Lecture 20

COMPILER DESIGN
Announcements

• HW5: OAT v. 2.0
  – Records, function pointers, type checking, array-bounds checks, etc.
  – **Due:** Thursday, November 28th at 23:59 (i.e., tonight)
  – *May submit by Saturday, November 30th at 23:59 with penalty*

• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – Available soon (before next Mon/Tue)
  – **Due:** Tuesday, December 17th at 23:59
  – *May submit by Thursday, December 19th at 23:59 with penalty*

• Final Exam
  – Scheduled for Friday, January 31st, 9-11 AM
Accessing Spilled Registers

• If optimistic coloring fails, we need to generate code to move the spilled temporary to & from memory

• **Option 1: Reserve registers specifically for moving to/from memory**
  – Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  – Pro: Only need to color the graph once.
  – Not good on X86 (especially 32bit) because there are too few registers & too many constraints on how they can be used.

• **Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory**
  – Pro: Need to reserve fewer registers.
  – Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph
Example Spill Code

- Suppose temporary \( t \) is marked for spilling to stack slot [rbp+offs]

- Rewrite the program like this:
  
  Initially:
  
  \[
  t = a \text{ op } b; \]
  
  ... 
  
  \[
  x = t \text{ op } c \]
  
  ... 
  
  \[
  y = d \text{ op } t
  \]

  After spilling:
  
  \[
  t = a \text{ op } b \quad // \text{ defn. of } t
  \]
  
  Mov [rbp+offs], t
  
  ...
  
  Mov t37, [rbp+offs] \quad // \text{ use } #1 \text{ of } t
  
  x = t37 \text{ op } c
  
  ...
  
  Mov t38, [rbp+offs] \quad // \text{ use } #2 \text{ of } t
  
  y = d \text{ op } t38

- Here, \( t37 \) and \( t38 \) are freshly generated temporaries that replace \( t \) for different uses of \( t \)

- Rewriting the code in this way breaks \( t \)'s live range up:
  
  \( t, t37, t38 \) are only live across one edge
Example Spilling using Spare Registers

- Suppose temporary $t$ is marked for spilling to stack slot [rbp+offs]
- Rewrite the program like this:

  $t = a \ op \ b$; 
  ... 
  $x = t \ op \ c$ 
  ... 
  $y = d \ op \ t$

- $u = u \ op \ v$;

  $t = a \ op \ b$ \hspace{1cm} // defn. of $t$
  Mov [rbp+offs], $t$
  ...
  $x = t \ op \ c$
  ...
  $y = d \ op \ t$

  $u = u \ op \ v$;
  Mov r1, [rbp+offs1] \hspace{1cm} // both $u, v$ spilled
  Mov r2, [rbp+offs2] \hspace{1cm} // $u@offs1, v@offs2$
  $r1 = r1 \ op \ r2$
  Mov [rbp+offs1], r1
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

- **Briggs’ strategy**: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors that have degree \( \geq k \)

- **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
Why Two Strategies?

• For Briggs, we need to look at all neighbors of x & y

• For George, we need to look at only the neighbors of x

• Precolored nodes have infinite degree

• Thus, we
  – Use George’s strategy if one of x & y is precolored
  – Use Briggs’ strategy if both are temporaries
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 Briggs’ 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 30 and rerun the whole register allocation algorithm starting at step 1.
Overall Algorithm

- Simplify, coalesce, and freeze
- Mark possible spills
- Color, and delete actual spills
- Rewrite code to implement actual spills
- Liveness analysis
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.
- Though, not an issue on X86.

Vast literature on register allocation.

