Lecture 25

CIS 341: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: Wednesday, April 27th
- Final Exam:
  - According to registrar: Monday, May 2nd noon - 2:00pm

See HW6: Dataflow Analysis

IMPLEMENTATION

REGISTER ALLOCATION

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs

- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed

- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

1. Compute liveness information: `live(x)`
   - recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `pal` be the set of usable registers
   - usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   - locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   - `used` = `{r | reg r = uid_loc(y)` s.t. `y ∈ live(x)}`
   - `available = pal - used`
   - If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n ; n = n + 1`
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

For HW6

- HW 6 implements two naive register allocation strategies:
  - `none`: spill all registers
  - `greedy`: uses linear scan
- Also offers choice of liveness
  - `trivial`: assume all variables are live everywhere
  - `dataflow`: use the dataflow algorithms

- Your job: do "better" than these.
- Quality Metric:
  - registers other than rbp count positively
  - rbp counts negatively (it is used for spilling)
  - shorter code is better

- Linear scan is OK
  - but… how can we do better?

GRAPH COLORING

Register Allocation

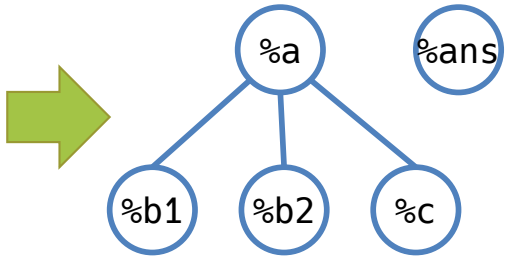
Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, "spill" a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

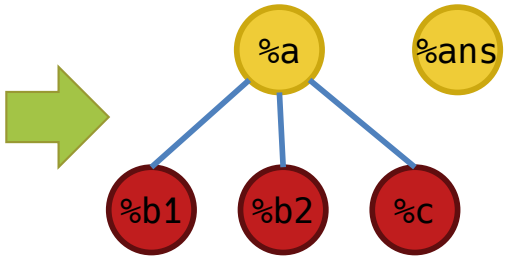
Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

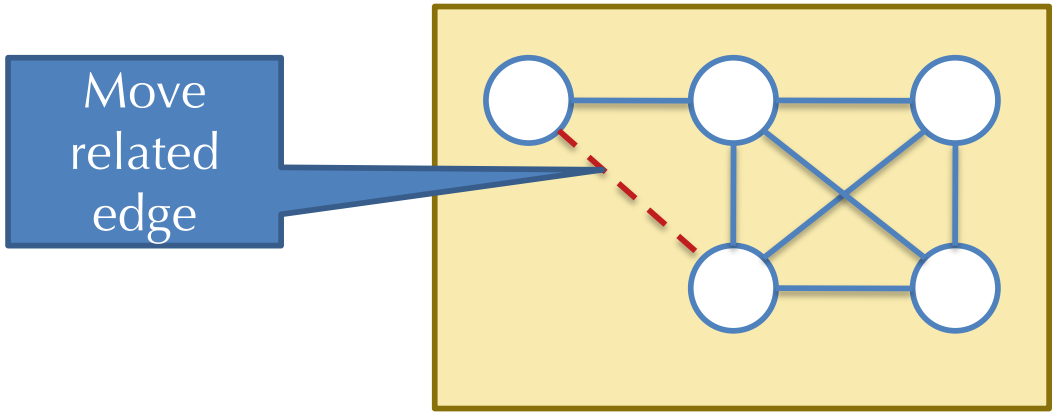
Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:
  `movq t1, t2`
  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

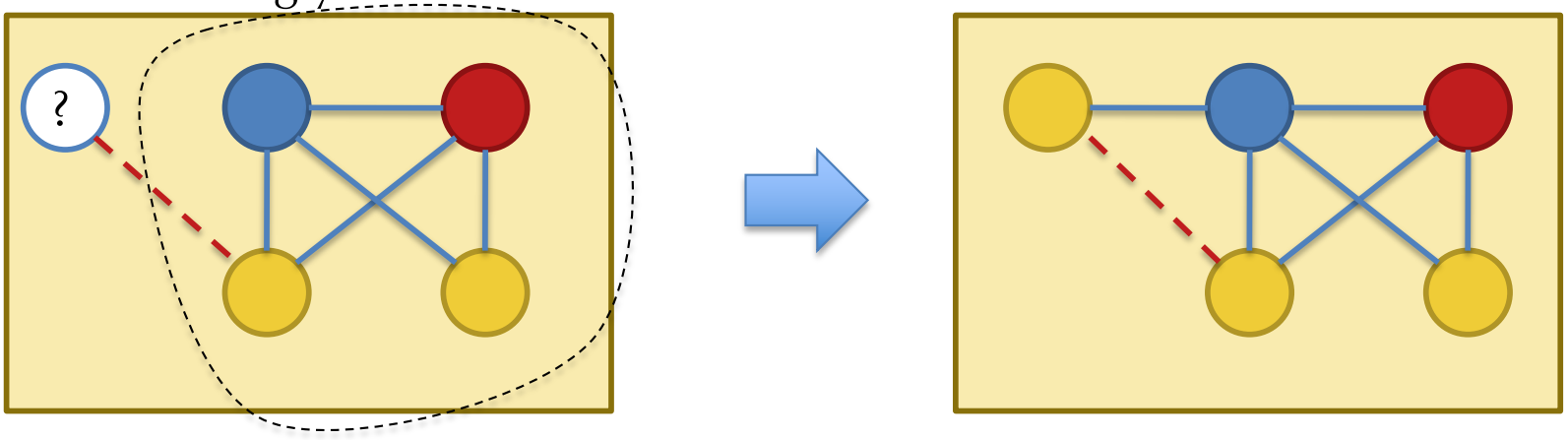
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible, pick a color of an already colored node reachable by a move-related edge.

Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

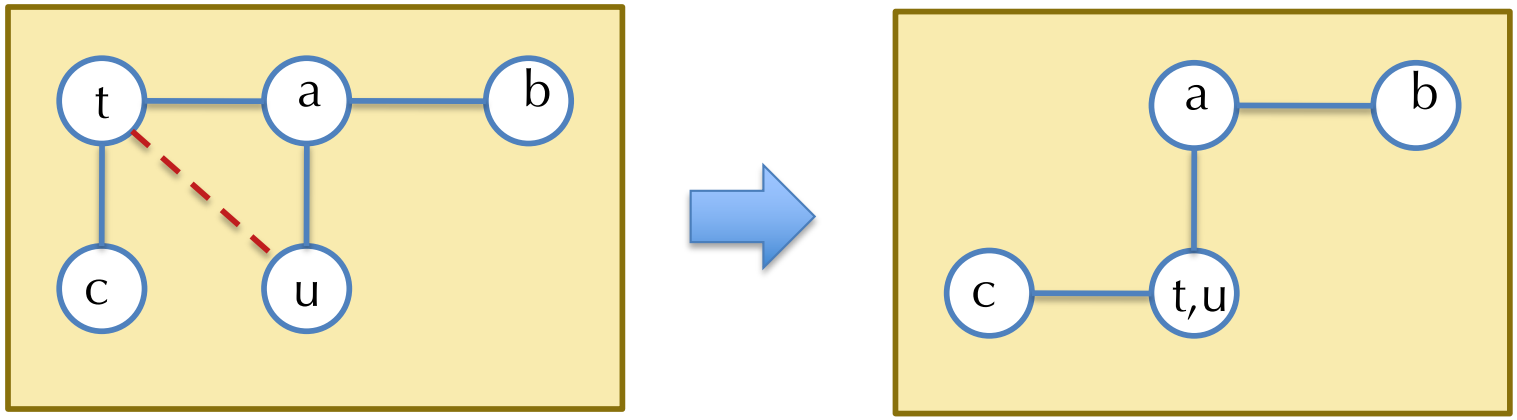


- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

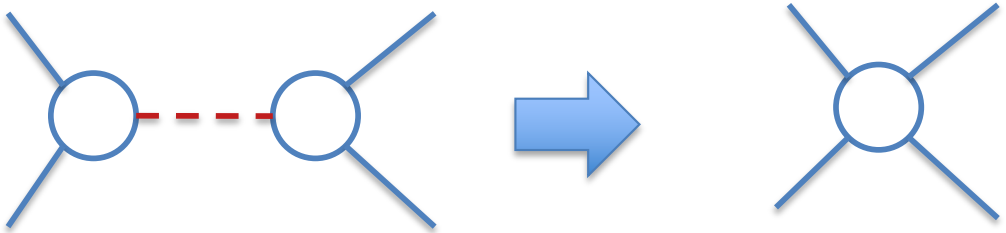


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

1. *Brigg's strategy*: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors (with degree ≥ k).

2. *George's strategy:* We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

Last details

- After register allocation, the compiler should do a peephole optimization pass to remove redundant moves.
- Some architectures specify calling conventions that use registers to pass function arguments.
  - It's helpful to move such arguments into temporaries in the function prelude so that the compiler has as much freedom as possible during register allocation. (Not an issue on X86, though.)

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"

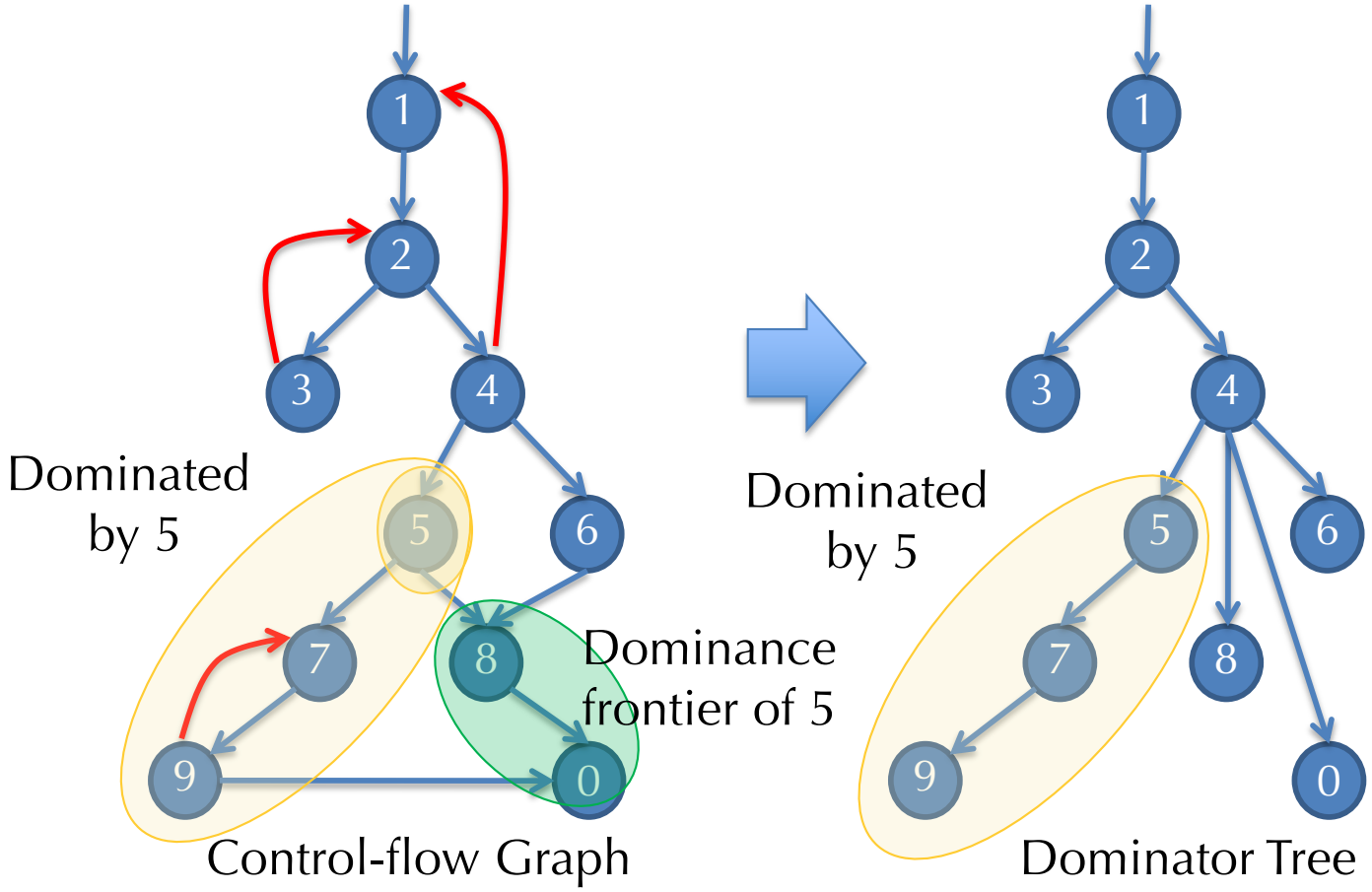
- Node A ***strictly dominates*** node B if A dominates B and A ≠ B.
  - Note: A does not strictly dominate B if A does not dominate B or A = B.

- 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.

