Dataflow Analysis

Colin Unger

February 27, 2020

1 Introduction

Dataflow analysis is a generalized framework for analyzing programs. A “program analysis” is an automated technique for synthesizing judgements about a program: for example, a program analysis might judge whether or not a buffer overflow could occur in a program, or what values in a program are constant, or which pointer values could potentially point to the same location. Program analysis is broadly used in the field of compilers and programming languages, but is also very important in reverse engineering in binary analysis—a decompiler like IDA Pro or Ghidra will typically perform tens to hundreds of analyses internally to transform the bytes of an executable into the pseudocode that reverse engineers often rely on.

1.1 Approximations

When analyzing programs, it is important to recognize that solving the problem of program analysis is, in general, impossible—similar to the halting problem, answering any nontrivial property of a program is ultimately undecidable. As such, we are forced to approximate. Since approximation is such a key aspect of a program analysis, we often group analyses based on the properties their approximations define: “sound”, “precise”, both, or neither.

A sound analysis guarantees that if the analysis shows that a property does not hold for a program, then there does not exist a concrete execution that displays this property. As an example, take a analysis that examines a program to determine if it contains a buffer overflow. If this analysis is sound, then if the analysis determines that there are no buffer overflows, then it is guaranteed that a buffer overflow will never occur in the execution of the program. On the other hand, if the analysis alerts us to the presence of a potential overflow, we cannot say for certain whether the overflow exists (a “true positive”) or is simply a result of “overapproximating” (a “false positive”). In dataflow analysis, we typically use sound analyses, as these have historically been a primary focus. However, dataflow analysis is capable of encoding other approximations too, and these other approximations can often be quite useful.

A precise analysis guarantees the opposite of a sound analysis—if it shows a property (in our example, the existence of a buffer overflow) then there must be a concrete execution that displays this buffer overflow, but if it claims the absence of a buffer overflow, we do not know if this is because the program is overflow-free or if it an artifact from the analysis’s “underapproximation” of the program. Said another way, a precise analysis guarantees no false positives, but allows false negatives. In contrast to a sound analysis, a precise analysis guarantees

However, it turns out that often times the approximations necessary to produce a sound or precise analysis are quite extreme, resulting in a large number of false positives or false negatives respectively.

Often, we can get significantly improved performance if we sacrifice both soundness and precision—now we can have both false positives *and* false negatives, but often times we can significantly decrease the total number of false results. These analyses are often jokingly termed "soundy."

Finally, it is possible to guarantee both soundness and precision at the cost of guaranteed termination—if our analysis terminates, the result will have no false positives or false negatives. This, for example, is the approach typically used in symbolic execution when loop depth is not bounded. However, for most real-world programs, termination of these analyses is unlikely in practice, so one of the above approximations are typically required.

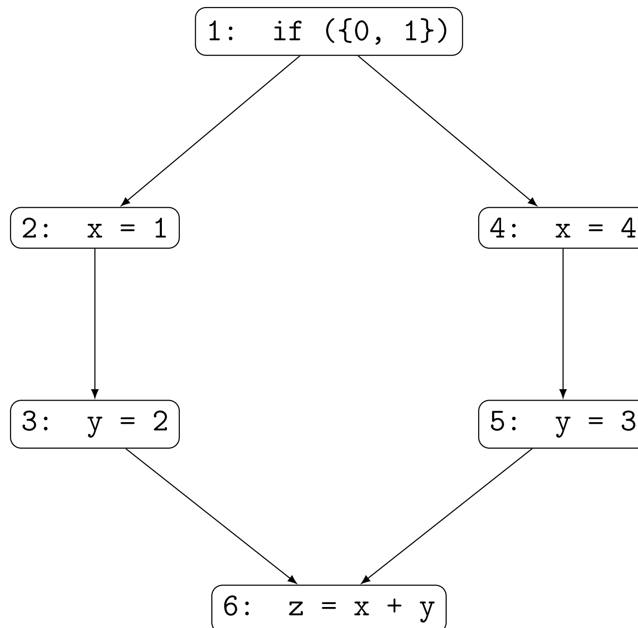
1.2 Intuition

Now that we understand the constraints of the problem we face in program analysis, we'll walk through a basic example of dataflow analysis that appeals to our intutition—we'll formalize this all later.

