# Lecture Notes:
# Dataflow Analysis Examples

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues[*]
clegoues@cs.cmu.edu

Zero analysis is useful for simply tracking whether a given variable is zero or not; even this simple didactic example can be used to find possible bugs in programs. We will now examine several more complex analyses, including certain well-known analyses that are, particularly (but not exclusively) useful within a compiler.

## 1 Integer Sign Analysis

Integer sign analysis tracks whether each integer in the program is positive, negative, or zero. The results of this analysis can be used to optimize a program or to circumvent errors like using a negative index into an array (or memory underflow gnerally). This analysis is broadly similar to the zero analysis discussed previously (ignoring the possibility of integer overflow, i.e., consider mathematical integers).

This problem admits natural alternatives in designing our analysis, starting with the abstract domain $L$. For example, we can prefer simplicity in favor of imprecision, defining $L$ to track only whether a value is less than zero, greater than zero, equal to zero, or unknown.

**Exercise 1**. Specify this on paper, indicating (A) the set of lattice elements, (B) the relation between them, (c) the top element and (d) the bottom element.

One way to increase precision in this analysis is to define a more precise abstract domain. For example, we might decide to track whether a value is less than zero, greater than zero, equal to zero, greater than or equal to zero, less than or equal to zero, non-zero, or unknown. In addition to (trivially) increasing the size of $L$, this also makes the ordering relation more interesting.

**Exercise 2**. Specify this on paper, indicating (A) the set of lattice elements, (B) the relation between them, (c) the top element and (d) the bottom element.

## 2 Constant Propagation

*Constant propagation analysis* attempts to track the constant values of variables in the program, where possible. Constant propagation has long been used in compiler optimization passes in order to turn variable reads and computations into constants. However, it is generally useful for

[*]These notes were developed together with Jonathan Aldrich

analysis for program correctness as well: any client analysis that benefits from knowing program values (e.g. an array bounds analysis) can leverage it.

For constant propagation, we want to track what is the constant value, if any, of each program variable. Therefore we will use a lattice where the set $L_{CP}$ is $\mathbb{Z} \cup \{\top, \bot\}$. The partial order is $\forall l \in L_{CP} : \bot \sqsubseteq l \wedge l \sqsubseteq \top$. In other words, $\bot$ is below every lattice element and $\top$ is above every element, but otherwise lattice elements are incomparable.

In the above lattice, as well as our earlier discussion of zero analysis, we used a lattice to describe individual variable values. We can lift the notion of a lattice to cover all the dataflow information available at a program point. This is called a *tuple lattice*, where there is an element of the tuple for each of the variables in the program. For constant propagation, the elements of the set $\sigma$ are maps from *Var* to $L_{CP}$, and the other operators and $\top/\bot$ are lifted as follows:

$$\begin{array}{rcl}
\sigma & \in & \mathit{Var} \rightarrow L_{CP} \\
\sigma_1 \sqsubseteq_{lift} \sigma_2 & \mathit{iff} & \forall x \in \mathit{Var} : \sigma_1(x) \sqsubseteq \sigma_2(x) \\
\sigma_1 \sqcup_{lift} \sigma_2 & = & \{x \mapsto \sigma_1(x) \sqcup \sigma_2(x) \mid x \in \mathit{Var}\} \\
\top_{lift} & = & \{x \mapsto \top \mid x \in \mathit{Var}\} \\
\bot_{lift} & = & \{x \mapsto \bot \mid x \in \mathit{Var}\}
\end{array}$$

We can likewise define an abstraction function for constant propagation, as well as a lifted version that accepts an environment $E$ mapping variables to concrete values. We also define the initial analysis information to conservatively assume that initial variable values are unknown. Note that in a language that initializes all variables to zero, we could make more precise initial dataflow assumptions, such as $\{x \mapsto 0 \mid x \in \mathit{Var}\}$:

$$\begin{array}{rcl}
\alpha_{CP}(n) & = & n \\
\alpha_{lift}(E) & = & \{x \mapsto \alpha_{CP}(E(x)) \mid x \in \mathit{Var}\} \\
\sigma_0 & = & \top_{lift}
\end{array}$$

We can now define flow functions for constant propagation:

$$\begin{array}{lll}
f_{CP}[\![x := n]\!](\sigma) & = \sigma[x \mapsto \alpha_{CP}(n)] & \\
f_{CP}[\![x := y]\!](\sigma) & = \sigma[x \mapsto \sigma(y)] & \\
f_{CP}[\![x := y \; op \; z]\!](\sigma) & = \sigma[x \mapsto \sigma(y) \; op_{lift} \; \sigma(z)] & \\
& \quad \text{where } \; n \; op_{lift} \; m = n \; op \; m & \\
& \quad \text{and } \; n \; op_{lift} \; \bot = \bot & \text{(and symmetric)} \\
& \quad \text{and } \; n \; op_{lift} \; \top = \top & \text{(and symmetric)} \\
f_{CP}[\![\text{goto } n]\!](\sigma) & = \sigma & \\
f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) & = \sigma[x \mapsto 0] & \\
f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) & = \sigma & \\
f_{CP}[\![\text{if } x < 0 \text{ goto } n]\!](\sigma) & = \sigma &
\end{array}$$

