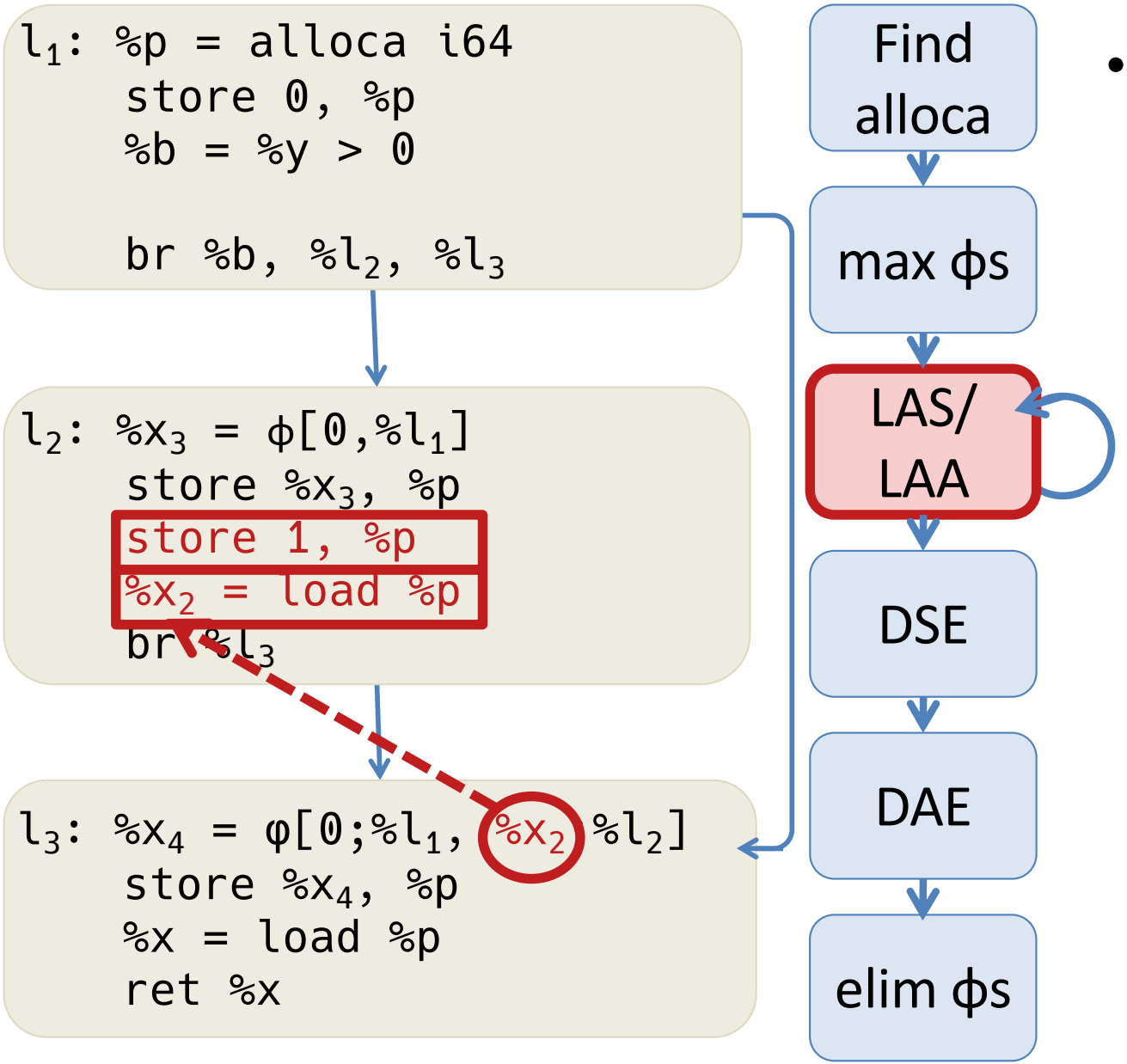
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



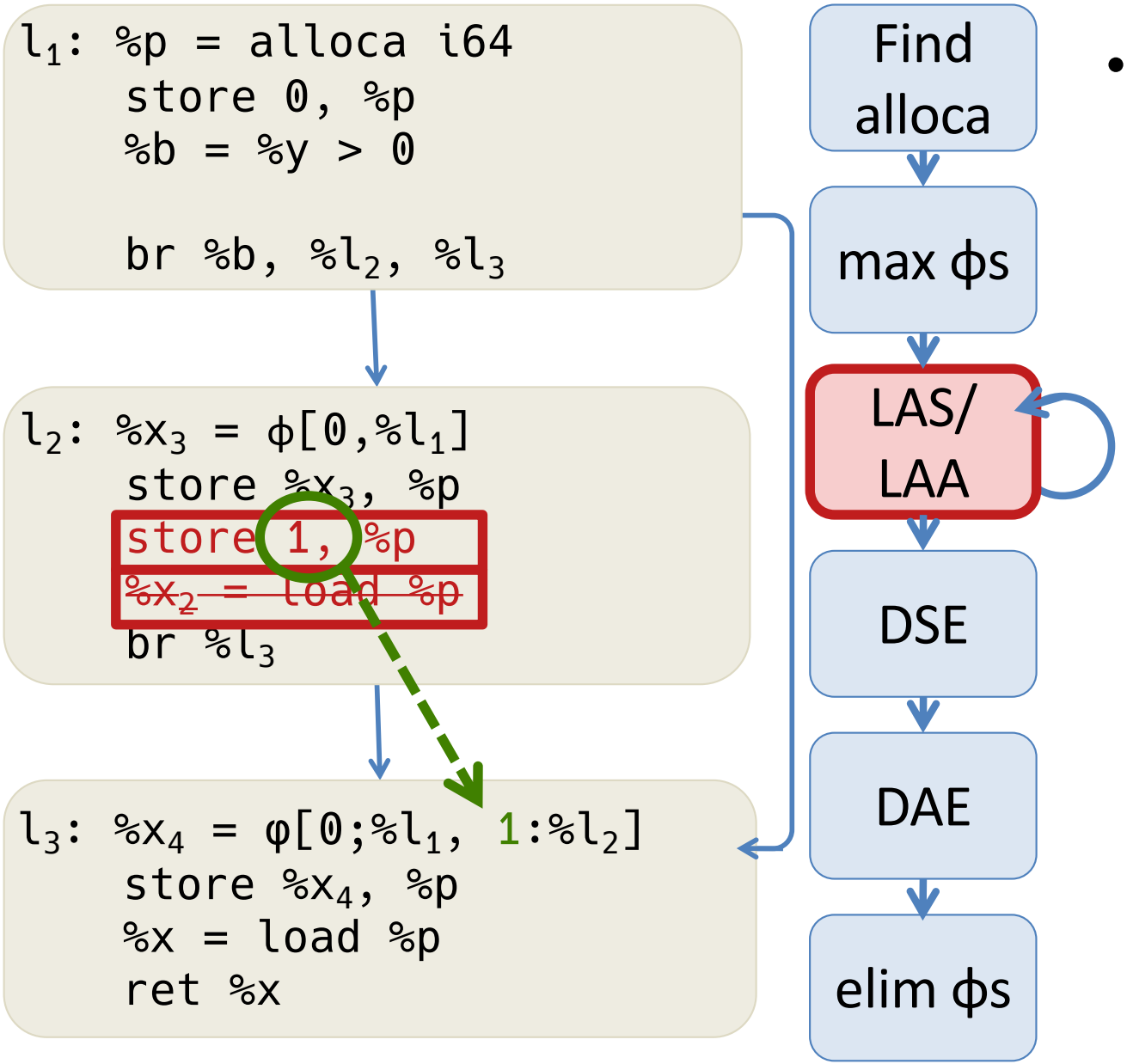
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



