Lecture 26

CS 131: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: December 30th
- Final Exam:
  - In class, Jan 2nd
  - Coverage: emphasizes material since the midterm
  - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

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”

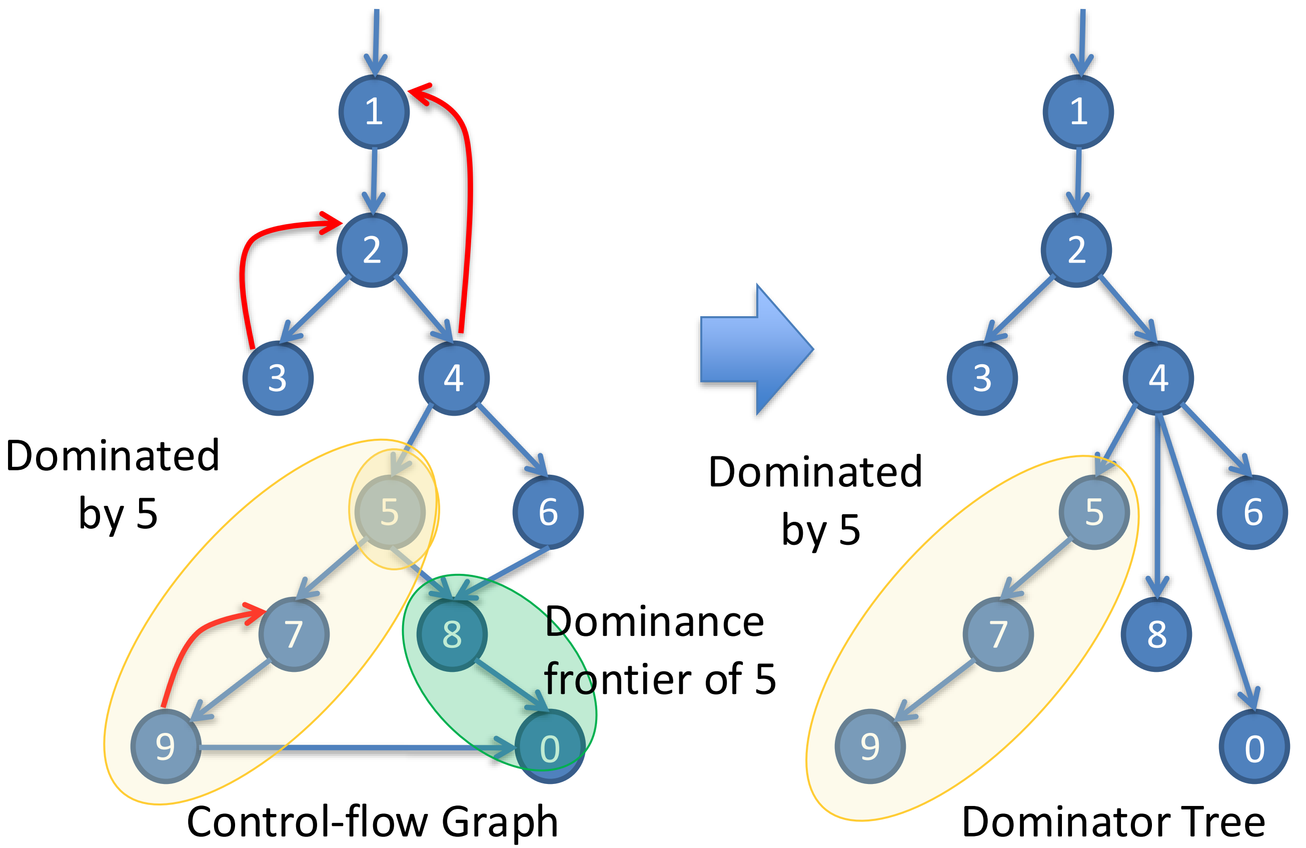
- Node A ***strictly dominates*** node B if A dominates B and A ≠ B.
  - Note: A does not strictly dominate B if A does not dominate B or A = B.

- 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].
  - $x = \phi(x, x, \ldots, x)$; (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?

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

  $= DF[DF_0\{3,6\} \cup \{3,6\}]$

  $= DF[\{2,3,6,8\}]$

  $= DF[2] \cup DF[3] \cup DF[6] \cup DF[8]$

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

  = ...

  $= \{1,2,8,0\}$

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

Phi Placement (Alternative)

- Less efficient than LLVM's true "dominance frontier" algorithm, 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

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

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

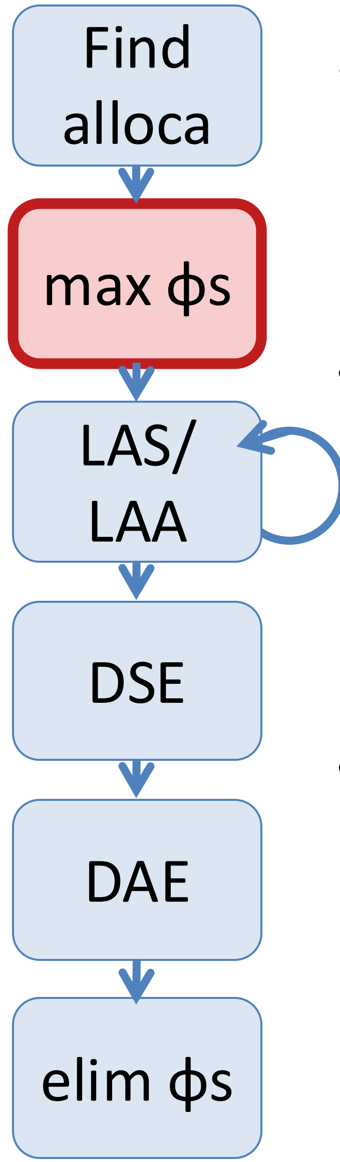
```
l2:

    store 1, %p

    br %l3
```