We can now look at an example of constant propagation. Below, the code is on the left, and the results of the analysis is on the right. In this table we show the worklist as it is updated to show how the algorithm operates:

| | |
|---|---|
| 1 : | $x := 3$ |
| 2 : | $y := x + 7$ |
| 3 : | if $z = 0$ goto 6 |
| 4 : | $z := x + 2$ |
| 5 : | goto 7 |
| 6 : | $z := y - 5$ |
| 7 : | $w := z - 2$ |

| stmt | worklist | x | y | z | w |
|---|---|---|---|---|---|
| 0 | 1 | $\top$ | $\top$ | $\top$ | $\top$ |
| 1 | 2 | 3 | $\top$ | $\top$ | $\top$ |
| 2 | 3 | 3 | 10 | $\top$ | $\top$ |
| 3 | 4,6 | 3 | 10 | $0_T, \top_F$ | $\top$ |
| 4 | 5,6 | 3 | 10 | 5 | $\top$ |
| 5 | 6,7 | 3 | 10 | 5 | $\top$ |
| 6 | 7 | 3 | 10 | 5 | $\top$ |
| 7 | $\emptyset$ | 3 | 10 | 5 | 3 |

## 3 Reaching Definitions

Reaching definitions analysis determines, for each use of a variable, which assignments to that variable might have set the value seen at that use. Consider the following program:

| | |
|---|---|
| 1 : | $y := x$ |
| 2 : | $z := 1$ |
| 3 : | if $y = 0$ goto 7 |
| 4 : | $z := z * y$ |
| 5 : | $y := y - 1$ |
| 6 : | goto 3 |
| 7 : | $y := 0$ |

In this example, definitions 1 and 5 reach the use of $y$ at 4.

**Exercise 3**. Which definitions reach the use of $z$ at statement 4?

Reaching definitions can be used as a simpler but less precise version of constant propagation, zero analysis, etc. where instead of tracking actual constant values we just look up the reaching definition and see if it is a constant. We can also use reaching definitions to identify uses of undefined variables, e.g. if no definition from the program reaches a use.

For reaching definitions, we define a new kind of lattice: a *set lattice*. Here, a dataflow lattice element is the set of definitions that reach the current program point. Assume that DEFS is the set of all definitions in the program. The set of elements in the lattice is the set of all subsets of DEFS—that is, the powerset of DEFS, written $\mathcal{P}^{\mathsf{DEFS}}$.

What should $\sqsubseteq$ be for reaching definitions? The intuition is that our analysis is more precise the *smaller* the set of definitions it computes at a given program point. This is because we want to know, as precisely as possible, where the values at a program point came from. So $\sqsubseteq$ should be the subset relation $\subseteq$: a subset is more precise than its superset. This naturally implies that $\sqcup$ should be *union*, and that $\top$ and $\bot$ should be the universal set DEFS and the empty set $\emptyset$, respectively.

In summary, we can formally define our lattice and initial dataflow information as follows:

$$\begin{array}{ccl} \sigma & \in & \mathcal{P}^{\mathsf{DEFS}} \\ \sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\ \top & = & \mathsf{DEFS} \\ \bot & = & \emptyset \\ \sigma_0 & = & \emptyset \end{array}$$

Instead of using the empty set for $\sigma_0$, we could use an artificial reaching definition for each program variable (e.g. $x_0$ as an artificial reaching definition for $x$) to denote that the variable is either uninitialized, or was passed in as a parameter. This is convenient if it is useful to track whether a variable might be uninitialized at a use, or if we want to consider a parameter to be a definition. We could write this formally as $\sigma_0 = \{x_0 \mid x \in \mathsf{Vars}\}$

We will now define flow functions for reaching definitions. Notationally, we will write $x_n$ to denote a definition of the variable $x$ at the program instruction numbered $n$. Since our lattice is a set, we can reason about changes to it in terms of elements that are added (called GEN) and elements that are removed (called KILL) for each statement. This GEN/KILL pattern is common to many dataflow analyses. The flow functions can be formally defined as follows:

$$
\begin{array}{lll}
f_{RD}[\![I]\!](\sigma) & = \sigma - KILL_{RD}[\![I]\!] \cup GEN_{RD}[\![I]\!] & \\
\\
\text{KILL}_{RD}[\![n\text{: } x := ...]\!] & = \{x_m \mid x_m \in \mathsf{DEFS}(x)\} & \\
\text{KILL}_{RD}[\![I]\!] & = \varnothing & \text{if } I \text{ is not an assignment} \\
\\
\text{GEN}_{RD}[\![n\text{: } x := ...]\!] & = \{x_n\} & \\
\text{GEN}_{RD}[\![I]\!] & = \varnothing & \text{if } I \text{ is not an assignment}
\end{array}
$$

