Lecture 23

CMSC 350: COMPILER DESIGN
CODE ANALYSIS
Motivating Code Analyses

• There are lots of things that might influence the safety/applicability of an optimization
  – What algorithms and data structures can help?

• How do you know what is a loop?
• How do you know an expression is invariant?
• How do you know if an expression has no side effects?
• How do you keep track of where a variable is defined?
• How do you know where a variable is used?
• How do you know if two reference values may be aliases of one another?
Moving Towards Register Allocation

• The Tiger compiler currently generates as many temporary variables as it needs
  – These are the %uids you should be very familiar with by now.

• Current compilation strategy:
  – Each %uid maps to a stack location.
  – This yields programs with many loads/stores to memory.
  – Very inefficient.

• Ideally, we’d like to map as many %uid’s as possible into registers.
  – Eliminate the use of the alloca instruction?
  – Only 10 general-purpose registers available on HERA
  – This means that a register must hold more than one slot

• When is this safe?
Liveness

- Observation: \%uid1 and \%uid2 can be assigned to the same register if their values will not be needed at the same time.
  - What does it mean for an \%uid to be “needed”? Ans: its contents will be used as a source operand in a later instruction.
- Such a variable is called “live”
- Two variables can share the same register if they are not live at the same time.
Scope vs. Liveness

• We can already get some coarse liveness information from variable scoping.
• Consider the following Tiger program:
  
  ```tiger
  function f(x: int): int = (  
    let var a := 0 in  
    if x > 0 then  
      let var b := x * x in  
      a := b + b  
      end  
    let var c := a * x in  
    c  
  end end)
  ```

• Note that due to Tiger’s scoping rules, variables \(b\) and \(c\) can never be live at the same time.
  – \(c\)’s scope is disjoint from \(b\)’s scope
• So, we could assign \(b\) and \(c\) to the same allocat’ed slot and potentially to the same register.
But Scope is too Coarse

• Consider this Java program:

```java
int f(int x) {
    int a = x + 2;
    int b = a * a;
    int c = b + x;
    return c;
}
```

• The scopes of a, b, c, x all overlap – they’re all in scope at the end of the block.
• But, a, b, c are never live at the same time.
  – So they can share the same stack slot / register
Live Variable Analysis

- A variable $v$ is *live* at a program point if $v$ is defined before the program point and used after it.
- Liveness is defined in terms of where variables are *defined* and where variables are *used*.

- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is 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, ...
Control-flow Graphs Revisited

• For the purposes of dataflow analysis, we use the control-flow graph (CFG) intermediate form.

• Recall that a basic block is a sequence of instructions such that:
  – There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  – There is a (possibly empty) sequence of non-control-flow instructions
  – The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

• A control flow graph
  – Nodes are blocks
  – There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  – There are no “dangling” edges – there is a block for every jump target.

• Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs:
  – at the HERA assembly level
  – an “imperative” C-like source level
  – at the LLVM IR level
  – Same general idea, but the exact details will differ
    • e.g. LLVM IR doesn’t have “imperative” update of %uid temporaries.
    • In fact, the SSA structure of the LLVM IR makes some of these analyses simpler.
Dataflow over CFGs

- For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…

Basic block CFG

“Exploded” CFG

Fall-through edges

in-edges

out-edges
Liveness is Associated with *Edges*

- This is useful so that the same register can be used for different temporaries in the same statement.
- Example: \( a = b + 1 \)

- Compiles to:

  - \( \text{MOVE}(a,b) \)
  - \( \text{INC}(a,1) \)
  - \( \text{MOVE}(\text{R1},\text{R1}) \)
  - \( \text{INC}(\text{R1},1) \)

  register allocation:
  - \( a \rightarrow \text{R1}, b \rightarrow \text{R1} \)
  - live: \( a, b \)
  - live: \( b, d, e \)
  - live: \( a \) (maybe)
Uses and Definitions

• Every instruction/statement uses some set of variables
  – i.e. reads from them
• Every instruction/statement defines some set of variables
  – i.e. writes to them