Other notable formulations include using ILP (for optimal assignment).
OTHER DATAFLOW ANALYSES
• The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  – Reaching definitions analysis
  – Available expressions analysis
  – Alias Analysis
  – Constant Propagation
  – These analyses follow the same 3-step approach as for liveness

• To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called quadruples
  – Allows easy definition of def[n] and use[n]
  – A slightly “looser” variant of LLVM’s IR that doesn’t require the “static single assignment” – i.e. it has mutable local variables
  – We will use LLVM-IR-like syntax
## Def / Use for SSA

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = op b c</code></td>
<td>{a}</td>
<td>{b,c}</td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>a = load b</code></td>
<td>{a}</td>
<td>{b}</td>
<td>load</td>
</tr>
<tr>
<td><code>store c, b</code></td>
<td>Ø</td>
<td>{b}</td>
<td>store</td>
</tr>
<tr>
<td><code>a = alloca t</code></td>
<td>{a}</td>
<td>Ø</td>
<td>alloca</td>
</tr>
<tr>
<td><code>a = bitcast b to u</code></td>
<td>{a}</td>
<td>{b}</td>
<td>bitcast</td>
</tr>
<tr>
<td><code>a = gep b [c,d, ...]</code></td>
<td>{a}</td>
<td>{b,c,d,...}</td>
<td>getelementptr</td>
</tr>
<tr>
<td><code>a = f(b_1,...,b_n)</code></td>
<td>{a}</td>
<td>{b_1,...,b_n}</td>
<td>call w/return</td>
</tr>
<tr>
<td><code>f(b_1,...,b_n)</code></td>
<td>Ø</td>
<td>{b_1,...,b_n}</td>
<td>void call (no return)</td>
</tr>
</tbody>
</table>

### Terminators

<table>
<thead>
<tr>
<th>Terminator</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>br L</code></td>
<td>Ø</td>
<td>Ø</td>
<td>jump</td>
</tr>
<tr>
<td><code>br a L1 L2</code></td>
<td>Ø</td>
<td>{a}</td>
<td>conditional branch</td>
</tr>
<tr>
<td><code>return a</code></td>
<td>Ø</td>
<td>{a}</td>
<td>return</td>
</tr>
</tbody>
</table>
REACHING DEFINITIONS
Reaching Definition Analysis

• Q: What variable definitions reach a particular use of the variable?

• This analysis is used for constant propagation & copy propagation
  – Constant propagation: If only one definition reaches a particular use, can replace use by the definition
  – Copy propagation: additionally requires that the copied value still has its same value – computed using an available expressions analysis (next)

• Input: Quadruple CFG
• Output: in[n] (resp. out[n]) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n
Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG

```
b = a + 2
out[1]: {1}
in[2]: {1}

c = b * b
out[2]: {1,2}
in[3]: {1,2}

b = c + 1
out[3]: {2,3}
in[4]: {2,3}

return b * a
```
Reaching Definitions Step 1

- Define the **sets of interest** for the analysis
- Let \( \text{defs}[a] \) be the set of nodes that define the variable \( a \)
- Define \( \text{gen}[n] \) and \( \text{kill}[n] \) as follows

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \ \text{op} \ c )</td>
<td>{n}</td>
<td>( \text{defs}[a] - {n} )</td>
</tr>
<tr>
<td>( a = \text{load} \ b )</td>
<td>{n}</td>
<td>( \text{defs}[a] - {n} )</td>
</tr>
<tr>
<td>store ( b, a )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( a = f(b_1,\ldots,b_n) )</td>
<td>{n}</td>
<td>( \text{defs}[a] - {n} )</td>
</tr>
<tr>
<td>( f(b_1,\ldots,b_n) )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{br} \ L )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{br} \ a \ L1 \ L2 )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>return ( a )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy
  - $\text{out}[n] \supseteq \text{gen}[n]$
    “The definitions that reach the end of a node at least include the definitions generated by the node”

- $\text{in}[n] \supseteq \text{out}[n'] \text{ if } n' \text{ is in } \text{pred}[n]$
  “The definitions that reach the beginning of a node include those that reach the exit of any predecessor”

- $\text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n]$
  “The definitions that come in to a node either reach the end of the node or are killed by it.”
  - Equivalently: $\text{out}[n] \supseteq \text{in}[n] - \text{kill}[n]$