For the rest of these notes, we'll be analyzing the following language. Since the focus of this class is not on formalizing program syntax and semantics, we appeal to the reader's intution in terms of the exact semantics of the operations represented. Our language is only be capable of computing on integers, and even then can only perform the operations of addition (+) and comparison (<=) on these numbers. To simulate interactions with the external environment (reading input from the user, generating random numbers, reading from the network, etc.) we add non-deterministic choice to our language: the expression `{1, 2, 3}` will non-deterministically evaluate to either `1`, `2`, or `3` at runtime. Similar to interactions with the environment, we are not able to predict the value that it will be evaluated to without running the program. Our language has mutable variables as well as typical control-flow structures (loops, branches, etc.). Below is a simple example program:

```
1: if ({0, 1}) {
2:   x = 1
3:   y = 2
   } else {
4:   x = 4
5:   y = 3
   }
6: z = x + y
```

For simplicity, we will examine a toy problem in program analysis: the question of determining the parity (even or odd) of each program variable before and after every point in our program (the numbers at the beginning of certain lines of our program). We will write the location right before the label `n` as $\mathtt{n}_\uparrow$ and the location right after the label as $\mathtt{n}_\downarrow$. The first key aspect of dataflow analysis is that we do not perform it on a standard syntactic representation of the program—instead, we first convert the program to a control-flow graph (CFG) similar to the one generated by your recursive disassembler and then perform our analysis on that graph. The CFG for our example program is



The second key aspect is that we are going to pretend that instead of operating on integers, our program actually operates on new domain of values that expresses the judgements we want to make about our program. This new domain is typically called the "abstract domain". In the case of our parity analysis, we will want to have the values EVEN and ODD, as these are the judgements we are interested in making. However, there are obviously program locations where a variable could take on both an even and an odd value—for example, at $6_\uparrow$ in our example program, `x` could be either 1 or 4. Thus, we will need a value to represent a value on which we cannot make judgements. Historically, this value is called "top", written as the symbol $\top$. There remains a few program points at which we still do not know the value—before the variable is assigned. For example, consider $1_\uparrow$: what is the parity of x at this point? It may seem intuitive to assign `x` to $\top$, but this only makes sense if the language sets `x` randomly at the start of the program. What would be more accurate is if we had a way of specifying that a variable has no possible parities (unlike $\top$ which represents both parities)—typically, this value is called "bottom" (written $\bot$). The distinction between $\top$ and $\bot$ may seem a bit arbitrary for now, but should become clear once we dive into the mathematical foundations of dataflow analysis.

With these four values, we have our full domain:

$$\textsc{Parity} = \{\top, \textsc{Even}, \textsc{Odd}, \bot\}$$

Now to compute with these, we'll have to do two things: first of all, we'll have to lift the concrete integers in the program to elements of PARITY, and we'll have to define how the operations our program performs on concrete integers maps to operations on elements of PARITY. Fortunately, these should be essentially exactly what you'd intuitively expect! For example, we'll define a function $\alpha : \text{Pow}\,\mathbb{Z} \to \textsc{Parity}$ that will lift sets of concrete integers into the abstract domain. Intuitively, the reason we must operate on sets of integers is that we can have non-deterministic choice of one of many values in our program, so we must be able to simultaneously lift more than one value (for example, `{0, 1}` at `1`). We'll see exactly more formally why we want α to operate on sets later when we discuss Galois connections.

The definition for α is quite simple:

$$\alpha(s) = \begin{cases} \bot & \text{if } s = \varnothing \\ \textsc{Even} & \text{if } \forall x \in s \;.\; x \text{ is even} \\ \textsc{Odd} & \text{if } \forall x \in s \;.\; x \text{ is odd} \\ \top & \text{otherwise} \end{cases}$$