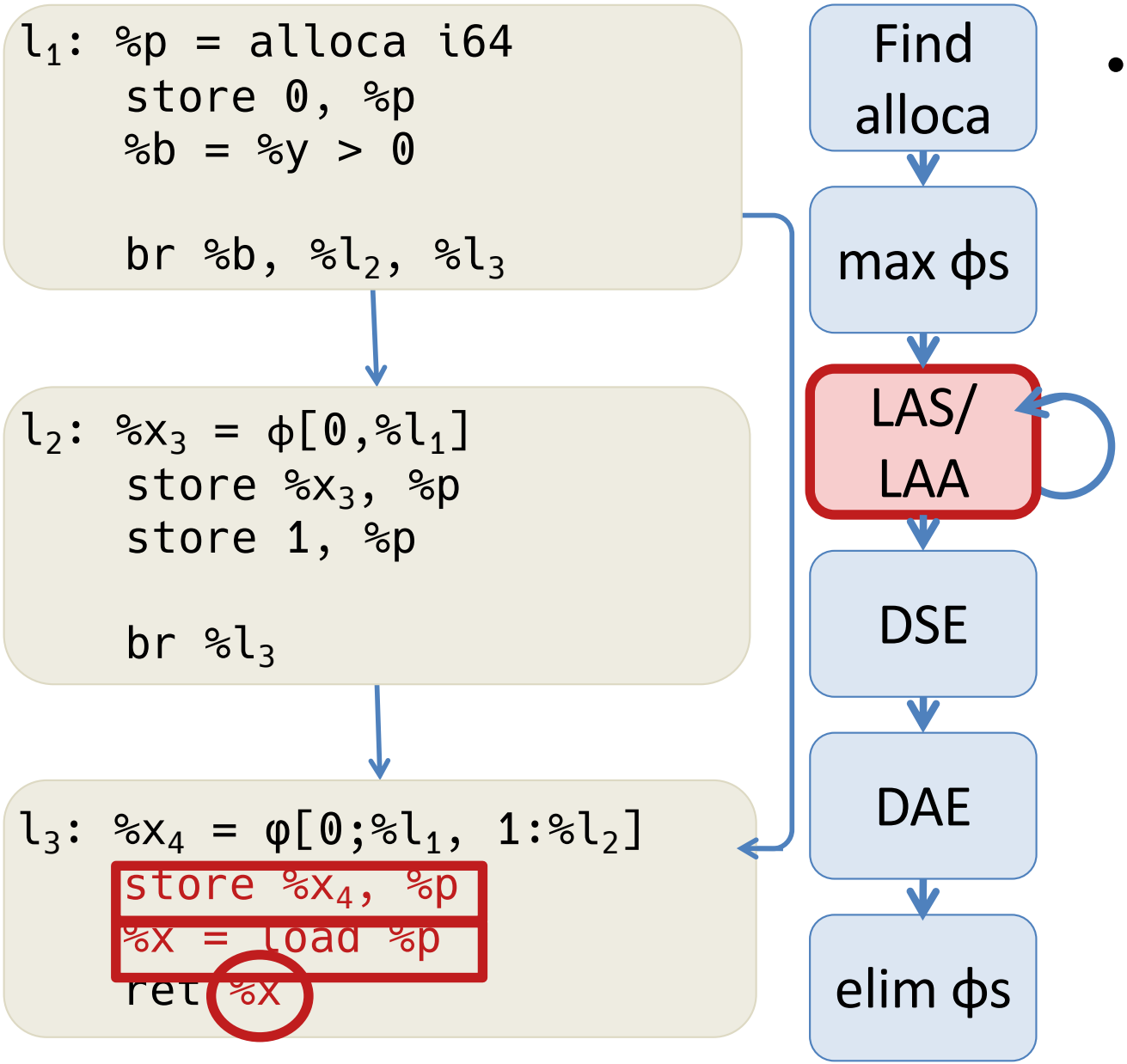
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



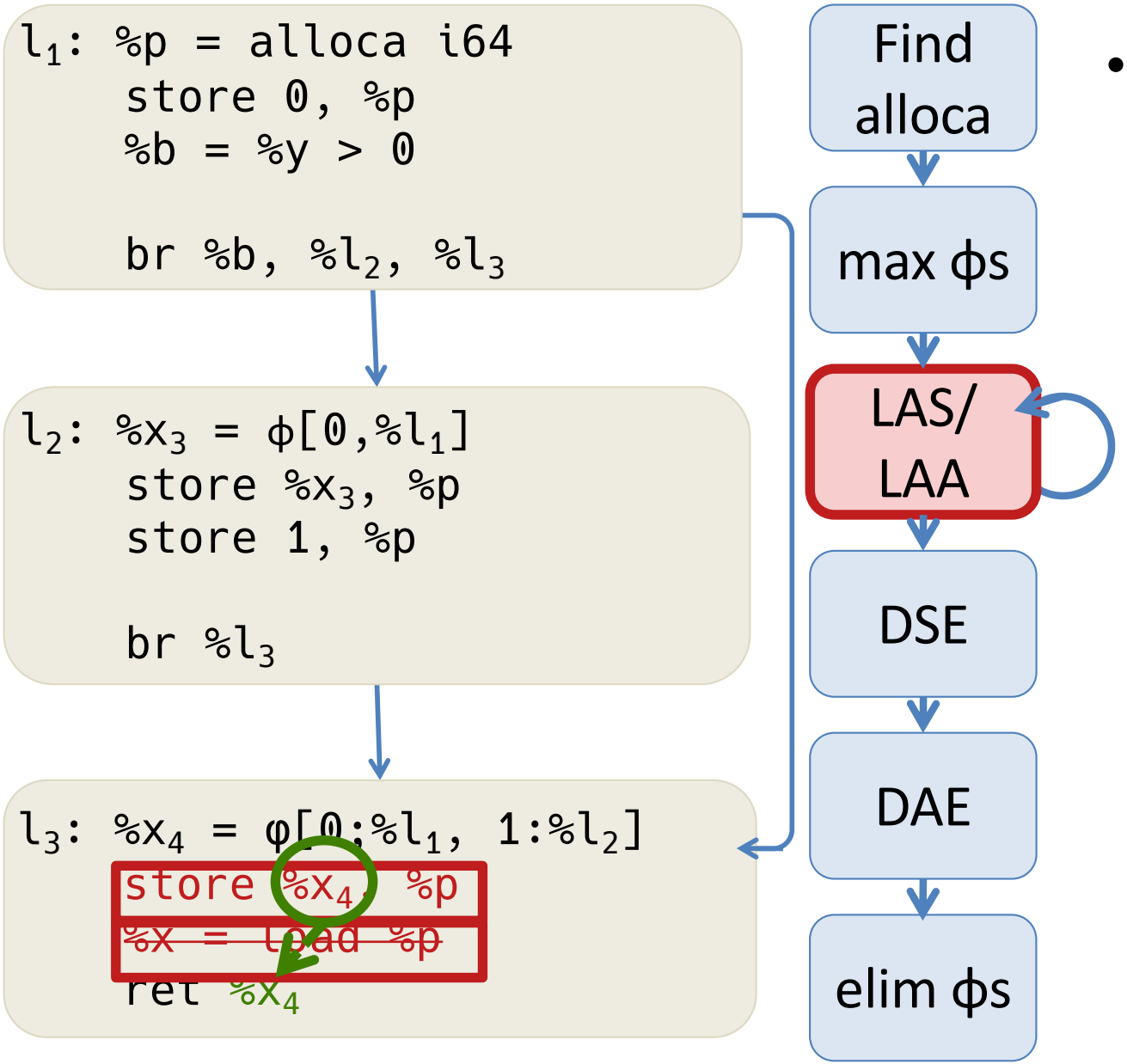
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



