

More Analyses

## Reaching definitions

- What definitions of a variable *reach* a particular program point
  - A definition of variable x from statement s reaches a statement t if there is a path from s to t where x is not redefined
- Especially important if x is used in t
  - Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses
    Used to determine dependences: if x is defined in s and that definition reaches t then there is a
    - Used to determine dependences: if x flow dependence from s to t
  - Example: determine if statements were loop invariant
    - All definitions that reach an expression must originate from outside the loop, or themselves be invariant

# Creating a reaching-def analysis

- Can we use a powerset lattice?
- - Can use powerset of set of definitions in the program
    - V is set of variables, S is set of program statements
    - Definition:  $d \in V \times S$ 
      - Use a tuple, <v, s>
  - How big is this set?
    - At most  $|V \times S|$  definitions

• At each program point, we want to know which definitions have reached a particular point



• What do you think?

#### Forward or backward?

### Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- the other?
  - We don't know which branch is taken!
  - We should union the two sets any of those definitions can reach

• What happens if we are at a merge point and a definition reaches from one branch but not

We want to avoid getting too many reaching definitions  $\rightarrow$  should start sets at  $\perp$ 

### Transfer functions for RD

- Forward analysis, so need a slightly different formulation
  - Merged data flowing into a statement

$$IN(s) = \bigcup_t OUT(s) = \mathbf{ge}$$

- What are gen and kill?
  - gen(s): the set of definitions that may occur at s
    - e.g.,  $gen(s_1: x = e)$  is  $\langle x, s_1 \rangle$ lacksquare
  - kill(s): all previous definitions of variables that are *definitely* redefined by s  $\bullet$ 
    - e.g., kill(s1: x = e) is  $\langle x, * \rangle$

 $e_{t \in pred(s)} OUT(t)$  $e_{n(s)} \cup (IN(s) - kill(s))$ 

- We've seen this one before
- What is the lattice?
  - powerset of all expressions appearing in a procedure
- Forward or backward?
- Confluence operator?

#### Available expressions

#### Transfer functions for meet

• What do the transfer functions look like if we are doing a meet?

$$IN(S) = \bigcap_{t \in OUT(S)} OUT(S) = generations of the second secon$$

- gen(s): expressions that *must be* computed in this statement
- kill(s): expressions that use variables that may be defined in this statement
  - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: gen and kill must never lead to incorrect results
  - Must not decide an expression is available when it isn't, but OK to be safe and say it isn't
  - Must not decide a definition doesn't reach, but OK to overestimate and say it does

 $\in pred(s) OUT(t)$  $\mathbf{n}(s) \cup (IN(S) - \mathbf{kill}(s))$ 

# Analysis initialization

- Remember our formalization
  - If we start with everything initialized to  $\bot$ , we compute the least fixpoint
  - If we start with everything initialized to T, we compute the greatest fixpoint
- Which do we want? It depends!
  - Reaching definitions: a definition that may reach this point
    - We want to have as few reaching definitions as possible  $\rightarrow$  use least fixpoint
  - Available expressions: an expression that was definitely computed earlier
    - We want to have as many available expressions as possible  $\rightarrow$  use greatest fixpoint
  - Rule of thumb: if confluence operator is  $\sqcup$ , start with  $\bot$ , otherwise start with  $\top$

# Analysis initialization (II)

- The set at the entry of a program (for forward analyses) or exit of a program (for backward analyses) may be different
  - e.g., no expressions available at the beginning of function
- One way of looking at this: start statement and end statement have their own transfer functions

- - Why does this matter?

  - Good candidates for loop invariant code motion

#### Very busy expressions

• An expression is very busy if it is computed on every path that leads from a program point

• Can calculate very busy expressions early without wasting computation (since the expression is used at least once on every outgoing path) – this can save space

- Lattice?
- Direction?  $\bullet$
- Confluence operator?
- Initialization?
- Transfer functions?
  - Gen? Kill?

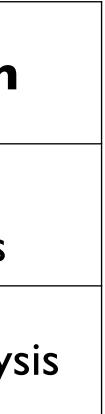
#### Very busy expressions

- Analysis can either be forward or backward
- Analysis can either be over all paths or over any pe
  - All paths: merges consider values from all paths
  - Any path: merges consider values from any pat

What kind of analysis is constant propagation?

#### Four types of dataflow

ath			
S		All paths	Any path
h	Forward	available expressions	reaching definitions
	Backward	very busy expressions	liveness analys



## Dataflow analysis precision

- So how good are the results of dataflow analysis?
- What is the best solution we can get?
  - Should determine information based on every path the actual program takes
  - This is undecidable! (what if the program loops?)
- More restrictive solution: *meet over all paths* 
  - program may not take)

Determine information based on every possible path in the program (including paths the actual

In general, this is also undecidable! (potentially infinite number of possible paths)

# Dataflow analysis precision

- - More formally, if confluence operator is  $\square$
  - Greatest fixpoint  $\sqsubseteq$  meet over all paths solution
    - MOP solution
  - If confluence operator is  $\Box$
  - Meet over all paths solution  $\sqsubseteq$  least fixpoint
    - the variable is definitely not constant

The solution to iterative dataflow analysis is less precise than the meet over all paths solution

e.g., for available expressions, calculated fixpoint does not have more available expressions than

e.g., for constant propagation, dataflow solution does not say a variable is constant if MOP says

## Distributive analysis

- A dataflow analysis is distributive if, for all transfer functions f
- $f(x \sqcup y) = f(x) \sqcup f(y)$  (equivalent definition for  $\Box$ )
- If a dataflow analysis is distributive, then meet over all paths solution = dataflow solution
- Powerset-based analyses are distributive
- Is constant propagation distributive?

## Dataflow analysis speed

- A dataflow analysis is *k-bounded* if, for all functions **f**
- $\forall x . f^{k}(x) = x \sqcup f(x) \sqcup ... \sqcup f^{k-1}(x)$  (and equivalently for  $\Box$ )
  - Consider a cycle, which contains a merge point at a loop header. If an analysis is *k-bounded*, then as long as the value coming in to the loop stays constant, you do not need more than *k* iterations to converge
- A dataflow analysis is *fast* if it is *k*-bounded and k = 2
  - Constant propagation is fast: after one cycle, a variable either stays constant or becomes T
- A dataflow analysis is *rapid* if it is *fast* and the solution for a cycle is independent of the entry node