Analysis and transformation

- Each optimization is one or more analyses followed by a transformation
- Analyze CFG and/or DFG by propagating information forward or backward along CFG and/or DFG edges
- Merges in graph require combining information
- Loops in graph require iterative approximation
- Perform improving transformations based on information computed
- Have to wait until any iterative approximation has converged
- Analysis must be conservative, so that transformations preserve program behavior

A simple analysis

- Let’s start with a simple analysis that can help us determine which assignments can be eliminated from a basic block
- The example is unreasonable as source, but perhaps not as intermediate code

Liveness analysis

- This analysis is a form of liveness analysis
  - It can help identify assignments to remove
  - It can also form the basis for memory and register optimizations
- The goal is to identify which variables are live and which are dead at given program points
- The analysis is usually performed backwards
  - When a variable is used, it becomes live in that statement and code before it
  - When a variable is assigned to, it becomes dead for all code before it
- Note the relationship to def-use, as we saw in the data flow graph

Work backwards

- This analysis shows we can eliminate the last assignment to a, which is no surprise
- Technically, assignments to a dead variable can be removed
  - The value isn’t needed below, so why do the assignment?
- Furthermore, you could show for this example that the declarations for n and x aren’t needed, since n nor x is ever live

So?

- This analysis shows we can eliminate the last assignment to a, which is no surprise
- Technically, assignments to a dead variable can be removed
  - The value isn’t needed below, so why do the assignment?
- Furthermore, you could show for this example that the declarations for n and x aren’t needed, since n nor x is ever live
Then…

- After eliminating the last assignment (and these two declarations), you can redo the analysis
- This analysis now shows that \( l \) is dead everywhere in the block, and it can be removed as a parameter
- The stack can be reduced because of this
- And the caller could, in principle, be further optimized

Well, that was easy

- But that’s for basic blocks
- Once we have control flow, it’s much harder to do because we don’t know the order in which the basic blocks will execute
- We need to ensure (for optimization) that every possible path is accounted for, since we must make conservative assumptions to guarantee that the optimized code always works

Global data flow analysis

- We’re going to need something called global data flow analysis
- The form we’re interested in for live variable analysis (across basic blocks) is any-path analysis
  - An any-path property is true if there exists some path through the control flow graph such that the given property holds
    - For example, a variable is live if there is some path leading to it being accessed
    - For example, a variable is uninitialized if there is some path that does not initialize it
- All-path is the other major form of analysis

Example (Dragon, p. 609)

- Let’s now consider this analysis over a control flow graph
- Basic blocks connected by edges showing possible control flow
- We will omit the conditionals and labels on edges, since that’s fine for any-path analysis
- This is extremely conservative (safe)

Some more terminology

- A definition of a variable \( x \) is a statement that assigns a value to \( x \)
  - (The book discussed unambiguous vs. ambiguous definitions, but we’ll ignore this)
- A definition \( d \) reaches a program point \( p \) if
  - There is a path from the point immediately following \( d \) to \( p \)
  - And \( d \) is not killed along that path
- We’re now really giving formal definitions to these terms, but we’ve used them before

Examples

- \( d_1, d_2, d_5 \) reach the beginning of B2
- \( d_2 \) does not reach B4, B5, or B6

- Note: this is a conservative analysis, since it may determine that a definition reaches a point even if it might not in practice
But how to compute in general?

- We’d like to be able to compute all reaching definitions (for example)
- Let’s consider a simple language
  - It turns out to be very material
  - Complex languages impose really serious demands on data flow analysis
  - $S ::= \text{id} \in S ; \text{if } E \text{ then } S \text{ else } S \text{ do } S \text{ while } E$

Data flow equations

- We’re now going to define a set of equations that represent the flow through different constructs in the language
- For example
  - $\text{out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S))$
  - “The information at the end of $S$ is either generated within the statement (\text{gen}(S)) or enters at the beginning of the statement (\text{in}(S)) and is not killed by the statement (\text{-kill}(S))”

Example: $d: a := b + c$

- $\text{gen}(S) = \{d\}$
- $\text{kill}(S) = D_a - \{d\}$
- $\text{out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S))$
- $D_a$ is the set of all definitions in the program for variable $a$

Example: $S1 ; S2$

- $\text{gen}(S) = \text{gen}(S1) \cup \text{gen}(S2)$
- $\text{kill}(S) = \text{kill}(S1) \cap \text{kill}(S2)$
- $\text{in}(S1) = \text{in}(S)$
- $\text{in}(S2) = \text{in}(S)$
- $\text{out}(S) = \text{out}(S1) \cup \text{out}(S2)$

Example: if $E$ then $S1$ else $S2$ fi

- $\text{gen}(S) = \text{gen}(S1) \cup \text{gen}(S2)$
- $\text{kill}(S) = \text{kill}(S1) \cap \text{kill}(S2)$
- $\text{in}(S1) = \text{in}(S)$
- $\text{in}(S2) = \text{in}(S)$
- $\text{out}(S) = \text{out}(S1) \cup \text{out}(S2)$

Example: while $E$ do $S1$

- $\text{gen}(S) = \text{gen}(S1)$
- $\text{kill}(S) = \text{kill}(S1)$
- $\text{in}(S1) = \text{in}(S) \cup \text{gen}(S1)$
- $\text{out}(S) = \text{out}(S1)$
Then what?

- In essence, this defines a set of rules by which we can write down the relationships for gen/kill and in/out for a whole (structured) program.
- This defines a set of equations that then need to be solved.
- This solution can be complicated:
  - We don’t know if/when branches are taken
  - Loops introduce complications
  - Merges introduce complications
- Approaches to solutions: next lecture.