Lecture 22

CIS 341: COMPILERS
• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – Due: Wednesday, April 25th
GENERAL DATAFLOW ANALYSIS
Example Liveness Analysis

Each iteration update:
\[
\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \\
\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

- **Iteration 5:**
  \[
  \text{out}[3]= e,x,z
  \]

Done!
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] := \( \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
  – in[n] := gen[n] \( \cup \) (out[n] - kill[n])

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

• Available Expressions: (forward)
  – in[n] := \( \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
  – 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 (U)
  – Available expressions uses intersection (∩)
  – 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 \( F_n : \mathcal{L} \to \mathcal{L} \)
   - 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 $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ whenever $\mathcal{L}_2$ provides at least as much information as $\mathcal{L}_1$.
  - The dataflow value $\mathcal{L}_2$ 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: $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \supseteq \mathcal{L}_2$

- 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: $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \subseteq \mathcal{L}_2$
\( \mathcal{L} \) 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 \cap is \cap
join \cup is \cup
Meets and Joins

• The combining operator $\sqcap$ is called the “meet” operation.
• It constructs the greatest lower bound:
  – $l_1 \sqcap l_2 \subseteq l_1$ and $l_1 \sqcap l_2 \subseteq l_2$
    “the meet is a lower bound”
  – If $l \subseteq l_1$ and $l \subseteq l_2$ then $l \subseteq l_1 \sqcap l_2$
    “there is no greater lower bound”

• Dually, the $\sqcup$ operator is called the “join” operation.
• It constructs the least upper bound:
  – $l_1 \subseteq l_1 \sqcup l_2$ and $l_2 \subseteq l_1 \sqcup l_2$
    “the join is an upper bound”
  – If $l_1 \subseteq l$ and $l_2 \subseteq l$ then $l_1 \sqcup l_2 \subseteq l$
    “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 $\text{out}[n]$ values:
  - let $x_n = \text{out}[n]$
  - Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it’s a vector of points in $\mathcal{L}$
    - $F(\mathbf{X}) = (F_1(\prod_{j \in \text{pred}[1]} \text{out}[j]), F_2(\prod_{j \in \text{pred}[2]} \text{out}[j]), \ldots, F_n(\prod_{j \in \text{pred}[n]} \text{out}[j]))$

- Any solution to the constraints is a fixpoint $\mathbf{X}$ of $F$
  - i.e. $F(\mathbf{X}) = \mathbf{X}$
Iteration Computes 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)
  - $\ldots \sqsubseteq F(F(X_0)) \sqsubseteq F(X_0) \sqsubseteq X_0$

- Therefore, # steps needed to reach a fixpoint is at most the height $H$ of $\mathcal{L}$ times the number of nodes: $O(Hn)$
Building Lattices?

• Information about individual nodes or variables can be lifted \textit{pointwise}:
  – If $\mathcal{L}$ is a lattice, then so is $\{ f : X \to \mathcal{L} \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

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

• Points in the lattice are sometimes called dataflow \textit{“facts”}
“Classic” Constant Propagation

• Constant propagation can be formulated as a dataflow analysis.

• Idea: propagate and fold integer constants in one pass:
  
x = 1;  
y = 5 + x;  
z = y * y;  

  x = 1;  
y = 6;  
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 $L$ for one variable like this:

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

$$... , -3, -2, -1, 0, 1, 2, 3, ...$$

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

- 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 \( x = y \text{ op } z \)
- \( F(\ell_x, \ell_y, \ell_z) = ? \)

- \( F(\ell_x, T, \ell_z) = (T, T, \ell_z) \)
- \( F(\ell_x, \ell_y, T) = (T, \ell_y, T) \)
  
  “If either input might have multiple values the result of the operation might too.”

- \( F(\ell_x, \bot, \ell_z) = (\bot, \bot, \ell_z) \)
- \( F(\ell_x, \ell_y, \bot) = (\bot, \ell_y, \bot) \)
  
  “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 $n_0, n_1, n_2, \ldots n_k$
• The best possible information along the path $p_1$ is:
  $\ell_{p_1} = F_{n_k}(\ldots F_{n_2}(F_{n_1}(F_{n_0}(T)))\ldots)$
• Best solution at the output is some $\ell \subseteq \ell_p$ for all paths $p$.
• Meet-over-paths (MOP) solution:
  $\bigcap_{p \in \text{paths_to}[n]} \ell_p$

Best answer here is:
$F_5(F_3(F_2(F_1(T)))) \sqcap F_5(F_4(F_2(F_1(T))))$
What about quality of iterative solution?

- Does the iterative solution: \( \text{out}[n] = F_n(\bigcap_{n' \in \text{pred}[n]} \text{out}[n']) \) compute the MOP solution?

- MOP Solution: \( \bigcap_{p \in \text{paths_to}[n]} \ell_p \)

- Answer: Yes, if the flow functions distribute over \( \bigcap \)
  - Distributive means: \( \bigcap_i F_n(\ell_i) = F_n(\bigcap_i \ell_i) \)
  - Proof is a bit tricky & beyond the scope of this class. (Difficulty: loops in the control flow graph might mean there are infinitely many paths…)

- Not all analyses give MOP solution
  - They are more conservative.
Reaching Definitions is MOP

- \( F_n[x] = \text{gen}[n] \cup (x - \text{kill}[n]) \)
- Does \( F_n \) distribute over meet \( \sqcap = \cup \)?
- \( F_n[x \sqcap y] \)
  \[ = \text{gen}[n] \cup ((x \cup y) - \text{kill}[n]) \]
  \[ = \text{gen}[n] \cup ((x - \text{kill}[n]) \cup (y - \text{kill}[n])) \]
  \[ = (\text{gen}[n] \cup (x - \text{kill}[n])) \cup (\text{gen}[n] \cup (y - \text{kill}[n])) \]
  \[ = F_n[x] \cup F_n[y] \]
  \[ = F_n[x] \sqcap F_n[y] \]
- Therefore: Reaching Definitions with iterative analysis always terminates with the MOP (i.e. best) solution.
\[
x = y + z
\]
\[
(T, T, T) \quad \text{iterative solution}
\]
MOP Solution ≠ Iterative Solution

\[
\begin{align*}
\text{if } x & > 0 \\
y & = 1 \\
z & = 2 \\
x & = y + z
\end{align*}
\]

MOP solution: \((3, 1, 2) \sqcap (3, 2, 1) = (3, T, T)\)
Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graph 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!
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 ($\sqcap$).

Dataflow analyses as presented work for an “imperative” intermediate representation.

- The 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
See HW6: Dataflow Analysis

IMPLEMENTATION
LOOPS AND DOMINATORS
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 $\text{Dom}[n]$ as a forward dataflow analysis.
  - Using the framework we saw earlier: $\text{Dom}[n] = \text{out}[n]$ where:
- “A node $B$ is dominated by another node $A$ if $A$ dominates all of the predecessors of $B$.”
  - $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
- “Every node dominates itself.”
  - $\text{out}[n] := \text{in}[n] \cup \{n\}$

- Formally: $\mathcal{L} =$ set of nodes ordered by $\subseteq$
  - $T = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ 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, looking for least common ancestor:

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

• Dominator analysis identifies \textit{back edges}:
  – Edge $n \rightarrow h$ where $h$ dominates $n$

• Each back edge has a \textit{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:
  – $\{n' \mid n$ is reachable from $n'$ in $G - \{h\} \cup \{h\}\}$

• 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
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.