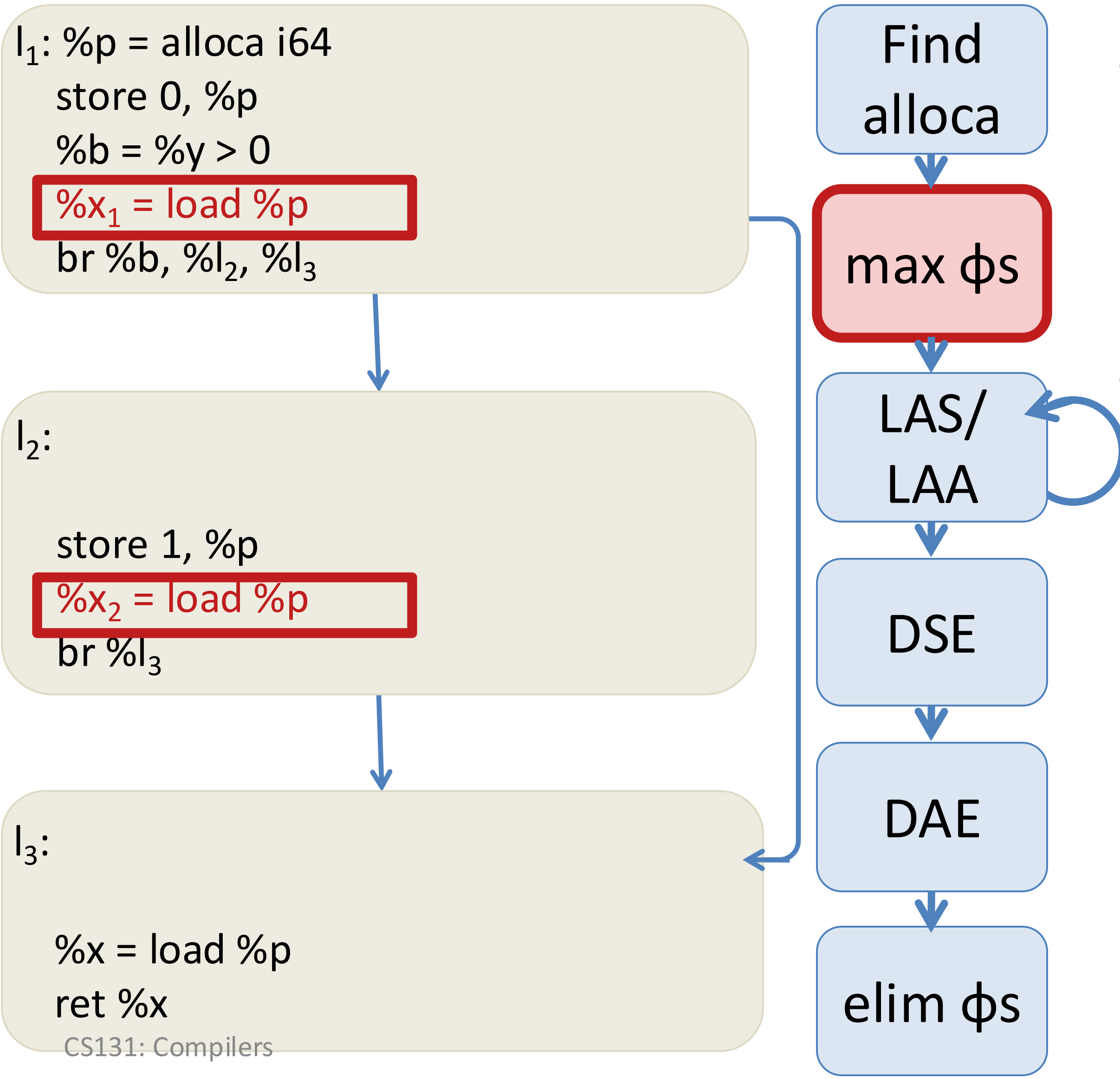
```
l3:

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



- How to place phi nodes without breaking SSA?
- 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

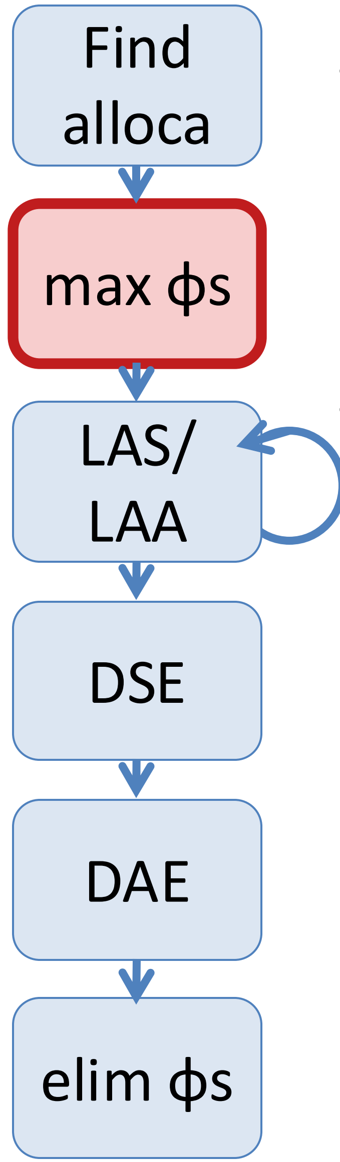
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

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
```