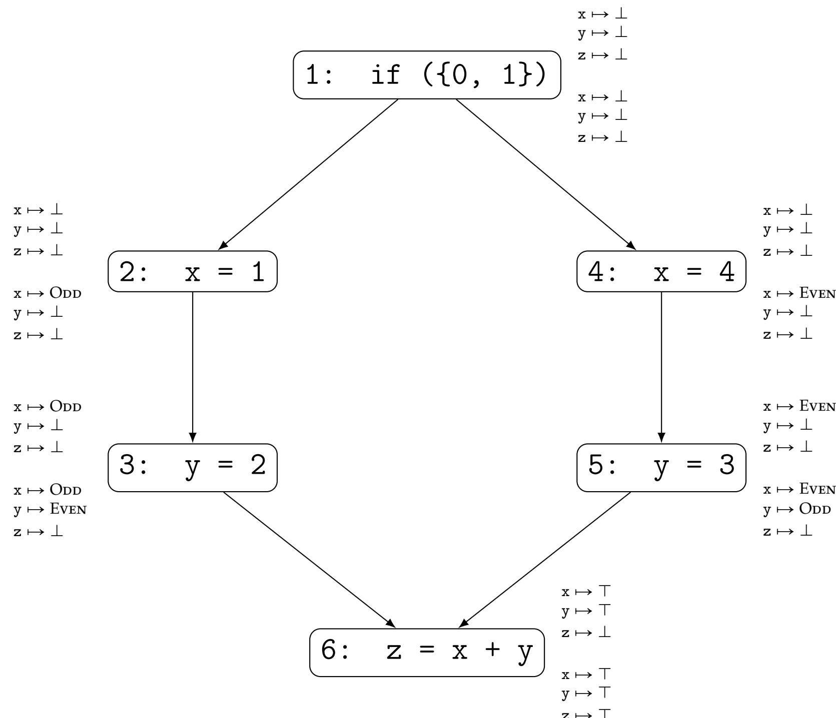
Now we'll need to define addition:

+	$\bot$	Odd	Even	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$
Odd	$\bot$	Even	Odd	$\top$
Even	$\bot$	Odd	Even	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

and similarly comparison:

<=	$\bot$	Odd	Even	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$
Odd	$\bot$	$\top$	$\top$	$\top$
Even	$\bot$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

With these, we can now step through our CFG in sequential order computing on these abstract values. The set of current abstract values is sometimes referred to as the "abstract state." The result of this is:



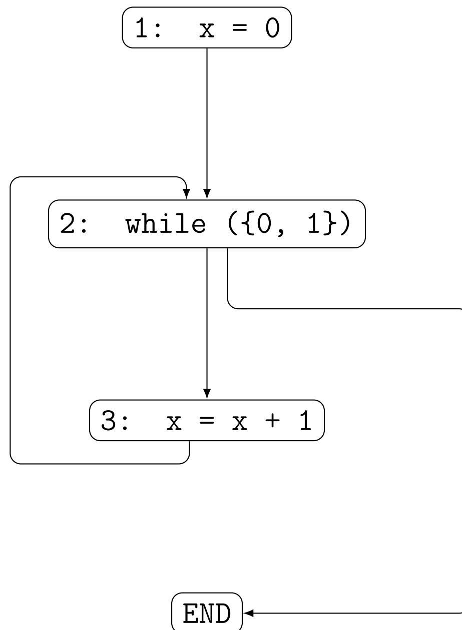
At nearly every single node, the abstract state at the end of the prior node is the same as the abstract state at the beginning of the following node. This makes sense—we would not expect the abstract state to spontaneously change without a program operation being performed. However, at $6_\uparrow$, this does not occur: we transition from $\{x \mapsto \textsc{Odd}, y \mapsto \textsc{Even}, z \mapsto \bot\}$ at $3_\downarrow$ and $\{x \mapsto \textsc{Even}, y \mapsto \textsc{Odd}, z \mapsto \bot\}$ at $5_\downarrow$ to $\{x \mapsto \textsc{Odd}, y \mapsto \textsc{Even}, z \mapsto \bot\}$ at $6_\uparrow$! This is because we have reached a so-called "join point"—we have two states incoming into the same block. Now, if we were performing standard symbolic execution, we would expect to simply execute each of the paths separately, and thus derive

the result that z ↦ Odd at $6_\downarrow$. However, this is the fundamental approximation that static analysis uses to maintain decidability: instead of executing the paths separately, we merge them back into one state at join points. Merging x ↦ Odd and x ↦ Even should give us x ↦ ⊤, since in the merged state x is capable of being odd (if it comes from 3) or even (if it comes from 5). Unfortunately, this forces us to make the judgement that z ↦ ⊤ at $6_\downarrow$—an overapproximation of the property we wanted to examine (parity). In fact, if we formally analyzed this analysis, we would find that it is in fact sound, as we sometimes incorrectly state a wider range of possible parities than is possible, but we never state a narrower range.