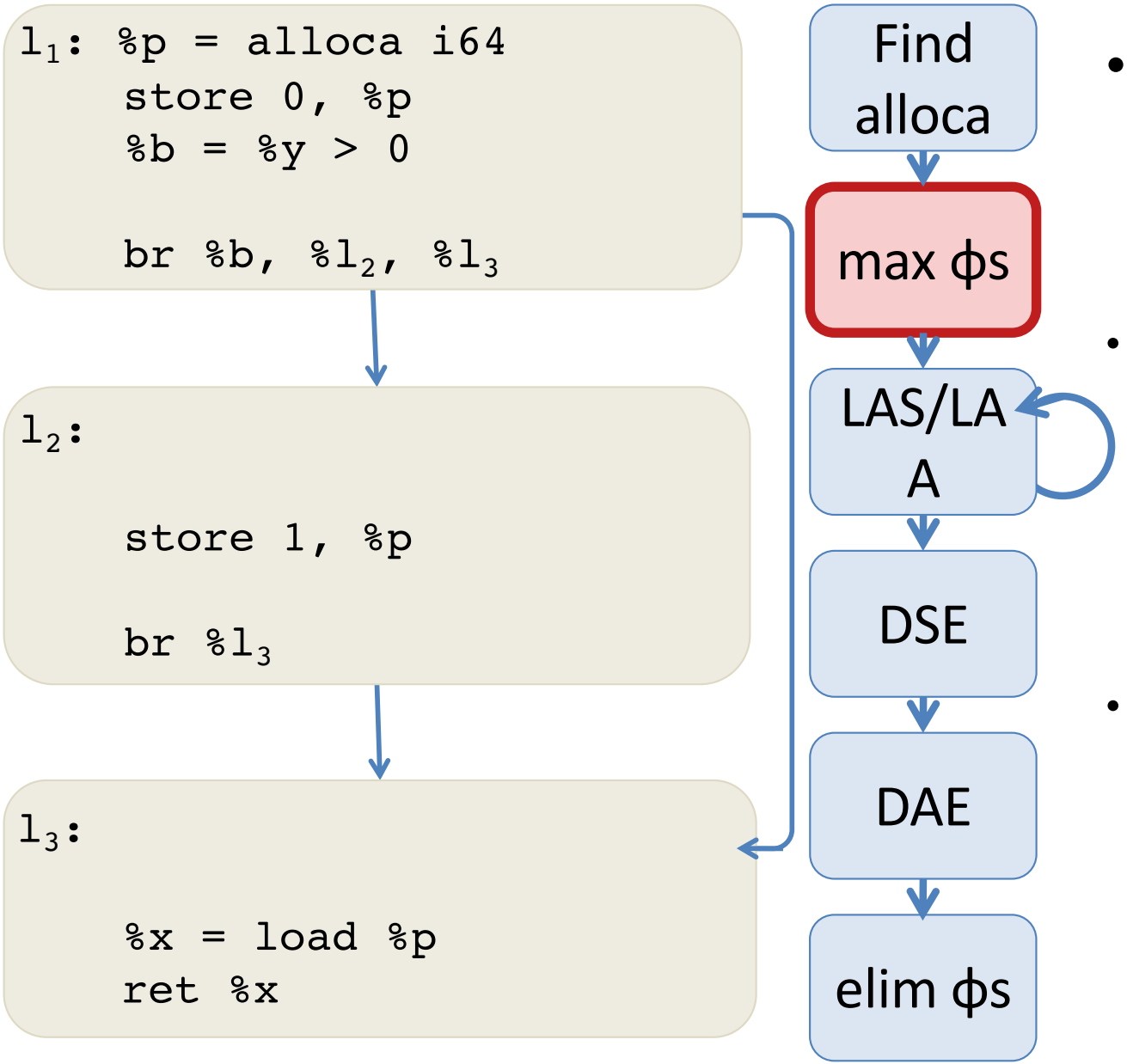
Phi Placement Alternative

- 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
```

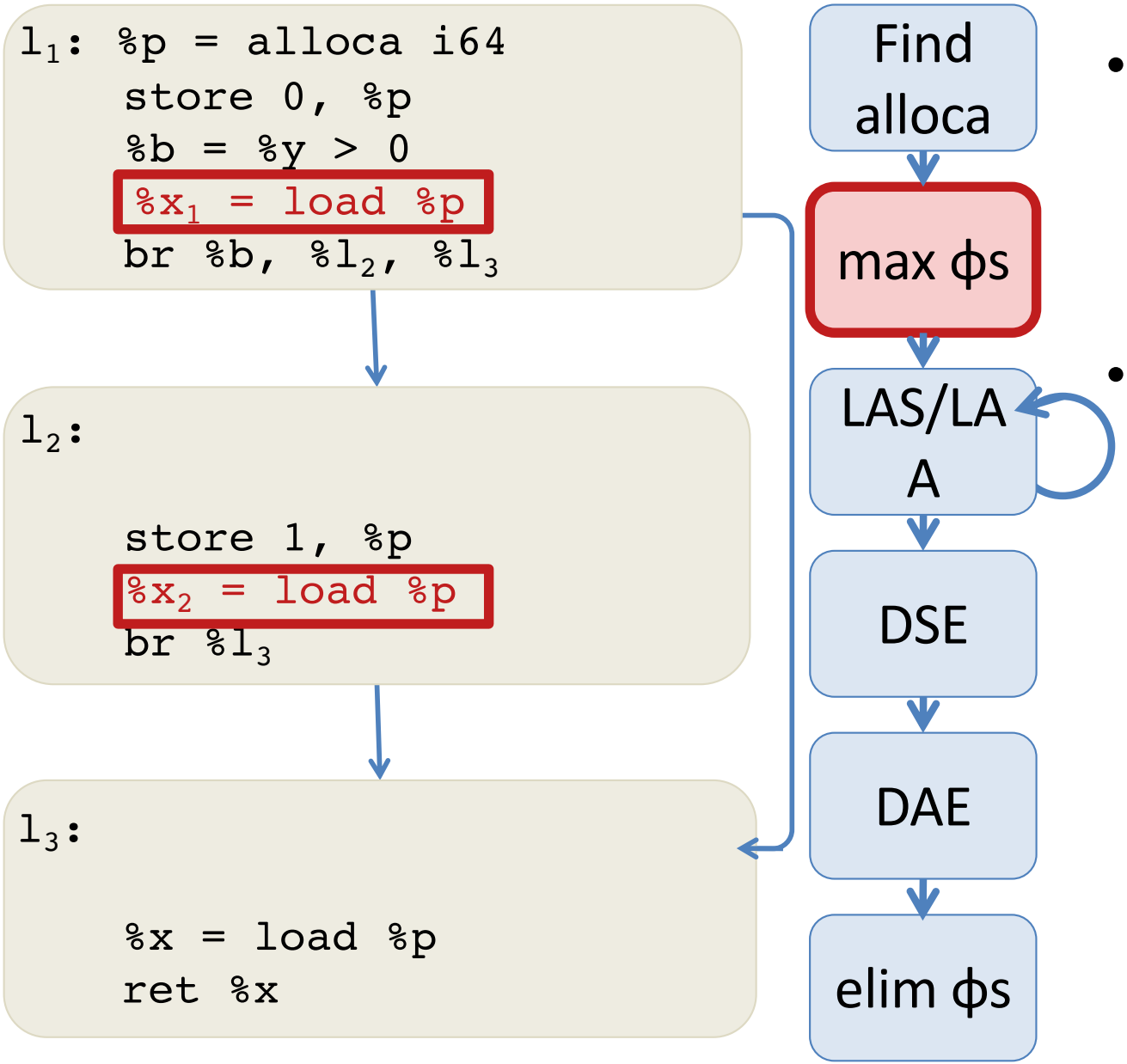
- Interleave with other optimizations
  - copy propagation
  - constant propagation
  - etc.

Example SSA Optimizations



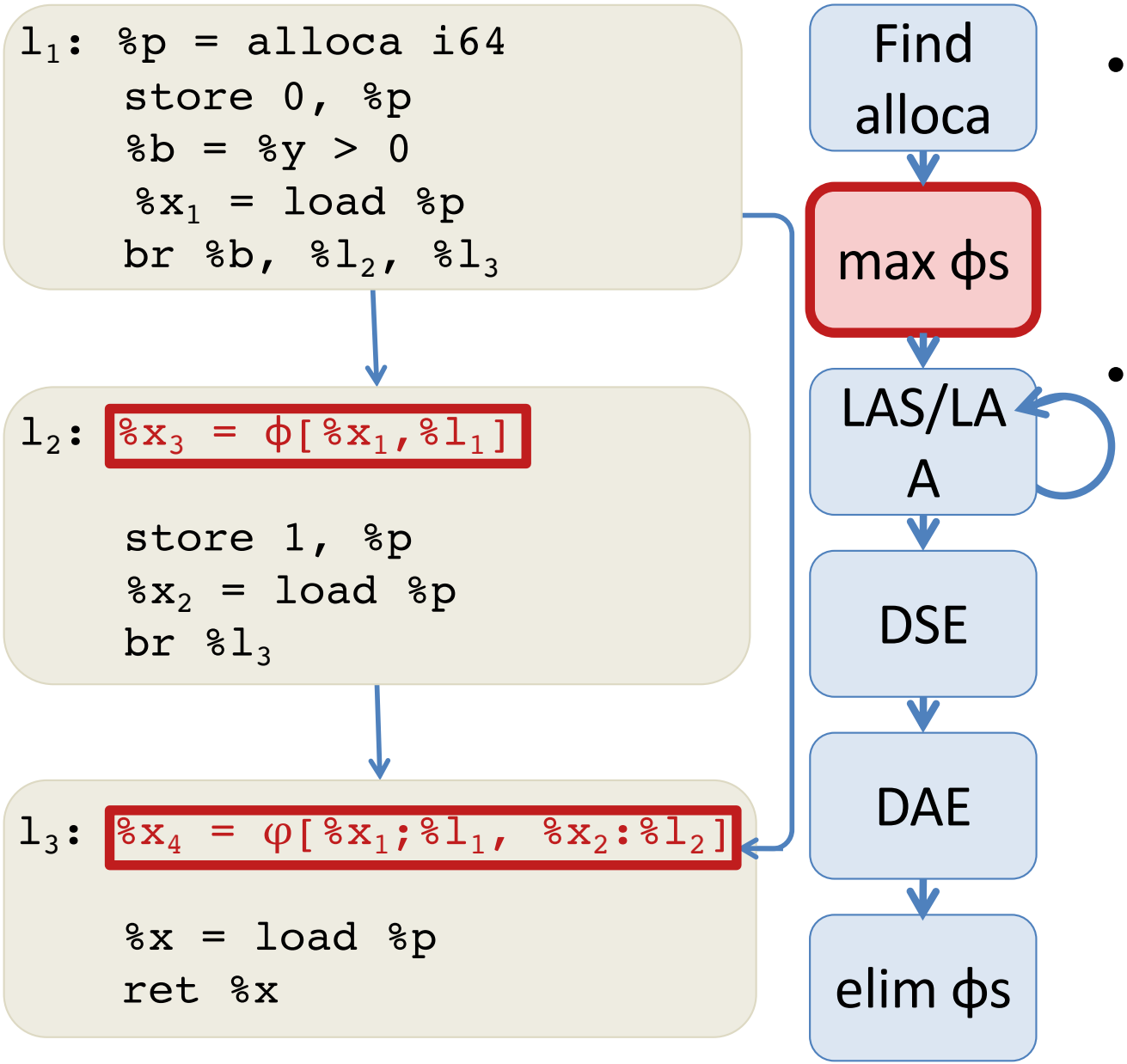
- How to place phi nodes without breaking SSA?
- Note: the "real" implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

Example SSA Optimizations



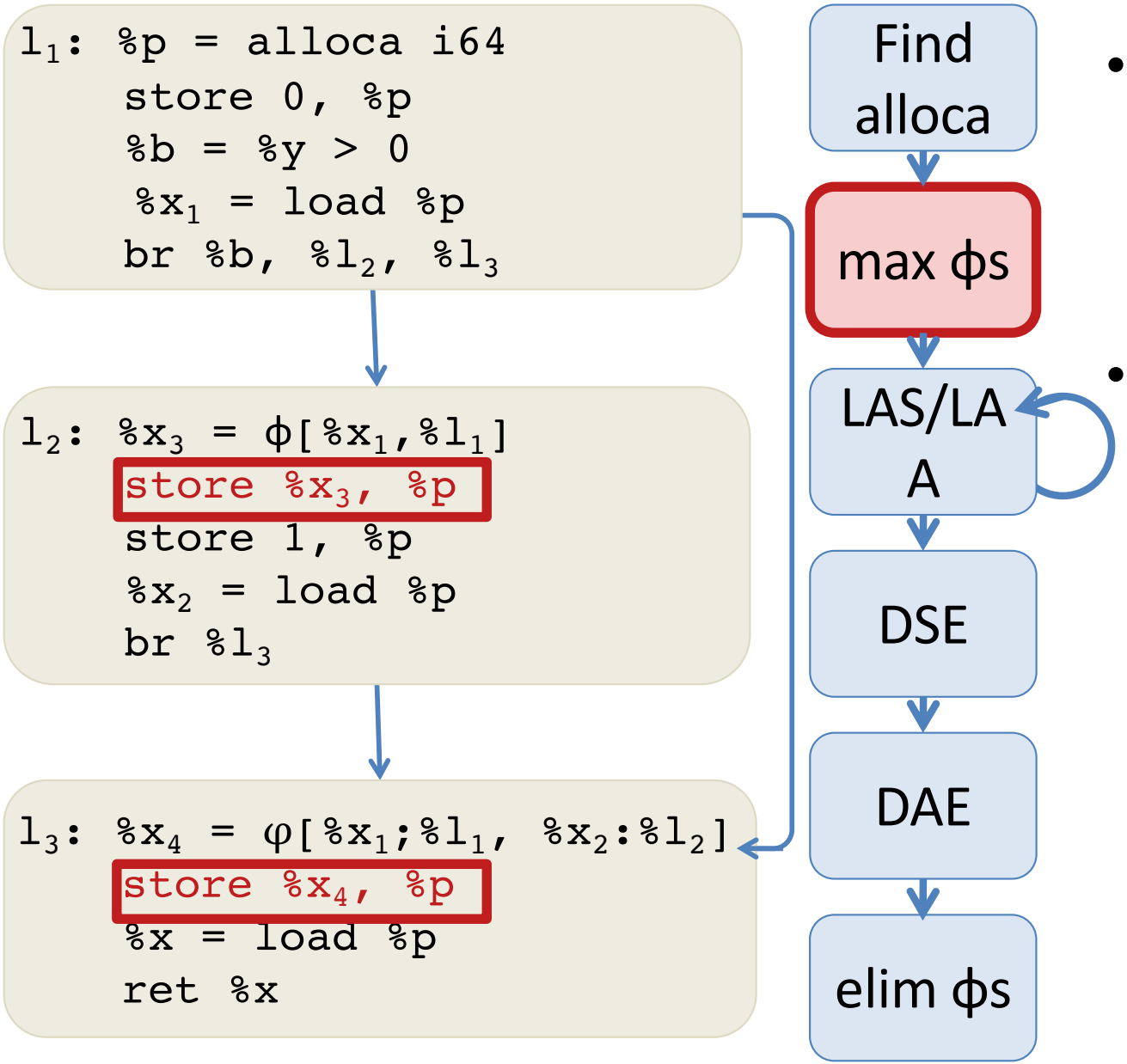
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example SSA Optimizations



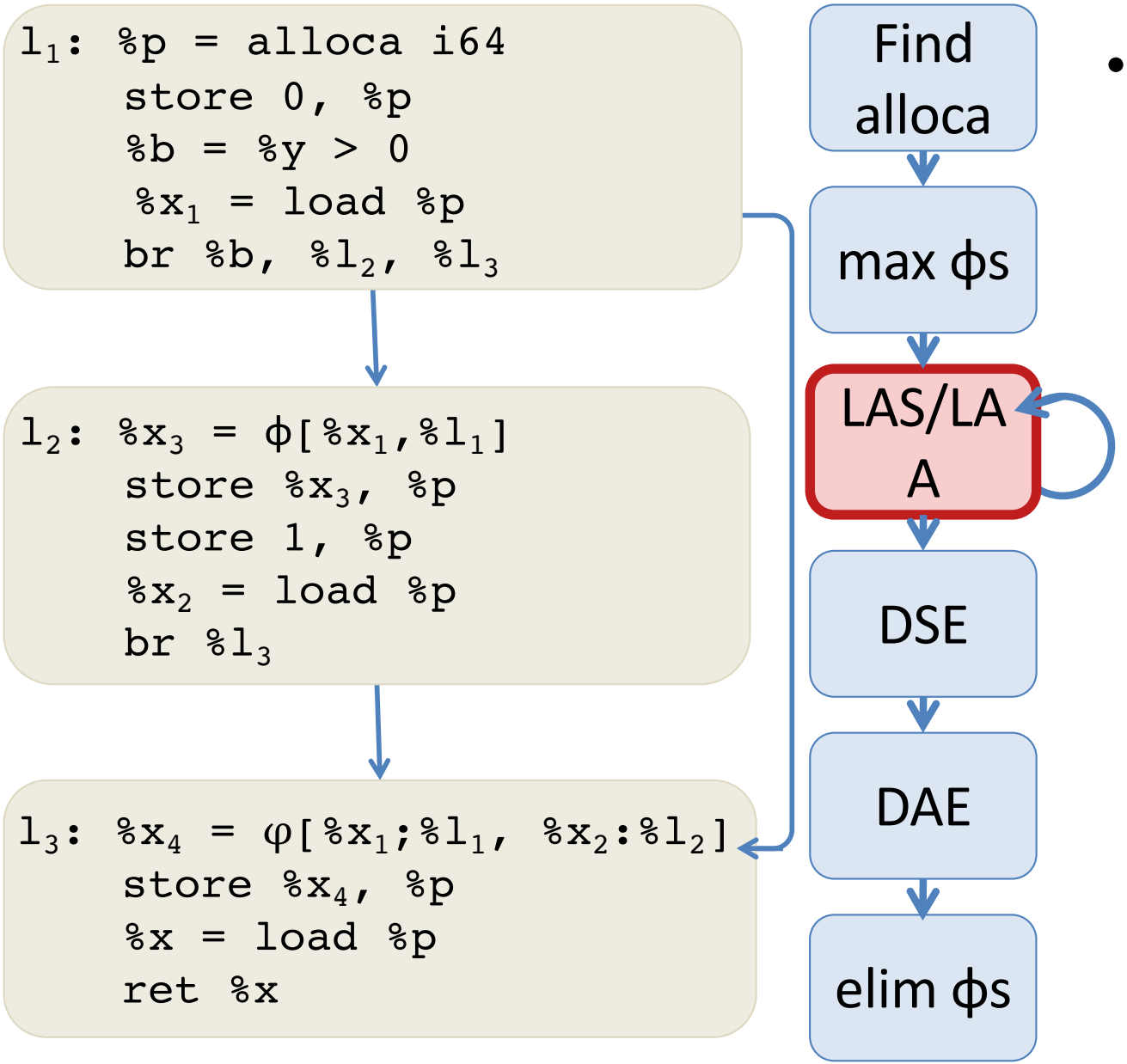
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block

