Intro to Abstract Interpretation (notes from September 10,2018)
Mark Goldstein

Last time: type inference for refinement/dependent types. Start off with traditional type inference, then build horn clauses that constrain the types. Need finite set of decidable refinements. Efficiency is also a concern. Abstract Interpretation gives a framework to systematically make such kinds of analyses and many more.

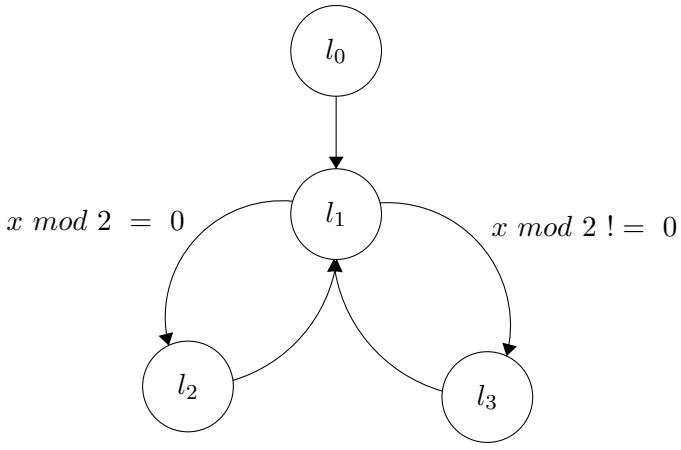
Consider the following simple imperative program:

```
x := 0;
while true do
    if x mod 2 = 0 then
        x := x+2;
    else x:= x+1;
end
```

You might notice that there is no loop bound here, but it isn't essential for the analysis that we will make). Consider the verification problem of proving that the else statement is never reached, perhaps because something bad happens there. An optimizer may also want to prove code is not reached to remove it and simplify the code.

First think about how to formalize semantics of such programs. Programs are control flow graphs.

```
l0: x:= 0
l1: while true do
              if x mod 2=0 then
l2:          x:=x+2;
l3:      else x:=x+1;
   end
```



Formalize this: $\texttt{Loc} : \{l_0, l_1, ...\}, \texttt{Var} : \{x, pc\}, S : \texttt{Var} \to \texttt{Val}(\{x\} \to Z)$ and $\{\texttt{pc}\} \to \texttt{Loc}$. Now think about what the meaning of each edge is. Let $-- >$ be a $\texttt{State} \times \texttt{State}$ *successor* relation.

$$\texttt{s---------->s'} \quad \texttt{(l0,x:=e,l')} \quad \texttt{def} \iff \texttt{s(pc)=l} \wedge \texttt{s'(pc) =l'} \wedge \texttt{s'(x) =} [[e]]_s$$

```
    (l,g,l')                 def
s--------------> s'          <=>   s(pc) = l ∧ s'(pc) = l' ∧ [[g]]_s=true
```

Now that the meaning of the edges has been formalized, there are many ways to proceed in defining the meaning of the program. Depending on what to analyze, we can make a semantics. So what is the meaning of a program? We could be interested in the value it computes, the set of reachable states, or something else. A trace semantics says that the meaning of a program is its trace (list of values that the variables take over time). Let's focus on the set of `Reach` of reachable states.

Define $\texttt{Init} = \{s | s(pc) = l_0\}$. Define the `Post` operator as the lifting of the state transition relation to a function on sets of states that gives the set of 1-step successors of a state. $\texttt{Post} : 2^{state} -> 2^{state}$.

$$\texttt{Post}(S) = \{s' \in \texttt{State} | \exists s \in S. s --> s'\}$$

$\texttt{Post}(S_1 \cup S_2) = \texttt{Post}(S_1) \cup \texttt{Post}(S_2)$ distributes because of exists operator. Easy to prove. From this it follows that `Post` is Monotone (in terms of the sizes of the sets). IMPORTANT: Let $F = \lambda S.\texttt{Init} \cup \texttt{Post}(S)$. Then the set of reachable states is least fix point $(lfp)_{\emptyset}^{\sqsubseteq}[F]$ of the function F, with respect to some ordering $\sqsubseteq$ and starting from the element $\emptyset$.

$$\texttt{Reach} = (lfp)_{\emptyset}^{\sqsubseteq}[\lambda S.\texttt{Init} \cup \texttt{Post}(S))$$

Why does the least fix point exist / why is it well defined? We are using some general results of fixed point theory / order theory.