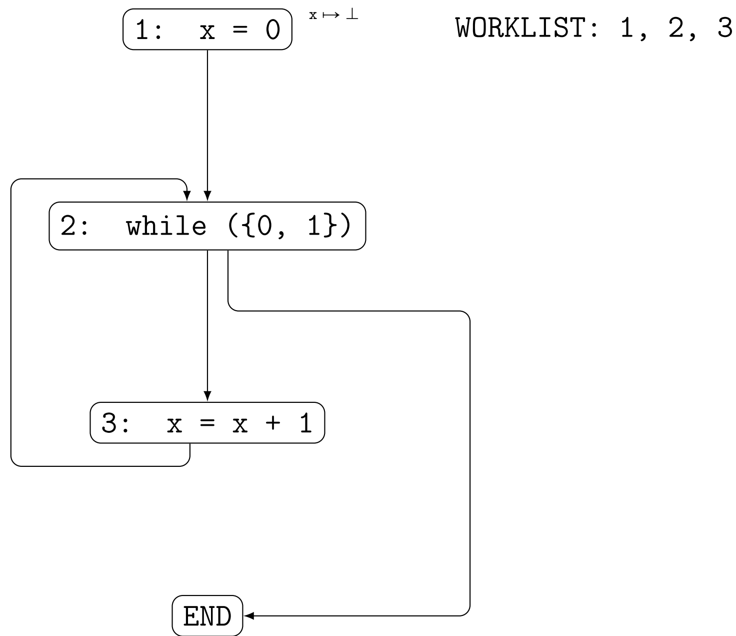
However, as we know from symbolic execution, the challenging part of analyzing a program is not typically branches, but loops. To see how dataflow analysis handles loops, we'll examine the following program:

```
1: x = 0
2: while ({0, 1}) {
3:   x = x + 1
   }
```

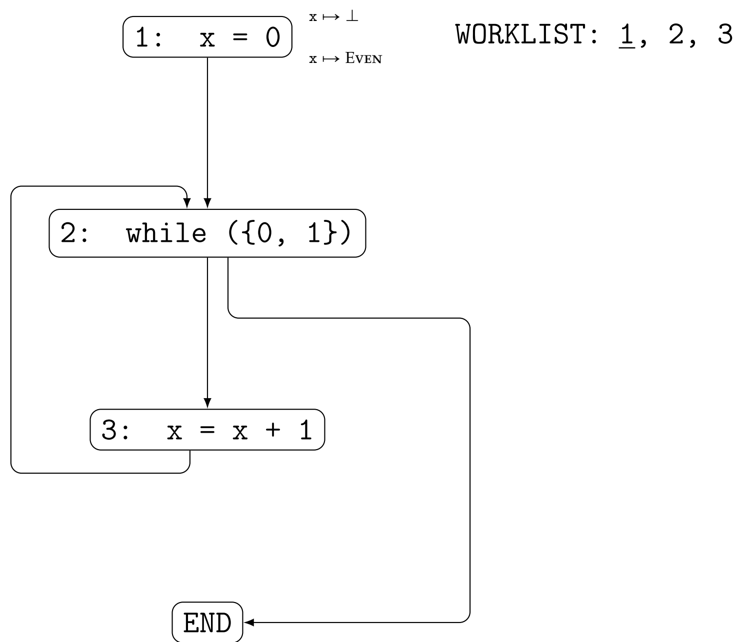
As always with dataflow analysis, we immediately transform this program into its CFG:



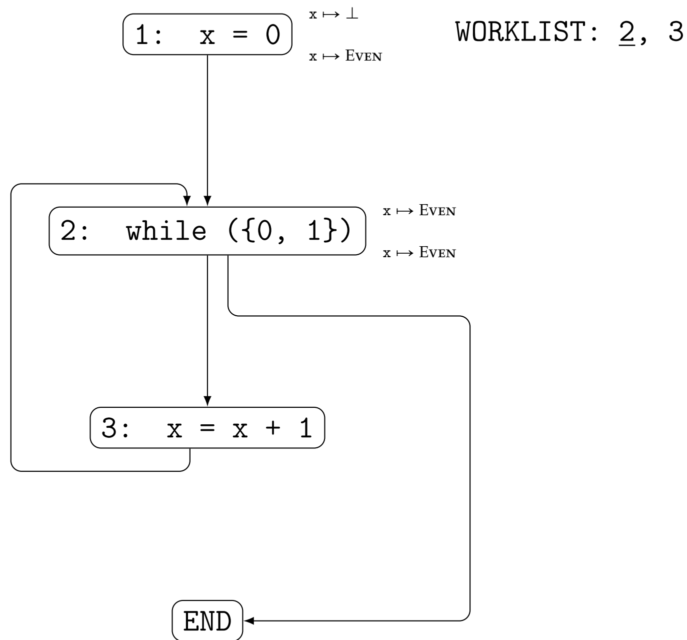
Now, as before, we traverse the nodes of our graph in order. However, now that our graph is cyclic, the order and stopping condition is not quite so obvious! The correct answer turns out to be to use a worklist algorithm as in your recursive disassembler. We start off with



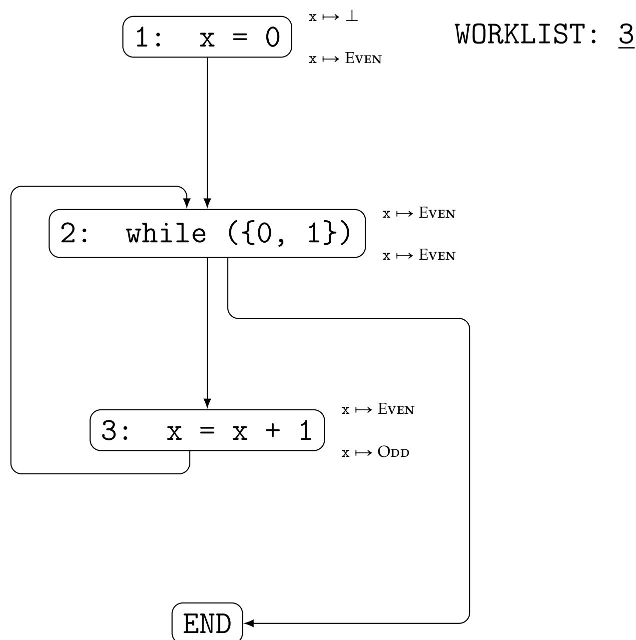
Following the worklist algorithm, we first remove 1 and perform its computation on the incoming state $1_\uparrow$. Note that we signify the current node which has been removed from the worklist and is being processed as still briefly in the worklist and underlined:



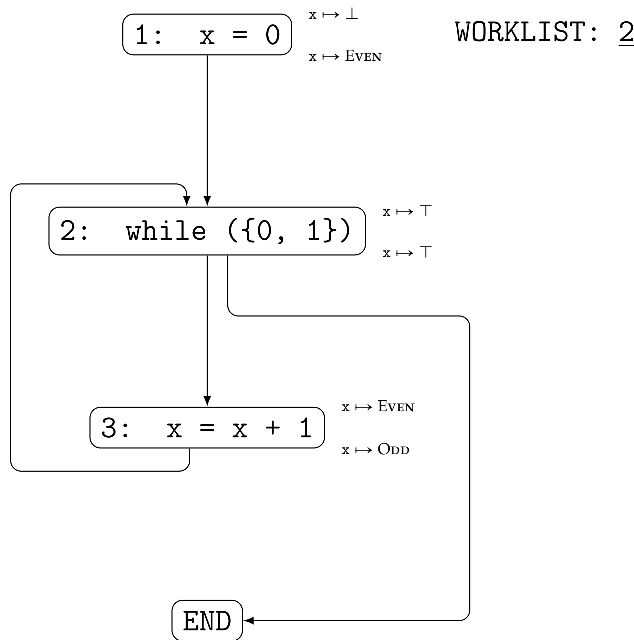
We then process 2, using the incoming abstract state from $1_\downarrow$:



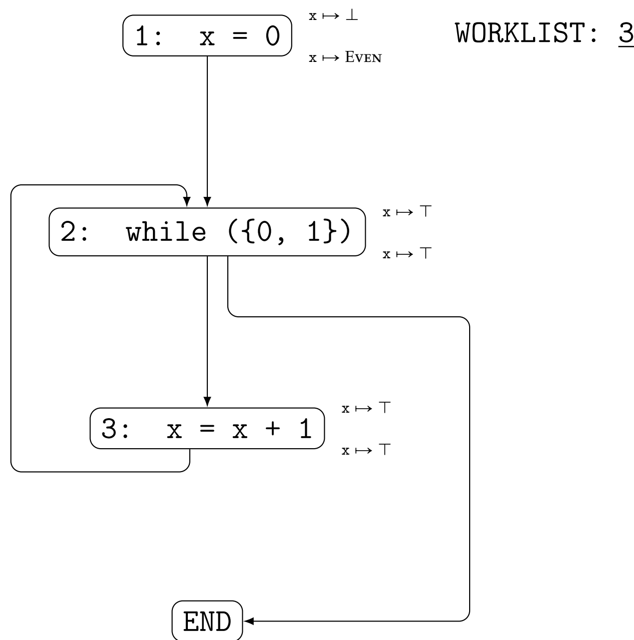
Then 3:



However, now we see the effect of loops: by updating $3_\downarrow$, we've rendered the state at $2_\uparrow$ incorrect as it should have been merged with $\{x \mapsto \textsc{Odd}\}$! To handle this, whenever an outgoing state changes and this change modifies the input state to a child state, we'll add these children to the work list if they aren't there already—therefore, in this case, we'll add 2 to the worklist. Now, since 2 is on the worklist, we remove it and process its node:



And now since the state at $2_\downarrow$ has changed, we need to add 3 to the worklist and process it:



Now since merging our new state for $3_\downarrow$ with the existing one at $2_\uparrow$ yields no change, we add nothing to the worklist. Our worklist is now empty, so our analysis is completed!

In our diagrams above, we started with all abstract states simply absent. While this is clearer for illustrative purposes, it serves to artificially complicate the algorithm: it is equivalent to simple starting with every abstract state set to $\{x \mapsto \bot\}$. Secondly, instead of viewing each state having a pair of abstract states, we can simply view the entire program as having one giant abstract states which is simply a giant map from before- and after-locations to our prior abstract states. While this may seem like a useless way of viewing the analysis, it is convenient since it allows us to view execution of the program on our concrete value as simply a single function application: given prior full-program abstract state s, we can simple step this state to a new state s' by applying a summary of our entire program as operating on abstract states using the abstract operations we defined earlier (we call this function F)—therefore, $s' = F(s)$. Our worklist algorithm then becomes equivalent to simply repeatedly applying F until s stops changing.

You may, however, ask if this is in fact guaranteed to always terminate. For example, if we use

the trivial abstract domain CONCRETE = Pow $\mathbb{Z}$, we will match our program's concrete execution exactly, and many programs (like this one) are not guaranteed to terminate. Thus, it is important to know what divides abstract domains with guaranteed termination with those that do not have it, as well as to establish certain properties of correctness between the concrete and abstract domains. Thus is what will explore in the next section.

2 Mathematical Foundations of Dataflow Analysis

2.1 Preliminaries

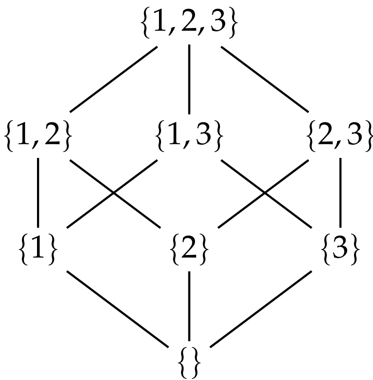
Definition 2.1. A "binary relation" on a set S is a subset of $S \times S$. Typically we denote a binary relationship with $\sqsubseteq$. We also often write $x \sqsubseteq y$ instead of $(x, y) \in (\sqsubseteq)$.