Example SSA Optimizations



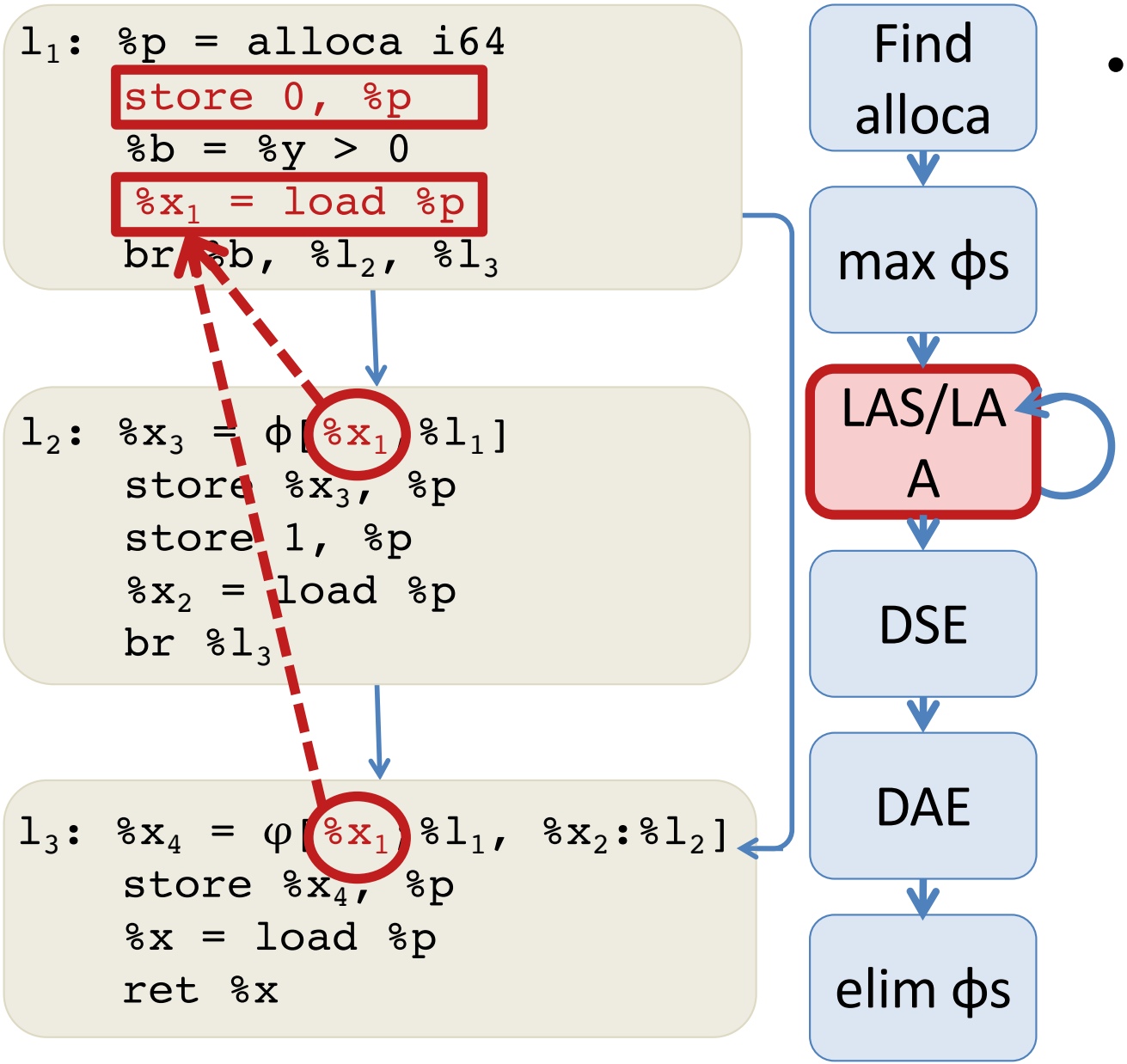
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

Example SSA Optimizations



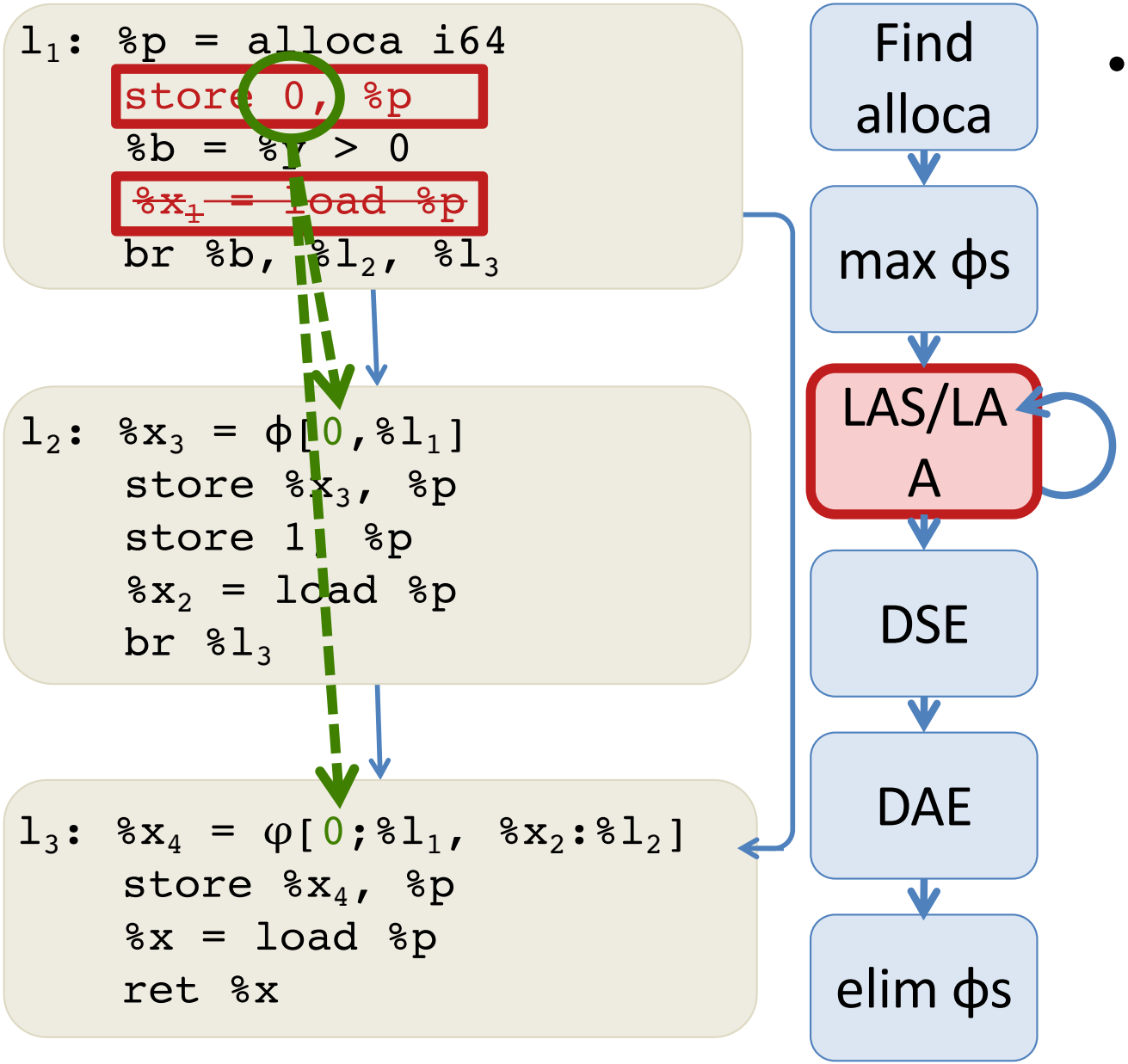
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



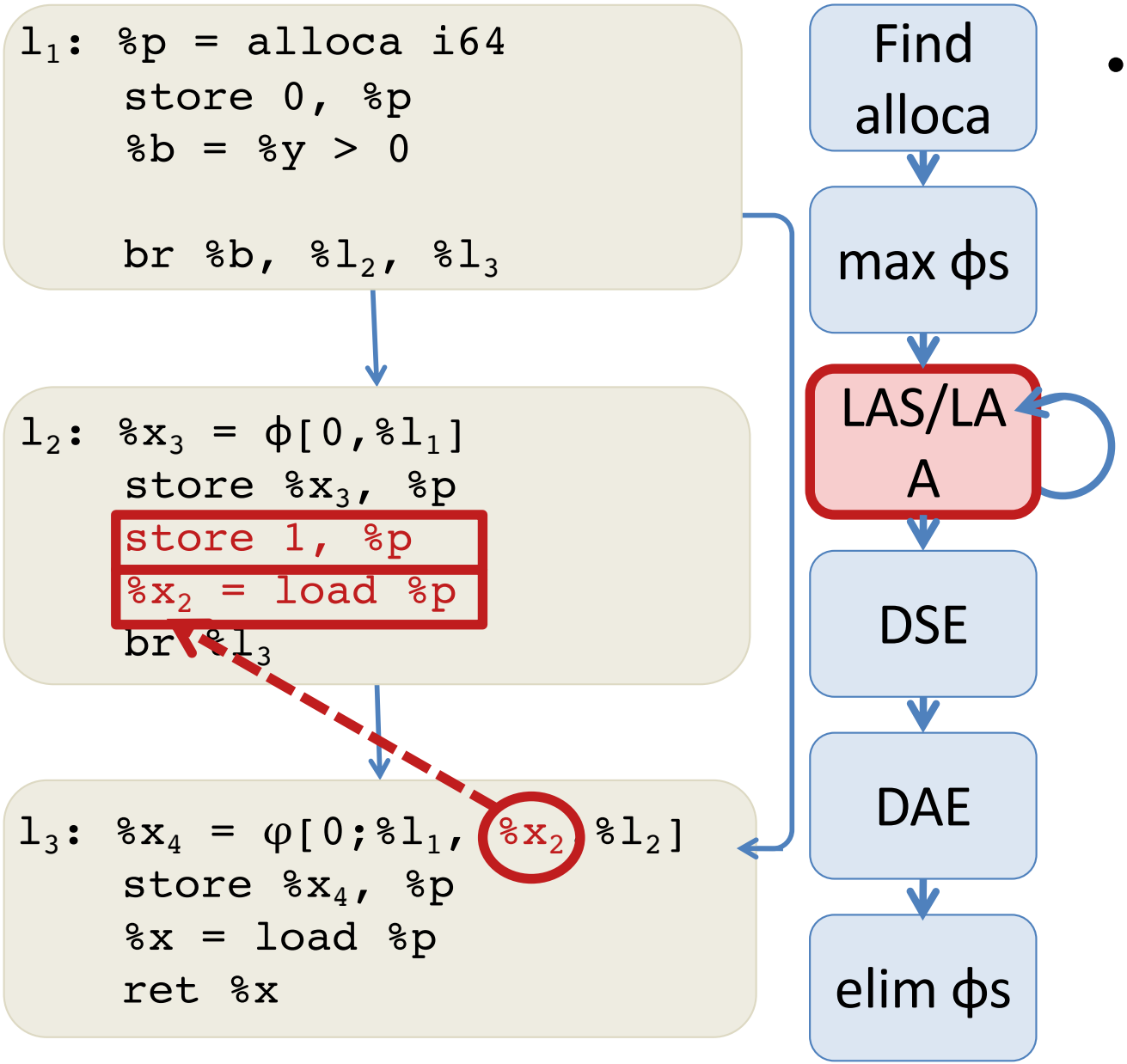
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



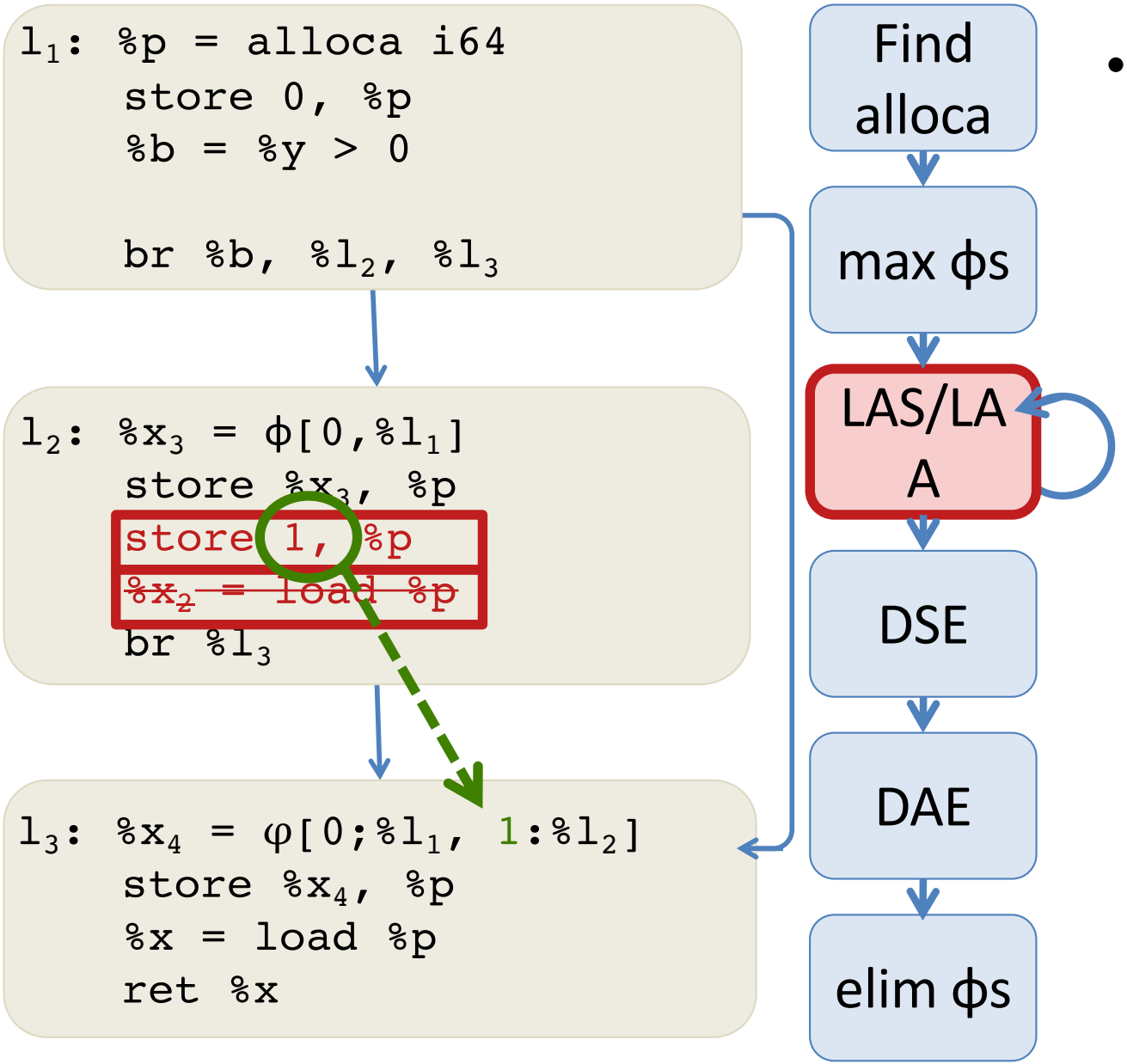
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



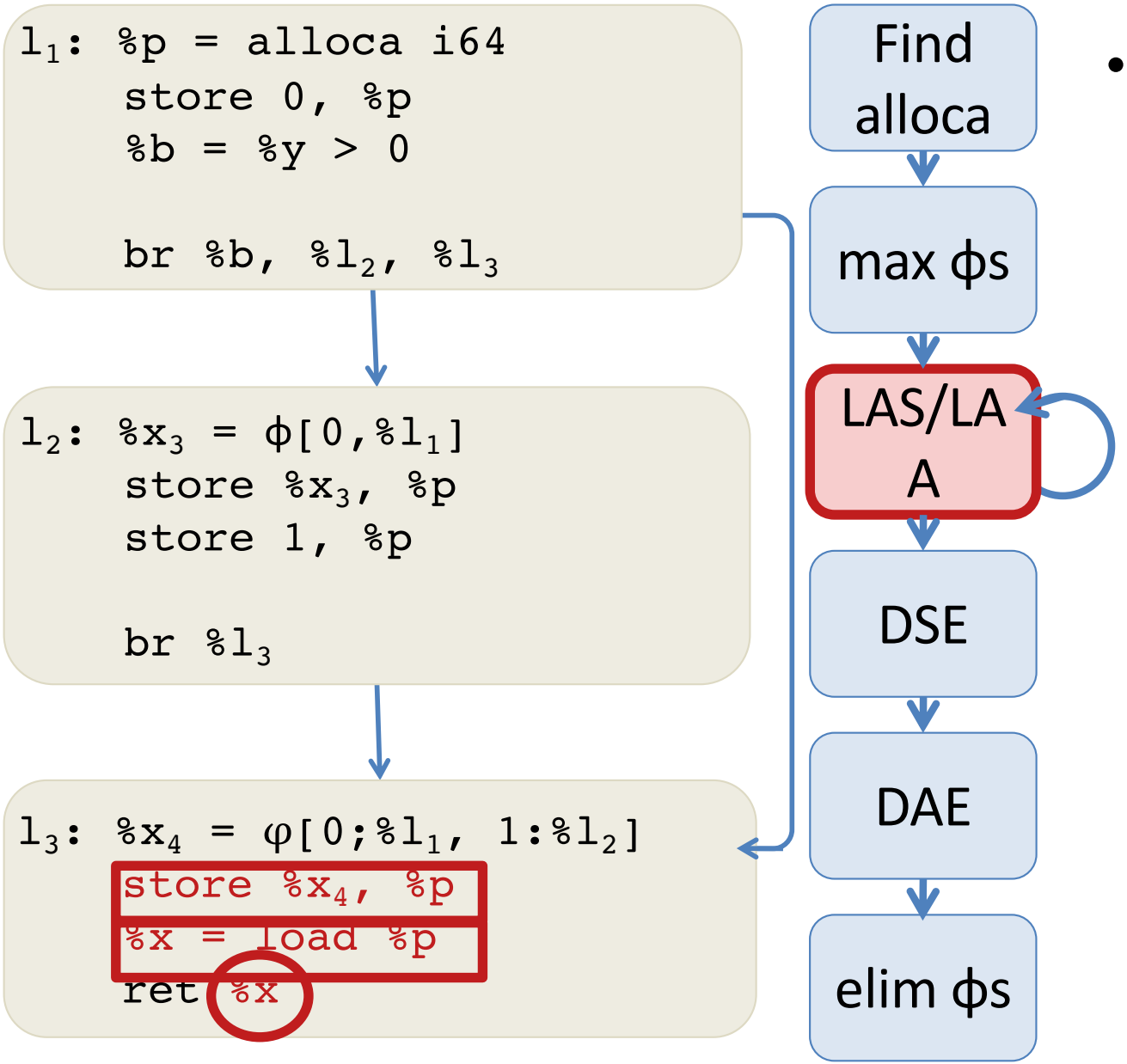
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



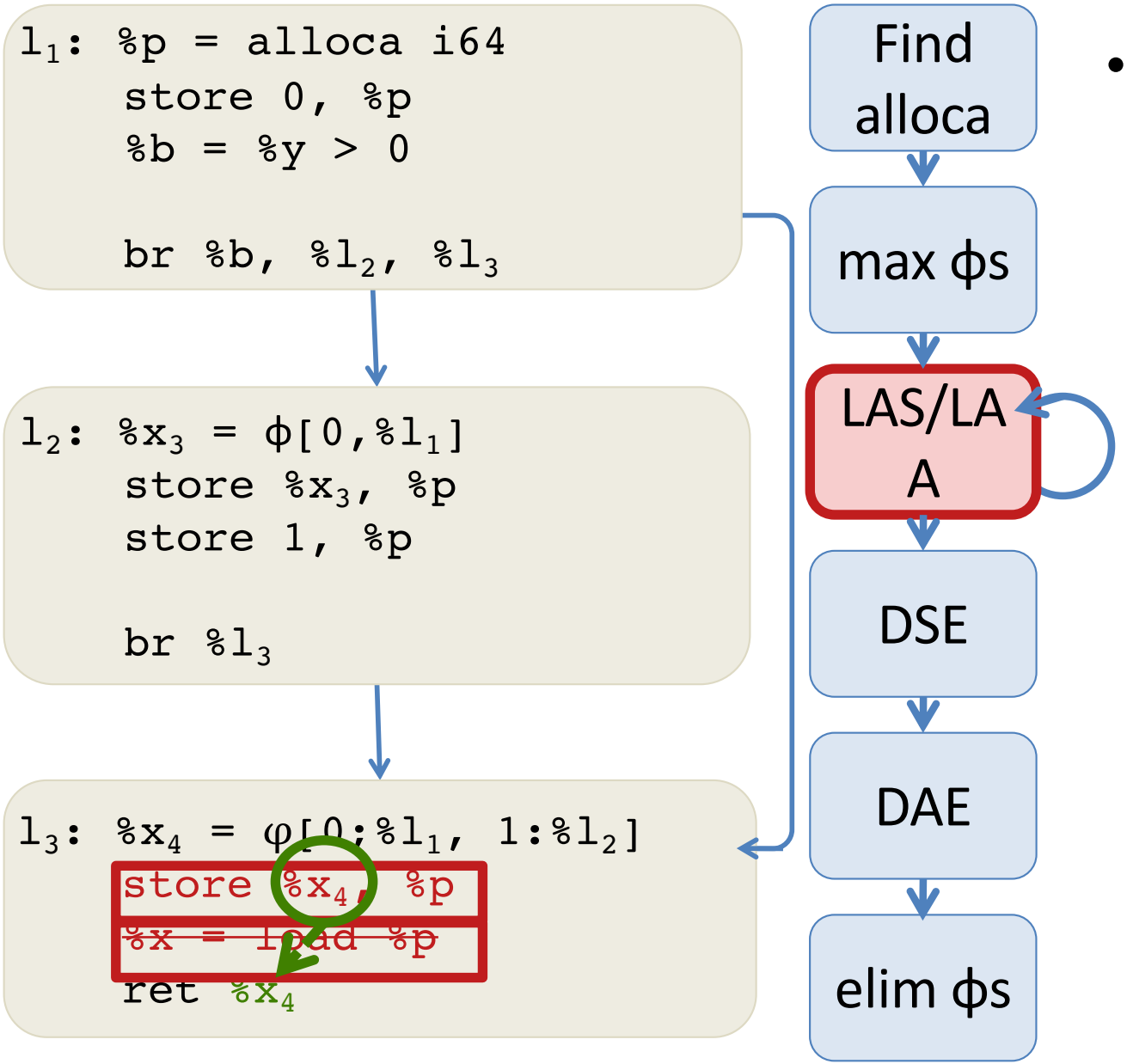
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



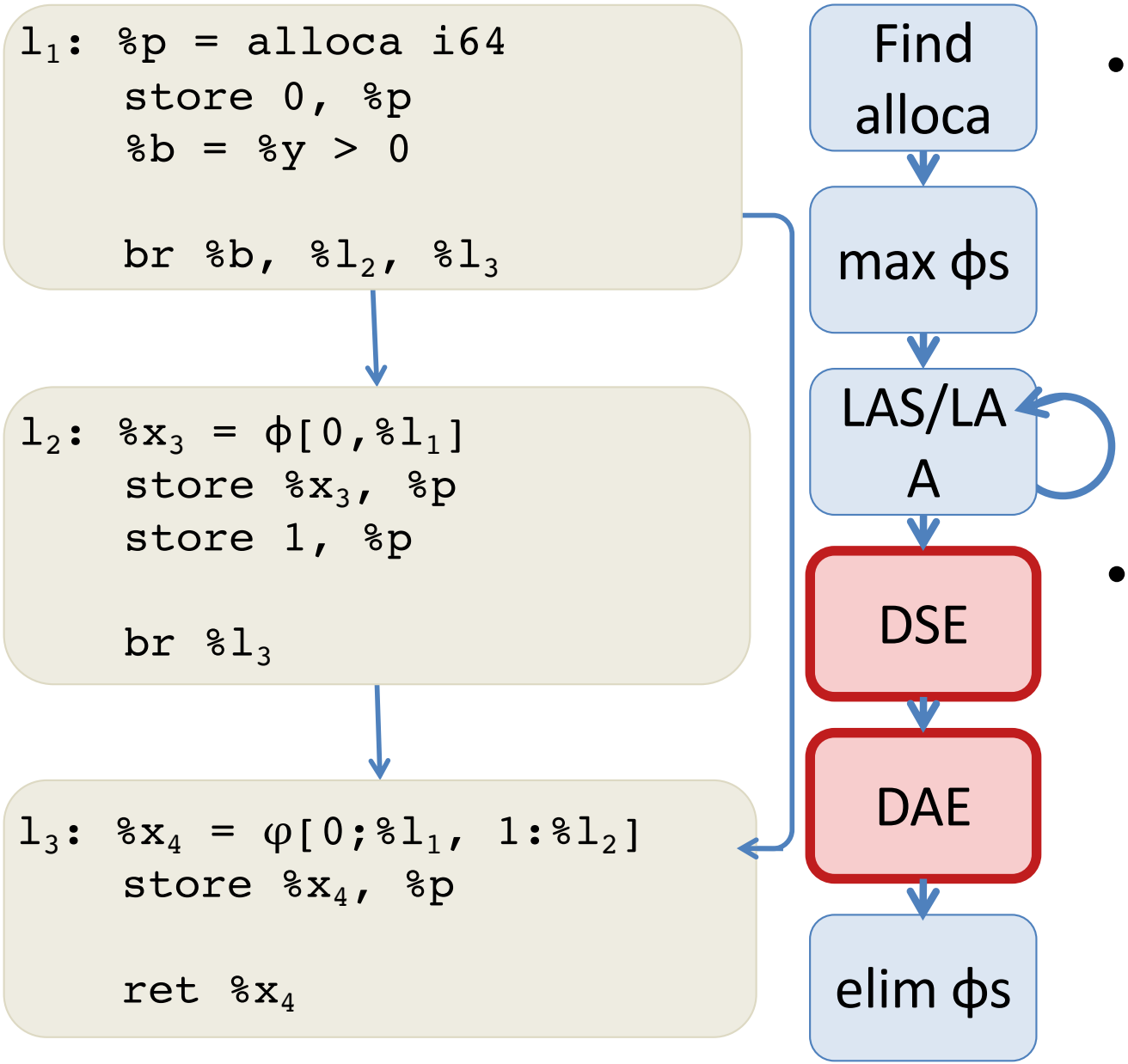
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



