Announcements

• **HW5: OAT v. 2.0**
  – Records, function pointers, type checking, array-bounds checks, etc.
Live Variable Analysis

• Variable v is *live* at a program point L if
  – v is defined before L
  – v is used after L

• Liveness is defined in terms of where variables are *defined and used*

• Liveness analysis: Compute the live variables between each statement
  – May be *conservative* (i.e. it may claim a variable live when it isn’t) because that’s a safe approximation
  – To be useful, it should be more *precise* than simple scoping rules

• Liveness analysis is one example of *dataflow analysis*
  – Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …
Liveness, Formally

- A variable $v$ is *live* on edge $e$ if
  - there is a node $n$ in the CFG such that $\text{use}[n]$ contains $v$, and
  - a directed path from $e$ to $n$ such that for every statement $s'$ on the path, $\text{def}[s']$ does not contain $v$

- The first clause says $v$ will be used on some path starting from edge $e$
- The second says that $v$ won’t be redefined on that path before the use

- Questions
  - How to compute this efficiently?
  - How to use this information (e.g. for register allocation)?
  - How does the choice of IR affect this?
    - e.g. LLVM IR uses SSA (doesn’t allow redefinition) ⇒ simplify liveness analysis
Simple, inefficient algorithm

• “A variable $v$ is live on an edge $e$ if there is a node $n$ in the CFG using it and a directed path from $e$ to $n$ passing through no def of $v$.”

• Backtracking Algorithm
  – For each variable $v$
  – Try all paths from each use of $v$, tracing backward through the control-flow graph until either $v$ is defined or a previously visited node is reached
  – Mark the variable $v$ live across each edge traversed

• Inefficient because it explores the same paths many times (for different uses and different variables)
Dataflow Analysis

• **Idea:** compute liveness information for all variables simultaneously
  – Keep track of sets of information about each node

• **Approach:** define *equations* that must hold by any liveness determination
  – Equations based on “obvious” constraints

• Solve the equations by iteratively converging on a solution
  – Start with a “rough” approximation to the answer
  – Refine the answer at each iteration
  – Keep going until no more refinement possible: a *fixpoint* has been reached

• This is an instance of a general framework for computing program properties: **dataflow analysis**
Dataflow Value Sets for Liveness

• Nodes are program statements, so
  – use[n] : set of variables used by n
  – def[n] : set of variables defined by n
  – in[n] : set of variables live on entry to n
  – out[n] : set of variables live on exit from n

• Associate in[n] and out[n] with the “collected” information about incoming/outgoing edges

• For Liveness: what constraints are there among these sets?
  • Clearly
    \[ \text{in}[n] \supseteq \text{use}[n] \]

• What other constraints?
Other Dataflow Constraints

• We have: \( \text{in}[n] \supseteq \text{use}[n] \)
  – “A variable must be live on entry to \( n \) if it is used by \( n \)”

• Also: \( \text{in}[n] \supseteq \text{out}[n] \setminus \text{def}[n] \)
  – “If a variable is live on exit from \( n \), and \( n \) doesn’t define it, it is live on entry to \( n \)”
  – Note: here ‘\( \setminus \)’ means “set difference”

• And: \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)
  – “If a variable is live on entry to a successor node of \( n \), it must be live on exit from \( n \)”
Iterative Dataflow Analysis

• Find a solution to those constraints by starting from a rough guess
• Start with: $in[n] = out[n] = \emptyset$
• They don’t satisfy the constraints
  – $in[n] \supseteq use[n]$
  – $in[n] \supseteq out[n] \setminus def[n]$
  – $out[n] \supseteq in[n']$ if $n' \in succ[n]$

• Idea: iteratively re-compute $in[n]$ & $out[n]$ where forced to by constraints
  – Each iteration will add variables to the sets $in[n]$ & $out[n]$
    (i.e. the live variable sets will increase monotonically)
• We stop when $in[n]$ & $out[n]$ satisfy these equations
  (which are derived from the constraints above)
  – $in[n] = use[n] \cup (out[n] \setminus def[n])$
  – $out[n] = \bigcup_{n' \in succ[n]} in[n']$
for all $n$, $in[n] := \emptyset$, $out[n] := \emptyset$
repeat until no change in ‘in’ and ‘out’
for all $n$

$out[n] := \bigcup_{n' \in \text{succ}[n]} in[n']$

$in[n] := \text{use}[n] \cup (out[n] \setminus \text{def}[n])$

end
end

• Finds a fixpoint of the in & out equations
  – The algorithm is guaranteed to terminate, why?
• Why do we start with $\emptyset$?
Example Liveness Analysis

- Example flow graph

e = 1;
while(x>0) {
    z = e * e;
    y = e * x;
    x = x - 1;
    if (x & 1) {
        e = z;
    } else {
        e = y;
    }
}
return x;
Example Liveness Analysis

Each iteration update:

\[
\begin{align*}
\text{out}[n] & := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \\
\text{in}[n] & := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\end{align*}
\]