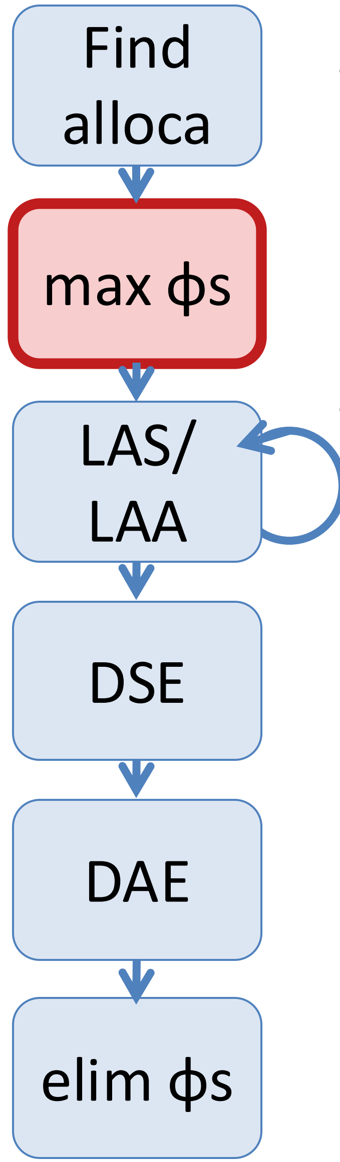
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block

Example SSA Optimizations

l_1: %p = alloca i64
store 0, %p
%b = %y > 0
$\%x_1$ = load %p
br %b, $\%l_2$, $\%l_3$

l_2: $\%x_3 = \phi[\%x_1, \%l_1]$
$\boxed{\text{store } \%x_3\text{, } \%p}$
store 1, %p
$\%x_2$ = load %p
br $\%l_3$

l_3: $\%x_4 = \phi[\%x_1;\%l_1, \%x_2:\%l_2]$
$\boxed{\text{store } \%x_4\text{, } \%p}$
%x = load %p
ret %x



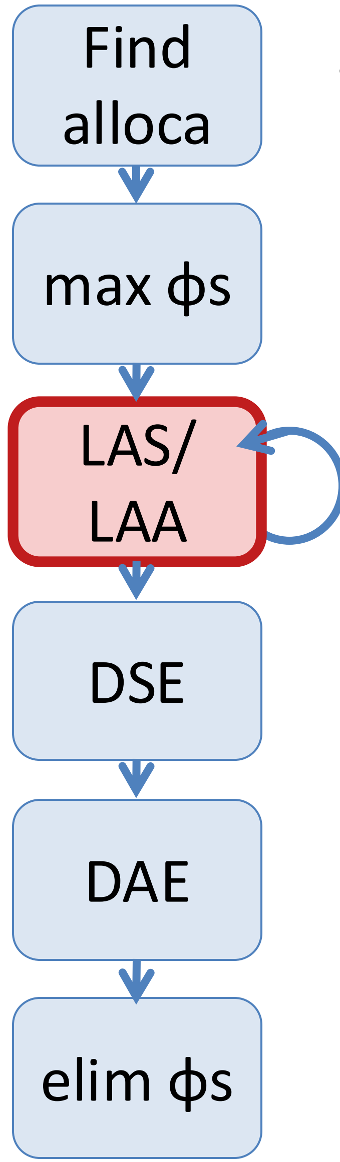
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block
  - Insert stores after ϕ-nodes

Example SSA Optimizations

```
l1: %p = alloca i64
    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
```



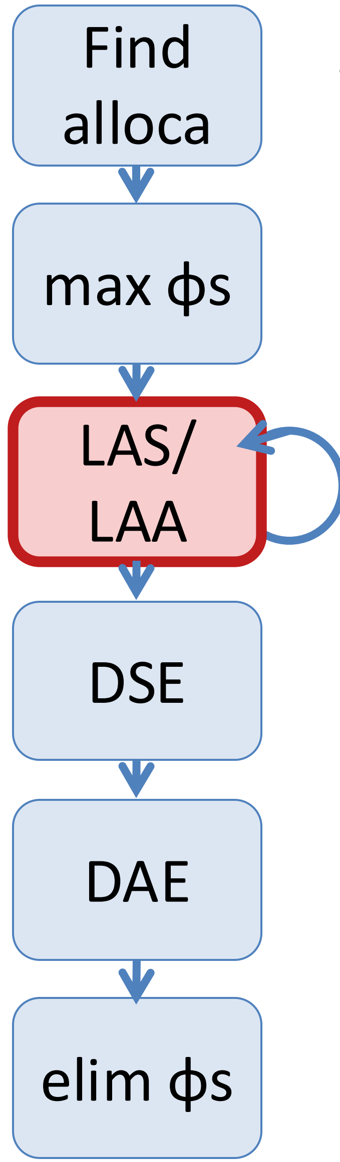
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    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 SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    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
```

Find alloca → max ϕs → LAS/LAA → DSE → DAE → elim ϕs

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

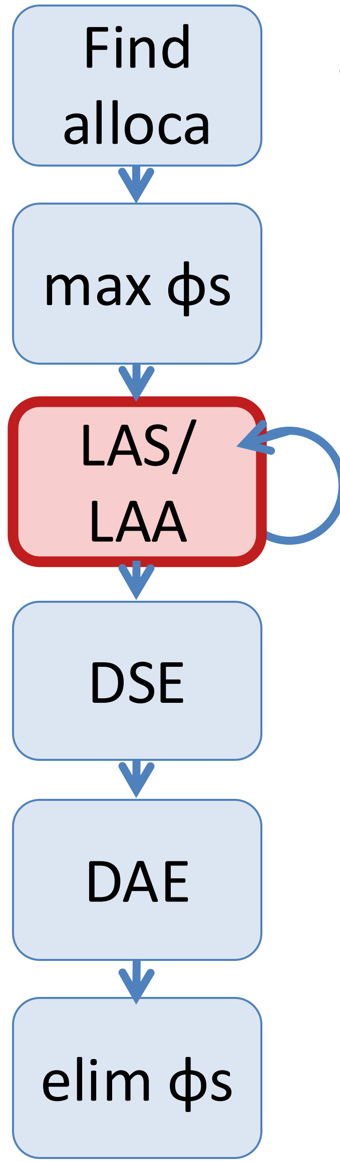
Example SSA Optimizations