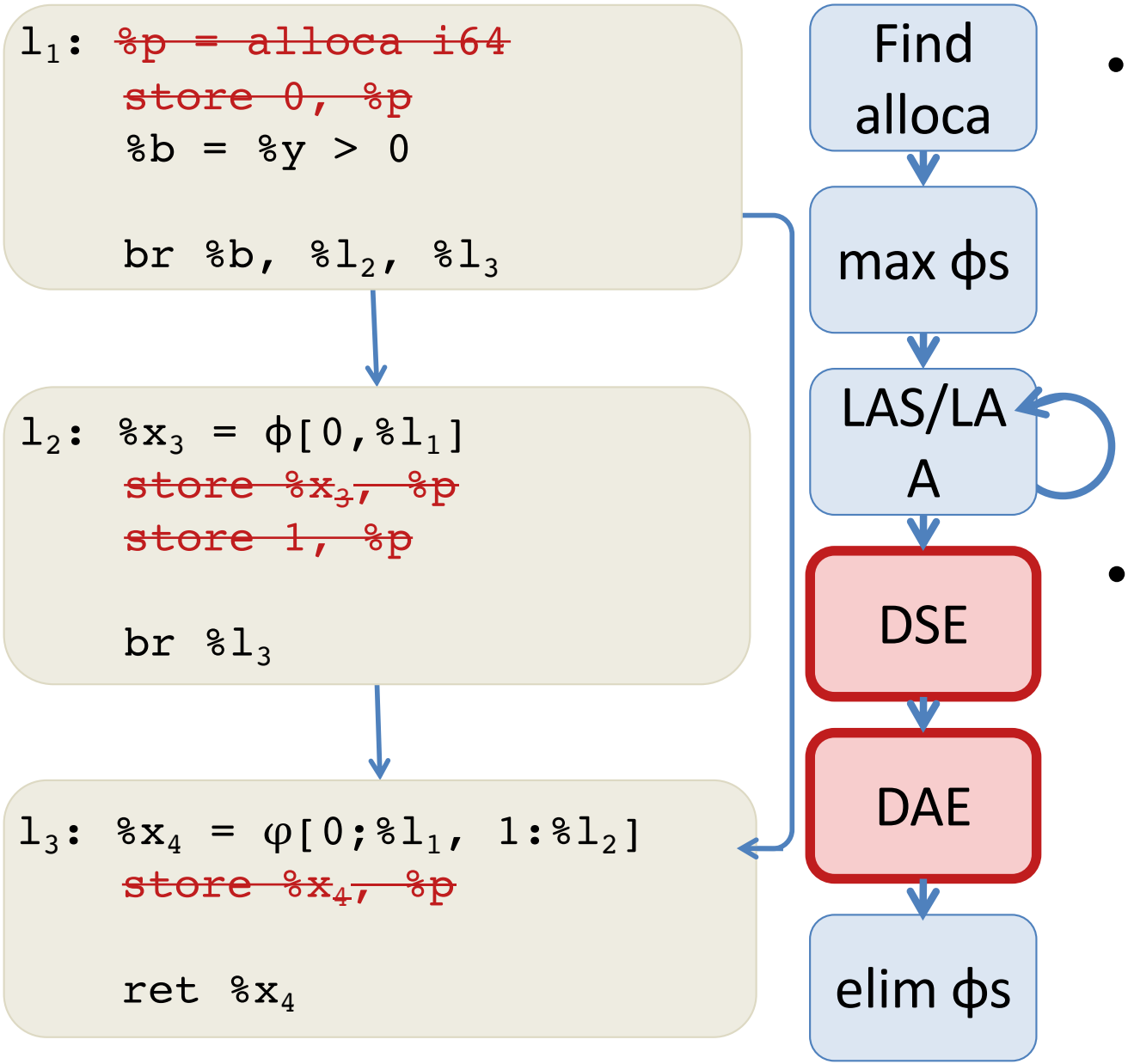
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



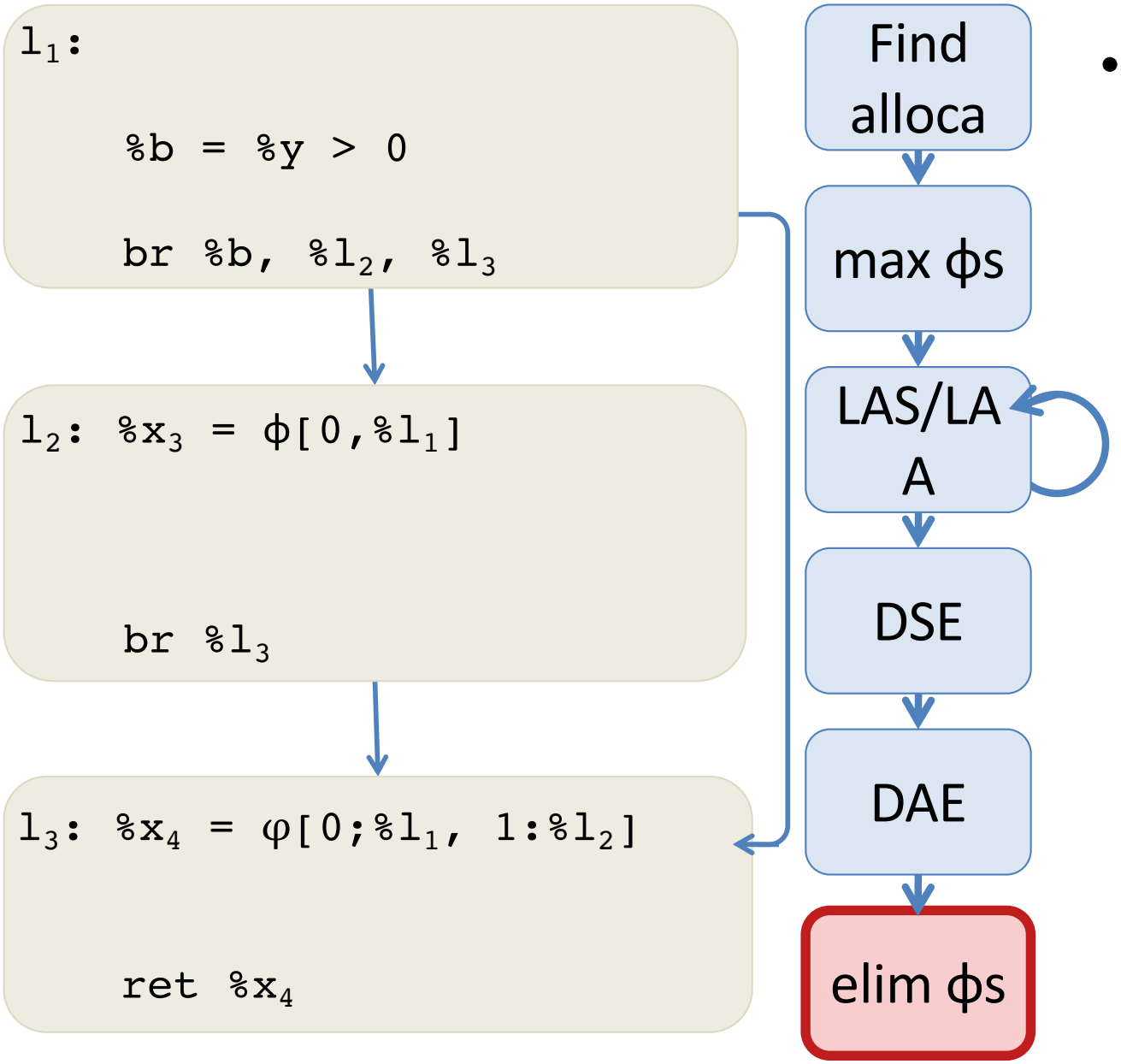
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations



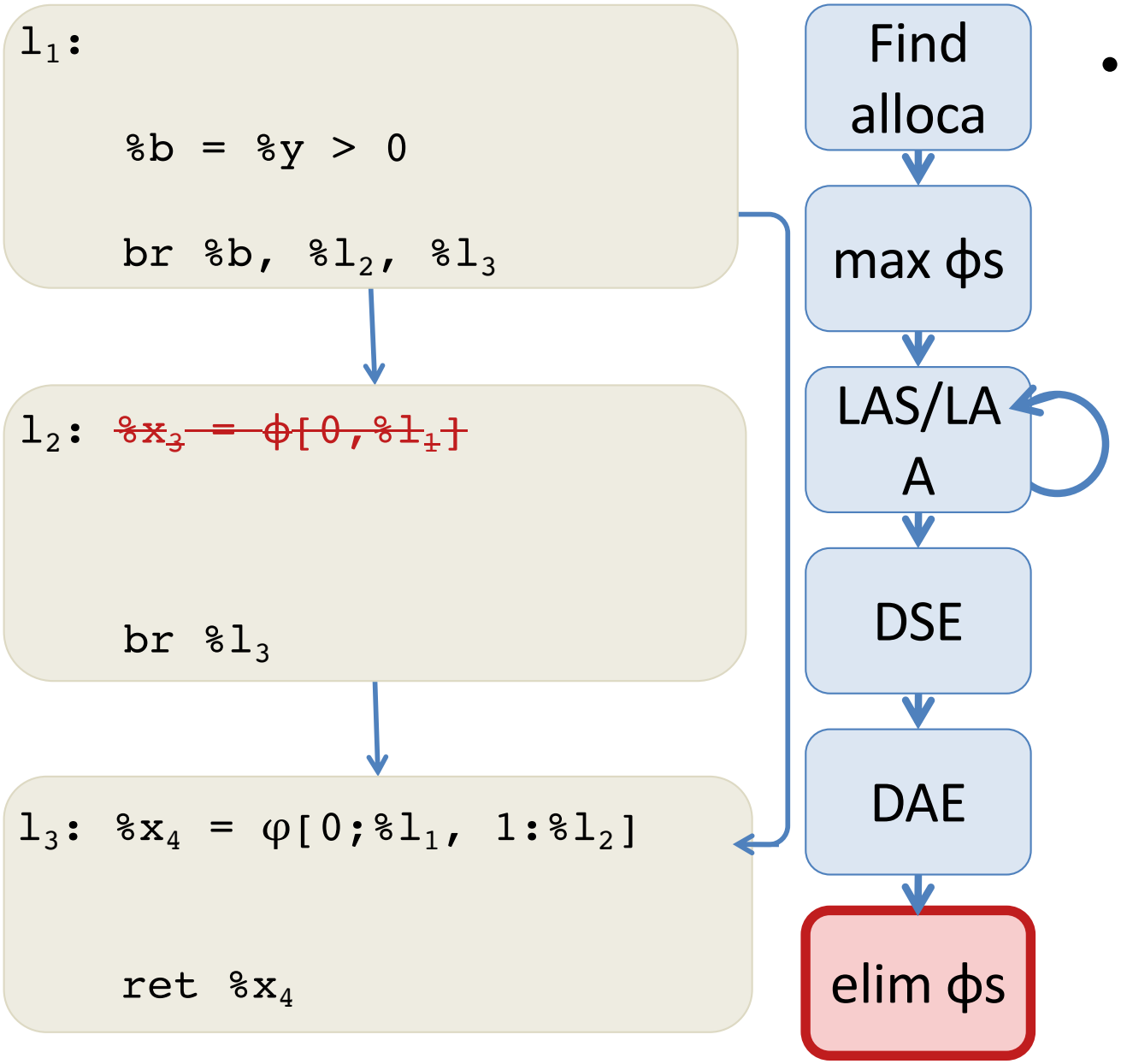
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations



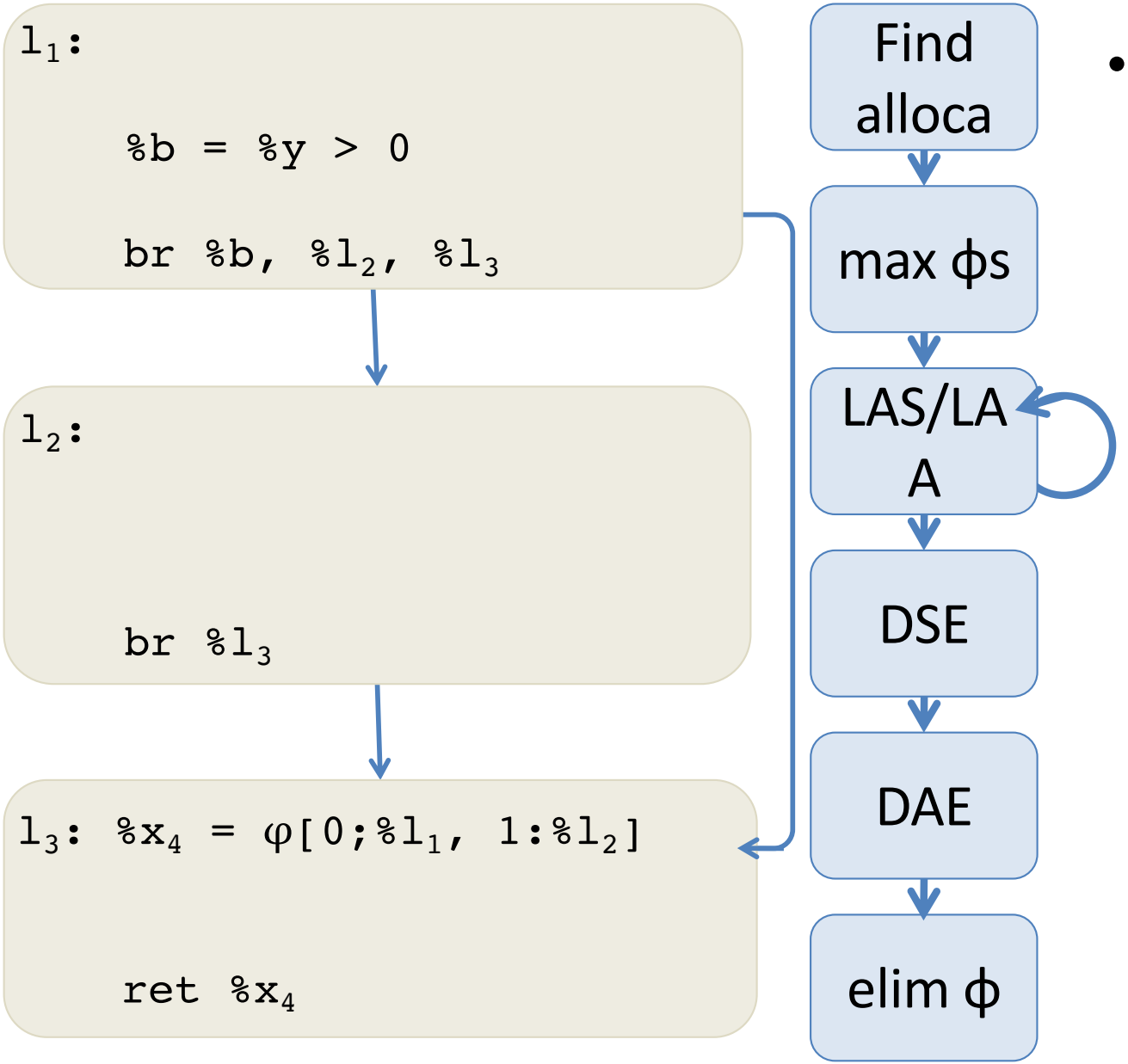
- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Example SSA Optimizations



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

Example SSA Optimizations



- Done!

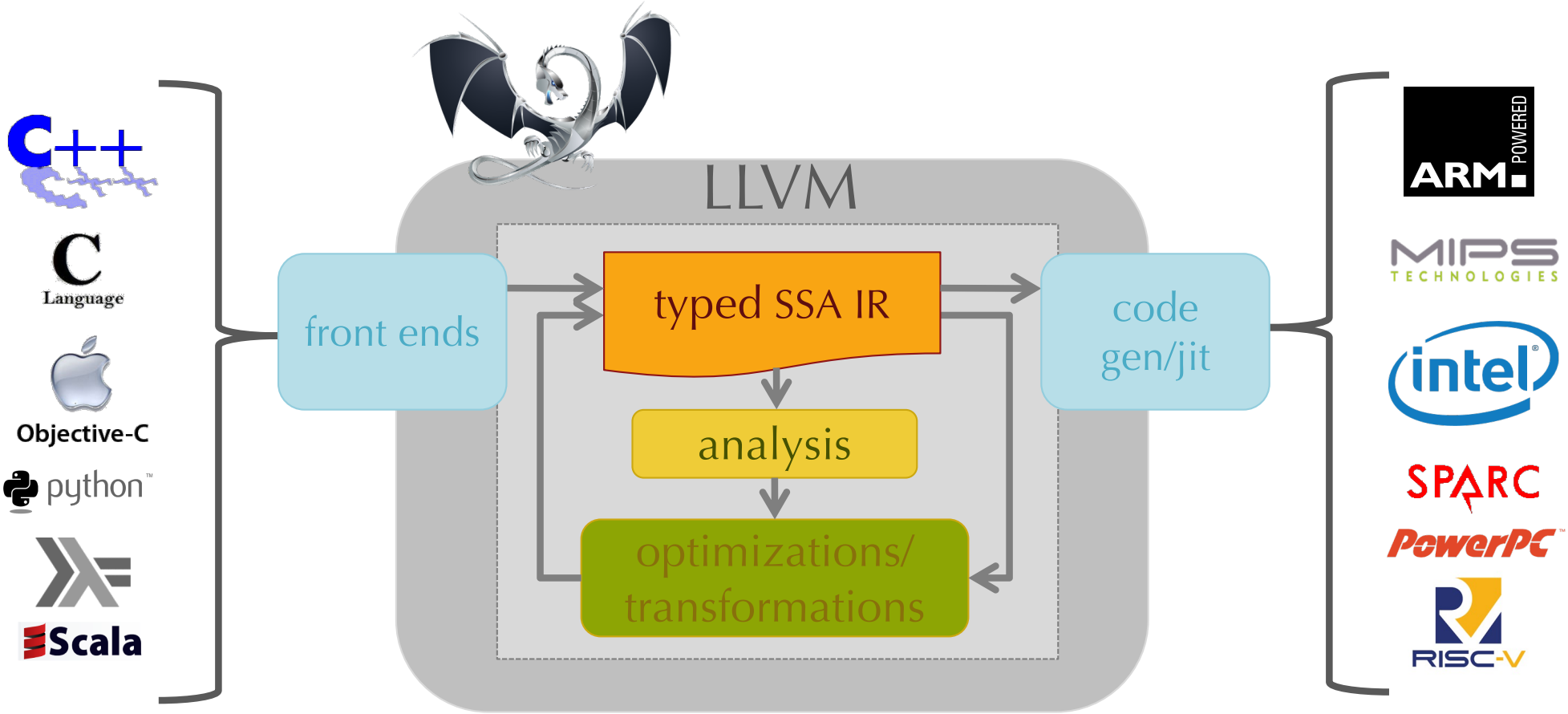
LLVM Phi Placement

- 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.]

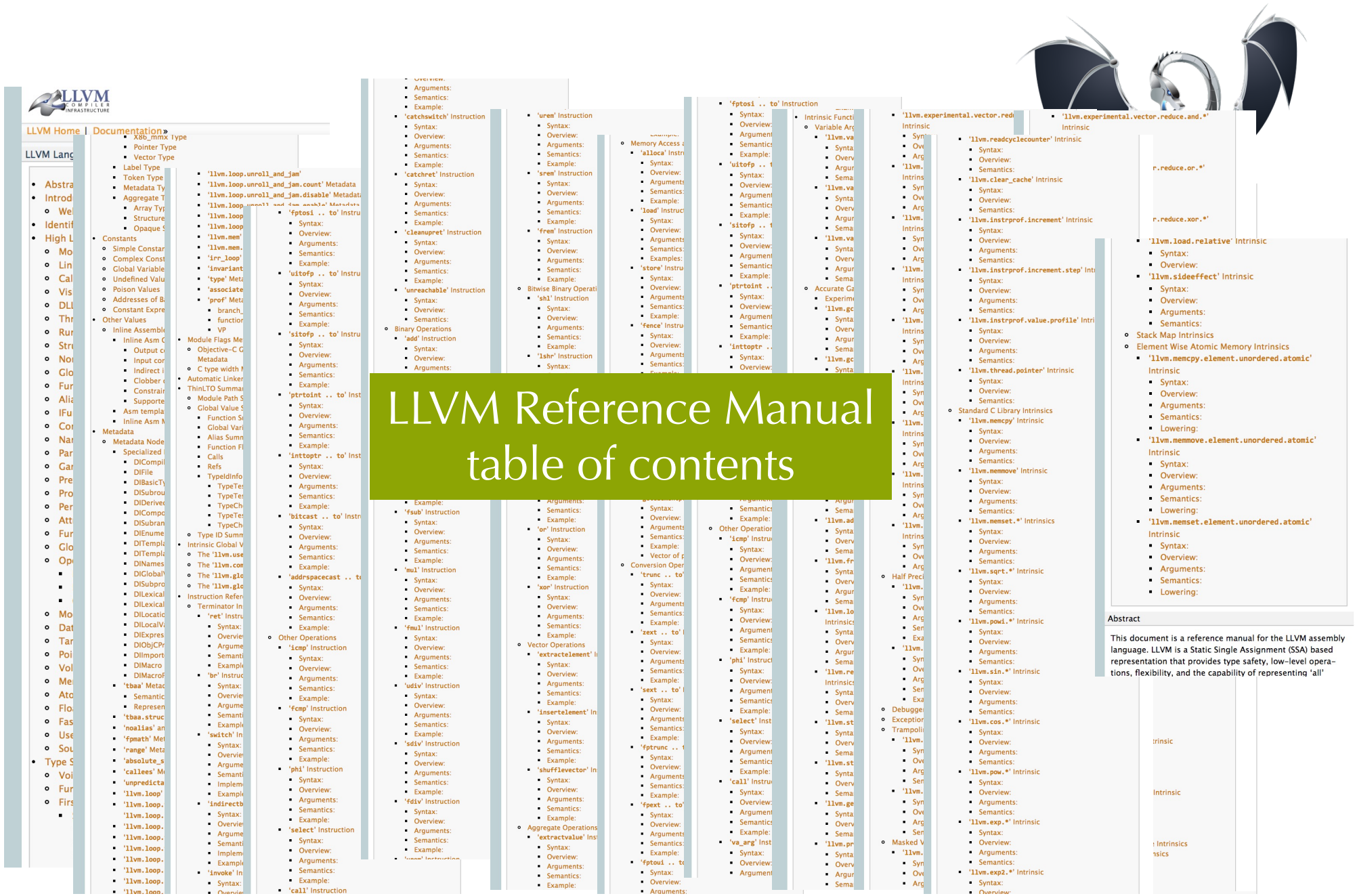


