02/01/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)
Jenna Wise
jlwise@andrew.cmu.edu

1 Reminders

- Homework 3 is due next **Thursday, February 7, 2019 at 11:59pm**. Homework instructions can be found on the course website

2 Lecture Review for Homework: Defining a Dataflow Analysis

A dataflow analysis computes some dataflow information at each program point in a control flow graph. This information can be used to detect errors in programs, such as if a program contains or may contain some division by zero(s). In fact, we can use a zero analysis (a particular dataflow analysis) to reduce the number of false positives we would incur by performing a more general division by zero analysis syntactically.
To define a dataflow analysis we need four things:

- a lattice $(L, \sqsubseteq)$ (Sec. 2.1)
- an abstraction function α (Sec. 2.2)
- initial dataflow analysis assumptions σ_0 (Sec. 2.3)
- a flow function f (Sec. 2.4)

2.1 Lattice

We will use σ to denote dataflow information at each program point. σ typically maps variables to abstract values taken from some set L (as is the case for a zero analysis or parity analysis):

$$\sigma \in \mathit{Var} \to L$$

L represents the set of abstract values we are interested in tracking in the analysis, which varies from one analysis to another. We require that these abstract values form a *join-semilattice* (lattice for short).

2.1.1 Partial Order

We define a partial order $\sqsubseteq$ over abstract values, where $l_1 \sqsubseteq l_2$ for $l_1, l_2 \in L$ means that l_1 is at least as precise as l_2. Recall that a partial order is any relation that is:

- reflexive: $\forall l : l \sqsubseteq l$
- transitive: $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
- anti-symmetric: $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

2.1.2 Join

A *join* operation, $\sqcup$, is helpful for generalizing the procedure of combining analysis results along multiple paths of a dataflow analysis. It means that when taking two abstract values $l_1, l_2 \in L$, the result of $l_1 \sqcup l_2$ is an abstract value l_j such that $l_1 \sqsubseteq l_j$ and $l_2 \sqsubseteq l_j$ for the partial order $\sqsubseteq$ (l_j is the least upper bound of l_1 and l_2).

2.1.3 Join-semilattice

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 is also in L, is called a *join-semilattice* (lattice for short). Any join-semilattice has a maximal element $\top$ (pronounced "top"). For all l, we have the identity $l \sqsubseteq \top$ and $\top \sqcup l = \top$.

2.1.4 $\bot$ Abstract Value & Meet

$\bot$ plays a dual role to the value $\top$: it sits at the bottom of the dataflow value lattice. For all l, we have $\bot \sqsubseteq l$ and $\bot \sqcup l = l$.

There is a greatest lower bound operator *meet*, $\sqcap$ (for $l_1, l_2 \in L$, the result of $l_1 \sqcap l_2$ is an abstract value l_j such that $l_j \sqsubseteq l_1$ and $l_j \sqsubseteq l_2$ for the partial order $\sqsubseteq$), which is dual to $\sqcup$. The meet of all dataflow values is $\bot$.

2.1.5 Complete Lattice

A set of values L that is equipped with a partial order $\sqsubseteq$, and for which both least upper bounds $\sqcup$ and greatest lower bounds $\sqcap$ exist in L and are unique, is called a *complete lattice*.

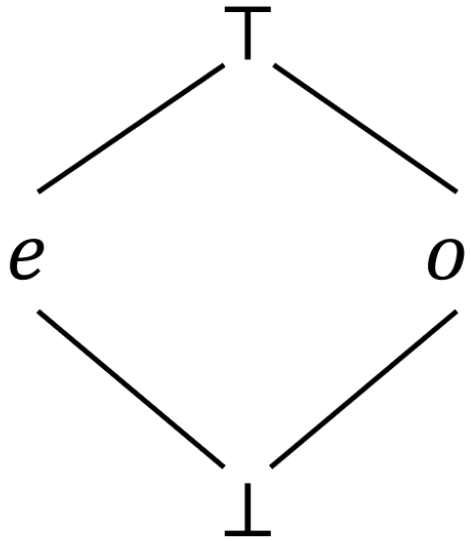
The theory of $\bot$ and complete lattices allows us to avoid following a specific path during a dataflow analysis by initializing σ at every instruction in the program, except at entry, to $\bot$, indicating that the instruction there has not yet been analyzed. We can then *always* merge all input values to a node, whether or not the sources of those inputs have been analysed, because we know that any $\bot$ values from unanalyzed sources will simply be ignored by the join operator $\sqcup$, and that if the dataflow value for that variable will change, we will get to it before the analysis is completed.