Zhendong Su  Compiler Design
Reaching Definitions Step 3

• Convert constraints to iterated update equations:
  \[
  \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
  \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
  \]

• Algorithm: initialize in[n] and out[n] to \(\emptyset\)
  – Iterate the update equations until a fixed point is reached

• Algorithm terminates since in[n] & out[n] increase only monotonically
  – At most to a maximum set that includes all variables in the program

• It is precise since it finds the smallest sets that satisfy the constraints
AVAILABLE EXPRESSIONS
Available Expressions

• Idea: want to perform common subexpression elimination:
  \[
  a = x + 1 \quad a = x + 1 \\
  \ldots \quad \ldots \\
  b = x + 1 \quad b = a
  \]

• This transformation is safe if \( x + 1 \) means computes the same value at both places (i.e., \( x \) hasn’t been assigned)
  – “\( x+1 \)” is an available expression

• Dataflow values
  \[
  \text{in}[n] = \text{set of nodes whose values are available on entry to } n \\
  \text{out}[n] = \text{set of nodes whose values are available on exit of } n
  \]
### Available Expressions Step 1

- Define the sets of values
- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>$\text{gen}[n]$</th>
<th>$\text{kill}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \text{ op } c$</td>
<td>${n} - \text{kill}[n]$</td>
<td>$\text{uses}[a]$</td>
</tr>
<tr>
<td>$a = \text{load } b$</td>
<td>${n} - \text{kill}[n]$</td>
<td>$\text{uses}[a]$</td>
</tr>
<tr>
<td>store $b, a$</td>
<td>$\emptyset$</td>
<td>$\text{uses}[{x}]$ (for all $x$ that may equal $a$)</td>
</tr>
<tr>
<td>$\text{br } L$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br } a \text{ L1 L2}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1, \ldots, b_n)$</td>
<td>$\emptyset$</td>
<td>$\text{uses}[a] \cup \text{uses}[{x}]$ (for all $x$)</td>
</tr>
<tr>
<td>$f(b_1, \ldots, b_n)$</td>
<td>$\emptyset$</td>
<td>$\text{uses}[{x}]$ (for all $x$)</td>
</tr>
<tr>
<td>return $a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Note the need for “may alias” information...

Note that functions are assumed to be impure
Available Expressions Step 2

• Define constraints that an available expressions solution must satisfy

  out[n] ⊇ gen[n]
  “The expressions made available by n that reach the end of the node”

• in[n] ⊆ out[n]’ if n’ is in pred[n]
  “The expressions available at the beginning of a node include those that reach the exit of every predecessor”

• out[n] ∪ kill[n] ⊇ in[n]
  “The expressions available on entry either reach the end of the node or are killed by it.”

Note similarities and differences with constraints for “reaching definitions”.
Available Expressions Step 3

- Convert constraints to iterated update equations:
  
in[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']
out[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])

- Algorithm: initialize $\text{in}[n]$ and $\text{out}[n]$ to the set of all nodes
  - Iterate the update equations until a fixed point is reached

- Algorithm terminates since $\text{in}[n]$ & $\text{out}[n]$ decrease only monotonically
  - At most to a minimum of the empty set

- It is precise since it finds the largest sets that satisfy the constraints
GENERAL DATAFLOW ANALYSIS
Comparing Dataflow Analyses

• Look at the update equations in the inner loop of the analyses

• Liveness: \( (\text{backward, may}) \)
  – Let \( \text{gen}[n] = \text{use}[n] \) and \( \text{kill}[n] = \text{def}[n] \)
  – \( \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
  – \( \text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \)

• Reaching Definitions: \( (\text{forward, may}) \)
  – \( \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
  – \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

• Available Expressions: \( (\text{forward, must}) \)
  – \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
  – \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)
Very Busy Expressions

• An expression $e$ is very busy at program point $p$ if every path from $p$ must evaluate $e$ before any variable in $e$ is redefined