Other LLVM IR Features

- 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:

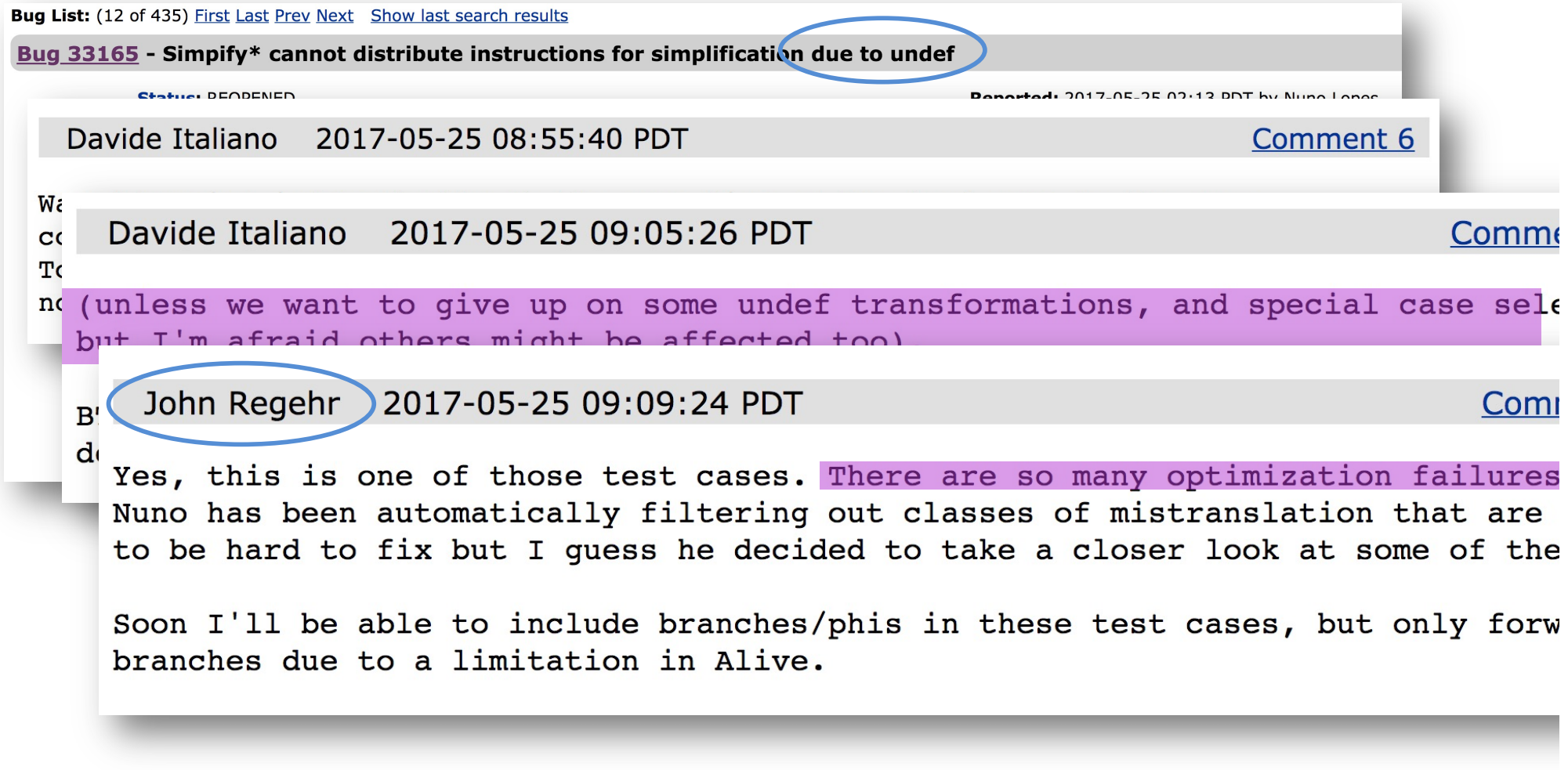
```
%y = mul i8 %x, 2
```

versus:

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

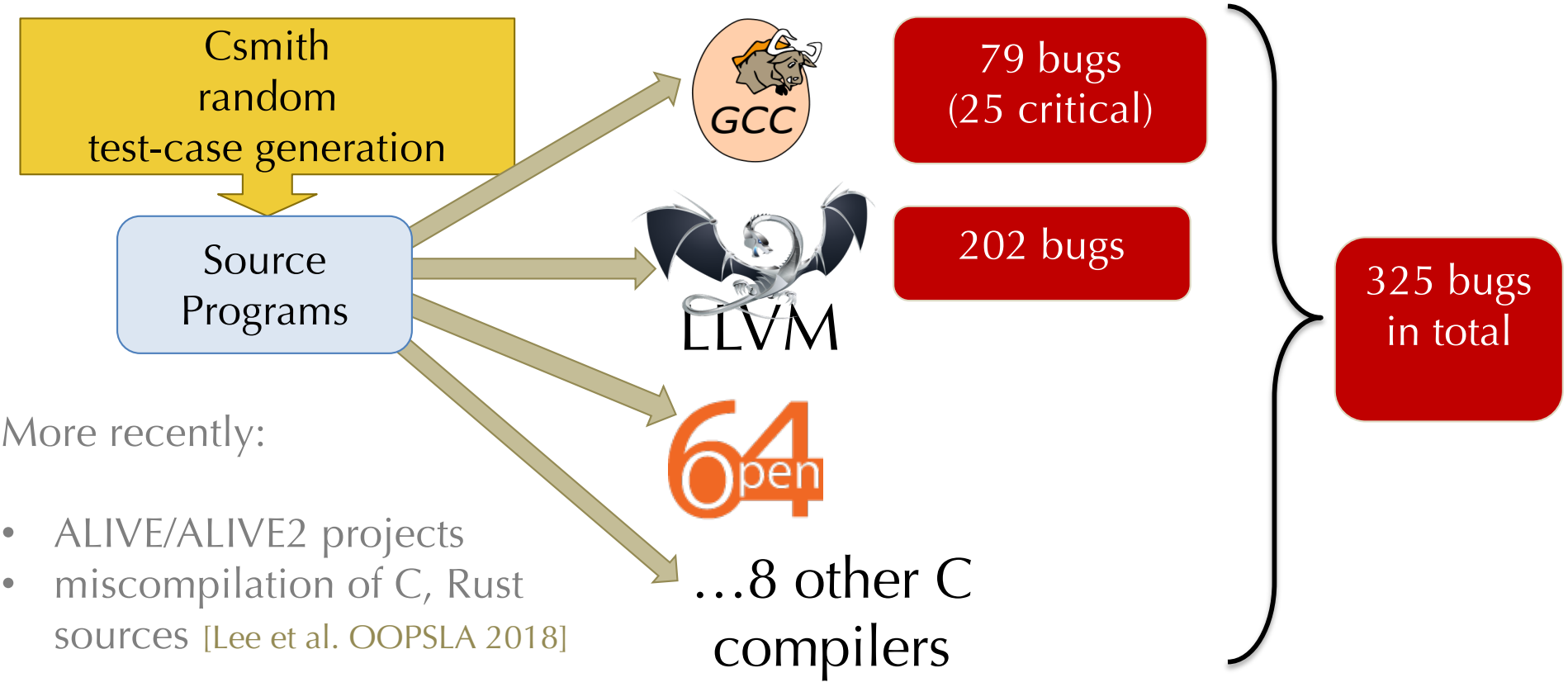
Upshot: if **%x** is **undef**, we can't optimize **mul** to **add** (or vice versa)!

What's the problem?



Compiler Bugs

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



More recently:

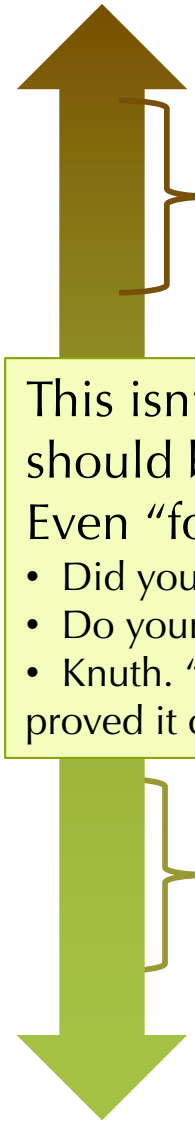
- ALIVE/ALIVE2 projects
- miscompilation of C, Rust sources [Lee et al. OOPSLA 2018]

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:

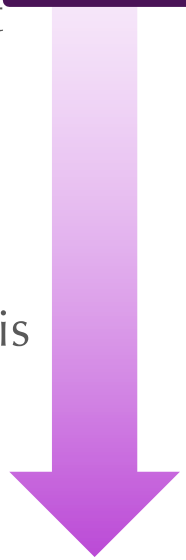
- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."