• For a node/statement \( s \), define the following:
  – \( \text{use}[s] \) : set of variables used by \( s \)
  – \( \text{def}[s] \) : set of variables defined by \( s \)

• Examples:
  – \( a = b + c \) \hspace{1cm} \text{use}[s] = \{b,c\} \hspace{1cm} \text{def}[s] = \{a\}
  – \( a = a + 1 \) \hspace{1cm} \text{use}[s] = \{a\} \hspace{1cm} \text{def}[s] = \{a\}
Liveness, Formally

• A variable $v$ is *live* on edge $e$ if:
  There is
  – a node $n$ in the CFG such that $use[n]$ contains $v$, *and*
  – a directed path from $e$ to $n$ such that for every statement $s'$ on the path, $def[s']$ does not contain $v$

• The first clause says that $v$ will be used on some path starting from edge $e$.
• The second clause 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, so it doesn’t allow redefinition $\Rightarrow$ 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 \( \text{and} \) 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 backwards through the control-flow graph until either \( v \) is defined or a previously visited node has been 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 be satisfied 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 is 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:
  - \text{use}[n] : set of variables used by n
  - \text{def}[n] : set of variables defined by n
  - \text{in}[n] : set of variables live on entry to n
  - \text{out}[n] : set of variables live on exit from n
  - \text{succ}[n] : set of nodes reachable in one step from n

- Associate \text{in}[n] and \text{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:  \( \text{in}[n] \supseteq \text{out}[n] - \text{def}[n] \)
  - “If a variable is live on exit from \( n \), and \( n \) doesn’t define it, it is live on entry to \( n \)”
  - Note: here ‘-' 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: \( \text{in}[n] = \emptyset \) and \( \text{out}[n] = \emptyset \)
- They don’t satisfy the constraints:
  - \( \text{in}[n] \supseteq \text{use}[n] \)
  - \( \text{in}[n] \supseteq \text{out}[n] - \text{def}[n] \)
  - \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)

- Idea: iteratively re-compute \( \text{in}[n] \) and \( \text{out}[n] \) where forced to by the constraints.
  - Each iteration will add variables to the sets \( \text{in}[n] \) and \( \text{out}[n] \)
    (i.e. the live variable sets will increase monotonically)

- We stop when \( \text{in}[n] \) and \( \text{out}[n] \) satisfy these equations:
  (which are derived from the constraints above)
  - \( \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \)
  - \( \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
Complete Liveness Analysis Algorithm

for all n, in[n] := Ø, out[n] := Ø
repeat until no change in ‘in’ and ‘out’
   for all n
      out[n] := \( \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
      in[n] := use[n] \( \cup (\text{out}[n] - \text{def}[n]) \)
   end
end

• Finds a fixpoint of the in and out equations.
  – The algorithm is guaranteed to terminate… Why?
• Why do we start with Ø?
Example Liveness Analysis

- Example flow graph:

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

- **Iteration 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:

- **Iteration 2:**

  out[1] = x
  in[1] = x
  out[2] = e, x
  in[2] = e, x
  out[3] = e, x
  in[3] = e, x
  out[5] = x
  in[5] = x
  out[6] = x
  in[6] = e, x
  out[7] = z, y
  in[7] = x, z, y
  out[8] = x
  in[8] = x, z
  out[9] = x
  in[9] = x, y
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 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:
\[
\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
\[
\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

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

Done!
Improving the Algorithm

• Can we do better?

• Observe: the only way information propagates from one node to another is using: \( \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
  – This is the only rule that involves more than one node

• If a node’s successors haven’t changed, then the node itself won’t change.

• 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.

for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty
  let n = w.pop() // pull a node off the queue
  old_in = in[n] // remember old in[n]
  out[n] := ∪_n'∈succ[n] in[n']
  in[n] := use[n] ∪ (out[n] - def[n])
  if (old_in != in[n]), // if in[n] has changed
     for all m in pred[n], w.push(m) // add to worklist
end