CH4: A Dataflow Analysis Framework for While3Addr

Jonathan Lam

9/11/21

Contents

1 Dataflow analysis fundamentals

- A **dataflow analysis** computes some dataflow information at each program point in the CFG. We can represent it as a map from all values to some abstract set, e.g., $\sigma \in \text{Var} \to L = \{Z, N, \top\}$.
  - Z represents the set of zero values, N represents the set of nonzero values, and $\top$ represents the set of unknown values (due to an imprecision in the analysis); these are called **abstract values**
  - In **zero analysis**, we have the **abstraction function** α_Z:

$$\alpha_Z(n) = \begin{cases} Z & \text{if } n = 0 \\ N & \text{if } n \neq 0 \end{cases} \tag{1}$$

- Zero analysis can be used to determine if a value will be zero at some point in the program.
- A **flow function** maps the dataflow information at the program point immediately before an instruction to the dataflow information directly after that instruction. It should represent the semantics of the instruction abstractly and in terms of the abstract values tracked by the analysis.
  - We define the flow function for zero analysis to be:

$$f_Z[x := 0](\sigma) = \sigma[x \mapsto Z] \tag{2}$$
$$f_Z[x := n](\sigma) = \sigma[x \mapsto N] \text{ where } n \neq 0 \tag{3}$$
$$f_Z[x := y](\sigma) = \sigma[x \mapsto \sigma(y)] \tag{4}$$
$$f_Z[x := y \text{ op } z](\sigma) = \sigma[x \mapsto \top] \tag{5}$$
$$f_Z[\text{goto } n](\sigma) = \sigma \tag{6}$$
$$f_Z[\text{if } x = 0 \text{ goto } n](\sigma) = \sigma \tag{7}$$

- The flow function for assignments to arithmetic expressions can be made more exact by breaking down different cases of operators and operands.
- Recap: σ denotes the mapping of values to abstract values at a program point (kind of like a "current state"); α denotes the function used to map values in σ, and f denotes the transitions between the σ map at consecutive program points (i.e., over a single instruction) in order to maintain α

- More notation: we can get information about a variable depending on whether a branch condition is true or false:

$$f_Z[\text{if } x = 0 \text{ goto } n]_T(\sigma) = \sigma[x \mapsto Z] \tag{8}$$
$$f_Z[\text{if } x = 0 \text{ goto } n]_F(\sigma) = \sigma[x \mapsto N] \tag{9}$$

- We will need some kind of **initial assumption** (initial σ) for the values of thee variables.

- When there are branch statements, then we have two incoming edges to a statement. The **top** ($\top$) value occurs when there are different values coming from different branches.

- The process of combining analysis results along multiple paths is called a **join** ($\sqcup$) operation. Joining two abstract values results in an abstract value that generalizes the two inputs. We first must define a partial order (a relation that is reflexive, transitive, and anti-symmetric) over the abstract values. A set of values L that is equipped with a partial order $\sqsubseteq$, and for which the least upper bound of any two values in that ordering $l_1 \sqcup l_2$ is unique and also in L, is called a **join-semilattice** (or simply **lattice**).

  - For zero analysis, the partial order is defined with $Z \sqsubseteq \top$ and $N \sqsubseteq \top$. Thus $Z \sqcup N = \top$.

- $\top$ is the **maximal** element of a lattice. I.e., $\forall l \in L.l \sqsubseteq \top$.

- The initial dataflow assumptions for all of the values is thus $\sigma_0 : \text{var} \to \top$, unless there are some known initial conditions.

2 Dataflow analysis for loops

- Do a straight-line analysis, then merge the loop point until a **fixpoint** (or **fixed point**) is reached. Then continue to the part after the loop.

- We sacrifice precision in exchange for coverage of all possible executions, a classic tradeoff.

- The intuition behind correctness is the invariant that at each program point, the analysis results approximate all the possible program values that could exist at that point. We can also state that it is true due to induction, as long as the initial state and the flow functions are correct.

- When merging multiple program points, we join the dataflow analysis from all instructions that could precede it. However, we may not have encountered all statements that could have preceded it (as is true for loops): thus we define the **bottom** ($\bot$) abstract value, which has the following properties:

$$\forall l \in L. \bot \sqsubseteq l \tag{10}$$

$$\bot \sqcup l = l \tag{11}$$

  This can be thought of as the most specific dataflow analysis, and thus is always taken to be less general than the value it is joined with.

  - A set of abstract values L with a partial order $\sqsubseteq$ and both a top and bottom value is called a **complete lattice**.

3 Analysis execution strategy

- It doesn't matter which order we evaluate branches; the fixed point should be reached no matter which branch of an `if` statement is traversed first.

- A simple algorithm:

```
for Node n in cfg
    results[n] = K
    results[0] = initialDataflowInformation

while not at fixed point
    pick a node n in program
    input = join { results[j] | j in predecessors(n) }
    output = flow(n, input)
    results[n] = output
```

  To implement the abstract condition, we keep track of whether any nodes have changed since the last iteration. When picking a node in the program, we can either say that the nodes are chosen fairly. (This is not the most efficient, of course – see below.)

- The intuition that this will always terminate (reach a fixpoint) is that the analysis will always be more general than the previous iteration, and the lattice is bounded by $\top$ (assuming a finite lattice).

- A more efficient worklist algorithm (**Kildall's algorithm**) to perform the analysis is shown below:

```
worklist = empty set

for Node n in cfg
    input[n] = output[n] = K
    add n to worklist
input[0] = initialDataflowInformation

while worklist is not empty
    take a Node n off the worklist
    output[n] = flow(n, input[n])
    for Node j in succs(n)
        newInput = input[j] \ output[n]
        if newInput != input[j]
            input[j] = newInput
            add j to worklist
```

  Here we have a set that keeps track of only the nodes that have changed or undetermined inputs. When there are no more changes, then the queue is empty and the algorithm terminates.

  - The complexity of the Kildall algorithm is $O(ceh)$, where c is the cost of each operation (a flow function, a join operation, or a comparison of dataflow values), e are the number of edges in the CFG (related to the number of nodes), and h is the height of the lattice.
  - This doesn't determine how to choose a node to analyze from the worklist. For efficiency, we should choose strongly-connected components (i.e., loops) and process them in topological order (inner loops are solved before outer loops).
  - To get the topological ordering, we perform a reverse post-order traversal. This means that a node is visited before any of its successor nodes has been visited, except when the successor is reached by a back edge (Wikipedia: Data-flow analysis).
  - These improvements (strongly-connected and reverse post-order) dramatically improves performance in practice but does not change the worst-case bound.