Definition: a complete lattice D is a set D with an ordering $\sqsubseteq$, smallest element $\bot$, largest element $\top$, join function $\sqcup$, meet function $\sqcap$, such that:

- $\sqsubseteq$ is a partial order on D
- join $\sqcup : 2^D \to D$ is the least upper bound with respect to $\sqsubseteq$
- meet $\sqcap : 2^D \to D$ is the greatest lower bound with respect to $\sqsubseteq$
- $\bot = \sqcap D$
- $\top = \sqcup D$
- least upper bound and greatest lower bound exist for any subset of D
- an incomplete lattice has join and meet only defined for finite subsets
- every powerset forms a complete lattice where the ordering $\sqsubseteq$ is subset inclusion $\subseteq$, join $\sqcup$ is union $\cup$, meet $\sqcap$ is intersection $\cap$, bottom $\bot$ is the empty set $\emptyset$, and top $\top$ is the full set itself.

The powerset of states forms a complete lattice. Tarski's fixed point theorems say that any monotone function over a complete lattice has a least fixed point. Abstract interpretation is approximating the set of fixed points for a function over a complete lattice.

Let's convince ourselves that definition of `Reach` makes sense. Let's look at some of the iterates of the fixed point functional $F = \lambda S.\texttt{Init} \cup \texttt{Post}(S)$:

- F applied 0 times to $\emptyset : F^0(\emptyset) = \emptyset$

- F applied 1 time $F^1(\emptyset) = \texttt{Init}$
- $F^2(\emptyset) = \texttt{Init} \cup \texttt{Post}(\texttt{Init})$
- $F^3(\emptyset) = \texttt{Init} \cup \texttt{Post}(\texttt{Init} \cup \texttt{Post}(\texttt{Init}))$

Because of $\texttt{Post}'s$ distribution over $\cup$, the last line is equal to $\texttt{Init} \cup \texttt{Post}(\texttt{Init}) \cup \texttt{Post}^2(\texttt{Init})$. This gives us that $F^k(\emptyset) = \cup_{i=0}^{k} F^i(\emptyset)$. Let k go to the limit, we compute all reachable states. `Reach` is the least fixed point starting from $\emptyset$ with respect to $\subseteq$ of F, which is equal to $\cup_{i=0}^{\infty} F^i(\emptyset)$

Kleene says that if D is a complete lattice, $F : D \to D$ is a function over that lattice, and F is continuous in the following sense ($\star$), then the least fixed point starting from $\bot$ with respect to the ordering $\sqsubseteq$ of F is equal to $\sqcup_{i=0}^{\infty} F^i(\bot)$.

($\star$) The definition of continuity here has various formulations, but the following is sufficient. If we take ascending chains in the ordering $d_0 \sqsubseteq d_1 \sqsubseteq d_2...$ (a sequence of potentially infinite elements increasing, perhaps not strictly increasing), and if F is known to be monotone, and if we apply $F(d_0) \sqsubseteq F(d_1) \sqsubseteq F(d_2)...$, then F distributes in the least upper bound of the new sequence so that $\sqcup_{i=0}^{\infty}\{F(d_i)\} = F(\sqcup_{i=0}^{\infty}\{d_i\})$. Winskel book has a discussion of this.

The least fixed point starting from $\bot$ with respect to $\sqsubseteq$ of F is $\sqcup_{i=0}^{\infty}\{F^i(\bot)\}$. It is an exercise to prove that our $F = \lambda S.\texttt{Init} \cup \texttt{Post}(S)$ is continuous.

This is a concrete semantics because it gives us the meaning of the program. This is already an abstraction in a way because it is not the trace semantics. But this is our initial semantics: the set of reachable states. This has less info than traces, which by the way are also abstract in some sense.

Question: are there any more semantics with more info than trace semantics. In our language we have deterministic behavior. But think of a non-deterministic program. Think about Temporal logics. How certain points of execution relate to each other. View of semantics of program as a tree. Like a non-deterministic Turing machine. You may want to know whether there exists a branch in the tree that satisfies some property. Temporal logics like CTL and CTL* let you define these sort of things, define some notion of trees. Trace semantics would be an abstraction of the non-deterministic tree semantics

To be addressed soon: how can we prove that one notion of semantics is a sound abstraction of another?

Our problem: the set `Reach` can't be computed in general because of undecidability. So rather than calculating `Reach` precisely, we approximate!