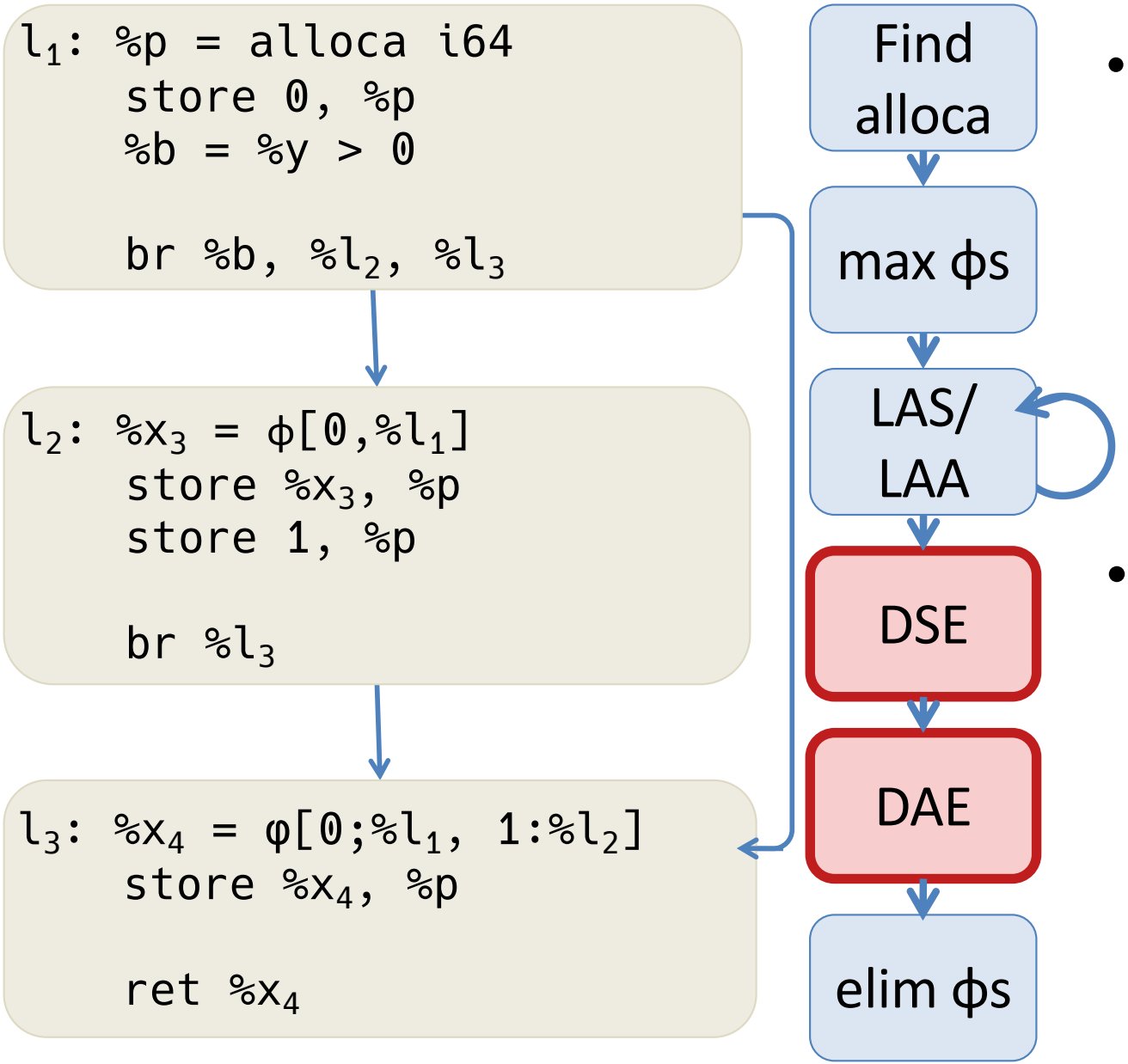
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



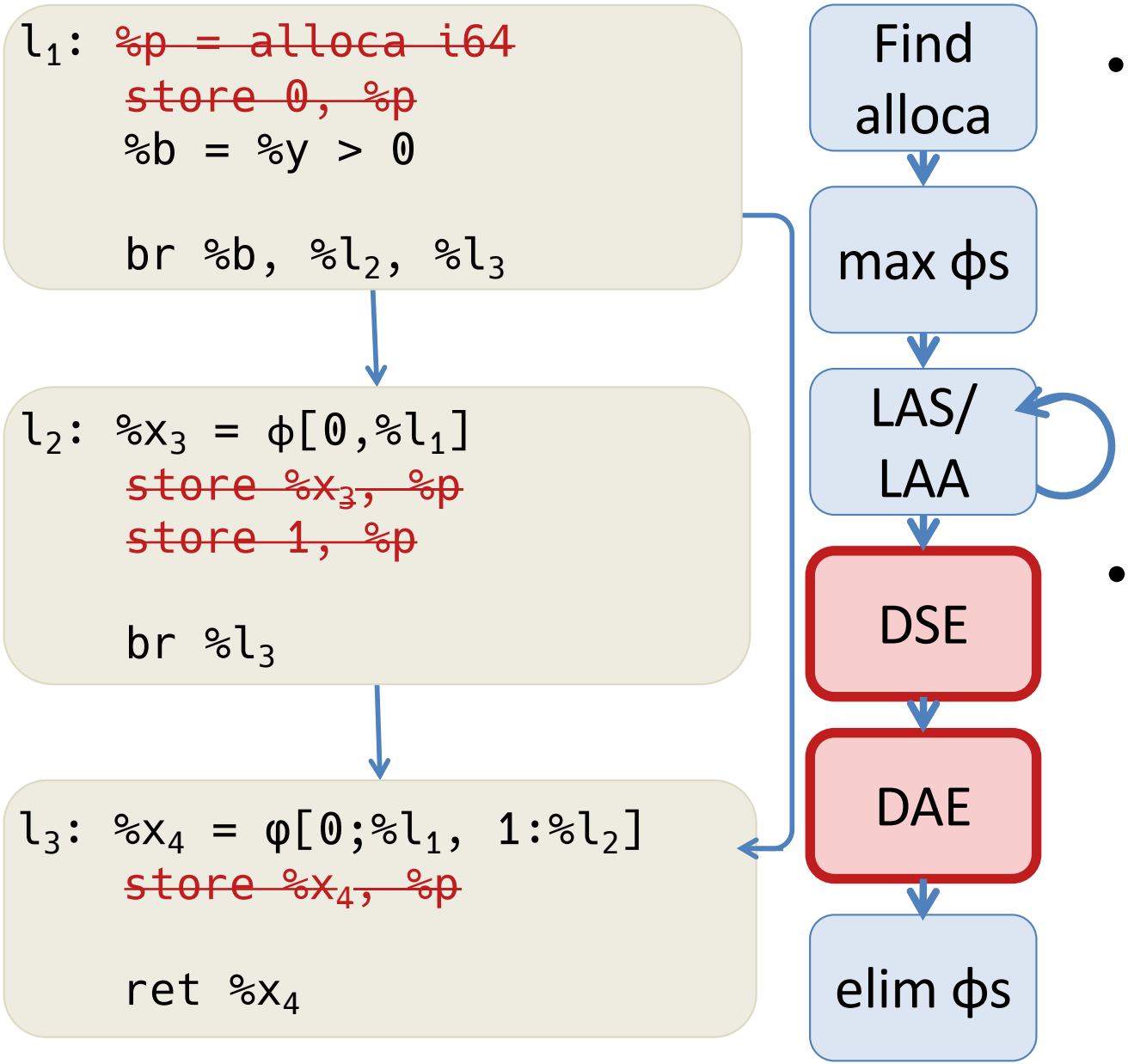
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



