

# CIS 341: Compilers

## Lecture 29

# The Plan

- Today:
  - Dataflow analysis
- Project 6: Simple optimizations
  - Due: Monday, November 17<sup>th</sup> at 11:59 pm
- Talk Announcement: Guy Blelloch of CMU
  - Thursday at 3:00 in Wu & Chen
  - He'll be talking about parallel/concurrent programming
  - He's the inventor of NESL, an influential parallel programming language.

# Available Expressions

- Idea: want to perform common subexpression elimination:
  - $a = x + 1$        $a = x + 1$   
  ...                     $\rightarrow$     ...  
   $b = x + 1$          $b = a$
- 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:
  - $in[n]$  = set of nodes whose values are available on entry to  $n$
  - $out[n]$  = set of nodes whose values are available on exit of  $n$

# Available Expressions Step 1

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

Quadruple forms n:	$gen[n]$	$kill[n]$
$a = b \text{ op } c$	$\{n\} - kill[n]$	$uses[a]$
$a = [b]$	$\{n\} - kill[n]$	$uses[a]$
$[a] = b$	$\emptyset$	$uses[ [x] ]$ (for all x that may equal a)
jump L	$\emptyset$	$\emptyset$
if a goto L1 else L2	$\emptyset$	$\emptyset$
L:	$\emptyset$	$\emptyset$
$a = f(b_1, \dots, b_n)$	$\emptyset$	$uses[a] \cup uses[ [x] ]$ (for all x)
$f(b_1, \dots, b_n)$	$\emptyset$	$uses[ [x] ]$ (for all x)
return a	$\emptyset$	$\emptyset$

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

Note that functions are assumed to be impure...

## Available Expressions Step 2

- Define the constraints that an available expressions solution must satisfy.
- $out[n] \supseteq gen[n]$   
“The expressions made available by  $n$  that reach the end of the node”
- $in[n] \subseteq out[n']$  if  $n'$  is in  $pred[n]$   
“The expressions available at the beginning of a node include those that reach the exit of *every* predecessor”
- $out[n] \cup kill[n] \supseteq in[n]$   
“The expressions available on entry either reach the end of the node or are killed by it.”
  - Equivalently:  $out[n] \supseteq in[n] - kill[n]$

Note similarities and differences with constraints for “reaching definitions”.

## Available Expressions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$
- Algorithm: initialize  $in[n]$  and  $out[n]$  to {set of *all* nodes}
  - Iterate the update equations until a fixed point is reached
- The algorithm terminates because  $in[n]$  and  $out[n]$  *decrease only monotonically*
  - At most to a minimum of the empty set
- The algorithm is precise because it finds the *largest* sets that satisfy the constraints.

# Comparing Dataflow Analyses

- Look at the update equations in the inner loop of the analyses
- Liveness: (backward)
  - Let  $gen[n] = use[n]$  and  $kill[n] = def[n]$
  - $out[n] := \bigcup_{n' \in succ[n]} in[n']$
  - $in[n] := gen[n] \cup (out[n] - kill[n])$
- Reaching Definitions: (forward)
  - $in[n] := \bigcup_{n' \in pred[n]} out[n']$
  - $out[n] := gen[n] \cup (in[n] - kill[n])$
- Available Expressions: (forward)
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
  - $out[n] := gen[n] \cup (in[n] - kill[n])$

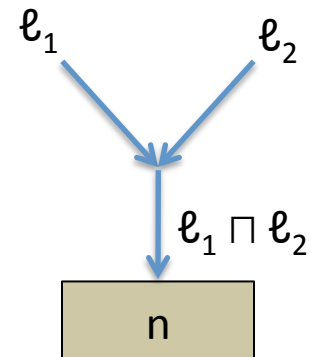
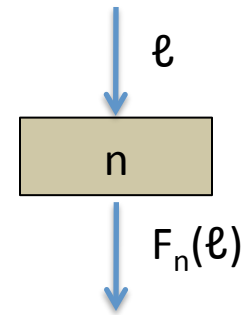
# Common Features

- All of these analyses have a *domain* over which they solve constraints.
  - Liveness, the domain is sets of variables
  - Reaching defns., 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 ( $\cup$ )
  - Available expressions uses intersection ( $\cap$ )
  - Union expresses a property that holds for *some* path (existential)
  - Intersection expresses a property that holds for *all* paths (universal)

# (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  $F_n(\ell) = \text{gen}[n] \cup (\ell - \text{kill}[n])$
  - 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  $\sqcap$ 
  - “If we know *either*  $\ell_1$  *or*  $\ell_2$  holds on entry to node  $n$ , we know at most  $\ell_1 \sqcap \ell_2$ ”
  - $\text{in}[n] := \sqcap_{n' \in \text{pred}[n]} \text{out}[n']$



# Generic Iterative (Forward) Analysis

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

- Here,  $\top \in \mathcal{L}$  (“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 $\mathcal{L}$