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

Goal: Verified Software Correctness

- Social
  - Code reviews
  - Extreme/Pair programmin
- 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?



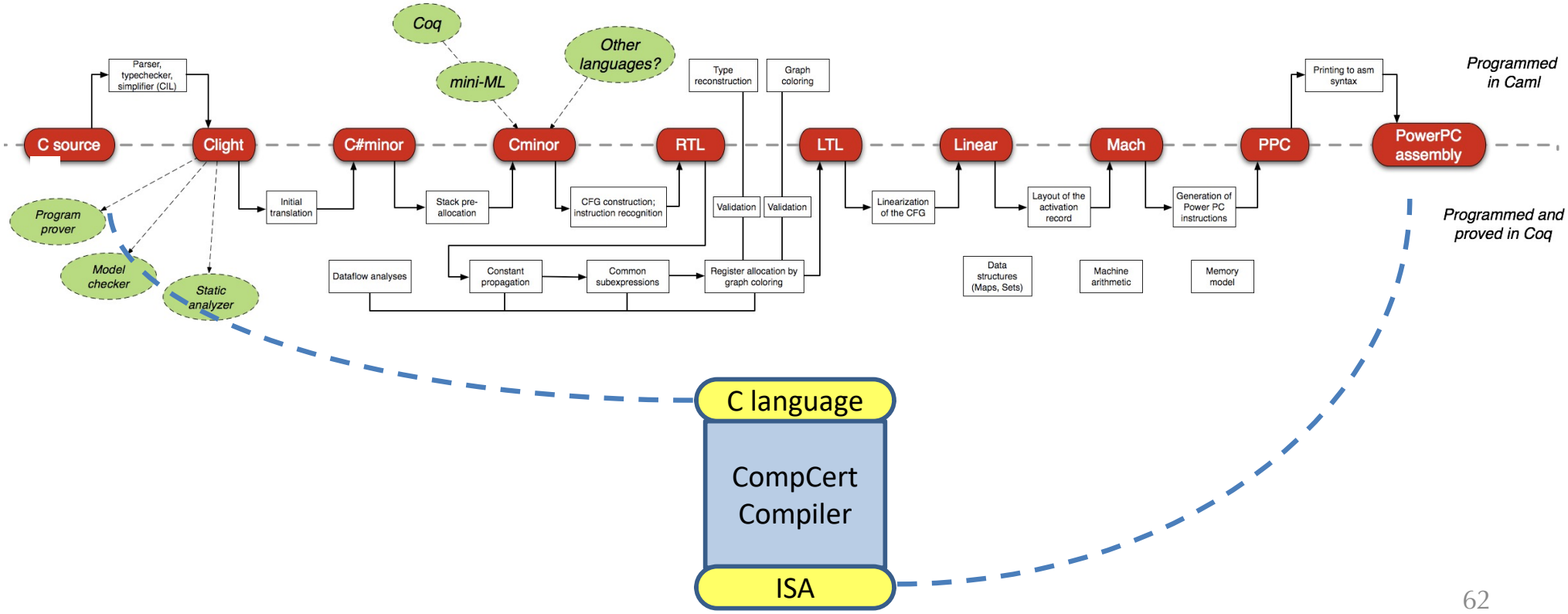
Taking advantage of advances in computer science:

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

CompCert – A Verified C Compiler

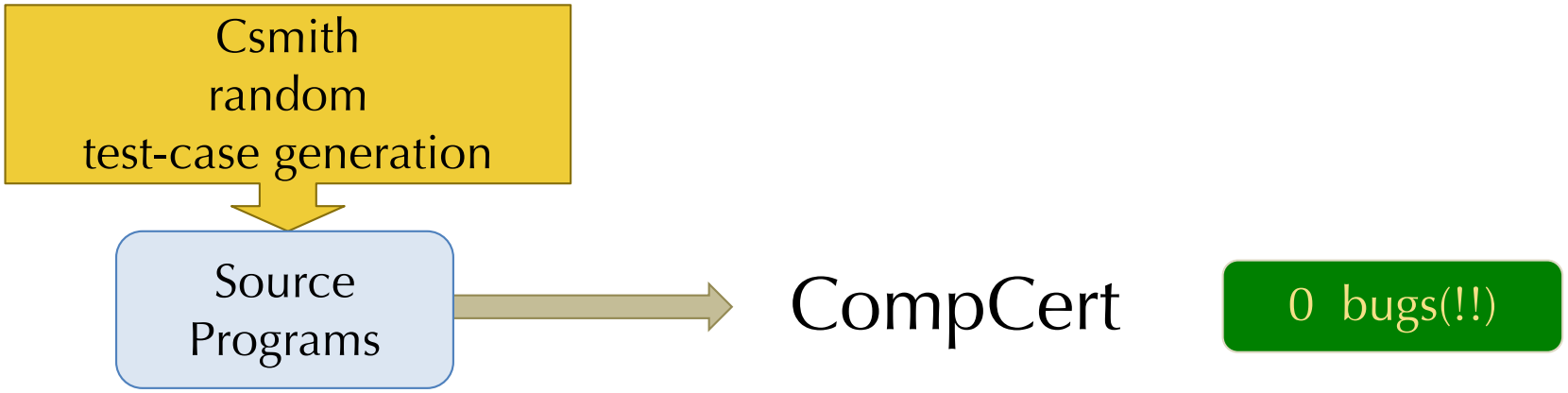
Xavier Leroy
INRIA

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

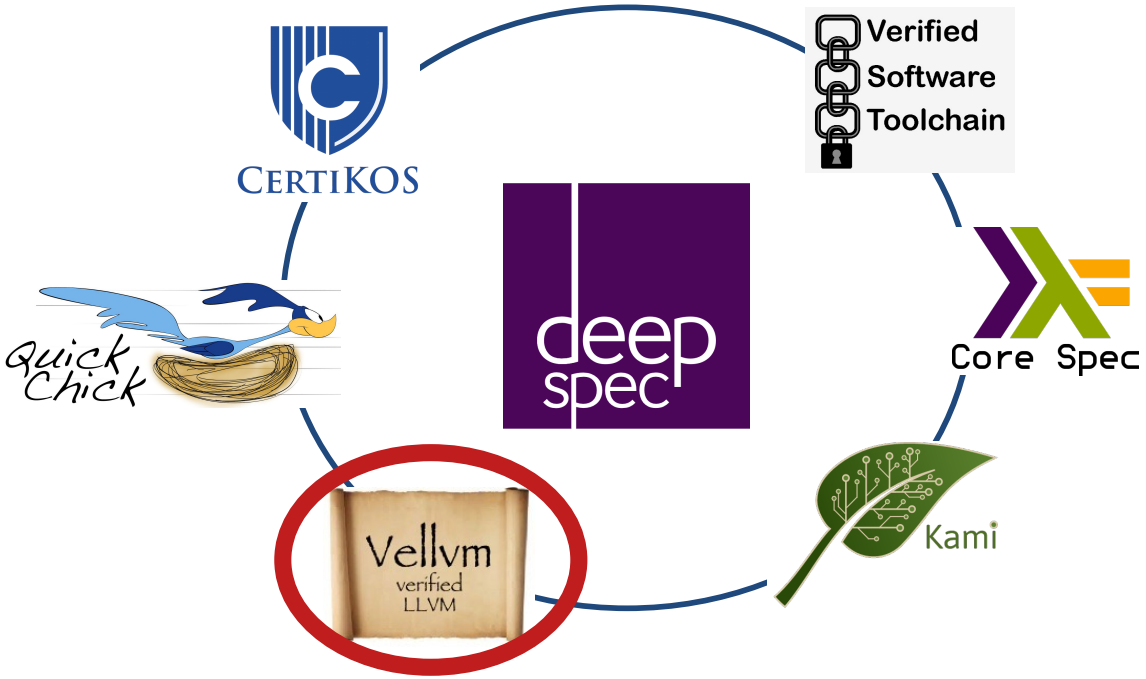
deepspec.org

Deep Specifications

[deepspec.org]

- ***Rich*** – expressive description
- ***Formal*** – mathematical, machine-checked
- ***2-Sided*** – tested from both sides