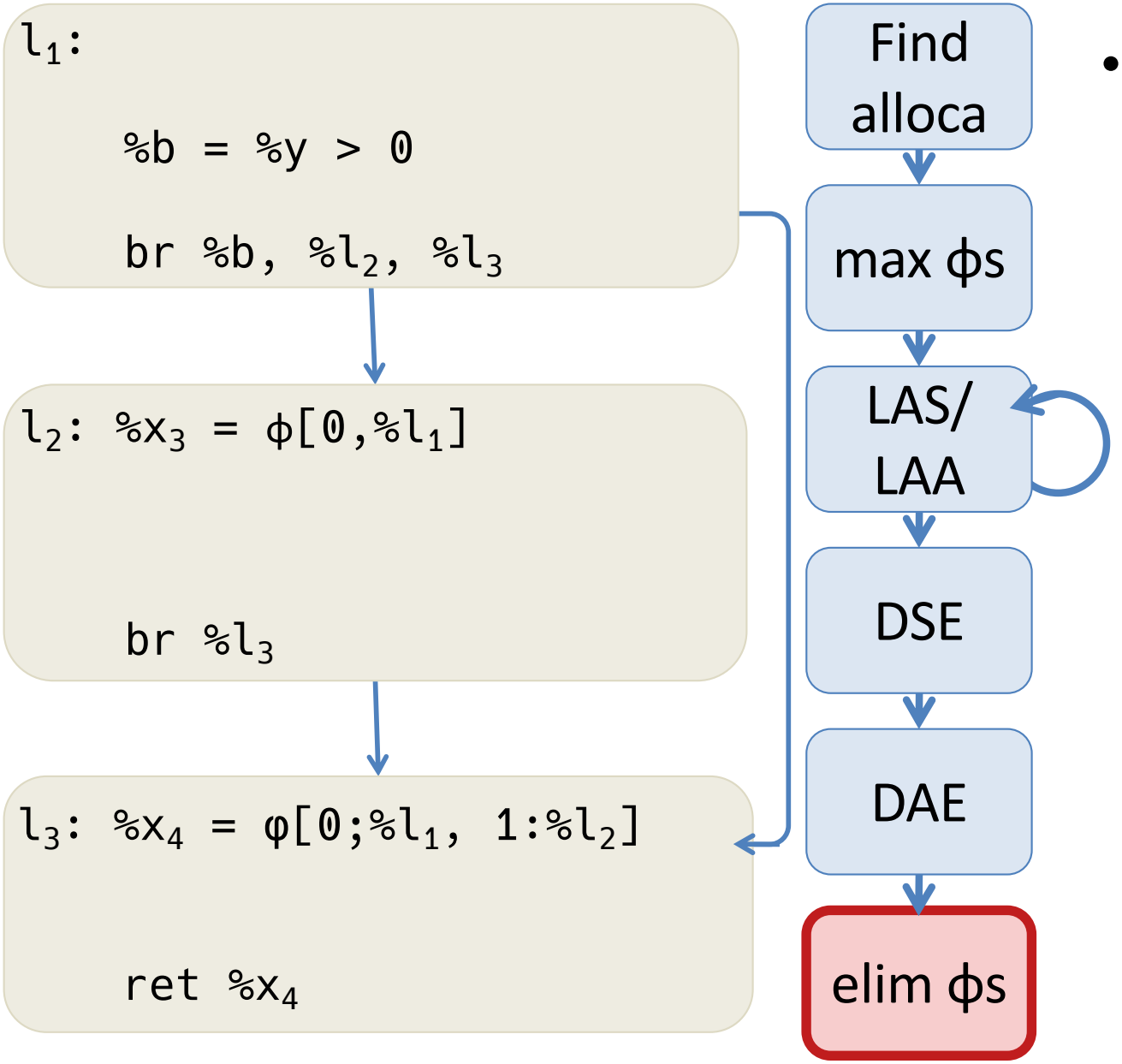
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations



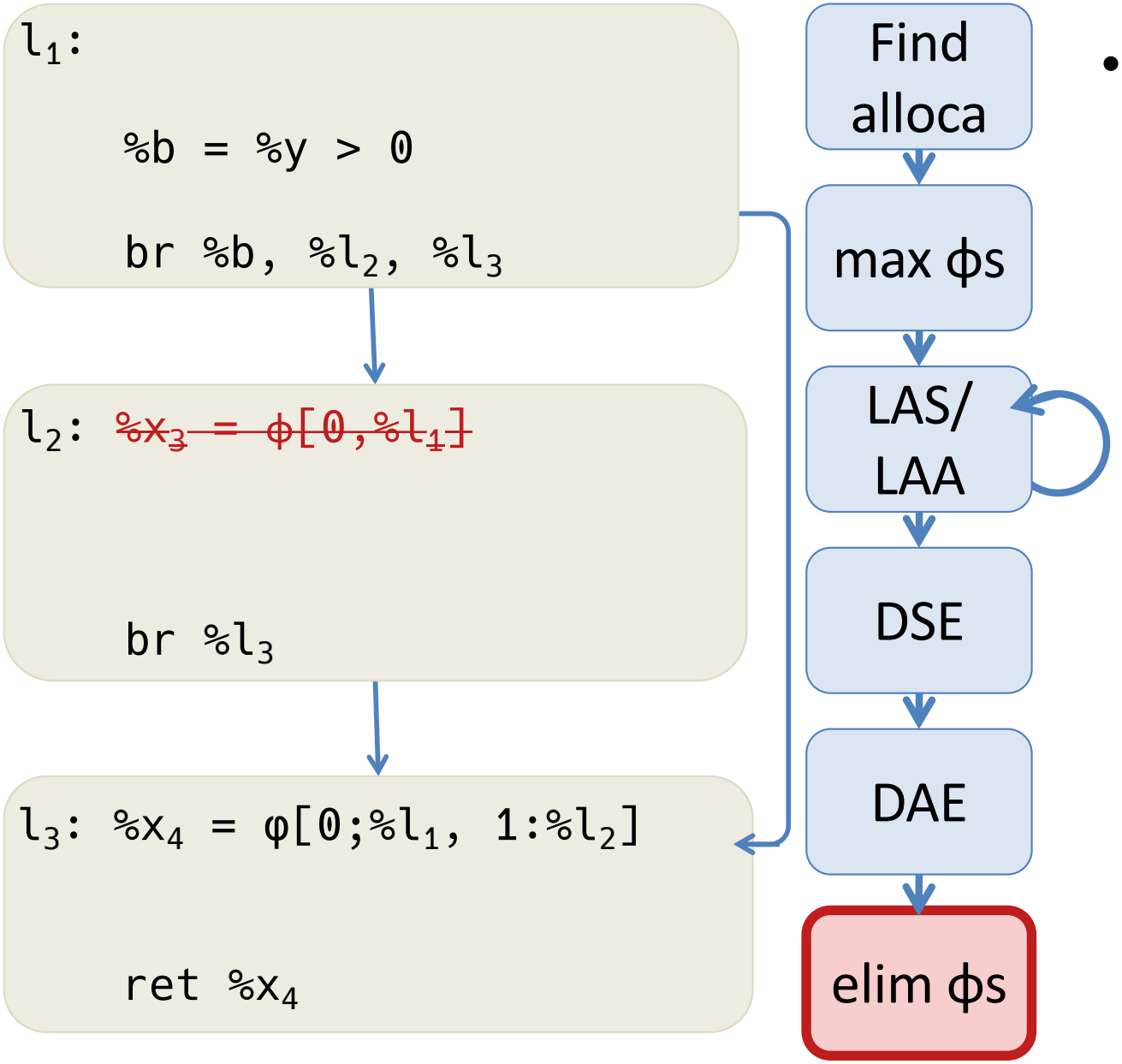
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations



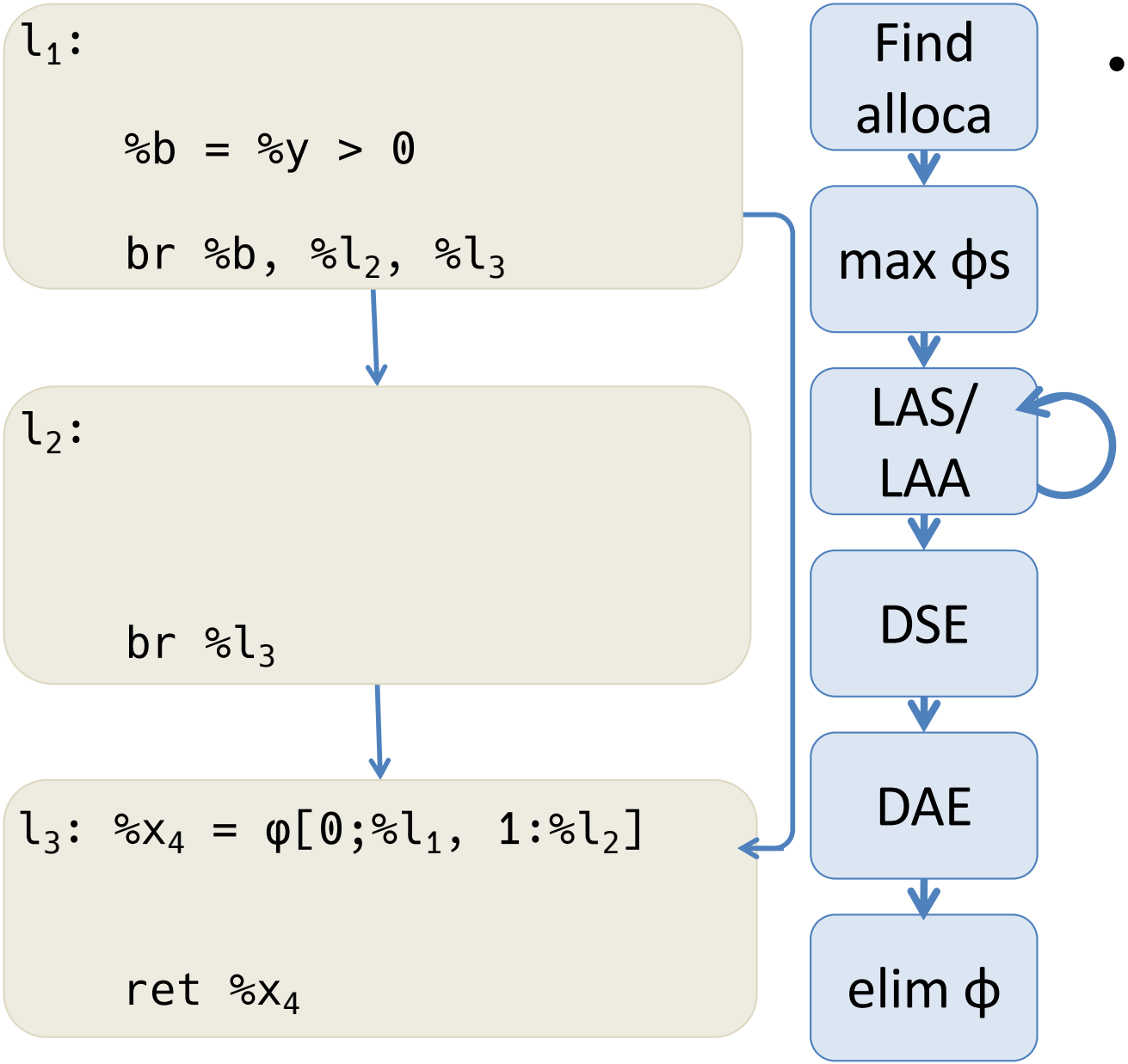
- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Example SSA Optimizations



- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor

Example SSA Optimizations



- Done!

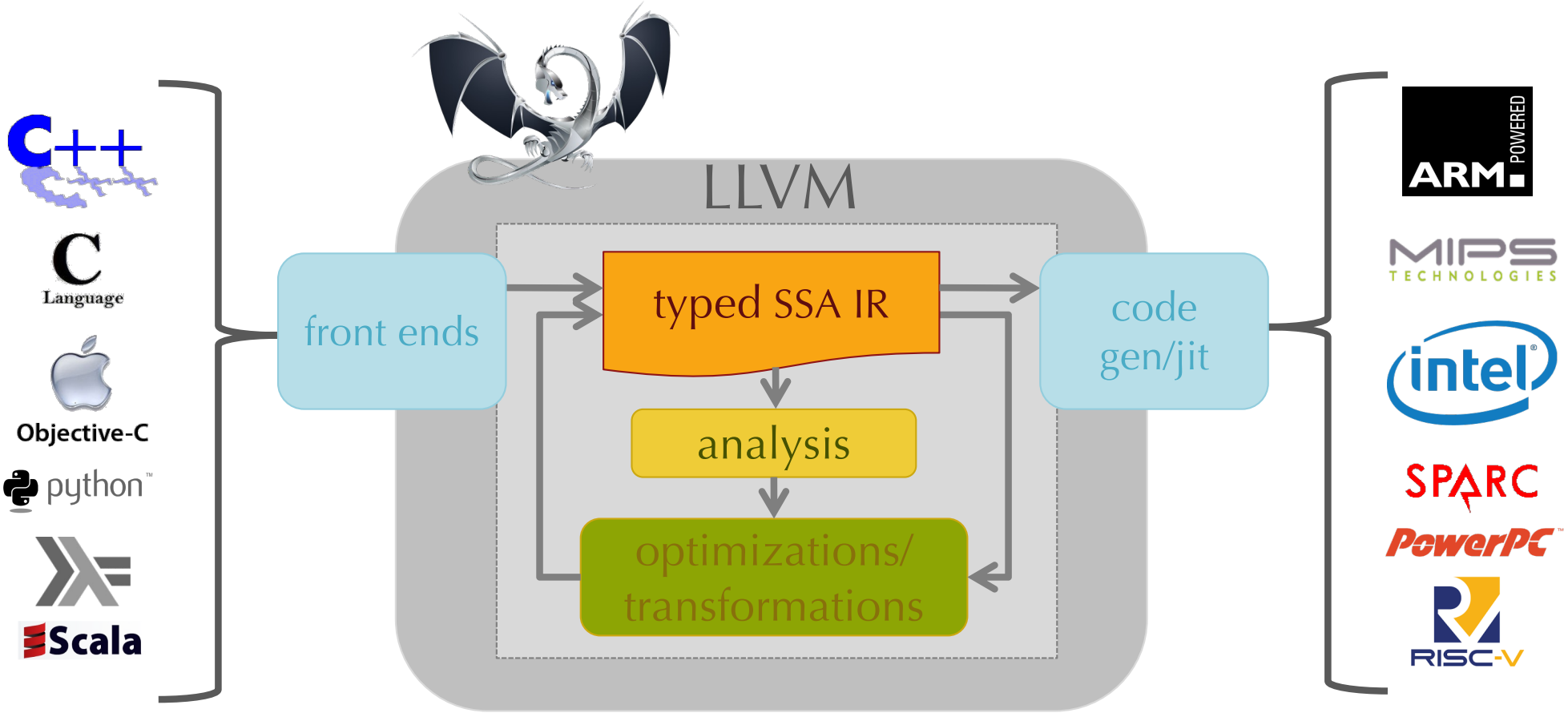
LLVM Phi Placement

- This transformation is also sometimes called register promotion
  - older versions of LLVM called this "mem2reg" memory to register promotion
- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
  - i.e. transforming loads/stores of structured data into loads/stores on register-sized data
- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  - Simplifies computing the DF

COMPILER VERIFICATION

LLVM Compiler Infrastructure

[Lattner et al.]