Remember our example program above. Consider a set of `Bad` states defined by a set of program counters $\{l_3\}$ (the else statement). Verification problem: is $\texttt{Reach} \cap \texttt{Bad} = \emptyset$? We need an inductive invariant, which is a set `Inv` of states that satisfies:

- $\texttt{Inv} \supset \texttt{Init}$
- $\texttt{Post}(\texttt{Inv}) \subseteq \texttt{Inv}$
- $\implies \texttt{Reach} \subseteq \texttt{Inv}$

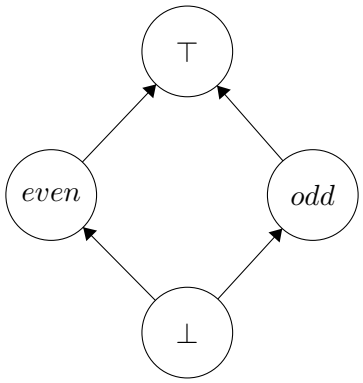
Look at this definition and thedefinition of `Reach`. Then `Reach` is the smallest, most precise invariant of the program. The set of all states is an invariant of the program too, but it doesn't tell us anything. The smallest set is the most informative. Anything that has the two invariant properties is guaranteed to be a superset of `Reach`. If we can show that $\texttt{Inv} \cap \texttt{Bad} = \emptyset$ then it also holds for `Reach`.

We want a way to automatically compute invariants to show that `Bad` states are not reachable. Aside from verification, we may also use invariants for optimization.

We approximate `Reach`. We approximate fixed points. How do we formalize this? When you do this analysis of the program, you approximate things on a different lattice, so that when you compute over that new lattice, you have the search for $F's$ fixed-point terminate.
Rough idea: Rather than keep track of the value of x, we could keep track of its parity (whether it is even or odd). Then we don't have to keep updating its value. Instead, we reason about whether or not we ever transition from even-ness to odd-ness, etc...

Abstract interpretation gives a mechanical process for coming up with another lattice for which the fixed-point search finishes and that is sound for the analysis we want to perform.

$D^\#$ (D sharp):



Even and Odd are incomparable, both larger than $\bot$, both smaller than $\top$. How to relate this lattice to the original one? This lattice keeps track of the parity of x. We probably also want to keep track of the program counter. $D^\#$ is a lattice. The underlying set $\{\texttt{Loc} \to D^\# p\}$ is a set of functions. The lattice over-approximates values of x.

The ordering we choose on two functions holds if for all inputs, their results are ordered: $s_1^\# \sqsubseteq s_2^\# \to \forall l \in \texttt{Loc}, s_1^\#(l) \sqsubseteq s_2^\#(l)$. Aside: If D is a complete lattice ordered by subset, then for all sets X, $X \to D$ is a complete lattice ordered by $\sqsubseteq$.

Galois Connections.

γ (concretization): $D^\# \to 2^{State}$ is a mapping from our abstract lattice to our original concrete

lattice.

$$\begin{aligned}\gamma(\top) &= \texttt{State}\\ \gamma(\bot) &= \emptyset\\ \gamma(\text{odd}) &= \{s \in \texttt{State} | s(x) \texttt{ mod } 2 \neq 0\}\\ \gamma(\text{even}) &= \{s \in \texttt{State} | s(x) \texttt{ mod } 2 = 0\}\end{aligned}$$

We need γ to exist and have a formal calculation, but we don't need to calculate it ever (what did this comment mean???)

To map concrete to abstract, we have α (abstraction): $2^Z \to D_p^\#$. Let's think about what we want for the relationship between γ and α. We want $Z \subset \gamma(\alpha(Z))$. If we do $a(Z) = T$, we lose too much info. We can differentiate between whether the set to be mapped has all evens, all odds, or a mix.

$$\begin{aligned}a(Z) = &|even, \text{ if } Z \subseteq \{z | z \texttt{ mod } 2 = 0\}\\ &|odd, \text{ if } Z \subseteq \{z | z \texttt{ mod } 2 \neq 0\}\\ &|\bot, \text{ if } Z = \emptyset\\ &|\top \text{ otherwise}\end{aligned}$$

Check the property we want, that $Z \subseteq \gamma(\alpha(Z))$. Is it the best possible abstraction? We want α to have the tightest over-approximation for the given abstraction. A Galois connection formalizes this idea. It's a pair of functions: $D \to^\alpha D\#$ and $D \leftarrow^\gamma D\#$ such that

- for all concrete values $x \in D$ and abstract values $y \in D\#$, $\alpha(x) \sqsubseteq^\# y \iff x \sqsubseteq \gamma(y)$
- $x \sqsubseteq \gamma(\alpha(x))$ [soundness]
- $\alpha(\gamma(y)) \sqsubseteq^\# y$ [bestness]
- going to concrete and back to abstract can only give more information