```
l1: %p = alloca i64
    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 SSA Optimizations

l_1: %p = alloca i64
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
~~$\%x_2$ = load %p~~
br $\%l_3$

l_3: $\%x_4 = \phi[0;\%l_1, 1{:}\%l_2]$
store $\%x_4$, %p
%x = load %p
ret %x

Find alloca
max ϕs
LAS/LAA
DSE
DAE
elim ϕs

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

Example SSA Optimizations

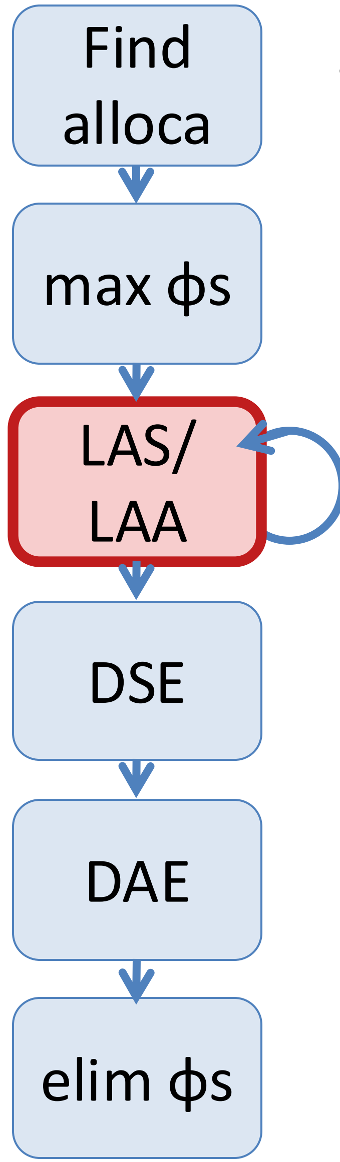
l_1: %p = alloca i64
store 0, %p
%b = %y > 0

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
%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 SSA Optimizations

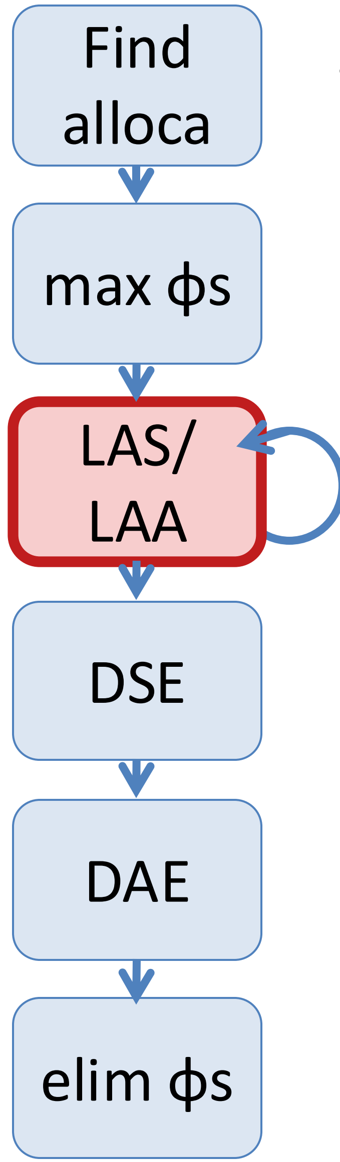
l_1: %p = alloca i64
store 0, %p
%b = %y > 0

