Lecture 21

COMPILER DESIGN
Announcements

• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – **Due**: Tuesday, December 17\textsuperscript{th} at 23:59
  – *May submit by Thursday, December 19\textsuperscript{th} at 23:59 with penalty*

• Final Exam
  – Scheduled for Friday, January 31\textsuperscript{st}, 9-11 AM
## One Cut at the Dataflow Design Space

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>
“Classic” Constant Propagation

• Constant propagation can be formulated as a dataflow analysis

• Idea: propagate and fold integer constants in one pass
  \[ x = 1; \quad x = 1; \]
  \[ y = 5 + x; \quad y = 6; \]
  \[ z = y \times y; \quad z = 36; \]

• Information about a single variable
  – Variable is never defined
  – Variable has a single, constant value
  – Variable is assigned multiple values
Domains for Constant Propagation

• We can make a constant propagation lattice $\mathcal{L}$ for one variable like

$$T = \text{multiple values}$$

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

$$\bot = \text{never defined}$$

• To accommodate multiple variables, we take the product lattice, with one element per variable
  – Assuming there are three variables, $x$, $y$, and $z$, the elements of the product lattice are of the form $(\ell_x, \ell_y, \ell_z)$
  – Alternatively, think of the product domain as a context that maps variable names to their “abstract interpretations”

• What are “meet” and “join” in this product lattice?

• What is the height of the product lattice?
Flow Functions

• Consider the node \( x = y \text{ op } z \)
• \( F(\ell_x, \ell_y, \ell_z) = ? \)

\[
\begin{align*}
F(\ell_x, T, \ell_z) &= (T, T, \ell_z) \\
F(\ell_x, \ell_y, T) &= (T, \ell_y, T)
\end{align*}
\]

“If either input might have multiple values
the result of the operation might too.”

\[
\begin{align*}
F(\ell_x, \bot, \ell_z) &= (\bot, \bot, \ell_z) \\
F(\ell_x, \ell_y, \bot) &= (\bot, \ell_y, \bot)
\end{align*}
\]

“If either input is undefined
the result of the operation is too.”

\[
F(\ell_x, i, j) = (i \text{ op } j, i, j)
\]

“If the inputs are known constants,
calculate the output statically.”

• Flow functions for the other nodes are easy...
• Monotonic?
• Distributes over meets?
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS
Best Possible Solution

• Suppose we have a control-flow graph

• If there is a path $p_1$ starting from the root node (entry point of the function) traversing the nodes $n_0, n_1, n_2, ..., n_k$

• The best possible information along the path $p_1$ is
  \[ \ell_{p_1} = F_{n_k}(...F_{n_2}(F_{n_1}(F_{n_0}(T)))...) \]

• Best solution at the output is some $\ell \subseteq \ell_p$ for all paths $p$

• Meet-over-paths (MOP) solution
  \[ \prod_{p \in \text{paths to } n} \ell_p \]

Best answer here is:

\[ F_5(F_3(F_2(F_1(T)))) \prod F_5(F_4(F_2(F_1(T)))) \]
What about quality of iterative solutions?

• Does the iterative solution $\text{out}[n] = F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n'])$ compute the MOP solution?

• MOP solution: $\prod_{p \in \text{paths}_t} \ell_p$

• Answer: Yes, if the flow functions distribute over $\prod$
  – Distributive means: $\prod_i F_n(\ell_i) = F_n(\prod_i \ell_i)$
  – Proof is a bit tricky & beyond the scope of this class
  – Difficulty: loops in the control-flow graph may mean that there are infinitely many paths

• Not all analyses give MOP solution
  – They are more conservative
• $F_n[x] = \text{gen}[n] \cup (x - \text{kill}[n])$

• Does $F_n$ distribute over meet $\prod = \cup$?

• $F_n[x \prod y]$
  $= \text{gen}[n] \cup ((x \cup y) - \text{kill}[n])$
  $= \text{gen}[n] \cup ((x - \text{kill}[n]) \cup (y - \text{kill}[n]))$
  $= (\text{gen}[n] \cup (x - \text{kill}[n])) \cup (\text{gen}[n] \cup (y - \text{kill}[n]))$
  $= F_n[x] \cup F_n[y]$
  $= F_n[x] \prod F_n[y]$

• Therefore, Reaching Definitions with iterative analysis always terminates with the MOP (i.e., best) solution

• In fact, the other three analyses (i.e., liveness, available expressions, & very busy expressions) are all MOP
Constprop Iterative Solution

\[
\begin{align*}
\text{if } x > 0 & \\
\text{y} = 1 & \quad \text{y} = 2 \\
\text{z} = 2 & \quad \text{z} = 1 \\
\text{iterate solution} & = \{1, 1, 2\} \land \{1, 2, 1\} = \{1, 1, 2\}
\end{align*}
\]
MOP Solution $\neq$ Iterative Solution

MOP solution $\ (3, 1, 2) \sqcap (3, 2, 1) = (3, T, T)$. 
What Problems are Distributive?