If $\{(x,x) \mid x \in S\} \subseteq (\sqsubseteq)$, we say that $\sqsubseteq$ is "reflexive." If $x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$, we say that $\sqsubseteq$ is "transitive." If $a \sqsubseteq b \iff b \sqsubseteq a$, we say that $\sqsubseteq$ is "symmetric." If $a \neq b \wedge a \sqsubseteq b \implies b \not\sqsubseteq a$, we say that $\sqsubseteq$ is "antisymmetric."

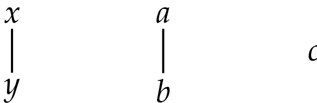
Definition 2.2. A "preorder" is a reflexive, transitive binary relation.

Definition 2.3. An "equivalence relation" is a symmetric preorder. A "partial order" is an antisymmetric preorder. A partially ordered set is typically abbreviated "poset."

We can visualize partial orderings as "Hasse diagrams." We represent the elements of S as nodes of a graph, and represent the binary relation as upward paths in the graph. Note thus that the direction in which you draw the graph is significant: larger elements must be drawn above smaller elements. For example, for $S = \text{Pow}\{1,2,3\}$ and $(\sqsubseteq) = (\subseteq)$, the Hasse diagram is:



Note that partial orders do not have to be connected—the following is a valid Hasse diagram:



Definition 2.4. A "total order" is a partial order that orders all pairs of elements.

Definition 2.5. The "least upper bound" or "join" (typically represented by the symbol $\sqcup$) of a set is the least element that is greater than all elements in the set. Note that such a least upper bound is not guaranteed to exist in a poset.

Similarly, the "greatest lower bound" or "meet" (typically represented by the symbol $\sqcap$) of a set is the greatest element that is less than all elements in the set. Again, this is not guaranteed to exist.

Definition 2.6. A "chain" is a totally ordered subset of a poset.

Definition 2.7. A "complete partial order" (typically abbreviated "CPO") is a partial order where every chain has a join.

Note that all finite posets are CPOs.

Definition 2.8. A "pointed CPO" is a CPO with a least element. Often, this element is a called "bottom" (represented by the symol $\bot$).

Definition 2.9. A "lattice" is a partial order where every two elements has a join and a meet.

Definition 2.10. A poset satisfies the "finite ascending chain condition" if there does not exist an infinite sequence $a_1, a_2, \ldots$ such that $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$. We call a poset that satisfies the finite ascending chain condition "noetherian."

Definition 2.11. A lattice is "bounded" if there exists a single greatest element $\top$ and a single least element $\bot$. A lattice is "complete" if every subset has a join and a meet. Note that all complete lattices are also bounded lattices.

Definition 2.12. Given a function $f : S \to S$, an element $x \in S$ is a "fixed point" (also termed "fixpoint") of f if $f(x) = x$.

Definition 2.13. Let S be a poset. The least fixed point of a function $f : S \to S$ is the fixed point u of f such that for any other fixed points y of f, $u \sqsubseteq y$. An element $x \in S$ is a "pre-fixed point" of f if $x \sqsubseteq f(x)$. Similarly, an element $x \in S$ is a "post-fixed point" of f if $f(x) \sqsubseteq x$.

Definition 2.14. Let R, S be posets. A function $f : R \to S$ is "monotonic" if $x \sqsubseteq_R y \implies f(x) \sqsubseteq_S f(y)$. Note that it is important to distinguish monotonicity from that of "extensivity": a function f is an "extensive" function if $x \sqsubseteq f(x)$. Extensivity captures the concept of "always increasing", while monotonicity captures the concept of "preserving ordering."

Theorem 2.1 (Tarski's Theorem)**.** Let L be a complete lattice and $f : L \to L$ monotone, then the set of fixed points of f is a complete lattice.

The significance of Tarski's Theorem may not immediately seem clear, so let's unpack what it's saying. The key aspect is that the set of fixed points is a complete lattice. By definition, in any complete lattice there exists a least element. Thus, if we satisfy the conditions of Tarski's Theorem, a function must have a least fixed point!