br %b, %l_2, %l_3

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

br %l_3

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



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

Example SSA Optimizations

l_1: %p = alloca i64
store 0, %p
%b = %y > 0

br %b, %l_2, %l_3

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

br %l_3

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

ret %x_4

Find alloca → max ϕs → LAS/LAA → DSE → DAE → elim ϕs

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

Example SSA Optimizations

l_1: ~~%p = alloca i64~~
~~store 0, %p~~
%b = %y > 0

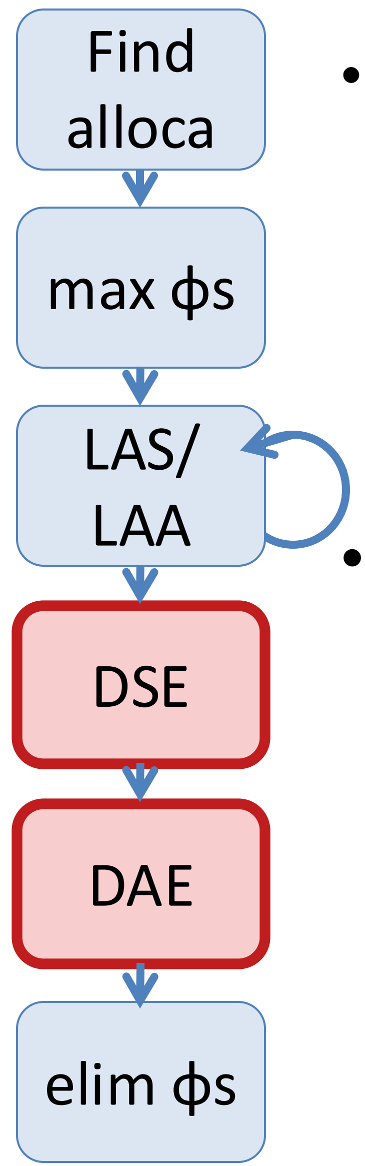
br %b, $\%l_2$, $\%l_3$

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

br $\%l_3$

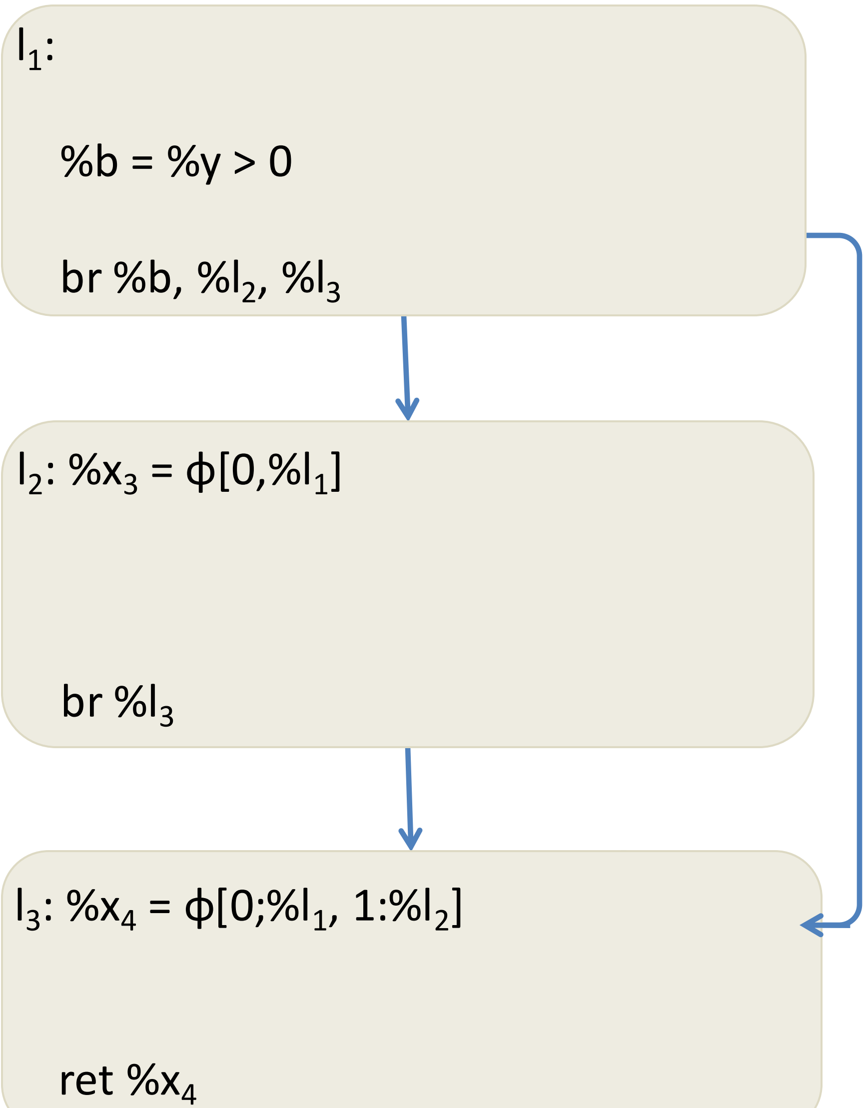
l_3: $\%x_4 = \phi[0;\%l_1, 1:\%l_2]$
~~store $\%x_4$, %p~~

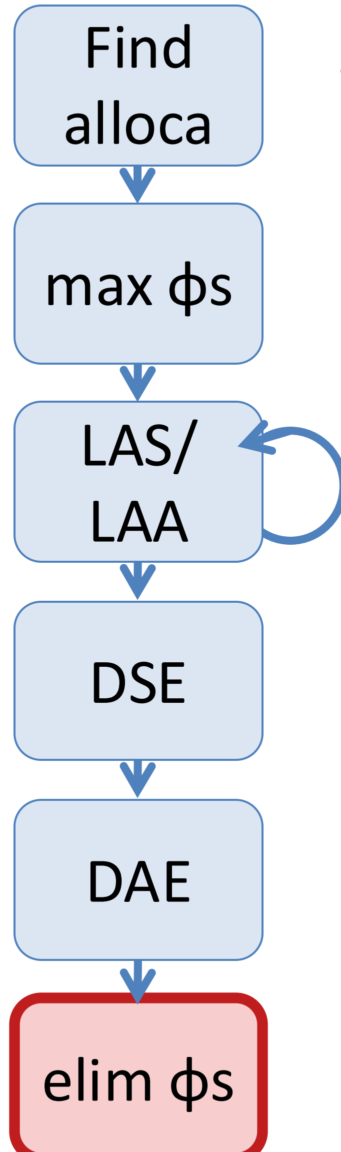
ret $\%x_4$



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

Example SSA Optimizations





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

Example SSA Optimizations

l_1:

%b = %y > 0

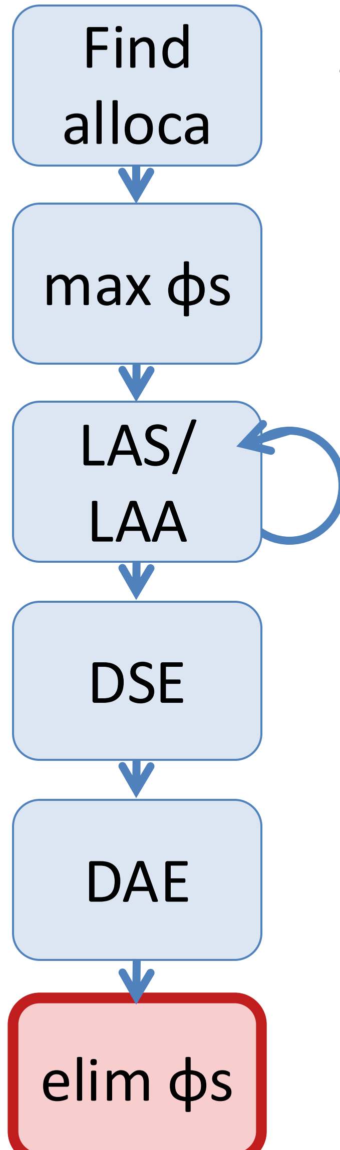
br %b, $\%l_2$, $\%l_3$

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

br $\%l_3$

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

ret $\%x_4$



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

Example SSA Optimizations

l_1:

%b = %y > 0

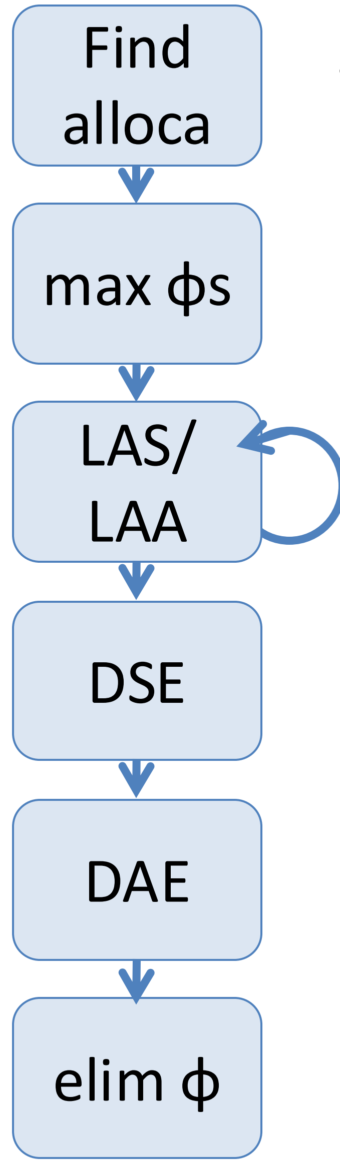
br %b, $\%l_2$, $\%l_3$

l_2:

br $\%l_3$

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

ret $\%x_4$



- Done!