- ***Live*** – connected to real, executable code

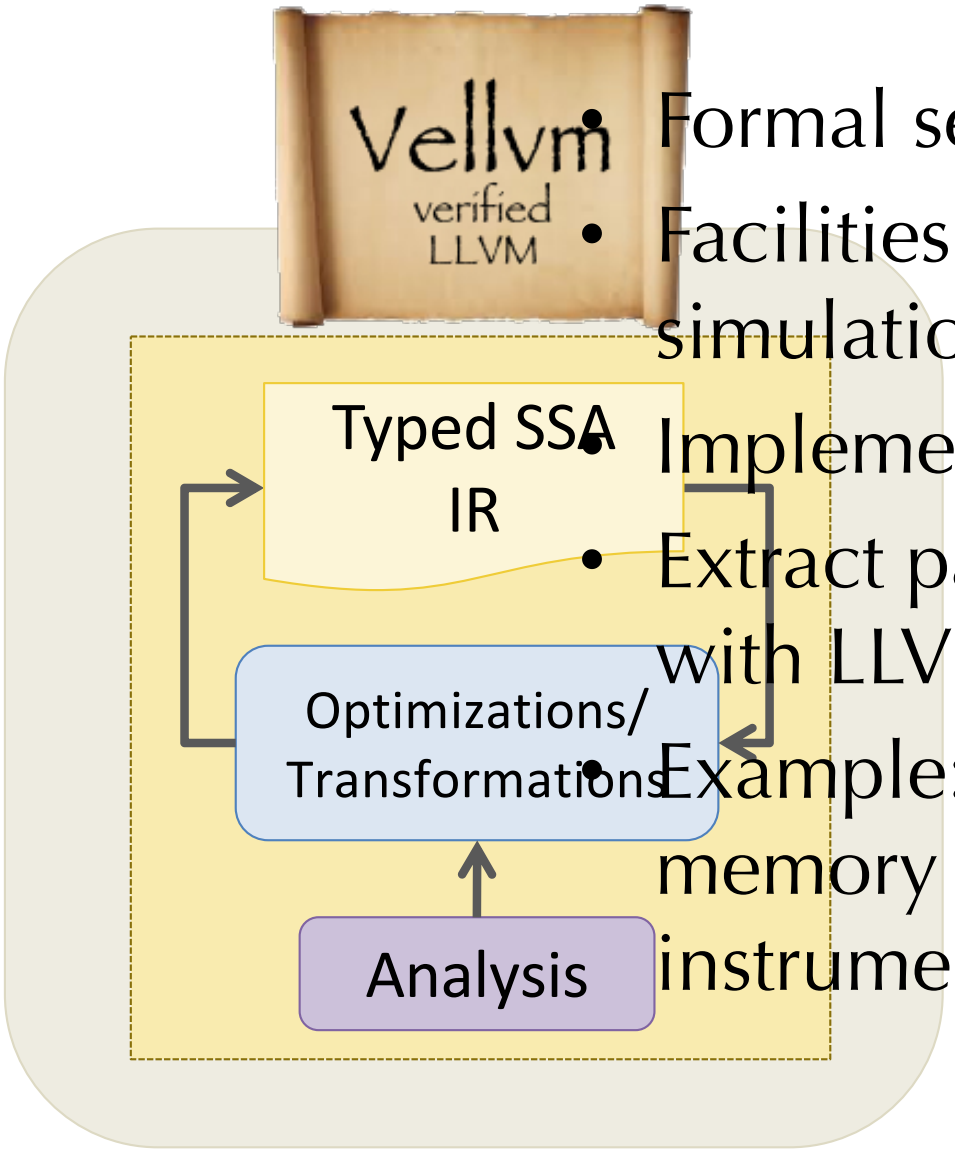
Goal: Advance the reliability, safety, security, and cost-effectiveness of software (and hardware).



deepspec.org

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

Vellvm Framework

Coq

Type System and SSA

Operational Semantics

Syntax

Memory Model

Proof Techniques & Metatheory

Extract

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

Caml

Parser

Printer

LLVM

C Source Code

LLVM IR

LLVM IR

Other Optimizations

Target

Writing Interpreters in Coq

Galina (Coq's Language)

- rich, dependent type system
- pure, total functional language

How do we write the interpretation function?