- **Iteration 1:**
  \[
  \begin{align*}
  \text{in}[2] & = x \\
  \text{in}[3] & = e \\
  \text{in}[4] & = x \\
  \text{in}[5] & = e, x \\
  \text{in}[6] & = x \\
  \text{in}[7] & = x \\
  \text{in}[8] & = z \\
  \text{in}[9] & = y
  \end{align*}
  \]

  (showing only updates that make a change)
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 2:**
  \[
  \begin{align*}
  \text{out}[1] &= x \\
  \text{in}[1] &= x \\
  \text{out}[2] &= e, x \\
  \text{in}[2] &= e, x \\
  \text{out}[3] &= e, x \\
  \text{in}[3] &= e, x \\
  \text{out}[5] &= x \\
  \text{out}[6] &= x \\
  \text{out}[7] &= z, y \\
  \text{in}[7] &= x, z, y \\
  \text{out}[8] &= x \\
  \text{in}[8] &= x, z \\
  \text{out}[9] &= x \\
  \text{in}[9] &= x, y
  \end{align*}
\]
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 3:**
  \( \text{out}[1] = e, x \)
  \( \text{out}[6] = x, y, z \)
  \( \text{in}[6] = x, y, z \)
  \( \text{out}[7] = x, y, z \)
  \( \text{out}[8] = e, x \)
  \( \text{out}[9] = e, x \)
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 4:
  \[
  \text{out}[5] = x, y, z \\
  \text{in}[5] = e, x, z
  \]
Example Liveness Analysis

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

• Iteration 5:
out[3] = e, x, z

Done!
Improving the Algorithm

• Can we do better?

• Observe: Information propagates between nodes only via

\[
\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]

– This is the only rule that involves more than one node

• If a node \( n \)'s successors haven’t changed, \( n \) won’t change

• Thus, idea for an improved version of the algorithm

– Keep track of which node’s successors have changed
A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated

\[
\text{for all } n, \text{ in}[n] := \emptyset, \text{ out}[n] := \emptyset
\]
\[
w = \text{new queue with all nodes}
\]
\[
\text{repeat until } w \text{ is empty}
\]
\[
\text{let } n = w.\text{pop}() \quad \text{// pull a node off the queue}
\]
\[
\text{old}_\text{in} = \text{in}[n] \quad \text{// remember old in}[n]
\]
\[
\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
\[
\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]
\[
\text{if } (\text{old}_\text{in} != \text{in}[n]), \quad \text{// if in}[n] \text{ has changed}
\]
\[
\text{for all } m \text{ in } \text{pred}[n], w.\text{push}(m) \quad \text{// add to worklist}
\]
end
Plan

• Next: Register allocation

• Upcoming
  – Dataflow analysis (part 2)
  – Control-flow analysis & SSA
  – Garbage collection (GC)
  – Compiler testing & validation
    • How to find thousands of bugs in GCC & LLVM?
  – Compiler verification
    • How to build a fully verified realistic compiler?
  – MLIR
  – Guest lecture on GraalVM, PGQL & Green-Marl
  – Summary
REGISTER ALLOCATION
Register Allocation Problem

• **Given:** an IR program using an unbounded number of temporaries
  – e.g. the uids of our LLVM programs

• **Find:** a mapping from temporaries to machine registers such that
  – program semantics is preserved (i.e. the behavior is the same)
  – register usage is maximized
  – moves between registers are minimized
  – calling conventions / architecture requirements are obeyed

• **Stack Spilling**
  – If $\exists k$ registers available & $m > k$ temps live at the same time,
    not all temps will fit into registers
  – So, "spill" the excess temps to the stack
Linear-Scan Register Allocation

Simple, greedy register-allocation strategy

1. Compute liveness information: $\text{live}(x)$
   - recall: $\text{live}(x)$ is the set of uids that are live on entry to $x$'s definition

2. Let $\text{pal}$ be the set of usable registers
   - usually reserve a couple for spill code
     (our implementation uses rax,rcx)

3. Maintain "layout" $\text{uid}_\text{loc}$ that maps uids to locations
   - locations include registers and stack slots $n$, starting at $n=0$

4. Scan through the program, for each instruction that defines a uid $x$
   - $\text{used} = \{r \mid \text{reg } r = \text{uid}_\text{loc}(y) \text{ s.t. } y \in \text{live}(x)\}$
   - $\text{available} = \text{pal} - \text{used}$
   - If $\text{available}$ is empty: // no registers available, spill
     $\text{uid}_\text{loc}(x) := \text{slot } n$ ; $n = n + 1$
   - Otherwise, pick $r$ in available: // choose an available register
     $\text{uid}_\text{loc}(x) := \text{reg } r$
For HW6

- HW 6 implements two simple register allocation strategies
  - no_reg_layout: spill all registers
  - greedy_layout: the linear scan algorithm

- Your job: do "better" than these
GRAPH COLORING
Register Allocation

Basic process
1. Compute liveness information for each temp
2. Create an *interference graph*
   - Nodes are temps
   - There is an edge between nodes n & m if they are live at the same time
3. Try to color the graph
   - Each color corresponds to a register
4. If step 3 fails, “spill” a register to the stack and repeat from step 1
5. Rewrite the program to use registers
Interference Graphs