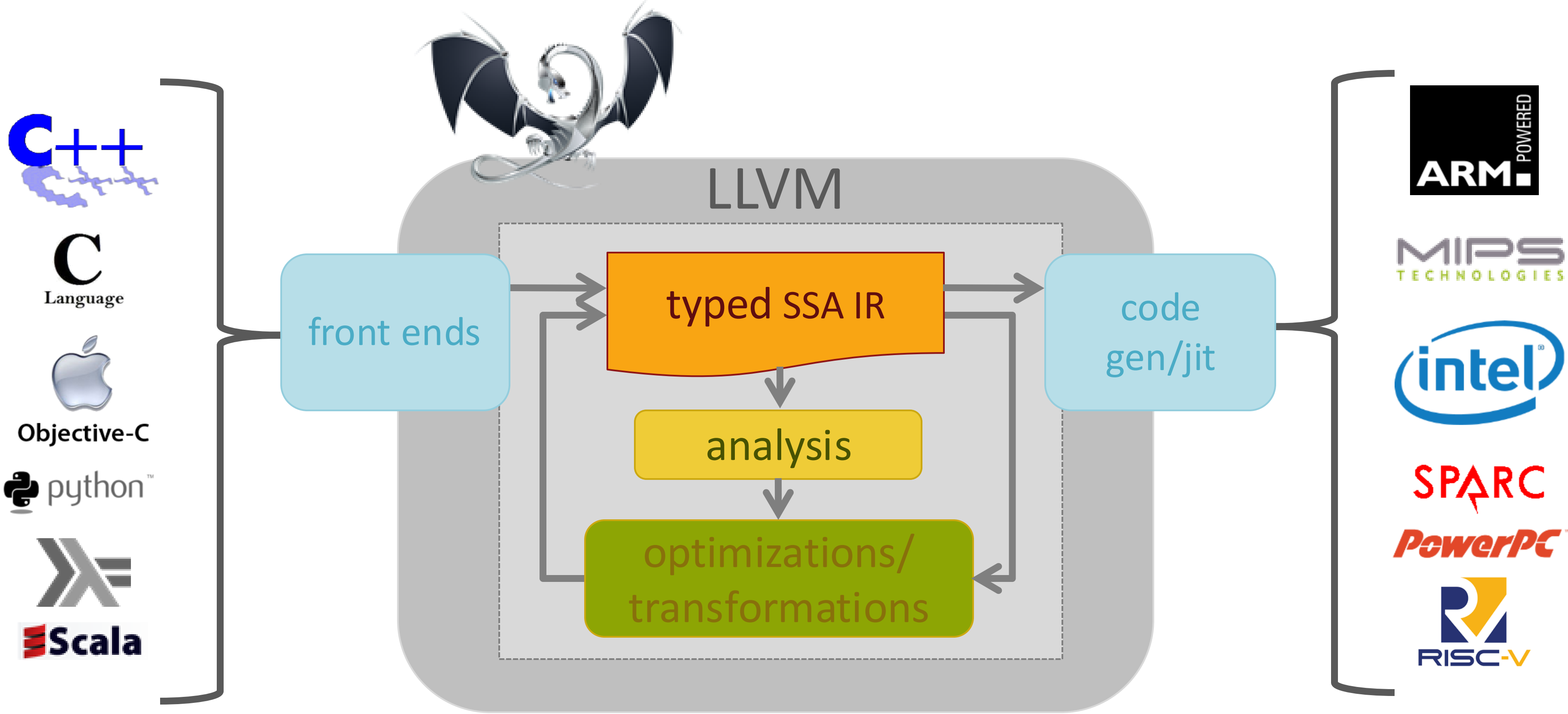
LLVM Phi Placement

- 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.]