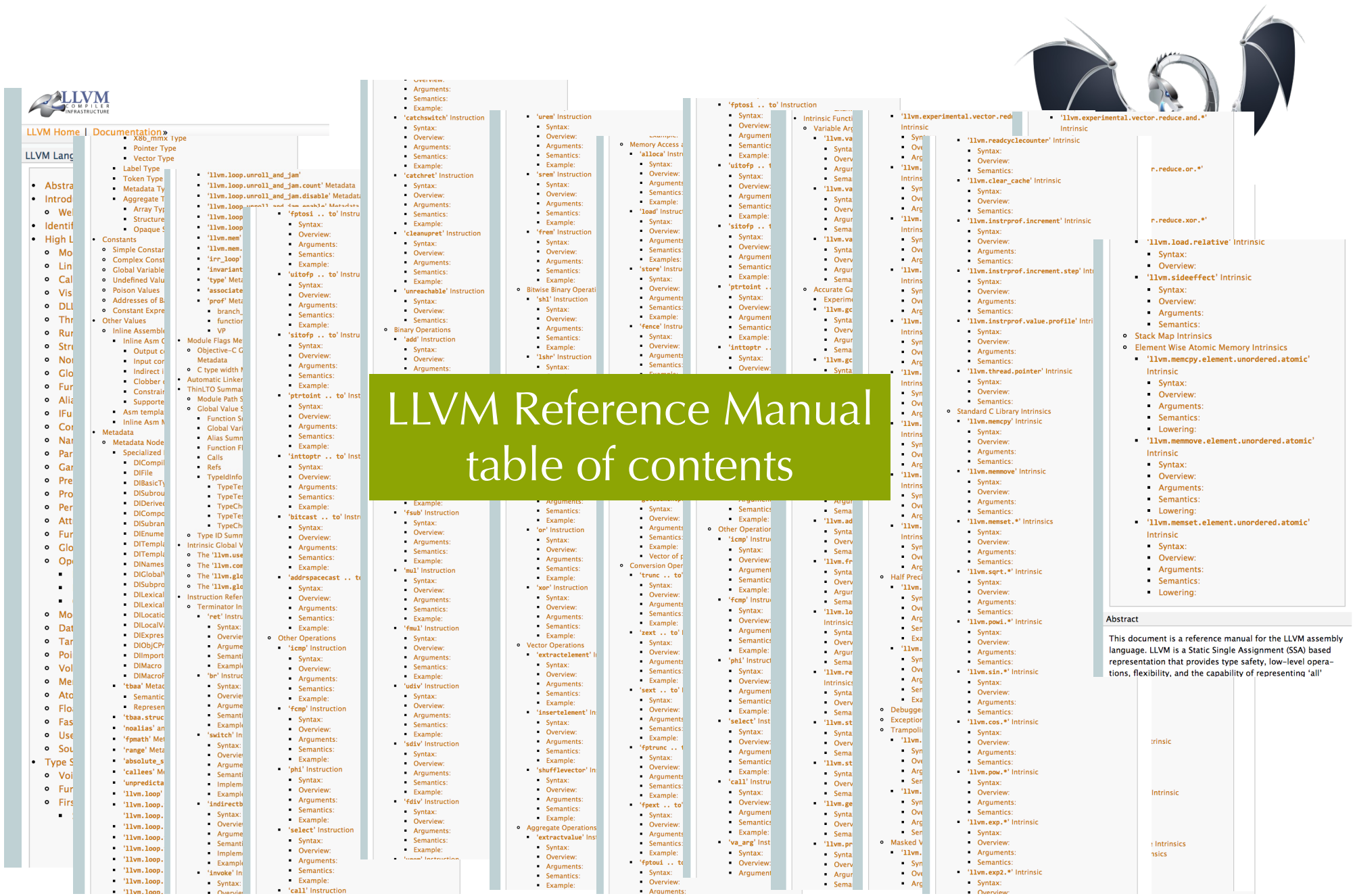


Other LLVM IR Features

- C-style data values
  - ints, structs, arrays, pointers, vectors
- Type system
  - used for layout/alignment/padding
- Relaxed-memory concurrency primitives
- Intrinsics
  - extend the language malloc, bitvectors, etc.
- Transformations & Optimizations

Make targeting LLVM IR easy and attractive for developers!

But… it's complex



One Example: `undef`

The **`undef`** "value" represents an arbitrary, but indeterminate bit pattern for any type.

Used for:

- uninitialized registers
- reads from volatile memory
- results of some underspecified operations

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)

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

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

```
⟦i8 undef⟧ = {0,…,255}
⟦i8 1⟧ = {1}
⟦%x⟧ = {a or b | a∈⟦i8 undef⟧, b ∈⟦1⟧}
     = {1,3,5,…,255}
⟦%y⟧ = {a xor b | a∈⟦%x⟧, b∈⟦%x⟧}
     = {0,2,4,…,254}
```

Interactions with Optimizations

Consider:

```
%y = mul i8 %x, 2
```

versus:

```
⟦%x⟧  = ⟦i8 undef⟧
      = {0,1,2,3,4,5,…,255}
⟦%y⟧ = {a mul 2 | a∈⟦%x⟧}
      = {0,2,4,…,254}
```

```
%y = add i8 %x, %x
```

```
⟦%x⟧  = ⟦i8 undef⟧
      = {0,1,2,3,4,5,…,255}
⟦%y⟧ = {a + b | a∈⟦%x⟧,
b∈⟦%x⟧}
      = {0,1,2,3,4,…,255}
```

Interactions with Optimizations

Consider:

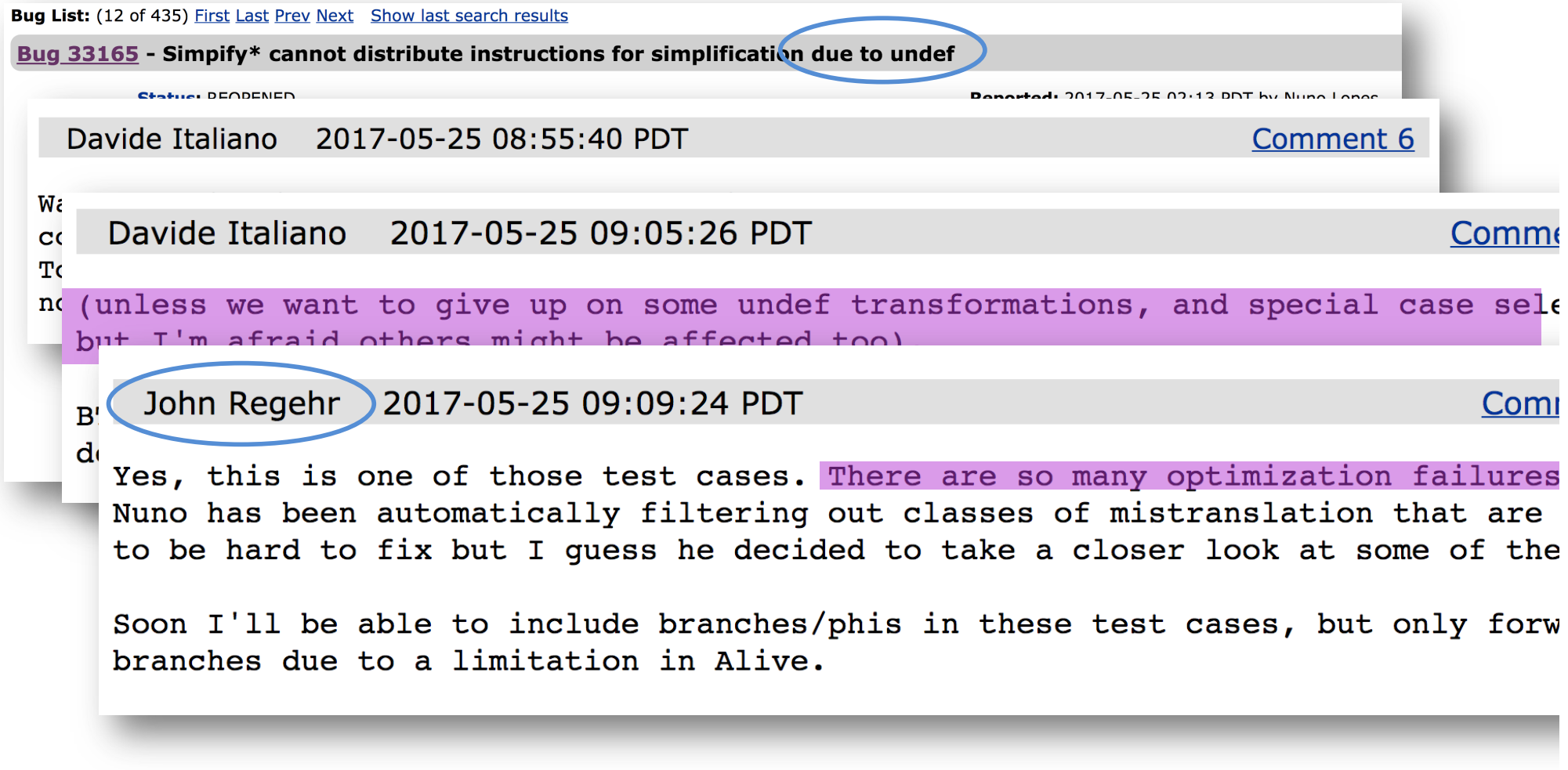
```
%y = mul i8 %x, 2
```

versus:

```
%y = add i8 %x, %x
```

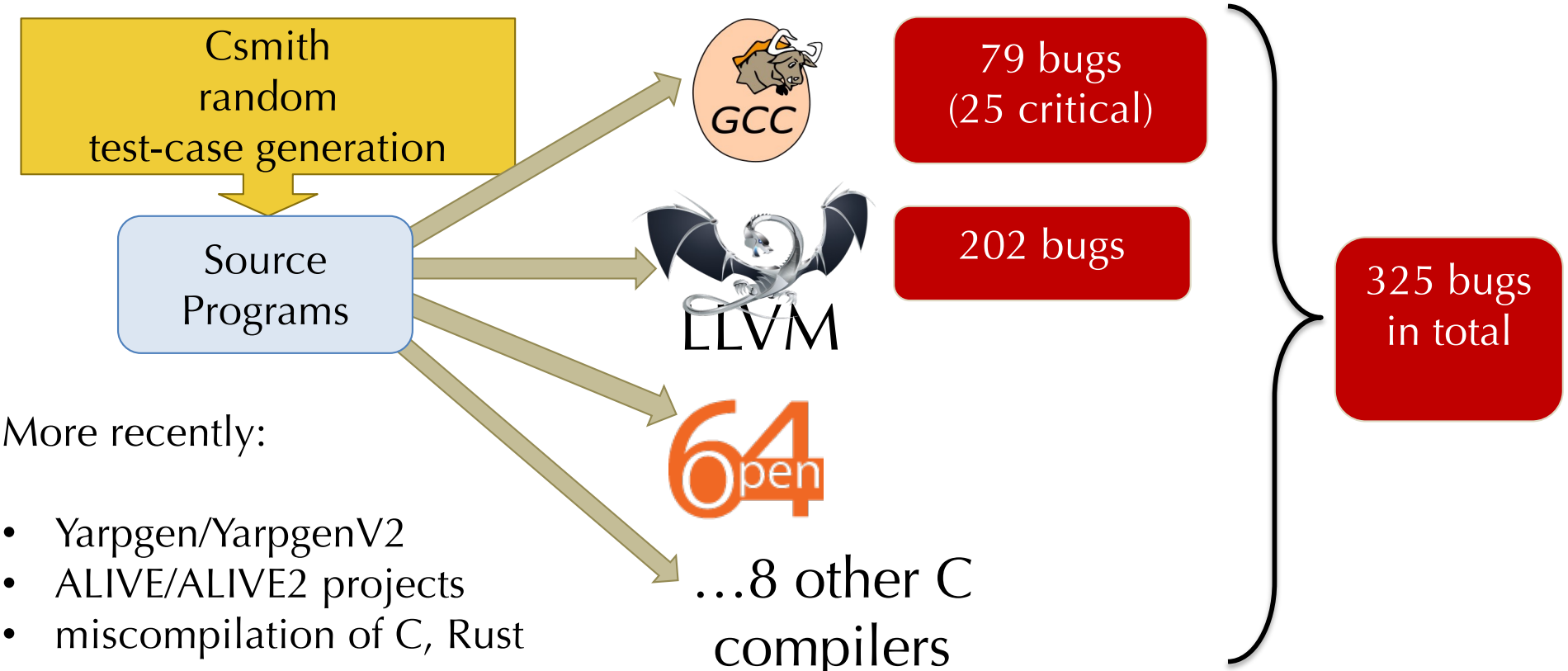
Upshot: if **%x** is **undef**, we can't optimize **mul** to **add** (or vice versa)!

What's the problem?



Compiler Bugs

[Regehr's group: Yang et al. PLDI 2011]



More recently:

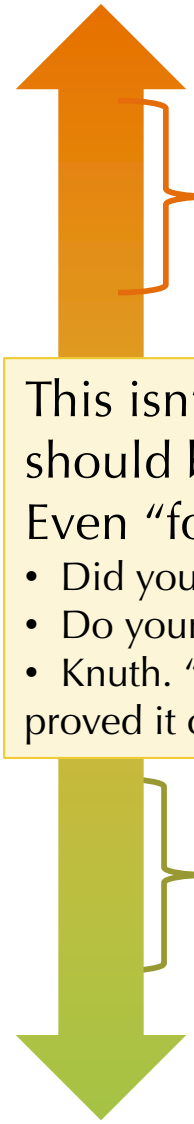
- Yarpgen/YarpgenV2
- ALIVE/ALIVE2 projects
- miscompilation of C, Rust sources [Lee et al. OOPSLA 2018]

LLVM is hard to trust
(especially for critical code)

What can we do about it?

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - "Formal" verification



Less "formal": Techniques may miss problems in programs

This isn't a tradeoff… all of these methods should be used.

Even "formal" methods can have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."

More "formal": eliminate with certainty as many problems as possible.

Goal: Verified Software Correctness

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - "Formal" verification

Q: How can we move the needle towards mathematical software correctness properties?

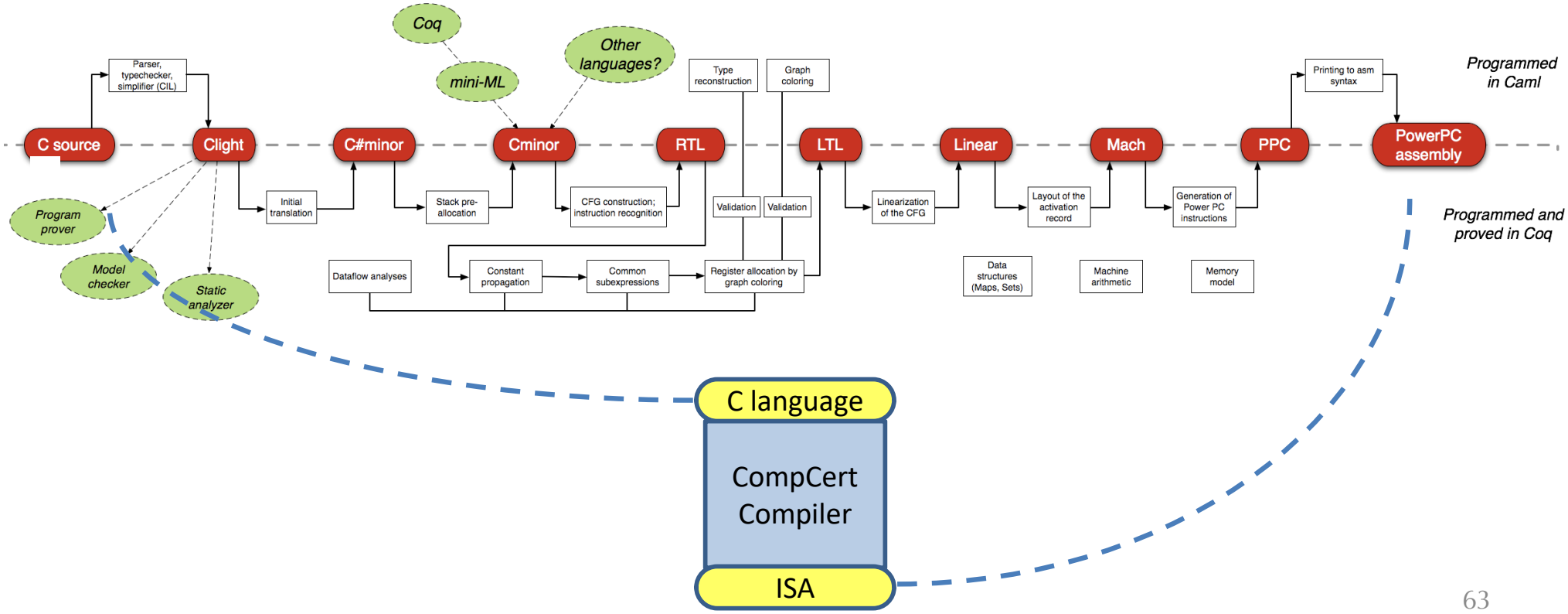
Taking advantage of advances in computer science:

- Moore's law
- improved programming languages & theoretical understanding
- better tools: interactive theorem provers

CompCert – A Verified C Compiler

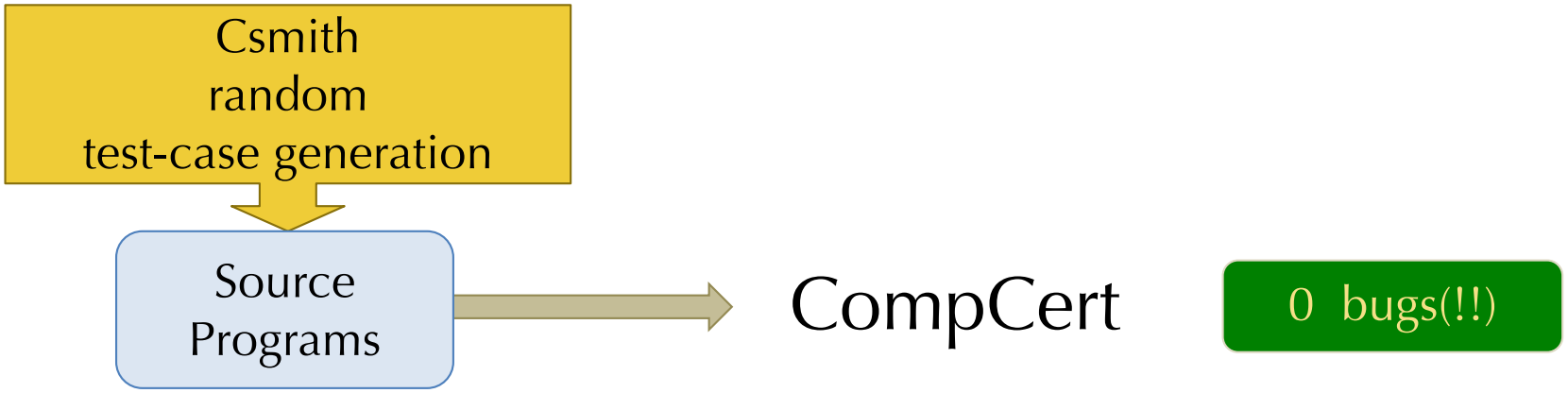
Xavier Leroy
INRIA

Optimizing C Compiler,
proved correct end-to-end
with machine-checked proof in Coq



Csmith on CompCert?

[Yang et al. PLDI 2011]



Verification Works!

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested ***for which Csmith cannot find wrong-code errors***. This is not for lack of trying: we have devoted about six CPU-years to the task. ***The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.***"

– *Regehr et. al 2011*

Our Approach: Formal Verification

Interactive theorem proving in Coq

- not model checking / SMT
- human-in-the-loop

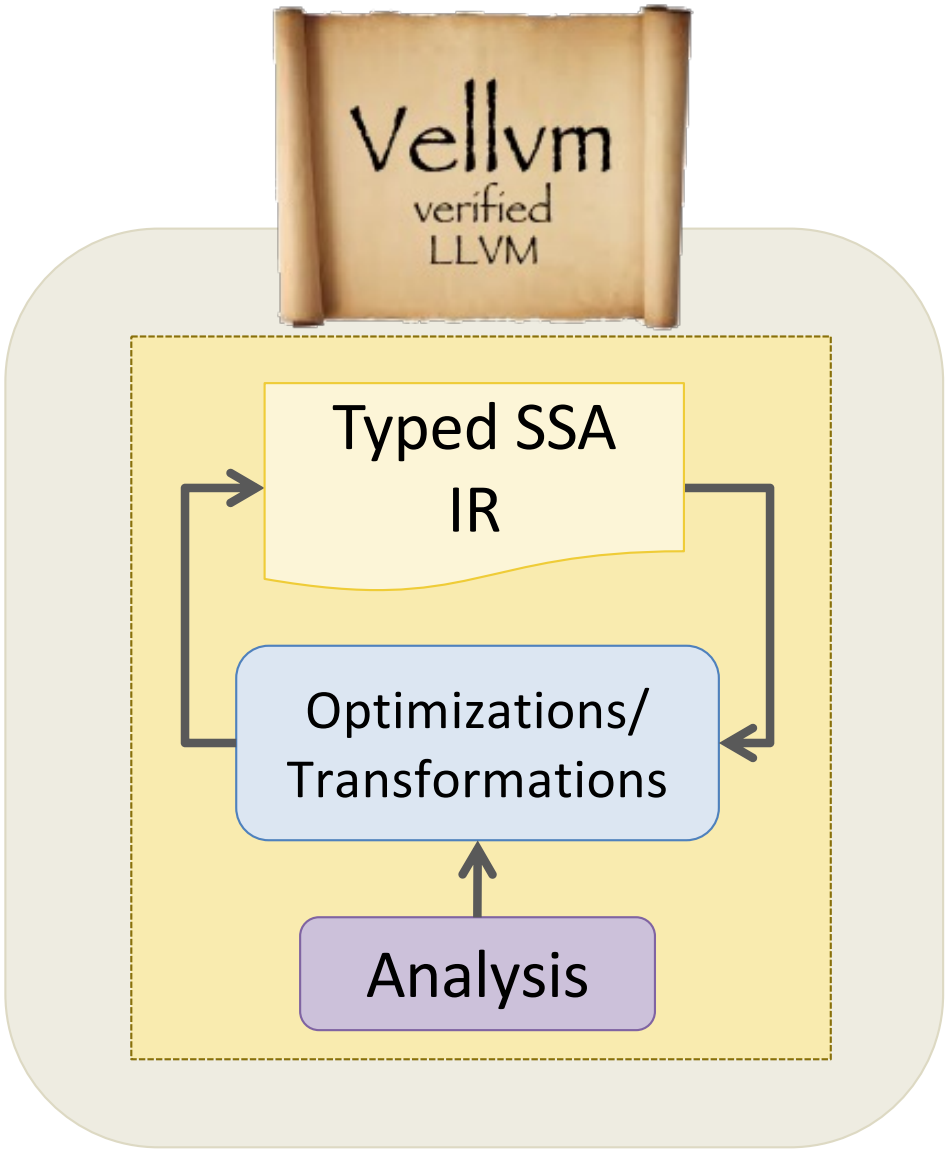
Using Coq ***is*** functional programming

…but some of your programs *are* proofs

⇒ proof engineering

The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Zackowski, et al. ICFP2021]



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