2.1.6 Parity Analysis Example

For example, we define a lattice for a *parity analysis*, which tracks whether each variable is odd or even at each program point.

For this analysis, we define L to be the set $\{e, o, \top, \bot\}$. The abstract value e represents all even values (including 0) and o represents all odd values. $\top$ is the "top element" and abstract value given to variables whose concrete value could be anything, even or odd, due to imprecision in the analysis. $\bot$ is the "bottom element" and abstract value given to variables that have not been analyzed yet for classification.

For parity analysis, we can define the partial order, $\sqsubseteq$, in two different ways:
1) Visually



2) Mathematically

$\forall l \in L.\ \bot \sqsubseteq l \sqsubseteq \top.$

Therefore, $\sqsubseteq$ is a partial order and $(L, \sqsubseteq)$ defines a complete lattice (and therefore, a join-semilattice). Additionally, $\top$ and $\bot$ can be proven to be the "top element" and "bot element" respectively.

2.2 Abstraction Function

Conceptually, each abstract value represents a set of one or more concrete values that may occur when a program executes. We define an abstraction function α that maps each possible concrete value of interest to an abstract value.

2.2.1 Parity Analysis Example

For parity analysis, we define $\alpha_P : \mathbb{Z} \to L$ so that all even integers (including 0) map to e and all even integers (all other integers) map to o:

$$\alpha_P(n) = e \text{ where } n \bmod 2 = 0$$
$$\alpha_P(n) = o \text{ where } n \bmod 2 \neq 0$$

2.3 Initial Dataflow Analysis Assumptions

Note that a side question comes up when we begin analyzing the first program instruction: what should we assume about the value of input variables? If we do not know anything about what the value of a variable can be, a good choice is to assume it can be anything; That is, in the initial environment σ_0, input variables are mapped to $\top$.

2.3.1 Parity Analysis Example

It makes the most sense for a parity analysis to map input variables (to the first program instruction) to $\top$, since the value of these variables could be anything (even or odd).

2.4 Flow Function

The core of any program analysis is how individual instructions in the program are analyzed and affect the analysis state σ at each program point. We define this using *flow functions* that map the dataflow information at the program point immediately *before* an instruction to the dataflow information *after* that instruction. A flow function should represent the semantics of the instruction, but abstractly, in terms of the abstract values tracked by the analysis.

2.4.1 Parity Analysis Example

For example, we define the flow functions f_P for parity analysis on WHILE3ADDR as follows:

$$f_P[\![x := n]\!](\sigma) = \begin{cases} \sigma[x \mapsto e] & \text{where } n \bmod 2 = 0 \\ \sigma[x \mapsto o] & \text{where } n \bmod 2 \neq 0 \end{cases} \quad (1)$$

$$f_P[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)] \quad (2)$$

$$f_P[\![x := y \; op \; z]\!](\sigma) = \sigma[x \mapsto \top] \quad (3)$$

$$f_P[\![\text{goto } n]\!](\sigma) = \sigma \quad (4)$$

$$f_P[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma \quad (5)$$

(1) is for assignment to a constant. If we assign an even integer to a variable x, then we should update the input dataflow information σ so that x maps to the abstract value e, and if we assign an odd integer to a variable x, then we should update the input dataflow information σ so that x maps to the abstract value o. Flow function (2) is for copies from a variable y to another variable x: we look up y in σ, written $\sigma(y)$, and update σ so that x maps to the same abstract value as y.