We would compute dataflow analysis information for the program shown above as follows:

| stmt | worklist | defs |
|---|---|---|
| 0 | 1 | $\varnothing$ |
| 1 | 2 | $\{y_1\}$ |
| 2 | 3 | $\{y_1, z_1\}$ |
| 3 | 4,7 | $\{y_1, z_1\}$ |
| 4 | 5,7 | $\{y_1, z_4\}$ |
| 5 | 6,7 | $\{y_5, z_4\}$ |
| 6 | 3,7 | $\{y_5, z_4\}$ |
| 3 | 4,7 | $\{y_1, y_5, z_1, z_4\}$ |
| 4 | 5,7 | $\{y_1, y_5, z_4\}$ |
| 5 | 7 | $\{y_5, z_4\}$ |
| 7 | $\varnothing$ | $\{y_7, z_1, z_4\}$ |

## 4 Live Variables

Live variable analysis determines, for each program point, which variables might be used again before they are redefined. Consider again the following program:

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

In this example, after instruction 1, $y$ is live, but $x$ and $z$ are not. Live variables analysis typically requires knowing what variable holds the main result(s) computed by the program. In the program above, suppose $z$ is the result of the program. Then at the end of the program, only $z$ is live.

Live variable analysis was originally developed for optimization purposes: if a variable is not live after it is defined, we can remove the definition instruction. For example, instruction 7 in the code above could be optimized away, under our assumption that $z$ is the only program result of interest.

We must be careful of the side effects of a statement, of course. Assigning a variable that is no longer live to null could have the beneficial side effect of allowing the garbage collector to collect memory that is no longer reachable—unless the GC itself takes into consideration which variables are live. Sometimes warning the user that an assignment has no effect can be useful for software engineering purposes, even if the assignment cannot safely be optimized away. For example, eBay found that FindBugs's analysis detecting assignments to dead variables was useful for identifying unnecessary database calls.[1]

For live variable analysis, we will use a set lattice to track the set of live variables at each program point. The lattice is similar to that for reaching definitions:

$$\begin{array}{ccl} \sigma & \in & \mathcal{P}^{\mathsf{Var}} \\ \sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\ \top & = & \mathsf{Var} \\ \bot & = & \varnothing \end{array}$$

What is the initial dataflow information? This is a tricky question. To determine the variables that are live at the start of the program, we must reason about how the program will execute...i.e. we must run the live variables analysis itself! There's no obvious assumption we can make about this. On the other hand, it is quite clear which variables are live at the *end* of the program: just the variable(s) holding the program result.

Consider how we might use this information to compute other live variables. Suppose the last statement in the program assigns the program result $z$, computing it based on some other variable $x$. Intuitively, that statement should make $x$ live immediately above that statement, as it is needed to compute the program result $z$—but $z$ should now no longer be live. We can use similar logic for the second-to-last statement, and so on. In fact, we can see that live variable analysis is a *backwards analysis*: we start with dataflow information at the *end* of the program and use flow functions to compute dataflow information at earlier statements.

Thus, for our "initial" dataflow information—and note that "initial" means the beginning of the program analysis, but the end of the program—we have:

$$\sigma_{end} \quad = \quad \{x \mid x \text{ holds part of the program result}\}$$

We can now define flow functions for live variable analysis. We can do this simply using GEN and KILL sets:

$$\begin{aligned} \text{KILL}_{LV}[\![I]\!] &= \{x \mid I \text{ defines } x\} \\ \text{GEN}_{LV}[\![I]\!] &= \{x \mid I \text{ uses } x\} \end{aligned}$$

We would compute dataflow analysis information for the program shown above as follows. Note that we iterate over the program backwords, i.e. reversing control flow edges between instructions. For each instruction, the corresponding row in our table will hold the information

[1]see Ciera Jaspan, I-Chin Chen, and Anoop Sharma, *Understanding the value of program analysis tools*, OOPSLA practitioner report, 2007

after we have applied the flow function—that is, the variables that are live immediately *before* the statement executes:

| stmt | worklist | live |
|---|---|---|
| end | 7 | $\{z\}$ |
| 7 | 3 | $\{z\}$ |
| 3 | 6,2 | $\{z, y\}$ |
| 6 | 5,2 | $\{z, y\}$ |
| 5 | 4,2 | $\{z, y\}$ |
| 4 | 3,2 | $\{z, y\}$ |
| 3 | 2 | $\{z, y\}$ |
| 2 | 1 | $\{y\}$ |
| 1 | $\emptyset$ | $\{x\}$ |