• Many analyses of program structure are distributive
  – Liveness Analysis
  – Available Expressions
  – Reaching Definitions
  – Very Busy Expressions

• These express properties on how the program computes
What Problems are Not Distributive?

• Analyses of what the program computes
  – The output is a constant, positive, and so on
  – Constprop is an example as we have just seen
Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graphs without loops)

- $O(n)$ nodes
- $O(n)$ edges
- $O(2^n)$ paths*
  - At each branch there is a choice of 2 directions

* Incidentally, a similar idea can be used to force ML / Haskell type inference to need to construct a type that is exponentially big in the size of the program!
Review of (& Additional) Terminology

Review
• Must vs. May
• Forward vs. Backward
• Distributive vs. non-Distributive

Additional
• Flow-sensitive vs. Flow-insensitive
• Context-sensitive vs. Context-insensitive
• Path-sensitive vs. Path-insensitive
Dataflow Analysis: Summary

• Many dataflow analyses fit into a common framework
• Key idea: Iterative solution of a system of equations over a lattice of constraints
  – Iteration terminates if flow functions are monotonic
  – Solution is equivalent to meet-over-paths answer if the flow functions distribute over meet (⊔)

• Dataflow analyses as presented work for an “imperative” IR
  – Values of temporary variables are updated (“mutated”) during evaluation
  – Such mutation complicates calculations
  – SSA = “Single Static Assignment” eliminates this problem
    • By introducing more temporaries --- each one assigned to only once
  – Next up: Converting to SSA, finding loops and dominators in CFGs
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 that 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 \text{pred}[n]} \text{out}[n']

• “Every node dominates itself.”
  – out[n] := in[n] \cup \{n\}

• Formally: \mathcal{L} = \text{set of nodes ordered by} \subseteq
  – T = \{\text{all nodes}\}
  – F_n(x) = x \cup \{n\}
  – \bigcap \text{ 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

• To compute Dom[b] walk through doms[b]
• Need to efficiently compute intersections of Dom[a] and Dom[b]
  – Traverse up tree & look for least common ancestor

• See “A Simple, Fast Dominance Algorithm” Cooper, Harvey, and Kennedy
Completing Control-flow Analysis

- Dominator analysis identifies *back edges*
  - Edge \( n \rightarrow h \) 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 \( n \rightarrow h \), find the natural loop
  - \( \{n' \mid n \text{ is reachable from } n' \text{ in } G \setminus \{h\}\} \cup \{h\} \)

- 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 a program computes
- It makes the order of evaluation explicit
- Each %uid is assigned to only once
  - Contrast with the mutable quadruple form
  - Note 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 to convert a source program to make maximal use of %uids, rather than alloca-created storage?
  - Two problems: control flow & location in memory

- Then, how to 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?

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

```c
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

\[
\%\text{uid} = \phi \text{ty} <v_1>, <\text{label}_1>, \ldots, v_n, <\text{label}_n>
\]

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

```assembly
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

```plaintext
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:
  ...
```
• 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 inst. is promotable if neither of the two above conditions holds

entry:
  %x = alloca i64          // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i64** %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 $\phi$ 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 $\phi$ functions?

- Need to calculate the “Dominance Frontier”

- Node $A$ strictly dominates node $B$ if $A$ dominates $B$ and $A \neq B$
  - 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
- \( \text{DF}[1] = \{1\}, \text{DF}[2] = \{1,2\}, \text{DF}[3] = \{2\}, \text{DF}[4] = \{1\}, \text{DF}[5] = \{8,0\}, \text{DF}[6] = \{8\}, \text{DF}[7] = \{7,0\}, \text{DF}[8] = \{0\}, \text{DF}[9] = \{7,0\}, \text{DF}[0] = \{\}\)
Algorithm For Computing DF[n]

- Assume that $\text{doms}[n]$ stores the dominator tree (so that $\text{doms}[n]$ 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[$B$]) \(\geq 2\) \hspace{1cm} // (just an optimization)

for each $p \in \text{pred}[B]$ {

runner := $p$ \hspace{1cm} // start at the predecessor of $B$

while (runner \(\neq\) doms[$B$]) \hspace{1cm} // walk up the tree adding $B$

\hspace{1cm} $\text{DF}[runner] := \text{DF}[runner] \cup \{B\}$

\hspace{1cm} runner := doms[runner]

}
Insert $\phi$ 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 $N$.

\[
\begin{align*}
DF_0[N] &= DF[N] \\
DF_{i+1}[N] &= DF[DF_i[N] \cup N]
\end{align*}
\]

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 $\phi$ 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?