• Optimization: hoisting expressions

• A must-analysis
• A backward analysis

• Dataflow equations?
# 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>
The Literature

• Vast literature on dataflow analyses

• 90+% can be described by
  – Forward or backward
  – May or must

• Some oddballs, but not many
  – Bidirectional analyses
Common Features

• All of these analyses have a domain over which they solve constraints
  – Liveness, the domain is sets of variables
  – Reaching defs., Available exprs. the domain is sets of nodes
• Each analysis has a notion of gen[n] and kill[n]
  – Used to explain how information propagates across a node
• Each analysis is propagates information either forward or backward
  – Forward: in[n] defined in terms of predecessor nodes’ out[]
  – Backward: out[n] defined in terms of successor nodes’ in[]
• Each analysis has a way of aggregating information
  – Liveness & reaching definitions take union (∪)
  – Available expressions uses intersection (∩)
  – Union expresses a property that holds for some path (existential) -- may
  – Intersection expresses a property that holds for all paths (universal) -- must
(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then “$x \in \ell$” means “$x$ has the property”

2. For each node $n$, a flow function $F_n : \mathcal{L} \rightarrow \mathcal{L}$
   - So far we’ve seen $F_n(\ell) = \text{gen}[n] \cup (\ell \setminus \text{kill}[n])$
   - So: $\text{out}[n] = F_n(\text{in}[n])$
   - “If $\ell$ is a property that holds before the node $n$, then $F_n(\ell)$ holds after $n$”

3. A combining operator $\prod$
   - “If we know either $\ell_1$ or $\ell_2$ holds on entry to node $n$, we know at most $\ell_1 \prod \ell_2$”
   - $\text{in}[n] := \prod_{n' \in \text{pred}[n]} \text{out}[n']$

Zhendong Su  Compiler Design
for all \( n \), \( \text{in}[n] := \top \), \( \text{out}[n] := \top \)
repeat until no change
  for all \( n \)
    \( \text{in}[n] := \prod_{n' \in \text{pred}[n]} \text{out}[n'] \)
    \( \text{out}[n] := F_n(\text{in}[n]) \)
end
end

• \( \top \in \mathcal{L} \) (“top”) represents having the “maximum” amount of information
  – Having “more” information enables more optimizations
  – “Maximum” amount could be inconsistent with the constraints
  – Iteration refines the answer, eliminating inconsistencies
Structure of $\mathcal{L}$

- The domain has structure that reflects the “amount” of information contained in each dataflow value
- Some dataflow values are more informative than others:
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever $\ell_2$ provides at least as much information as $\ell_1$
  - The dataflow value $\ell_2$ is “better” for enabling optimizations

- Example 1: liveness analysis --- smaller sets of variables more informative
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments
  - So: $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \supseteq \ell_2$

- Example 2: available expressions --- larger sets of nodes more informative
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination
  - So: $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \subseteq \ell_2$
\( \mathcal{L} \) as a Partial Order

- \( \mathcal{L} \) is a *partial order* defined by the ordering relation \( \sqsubseteq \)
- A partial order is an ordered set
- Some of the elements might be *incomparable*
  - That is, there might be \( \ell_1, \ell_2 \in \mathcal{L} \) such that neither \( \ell_1 \sqsubseteq \ell_2 \) nor \( \ell_2 \sqsubseteq \ell_1 \)

- Properties of a partial order
  - *Reflexivity*: \( \ell \sqsubseteq \ell \)
  - *Transitivity*: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_3 \) implies \( \ell_1 \sqsubseteq \ell_3 \)
  - *Anti-symmetry*: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_1 \) implies \( \ell_1 = \ell_2 \)

- Examples
  - Integers ordered by \( \leq \)
  - Types ordered by \(<:\)
  - Sets ordered by \( \subseteq \) or \( \supseteq \)
Subsets of \{a,b,c\} ordered by \(\subseteq\)