With an abstraction in mind, we have a γ and can we come up with an α. When can you do this? You can do this when your gamma has certain properties. γ must be a complete $\sqcap$-morphism, which means that $\forall y \subseteq D\#$ (arbitrary subsets) $\gamma(\sqcap^\# y) = \sqcap(\{\gamma(y) | y \in Y\})$ and $\gamma(\top^\#) = \top$. If that's true, we can define $\alpha = \lambda x \in D. \sqcap \{y \in D\# | x \subseteq \gamma(y)\}$

Sometimes having α first is more natural. In that case you need to come up with a dual where you define a $\sqcup$-Morphism and construct γ, but normally you have a γ in mind first and need to construct α. $\gamma(s\#) = \{s \in \texttt{State} | \forall l \in \texttt{Loc}. s(pc) = l \to s \in \gamma(s\#(l))\}$, which we can write as $\cup_{l \in \texttt{loc}} \{s \in \texttt{State} | s \in \gamma(s\#(l)) \wedge s(pc) = l\}$.

Let's abstract `Post`. `Post#` : $D\# \to D\#$. `Post#`$(s\#) = \alpha($`Post`$(\gamma(s\#)))$. Composition of monotone functions is monotone. So the fixed point of `Post#` exists on the abstract lattice. $F\# = \lambda s\#.$`Init#` $\cup$ `Post#`$(s\#)$. Let $p\#$ be the least fixed point of $F\#$ with respect to $\sqsubseteq$ # starting on $\bot$. Then $\gamma(p\#) \supset$ `Reach`. `Init#`$(l) = \top$ if $l = l_0$ and $\bot$ otherwise.

$$s\# \xrightarrow{(l, x := e, l')\#} s'\# \iff s'\#(l')(x) = [[x]]^l_{s\#} \wedge \texttt{(etc)}$$

For the semantics of expressions like $[[e1 + e2]]$, we can do $\alpha(\{z1 + z2 | z_1 \in \gamma([[e1]]^l_{s\#}) \wedge z_2 \in \gamma([[e2]]^l_{s'\#})\})$. Abstract evaluation of expressions. Less precise than going from abstract state to concrete state, then evaluate whole expression in concrete, then back to abstract.

One way to keep track of a parity value of x per location in the program is to do something like: (p_0 at l_0, p_1 at l_1, p_2 at l_2, p_3 at l_3). So our abstract function $F\#$ maps abstract states to abstract states. $s_0\# - -(x := (e = 0)) - -- > s_1\# \iff (\bot, p1', \bot, \bot)$ where $p'_1 = [[e]]^{l_0}_{s_0\#}$. Then $[[e = 0]]$ gives us $(\bot, even, \bot, \bot)$.

`Post`$\#$ is do `Post` for each transition, then take join of them. This is a function from `Loc` to parity Start from $\bot$ and look at iterates of $F\#$:

$$\begin{aligned}
\bot\# = (def) &= (\bot, \bot, \bot, \bot) \\
F\#0(\bot) &= (\bot, \bot, \bot, \bot) \\
F\#1(\bot) &= (\top, \bot, \bot, \bot) \\
F\#2(\bot) &= (\top, even, \bot, \bot) \\
F\#3(\bot) &= (\top, even, even, \bot) \\
F\#4(\bot) &= (\top, even, even, \bot)
\end{aligned}$$

So things left as bottom in the fixed point are unreachable, $l3$.

Consider x taking on intervals rather than even/odd. This corresponds to an infinite abstract domain. The interval of x will get larger as we iterate. How do we make sure the iterates converge so we find a fixed point? The answer is the Widening operator:

- $\nabla : D\# \times D\# \to D\#$
- ∇ approximates $\sqcup : y_1 \nabla y_2 \geq y1, y2$
- sequence in abstract $y0 \sqsubseteq y1 \sqsubseteq ...$
- define $z0$ as $y0$, z_i defined as $z_{i-1} \nabla yi, i > 0$
- $\to z0 \sqsubseteq z1 \sqsubset z2...$ (also increasing due to upper bound property of ∇ operator) must stabilize, so when we iterate we reach a fixed point.

Take the least fixed point starting on $\bot$ with respect to $\sqsubseteq$ of $[\lambda y.y \nabla F\#(y)]$ for the old $F\#$ that we had. For each step, you widen with the previous iterate. How you define the widening operator is specific to the abstract domain. Here you could do something simple, say, if x increased then go right to ∞ for the right-side limit of the interval. Or perhaps wait a while (e.g. x goes beyond 100), and then go to ∞. Choice of ∇ is another layer of approximation! Though it seems that interval domain is more expressive than parity domain, it is sort of orthogonal. We get information on whether x takes on values in a range like $[0, \infty]$ (that is, x is positive), but we don't know if its even or odd.

Galois connections compose