To relate this back to dataflow analysis, remember that we can view the algorithm as computing with one giant abstract state mapping program locations to "mini abstract states", and executing on this abstract state as a single function F that summarizes a single step of our whole program, and thus that we terminate and return the abstract state s when $F(s) = s$. But this is just a fixed point! It turns out that we don't just want any fixed point though: for any reasonable program analysis, assigning every variable at every program location to $\top$ is immediately a fixed point since intutively, if all variables could be random at every point, we'll never be able to make any judgements about the program's behavior. Thus, we don't want just any fixed point, we want to the most precise fixed point, i.e. the least fixed point. With Tarski's theorem, we know that if we design our lattice of combined abstract states to be a lattice and make f (typically called the transition function)

monotonic, then this least fixed point exists. However, at the moment we have no algorithmic way to find it. That takes us to our second key theorem:

Definition 2.15 (Kleene)**.** We will denote the least fixed point of a function f greater than an element x by $\mathrm{lfp}_x f = f^n(x)$, where $f^n(x) = f(f(\overset{n}{\cdots}(f(x))\cdots))$. Let L be a complete Noetherian lattice and $f : L \to L$ monotone and x a pre-fixed point of f; then $\exists(n \in \mathbb{N})$ such that $\mathrm{lfp}_x f = f^n(x)$.

Thus, if we satisfy the conditions of Theorem 2.15, all we have to do to find a least fixed point is repeatedly call the function—the same algorithm that we intuitively came up with in the introduction! To find the least fixed point we'll also need an element $x \sqsubseteq \mathrm{lfp}_x f$, but fortunately, we can just use the abstract state with every variable at every program location assigned to $\bot$, and thus we easily satisfy this inequality—notice that this is also the initial state we used in the introduction.

Now we have a way to find a fixed point in our lattice of abstract values, but it would be nice to be able to formalize the connection between our abstract values and the actual execution of our program. For example, in the case of the Parity domain, intuitively Even represents the set of all even integers, Odd represents the set of all odd integers, $\top$ represents $\mathbb{Z}$, and $\bot$ represents $\varnothing$. However, as of now, we have no way to express this connection, or to use the connection to understand whether or not our analysis actually correctly approximates the execution of our program. We could imagine defining a pair of functions $\alpha : \mathrm{Pow}\,\mathbb{Z} \to \textsc{Parity}$ and $\gamma : \textsc{Parity} \to \mathrm{Pow}\,\mathbb{Z}$, where α maps sets of concrete values (we can think of α mapping single values by mapping their singleton sets) to the corresponding abstract values, and γ capturing what concrete values these abstract values represent. We've already run into α, typically referred to as the "abstraction function" in the introduction. γ is simply its opposite, and is typically referred to as the "concretization function." However, there are still properties we would want to hold for these α and γ: for example, if $\alpha(\{1\}) = \textsc{Odd}$, it would seem wrong for $1 \notin \gamma(\textsc{Odd})$—essentially, we wouldn't be lifting our values to the proper abstract value. It turns out that the conditions on our abstraction and concretization function can be expressed by a "Galois connection":

Definition 2.16. Let R, S be posets and $F : A \to B$ and $G : B \to A$. R and S form a Galois connection (typically written $R \underset{F}{\overset{G}{\leftrightarrows}} S$) if the following conditions hold:

1. F, G are monotone, and
2. $\forall r \in R, s \in S,\ F(r) \sqsubseteq_S s \iff r \sqsubseteq_R G(r)$.

The condition that we will want is that our concrete lattice $\mathrm{Pow}\,\mathbb{Z}$ where $\sqsubseteq = \subseteq$ and our abstract lattice Parity form a Galois connection under α and γ:

$$\mathrm{Pow}\,\mathbb{Z} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \textsc{Parity}$$

Once we have defined α and γ, we now have a well-defined relationship between concrete and abstract values. It feels like this should also give us a well-defined relationship between concrete and abstract operations, and indeed it does: for any abstract operation f and corresponding concrete operation $f^{\#}$, the most precise f possible is $f = \alpha \circ f^{\#} \circ \gamma$. Intutively, this corresponds to simply concretizing the given abstract values, calculating the set of results from performing the operation on the set of concrete values, and then abstracting the set of results. Note that we cannot use this while actually performing datafloew analysis on an actual program, since concretizing the abstract value may lead to an infinite set. However, we can use it to check that the abstract operation we define is as precise as possible by proving its equivalence.