Partial order presented as a Hasse diagram

order \(\subseteq\) is \(\subseteq\)  
meet \(\sqcap\) is \(\cap\)  
join \(\sqcup\) is \(\cup\)

Height is 3

\(\ell_1 \subseteq \ell_2\)
Meets and Joins

- The combining operator $\sqcap$ is called the “meet” operation
- It constructs the greatest lower bound
  - $\ell_1 \sqcap \ell_2 \subseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \subseteq \ell_2$
    “the meet is a lower bound”
  - If $\ell \subseteq \ell_1$ and $\ell \subseteq \ell_2$ then $\ell \subseteq \ell_1 \sqcap \ell_2$
    “there is no greater lower bound”

- Dually, the $\sqcup$ operator is called the “join” operation
- It constructs the least upper bound
  - $\ell_1 \subseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \subseteq \ell_1 \sqcup \ell_2$
    “the join is an upper bound”
  - If $\ell_1 \subseteq \ell$ and $\ell_2 \subseteq \ell$ then $\ell_1 \sqcup \ell_2 \subseteq \ell$
    “there is no smaller upper bound”

- A partial order that has all meets and joins is called a lattice
  - If it has just meets, it’s called a meet semi-lattice
Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node \( n \))
  \[
  \text{out}[n] := F_n(\text{in}[n])
  \]

- Equivalently:
  \[
  \text{out}[n] := F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n'])
  \]
  - By definition of \( \text{in}[n] \)

- We can write this as a simultaneous update of the vector of \( \text{out}[n] \):
  - Let \( x_n = \text{out}[n] \)
  - Let \( X = (x_1, x_2, \ldots, x_n) \) it’s a vector of points in \( \mathcal{L} \)
  - \( F(X) = (F_1(\prod_{j \in \text{pred}[1]} \text{out}[j]), F_2(\prod_{j \in \text{pred}[2]} \text{out}[j]), \ldots, F_n(\prod_{j \in \text{pred}[n]} \text{out}[j])) \)

- Any solution to the constraints is a fixpoint \( X \) of \( F \), i.e., \( F(X) = X \)
Itération Compute Fixpoints

- Let $X_0 = (T, T, \ldots, T)$
- Each loop through the algorithm apply $F$ to the old vector $X_1 = F(X_0)$
  $X_2 = F(X_1)$
  …
- $F^{k+1}(X) = F(F^k(X))$
- A fixpoint is reached when $F^k(X) = F^{k+1}(X)$
  – That’s when the algorithm stops

- Wanted: a maximal fixpoint
  – Because that one is more informative/useful for performing optimizations
Monotonicity & Termination

• Each flow function $F_n$ maps lattice elements to lattice elements; to be sensible is should be *monotonic*:

• $F : \mathcal{L} \rightarrow \mathcal{L}$ is *monotonic* iff:
  \[ \ell_1 \sqsubseteq \ell_2 \text{ implies that } F(\ell_1) \sqsubseteq F(\ell_2) \]
  
  – Intuitively: “If you have more information entering a node, then you have more information leaving the node.”

• Monotonicity lifts point-wise to the function: $F : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  
  – vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each $i$

• Note that $F$ is consistent: $F(X_0) \sqsubseteq X_0$
  
  – So each iteration moves at least one step down the lattice (for some component of the vector)
  
  – \[ \cdots \sqsubseteq F(F(X_0)) \sqsubseteq F(X_0) \sqsubseteq X_0 \]

• Therefore, # steps needed to reach a fixpoint is at most the height $H$ of $\mathcal{L}$ times the number of nodes: $O(Hn)$
• Information about individual nodes or variables can be lifted pointwise
  – If $L$ is a lattice, then so is $\{ f : X \rightarrow L \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

• Like *types*, the dataflow lattices are *static approximations* to the dynamic behavior
  – Could pick a lattice based on subtyping
  – Or other information

• Points in the lattice are sometimes called dataflow “facts”
“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
  \[
  \top = \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?