```
Inductive exp : Set :=
| EXP_Ident   (id:ident)
| EXP_Integer (x:int)
| EX
| ..
```

```
Inductive instr : Set :=
| INSTR_Op   (op:exp)                                   (* IN
| INSTR_Call (fn:texp) (args:list texp)                 (* CO
| INSTR_Alloca (t:typ) (nb: option texp) (align:
| INSTR_
| INSTR_
```

```
Definition code := list (instr_id * instr).

Record block : Set :=
  mk_block
    {
      blk_id    : block_id;
      blk_phis  : list (local_id * phi);
      blk_code  : code;
      blk_term  : instr_id * terminator;
    }.
```

Datatypes for Abstract Syntax

LLMV Memory Model (simplified)

```
(* IO interactions for the LLVM IR *)
Inductive IO : Type -> Type :=
| Alloca : ∀ (t:dtyp), (IO dvalue)
| Load   : ∀ (t:dtyp) (a:dvalue), (IO dvalue)
| Store  : ∀ (a:dvalue) (v:dvalue), (IO unit)
| GEP    : ∀ (t:dtyp) (v:dvalue) (vs:list dvalue), (IO dvalue)
| ItoP   : ∀ (i:dvalue), (IO dvalue)
| PtoI   : ∀ (a:dvalue), (IO dvalue)
| Call   : ∀ (f:string) (args:list dvalue), (IO dvalue)
.
```

Describes the interface for "observations" of LLVM IR programs.

output values of the Call event

type of the result provided by the environment

LLVM Interpreter in Coq

```
Definition step (s:state) : LLVMTrace result
  let '(g, pc, e, k) := s in

  do cmd ← trywith ("CFG has no instruction at " ++ string_of
  match cmd with
  | Term (TERM_Ret (t, op)) ⇒
    'dv ← eval_exp (Some (eval_typ t)) op;
     match k with
     | [] ⇒ halt dv
     | (KRet e' id p') :: k' ⇒ cont (g, p', add_env id dv e', k'
     | _ ⇒ raise_p pc "IMPOSSIBLE: Ret op in non-return configur
     end

  | Inst insn ⇒  (* instruction *)
    do pc_next ← trywith "no fallthrough instruction" (incr_pc CFG pc);
    match (pt pc), insn  with

      | IId id, INSTR_Op op ⇒
         'dv ← eval_op g e op;
          cont (g, pc_next, add_env id dv e, k)

      | IId id, INSTR_Alloca t _ _ ⇒
        Trace.Vis (Alloca (eval_typ t))
                  (λ (a:dvalue) ⇒ cont (g, pc_next, add_env id a e, k))

      | IId id, INSTR_Load _ t (u,ptr) _ ⇒
        'dv ← eval_exp (Some (eval_typ u)) ptr;
         Trace.Vis (Load (eval_typ t) dv)
                   (λ dv ⇒ cont (g, pc_next, add_env id dv e, k))
```

interpreter returns an interaction tree with "LLVM" effects. LLVMTrace := itree IO

Extract to executable interpreter (Ocaml).

The interpreter "calls out" to the memory model by generating visible effects…

So What?

- Find bugs in the existing LLVM infrastructure
  - thinking hard about corner cases while formalizing is a good way to find real bugs
  - identify inconsistent assumptions on the LLVM compiler
- Automated Tests against other implementations
  - e.g., integrate with Csmith
- Formally validate program transformations
  - is a particular optimization correct?
  - improve confidence in novel program transformations
- Eventually… verify compiler front ends and/or back ends
  - to obtain a fully-verified CompCert-like compiler

Interactive Theorem Proving

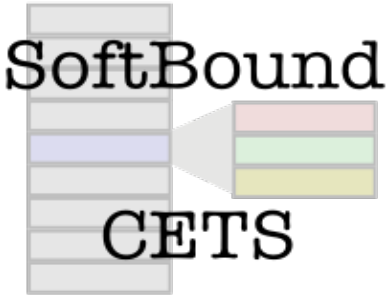
In Coq, one can state Lemmas just as easily as any other kind of function.

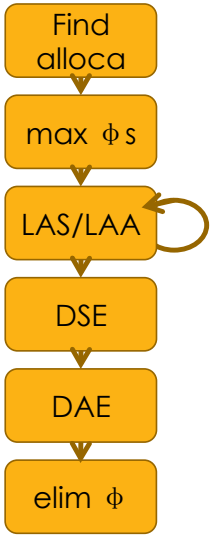
```
Theorem block_fusion_cfg_correct :
  ∀ (G : cfg dtyp),
    wf_cfg G →
    ⟦ G ⟧cfg ≈ ⟦ block_fusion_cfg G ⟧cfg.
Proof.
  intros G [WF₁ WF₂].
  unfold denote_cfg.
  simpl bind.
  unfold block_fusion_cfg.
  destruct (b   k_fusion G.(blks))                                   :EQ.
  - break_match_goa    [reflexivit
    simpl.
    apply Bool.orb_false_elim in
    unfold Eqv.eqv_dec in *.
```

You can prove those lemmas interactively. Coq checks each step as you do it.

VELLVM [Previous Results]

- Verified SoftBound
  - Memory Safety
- Verified mem2reg
  - Register promotion, defined in terms of a stack of "micro-optimizations"
- Verified dominator analysis
  - Cooper-Harvey-Kennedy Algorithm
- Better memory models
  - ptrtoint casts
  - modular formalization





Can it Scale?

- Use of theorem proving to verify "real" software is still considered to be the bleeding edge of research.

- **CompCert** – fully verified C compiler
  Leroy, INRIA
- **Vellvm** – formalized LLVM IR
  Zdancewic, Penn
- **Ynot** – verified DBMS, web services
  Morrisett, Harvard
- **Verified Software Toolchain**
  Appel, Princeton
- **Bedrock** – web programming, packet filters
  Chlipala, MIT
- **CertiKOS** – certified OS kernel
  Shao & Ford, Yale
- **CakeML** – certified compiler
- **SEL4** – certified secure OS microkernel
- **Kami** – verified RISCV architecture
- **DaisyNSF** – verified NFS file system
- …

Where next?

- Proof engineering is still nascent
  - automation, scale, maintenance
  - software engineering++
  - new theory needed: dealing with equality
- Verification is still hard
  - labor intensive, difficult, $$$$
- Deep Specifications
  - what are the principles?
  - compositionality?
- Real-time, cyberphysical,…