- Nodes of the graph are %uids
- Edges connect variables that interfere with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live)
- Register assignment is a graph coloring
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors
- Example

```c
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a, %b1}
%c = mult i32 %b1, %b1
// live = {%a, %c}
%b2 = add i32 %c, 1
// live = {%a, %b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```
Register Allocation Questions

• **Q1**: Can we efficiently find a k-coloring of the graph whenever possible?
  – Answer: in general the problem is NP-complete (it requires search)
  – But, we can do an efficient approximation using heuristics

• **Q2**: How do we assign registers to colors?
  – If we do this in a smart way, we can eliminate redundant MOVs

• **Q3**: What do we do when there aren’t enough colors/registers?
  – We have to use stack space, but how do we do this effectively?
Kempe [1879] provides this algorithm for K-coloring a graph.

It's a recursive algorithm that works in three steps:

- **Step 1**: Find a node with $\text{degree} < K$ and cut it out of the graph
  - Remove the nodes and edges
  - This is called *simplifying* the graph

- **Step 2**: Recursively K-color the remaining subgraph

- **Step 3**: When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was $< K$). Pick such a color.
Example: 3-color this Graph

Recursing Down the Simplified Graphs
Example: 3-color this Graph

Assigning Colors on the way back up
Failure of the Algorithm

- If the graph cannot be colored, it’ll simplify to a graph where every node having $\geq K$ neighbors
  - This can happen even when the graph is $K$-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:
Spilling

• Idea: If we can’t K-color the graph, we need to store 1 temp on the stack

• Which variable to spill?
  – Pick one that isn’t used very frequently
  – Pick one that isn’t used in a (deeply nested) loop
  – Pick one that has high interference
    (since removing it will make the graph easier to color)

• In practice: some weighted combination of these criteria

• When coloring
  – Mark the node as spilled
  – Remove it from the graph
  – Keep recursively coloring
Spilling, Pictorially

- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring
Optimistic Coloring

• Sometimes it is possible to color a node marked for spilling
  – If we get “lucky” with the choices of colors made earlier

• Example: When 2-coloring this graph

- Even though the node was marked for spilling, we can color it
- So, on the way down, mark for spilling, but don’t actually spill
Accessing Spilled Registers

• If optimistic coloring fails, we need to generate code to move the spilled temps 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 temp 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 temp \( t \) is marked for spilling to stack slot \([rbp+offs]\)

• Rewrite the program like this

\[
\begin{align*}
  t &= a \text{ op } b; \\
  & \quad \text{Mov} \ [rbp+offs], \ t \\
  \ldots \\
  x &= t \text{ op } c \\
  & \quad \text{Mov} \ t37, \ [rbp+offs] \quad // \text{ use } #1 \text{ of } t \\
  \ldots \\
  y &= d \text{ op } t \\
  & \quad \text{Mov} \ t38, \ [rbp+offs] \quad // \text{ use } #2 \text{ of } t \\
\end{align*}
\]

• \( t37 \) \& \( t38 \) are fresh temps to replace \( t \) for its different uses

• 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 temp $t$ is marked for spilling to stack slot $[rbp+offs]$.

• Rewrite the program like this.

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

  $t = a \ op \ b$  // defn. of $t$
  Mov $[rbp+offs]$, t
  $\ldots$
  Mov $r$, $[rbp+offs]$  // use #1 of $t$
  $x = r \ op \ c$
  $\ldots$
  Mov $r$, $[rbp+offs]$  // use #2 of $t$
  $y = d \ op \ r$

• $u = u \ op \ v$;

  Mov $r1$, $[rbp+offs1]$  // both $u, v$ spilled
  Mov $r2$, $[rbp+offs2]$  // $u@offs1, v@offs2$
  $r1 = r1 \ op \ r2$
  Mov $[rbp+offs1]$, $r1$
Precolored Nodes

• Some variables must be pre-assigned to registers
  – E.g. on X86 the multiplication instruction: IMul must define %rax
  – The “Call” instruction should kill caller-save registers %rax, %rcx, %rdx
  – Any temp live across a call interferes with the caller-save registers

• To properly allocate temps, we treat registers as nodes in the interference graph with pre-assigned colors
  – Pre-colored nodes can’t be removed during simplification
  – Trick: Treat pre-colored nodes as having “infinite” degree in the interference graph to guarantee they won’t be simplified
  – When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes
Picking Good Colors

- When choosing colors during the coloring phase, any choice is semantically correct, but some choices are better for performance.

- Example: `movq t1, t2`
  - If `t1` and `t2` can be assigned the same register (color), this move is redundant and can be eliminated.

- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of “move related” edge between the nodes for `t1` and `t2` in the interference graph.
  - When choosing a color for `t1` (or `t2`), if possible pick a color of an already colored node reachable by a move-related edge.
Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temp to another.

- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.
A more aggressive strategy is to coalesce nodes of the interference graph if they are connected by move-related edges. Coalescing nodes forces the two temps to be assigned the same register.

Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.

Problem: coalescing may 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 shown earlier 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
- Liveness analysis
- Rewrite code to implement actual spills
Last details

- 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 temps in the function prelude so that 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)