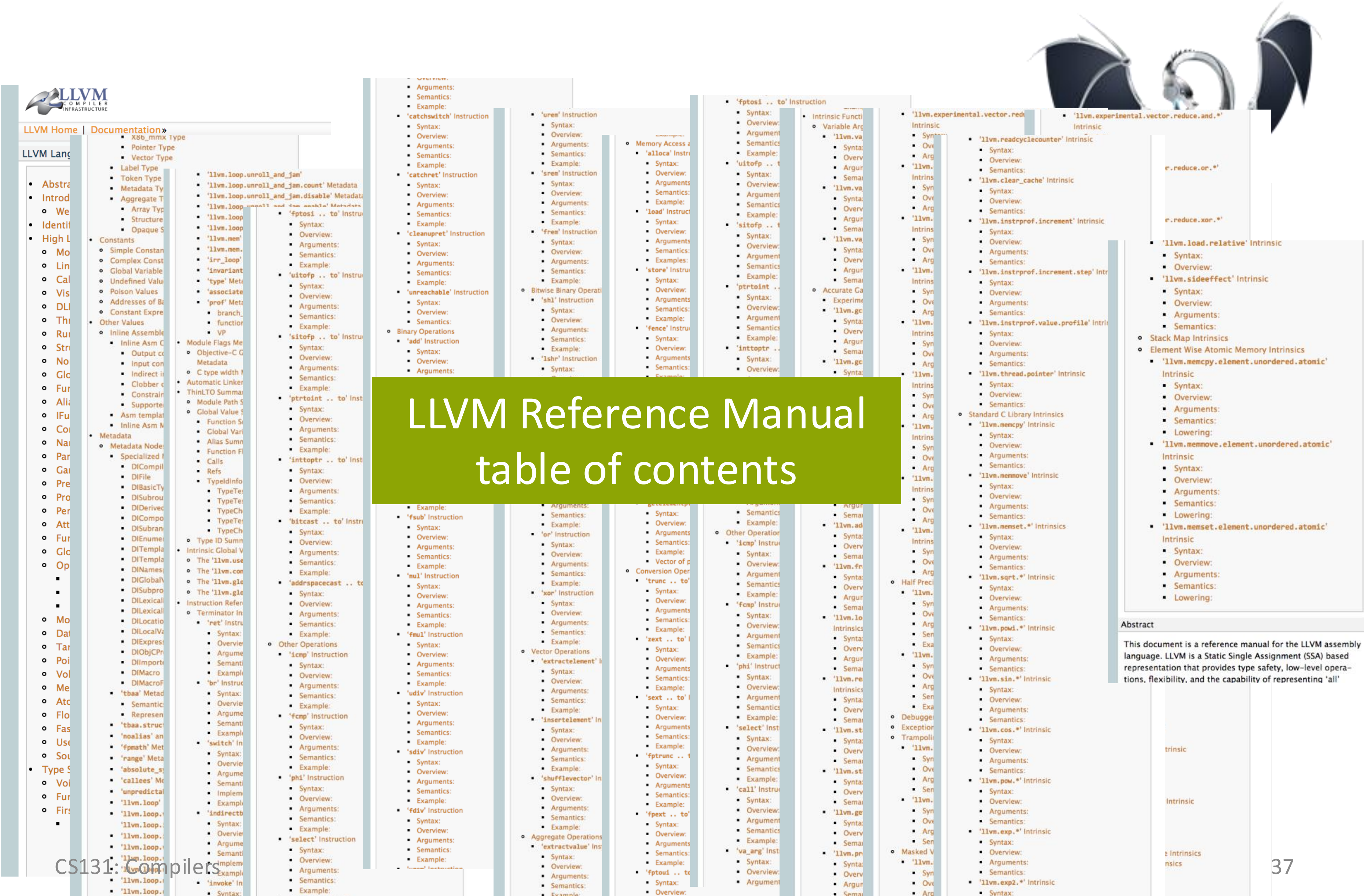


Other LLVM IR Features

- 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



LLVM Reference Manual table of contents

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
```

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

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

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

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

Interactions with Optimizations

Consider:

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

versus:

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

$\neq$

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

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

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?

Bug List: (12 of 435) First Last Prev Next Show last search results

Bug 33165 - Simpify* cannot distribute instructions for simplification due to undef

Davide Italiano 2017-05-25 08:55:40 PDT Comment 6

Davide Italiano 2017-05-25 09:05:26 PDT Comme

(unless we want to give up on some undef transformations, and special case sele

John Regehr 2017-05-25 09:09:24 PDT Comm

Yes, this is one of those test cases. There are so many optimization failures
Nuno has been automatically filtering out classes of mistranslation that are
to be hard to fix but I guess he decided to take a closer look at some of the

Soon I'll be able to include branches/phis in these test cases, but only forw
branches due to a limitation in Alive.

Compiler Bugs

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

Csmith
random
test-case generation

Source
Programs

GCC

LLVM

…8 other C
compilers

79 bugs
(25 critical)

202 bugs

325 bugs
in total

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?

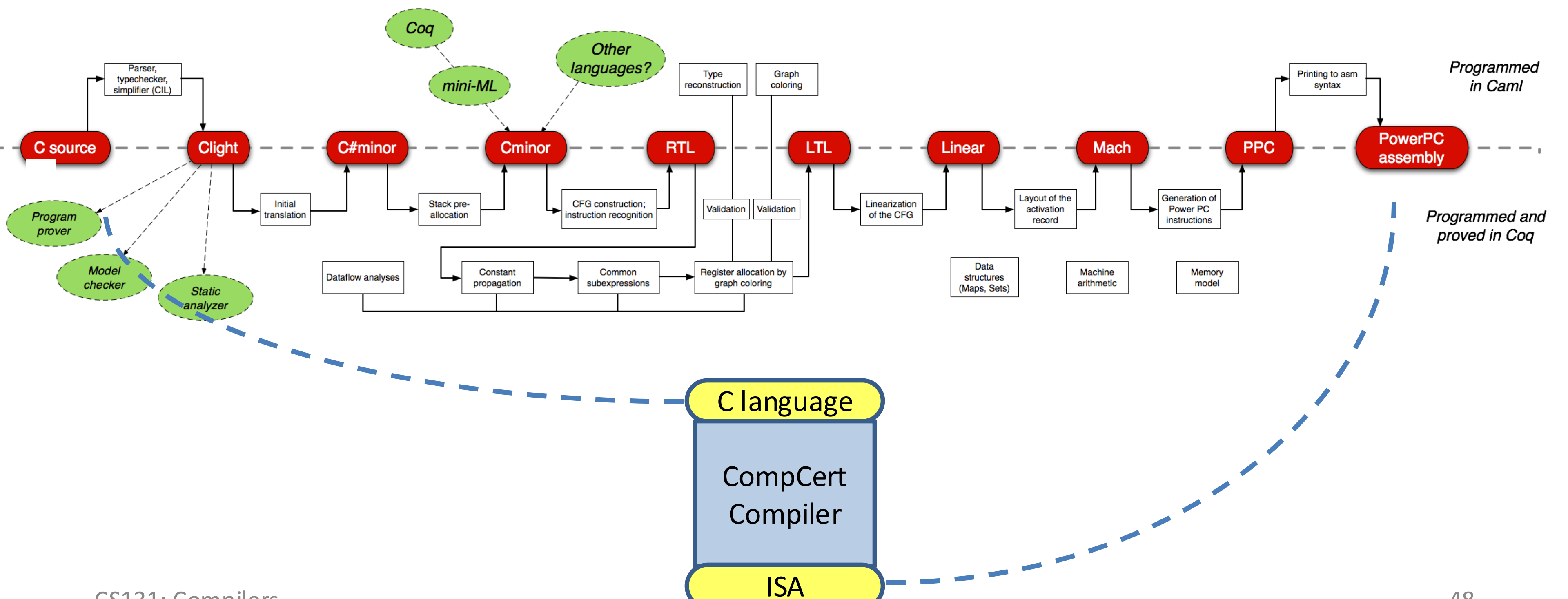
Taking advantage of advances in computer science:

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

CompCert – A Verified C Compiler

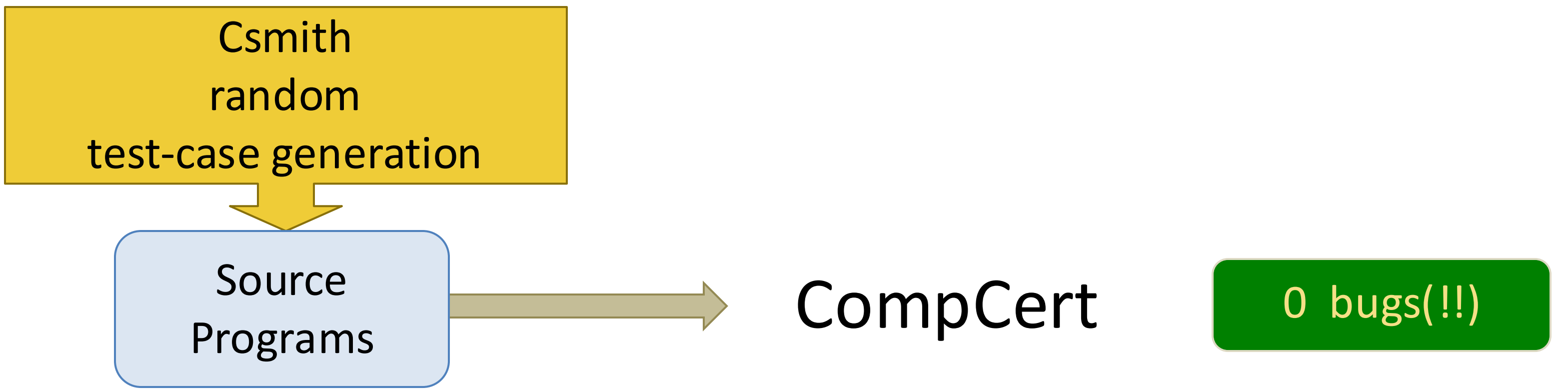
Xavier Leroy
INRIA

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