• \( DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\} \)
• \( DF_1[\{3,6\}] \)
  = \( DF[DF_0[\{3,6\}] \cup \{3,6\}] \)
  = \( DF[\{2,3,6,8\}] \)
  = \( \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\} \)
• \( DF_2[\{3,6\}] \)
  = ... 
  = \( \{1,2,8,0\} \)

• So \( J[\{3,6\}] = \{1,2,8,0\} \), and we need to add phi nodes at those 4 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:
  \[
  \%x = \text{phi} \ t \ \%y, \ %\text{pred1} \ t \ %y \ %\text{pred2} \ ... \ t \ %y \ %\text{predK}
  \]
  
  // 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

\[
\begin{align*}
l_1: \ & \%p = \text{alloca i64} \\
& \text{store } 0, \ %p \\
& \%b = \%y > 0 \\
& \text{br } \%b, \ %l_2, \ %l_3 \\
\end{align*}
\]

\[
\begin{align*}
l_2: \ & \text{store } 1, \ %p \\
& \text{br } \%l_3 \\
\end{align*}
\]

\[
\begin{align*}
l_3: \ & \%x = \text{load } \%p \\
& \text{ret } \%x \\
\end{align*}
\]
Example SSA Optimizations

l₁: %p = alloca i64
    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

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

\[
\begin{align*}
l_1 &: \%p = \text{alloca i64 } \\
& \text{store 0, } \%p \\
& \%b = \%y > 0 \\
& \%x_1 = \text{load } \%p \\
& \text{br } \%b, \%l_2, \%l_3 \\

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

l_3 &: \%x_4 = \phi[\%x_1; \%l_1, \%x_2; \%l_2] \\
& \%x = \text{load } \%p \\
& \text{ret } \%x
\end{align*}
\]
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

```
\begin{verbatim}
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]
    store \%x_3, \%p
    store 1, \%p
    \%x_2 = load \%p
    br \%l_3

l_3: \%x_4 = \phi[\%x_1; \%l_1, \%x_2; \%l_2]
    store \%x_4, \%p
    \%x = load \%p
    ret \%x
\end{verbatim}
```
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

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

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

l3: %x4 = phi[%x1, %l1, %x2, %l2]
    store %x4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

```
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
```
Example SSA Optimizations

l₁: %p = alloca i64

store 0, %p
%b = %b > 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

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

Find alloca
max φs
LAS/LA
A
DSE
DAE
elim φs
Example SSA Optimizations

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

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,%l₂]
store %x₄, %p
%x = load %p
ret %x
Example SSA Optimizations

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

\[
\begin{align*}
l_1: \quad & \%p = \text{alloca } \text{i64} \\
& \text{store } 0, \%p \\
& \%b = \%y > 0 \\
& \text{br } \%b, \%l_2, \%l_3 \\
\end{align*}
\]

\[
\begin{align*}
l_2: \quad & \%x_3 = \phi[0,\%l_1] \\
& \text{store } \%x_3, \%p \\
& \text{store } 1, \%p \\
& \%x_2 = \text{load } \%p \\
& \text{br } \%l_3 \\
\end{align*}
\]

\[
\begin{align*}
l_3: \quad & \%x_4 = \phi[0;\%l_1, 1;\%l_2] \\
& \text{store } \%x_4, \%p \\
& \%x = \text{load } \%p \\
& \text{ret } \%x \\
\end{align*}
\]
Example SSA Optimizations

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

```plaintext
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
    br %l3

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

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

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
    br %l₃

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

```
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
```
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.

```
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
```
Example SSA Optimizations

\[
\begin{align*}
&l_1: \quad \% b = \% y > 0 \\
&\quad \text{br} \% b, \% l_2, \% l_3

&l_2: \quad \% x_3 = \phi[0, \% l_1] \\
&\quad \text{br} \% l_3

&l_3: \quad \% x_4 = \phi[0; \% l_1, 1: \% l_2] \\
&\quad \text{ret} \% x_4
\end{align*}
\]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002
Example SSA Optimizations

\[ l_1: \]
\[ b = y > 0 \]
\[ b, l_2, l_3 \]

\[ l_2: x_3 = \phi[0, l_1] \]
\[ l_3: x_4 = \phi[0; l_1, 1; l_2] \]
\[ l_4: ret x_4 \]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
Example SSA Optimizations

l₁:
\%
b = \%y > 0
br %b, %l₂, %l₃

l₂:
br %l₃

l₃: \%x₄ = φ[0;%l₁, 1:%l₂]
ret \%x₄

- 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