We start with a generic flow function for arithmetic instructions (3). Arithmetic can produce either an even or odd value, so we use the abstract value $\top$ to represent our uncertainty. More precise flow functions are available based on certain instructions or operands. For example, if we multiply two operands one of which is even, then the result is definitely even. Similarly, if we multiply two operands both of which are odd, then the result is definitely odd. Not so obvious, is if one operand of multiplication is bottom and the other operand is odd (or top), then the result should be bottom to ensure the highest level of precision. This is because once we analyze and classify the bottom operand we could achieve a more precise result than top for x (we would still need the generic case above for instructions that do not fit such special cases):

$$\begin{aligned}
f_P[\![x := y * z]\!](\sigma) &= \sigma[x \mapsto e] && \text{where } \sigma(y) = e \vee \sigma(z) = e \\
f_P[\![x := y * z]\!](\sigma) &= \sigma[x \mapsto o] && \text{where } \sigma(y) = o \wedge \sigma(z) = o \\
f_P[\![x := y * z]\!](\sigma) &= \sigma[x \mapsto \bot] && \text{where } \sigma(y) = \bot \wedge \sigma(z) = o \\
f_P[\![x := y * z]\!](\sigma) &= \sigma[x \mapsto \bot] && \text{where } \sigma(y) = \bot \wedge \sigma(z) = \top
\end{aligned}$$

The flow function for branches ((4) and (5)) is trivial: branches do not change the state of the machine other than to change the program counter, and thus the analysis result is unaffected.

However, we can provide a better flow function for conditional branches if we distinguish the analysis information produced when the branch is taken or not taken. For example, for the true condition of the flow function for conditional branches, we know that x is zero (and therefore even) so we can update σ with the e lattice value. Conversely, in the false condition we know

nothing, $x \neq 0$ implies x may be even or odd, so we should propagate the information we already know about x to be the most precise:

$$\begin{aligned} f_P[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) &= \sigma[x \mapsto e] \\ f_P[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) &= \sigma \end{aligned}$$

2.5 Running a Dataflow Analysis

The point of developing a dataflow analysis is to compute information about possible program states at each point in a program. Even though this information contains approximations of values of variables and is, therefore, inevitably imprecise in certain situations, in practice, well-designed approximations can often allow dataflow analysis to compute quite useful information. For example, for zero analysis with fully precise flow functions, whenever we divide some expression by a variable x, we can determine in a lot of instances (more so than with a syntactic analysis) whether x must be zero (the abstract value Z) and warn the developer. Imprecision is handled by warning the developer that x may be zero in the case the abstract value of x is $\top$.

Kildall's worklist algorithm with the strongly-connected component and reverse postorder heuristics is the most efficient and correct algorithm for executing dataflow analyses that we know of from class. Therefore, we will simulate running dataflow analyses on programs using Kildall's worklist algorithm with the strongly-connected component and reverse postorder heuristics.
But first, here is the pseudocode for Kildall's worklist algorithm:

```
for Instruction i in program
    input[i] = ⊥
input[firstInstruction] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
    output = flow(i, input[i])
    for Instruction j in succs(i)
        if output ⋢ input[j]
            input[j] = input[j] ⊔ output
            add j to worklist
```

2.5.1 Running Parity Analysis with Kildall's Worklist Algorithm: An Example

We will simulate the parity analysis (defined previously) on the following program using Kildall's worklist algorithm with the strongly-connected component and reverse postorder heuristics. We will use the most precise versions of the flow functions for parity analysis defined previously, eg. the special cases for multiplication and conditional branches instead of their less precise generic counterparts. We track and report the analysis information using a table with a column for the program point, a column for the worklist, and a column for the abstract value of each variable. Each row tracks the value after the execution of the corresponding statement. The rows show how the analysis executes, examining one statement at a time:

```
1 :  x := 2
2 :  y := 3
3 :  z := 6
4 :  if y = 0 goto 8
5 :  x := x * y
6 :  y := y − 1
7 :  goto 4
8 :  y := x * z
```

Program Pt	Worklist	x	y	z
0	1	$\top$	$\top$	$\top$
1	2	e	$\top$	$\top$
2	3	e	o	$\top$
3	4	e	o	e
4	5, 8	e	e_T, o_F	e
5	6, 8	e	o	e
6	7, 8	e	$\top$	e
7	4, 8	e	$\top$	e
4	5, 8	e	$e_T, \top_F$	e
5	6, 8	e	$\top$	e
6	8	e	$\top$	e
8	$\varnothing$	e	e	e