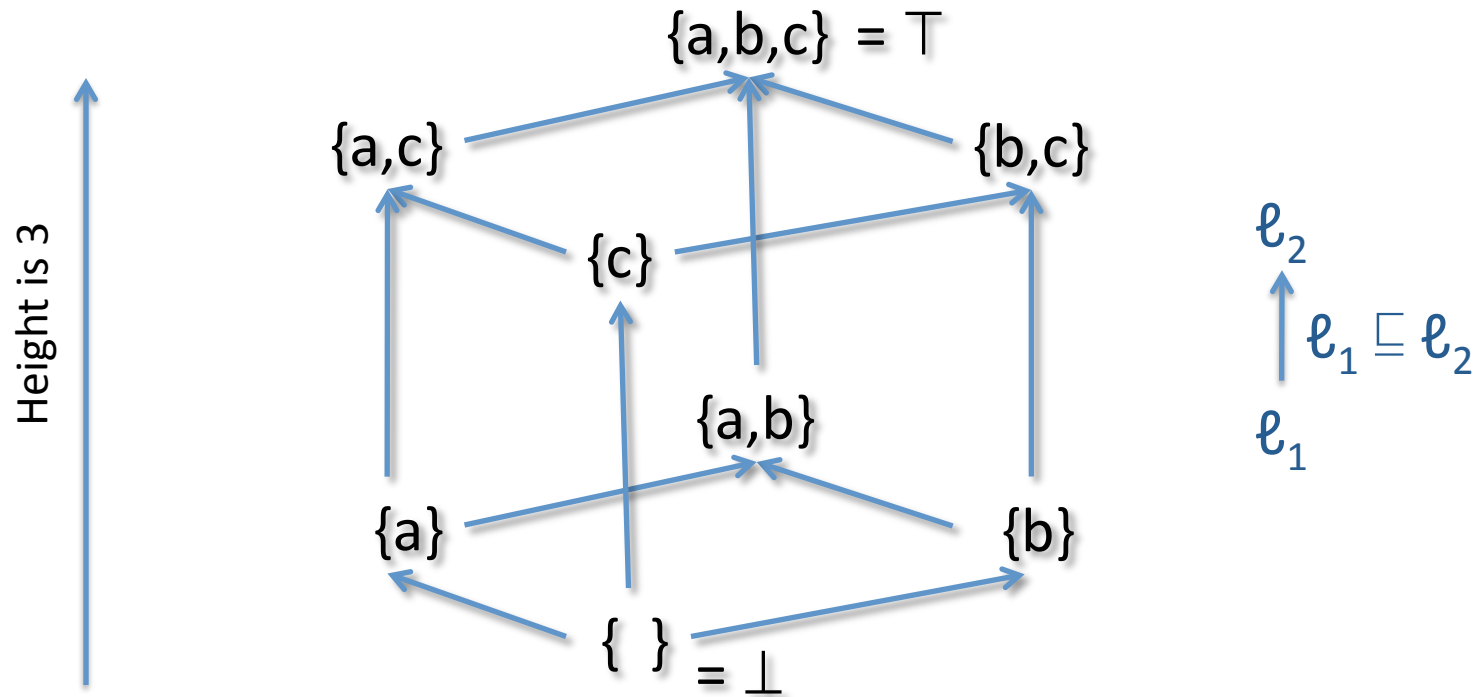
- The domain has structure that reflects the “amount” of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write  $\ell_1 \sqsubseteq \ell_2$  whenever  $\ell_2$  provides at least as much information as  $\ell_1$ .
  - The dataflow value  $\ell_2$  is “better” for enabling optimizations.
- Example 1: for liveness analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So:  $\ell_1 \sqsubseteq \ell_2$  if and only if  $\ell_1 \supseteq \ell_2$
- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So:  $\ell_1 \sqsubseteq \ell_2$  if and only if  $\ell_1 \subseteq \ell_2$

# $\mathcal{L}$ 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  $\sqsubseteq$  is  $\subseteq$

meet  $\sqcap$  is  $\cap$

join  $\sqcup$  is  $\cup$

# Meets and Joins

- The combining operator  $\sqcap$  is called the “meet” operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$  and  $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$   
“the meet is a lower bound”
  - If  $\ell \sqsubseteq \ell_1$  and  $\ell \sqsubseteq \ell_2$  then  $\ell \sqsubseteq \ell_1 \sqcap \ell_2$   
“there is no greater lower bound”
- Dually, the  $\sqcup$  operator is called the “join” operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$  and  $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$   
“the join is an upper bound”
  - If  $\ell_1 \sqsubseteq \ell$  and  $\ell_2 \sqsubseteq \ell$  then  $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$   
“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  $n$ ):
- $out[n] := F_n(in[n])$
- Equivalently:  $out[n] := F_n(\prod_{n' \in pred[n]} out[n'])$ 
  - By definition of  $in[n]$
- We can write this as a simultaneous update of the vector of  $out[n]$  values:
  - let  $x_n = out[n]$
  - Let  $\mathbf{X} = (x_1, x_2, \dots, x_n)$  it's a vector of points in  $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\prod_{j \in pred[1]} out[j]), F_2(\prod_{j \in pred[2]} out[j]), \dots, F_n(\prod_{j \in pred[n]} out[j]))$
- Any solution to the constraints is a *fixpoint*  $\mathbf{X}$  of  $\mathbf{F}$ 
  - i.e.  $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

# Iteration Computes Fixpoints

- Let  $\mathbf{X}_0 = (\top, \top, \dots, \top)$
- Each loop through the algorithm apply  $F$  to the old vector:  
 $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$   
 $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$   
...
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^k(\mathbf{X}))$
- A fixpoint is reached when  $\mathbf{F}^k(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$ 
  - 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 it should be *monotonic*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$  is *monotonic* iff:  
 $\ell_1 \sqsubseteq \ell_2$  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:  $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$ 
  - vector  $(x_1, x_2, \dots, x_n) \sqsubseteq (y_1, y_2, \dots, y_n)$  iff  $x_i \sqsubseteq y_i$  for each  $i$
- Note that  $\mathbf{F}$  is consistent:  $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$ 
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\dots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
- Therefore, # steps needed to reach a fixpoint is at most the height  $H$  of  $\mathcal{L}$  times the number of nodes:  $O(Hn)$