Program Analysis, Lecture 5

Professor Mooly Sagiv

Class notes by Yaron Koral and Hila Peleg

In this lecture complete the material of Chaotic Iterations and proceed to the theoretical foundation for Abstract Interpretation.

Chaotic Iteration (continued)

Reminder: the purpose of the chaotic iterations process is to find a least fixed point in the function comprised of all nodes in the control flow graph. The following function

$$F_S(X)[s] = \iota$$
$$F_S(X)[v] = \sqcup\{f(u,v)(X[u]) \mid (u,v) \in E\}$$

for which we are seeking the least fixed point, is defined by f, the effect of the edge (u,v) on the state when it is crossed. The value at each v is set by running the function on each of its neighboring vertices. This calculation may be recursive in its dependencies.

We are operating under the assumption that the starting state of the program or procedure, marked above as ι, is known. Later in the course we will discuss the implications of not knowing the starting state. We are also operating under the assumption that the height of the lattice is finite. Under this assumption, the function, and its corresponding set of |V| equations is computable. We will later show that the assumption that the height of the lattice is finite can be dispensed with.

The Chaotic Iterations algorithm we saw last time uses a Work List of vertices that still need to be calculated, and for each calculation performs a join between the existing value and the new value. This join means the value will only ever grow (in terms of the order of the lattice, since join is monotonous). For n vertices and a lattice of height h, the algorithm will converge after at most n*h iterations, as each move up the lattice height for one node must propagate out to all the others.

Chaotic Iteration – Python Implementation

For chaotic iterations to run, it requires first an implementation of join, top and bottom. These are found in cp_one.py:

```
bottom = "bottom"
top = "top"
```

Top and bottom are simply defined as constants for later use, while the join operation is defined thus:

```
def join(a, b):
    if a == bottom:
        return b
    elif b == bottom:
        return a
    elif a == b:
        return a
    else: return top
```

The definition of this function assumes only one variable in the state, which dictates the height of the lattice to be 3. Should we wish to run the chaotic function on a more complicated state, we will have to write a new join function – it might use this one as its basis to apply to each individual variable in the state, but since the chaotic function does not make distinction as to the state it runs on (especially since it is weakly typed in python) the responsibility for expanding the state is entirely on the join function.

The chaotic iterations function itself is defined in chaotic.py thus:

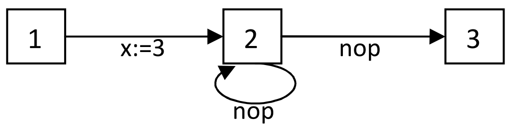
```
def chaotic(succ, s, i, join, bottom, tr):
# succ is the successor nodes in the CFG
# s in nodes is the start vertex
# i is the initial value at the start
# bottom is the minimum value
# tr is the transfer function
    wl = [s]
    df = dict([(x, bottom) for x in succ])
    df[s] = i
    while wl != []:
        u = wl.pop()
        print "Handling:", u
        for v in succ[u]:
            new = join(tr[(u,v)](df[u]), df[v])
            if (new != df[v]):
                print "changing the dataflow value at node:",
v, "to" , new
                df[v] = new
                wl.append(v)
                wl.sort(key=lambda x:-x)


    print "Dataflow results"
    for node in succ:
           print "%s: %s"%(node, df[node])
```

The initializations build a new work list to include the start node, as well as df, a dictionary (key, value) of the state for each node in the data flow graph. (Bottom is passed to the function from without, so that it can be changed in accordance with the join function to suit the lattice being used. The function uses the bottom variable to set a “global” bottom value to the state of the node, when it might represent something like [x↦⊤,y↦⊤].)

The start node is then initialized with the initial value for a variable given to the function (the parameter i) and the algorithm is run until the work list is empty. For each node in the work list its new value is calculated using the join function that chaotic receives as an argument, and if the value is changed, the node’s neighbors are added to the work list.

Assuming that the nodes of the graph themselves are numbered by DFS order, the descending sort of the work list after each addition ensures that the graph will be handled in DFS order. (This is not necessary, since the algorithm will finish with the same result regardless of the order, but in this order will simply finish faster, as will be shown later.)

Here is an example of code that initializes and runs chaotic on the following Flow Control Graph (assumption, there is only one variable in the state):

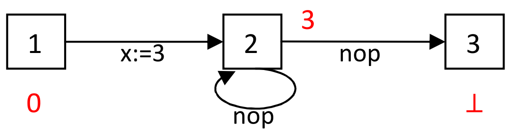


```
succ = {1:{2}, 2:{2,3}, 3:{}} # CFG edges
tr = {(1,2): lambda x: 3, (2, 2): lambda x: x, (2, 3): lambda
x: x} # transfer function
tr_txt  = {(1,2): "x := 3", (2, 2): "nop", (2, 3): "nop"}  #
for debugging


chaotic(succ, 1, 0, join, bottom, tr)
```

succ is the flow control graph (as shown above).

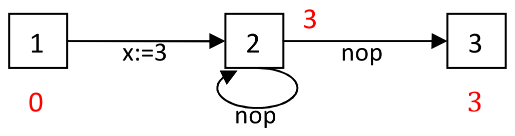
tr is the transfer function for each edge of the graph, and *tr_txt* is its text printout which can be used for debug printouts or printouts of the graph.

The run will start with node #1 in the work list, and nodes 2 and 3 initialized to bottom. The calculation of new values will run on succ[1], which is a list containing only node #2. The transfer function for the edge (1,2) is run on the state at node 1. In the case of (1,2) this is x:=3, which sets the state to 3.



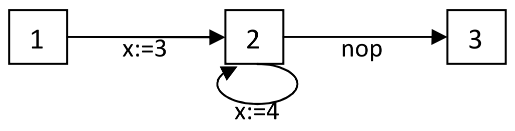
Then, since the old value is different than the new value, node #2 is added to the work list. It's then pulled from the work list (being the only node in it) and the list of nodes it is connected to is taken from succ[2], which includes nodes 2 and 3, and calculated.

The transfer function for the edge (2,2) is run on the state, an yields the same state, 3. As this is the same as the old value of the target node (node 2), node #2 is not put back in the work list. For the edge (2,3) the transfer function is run on the state, and also yields the state 3. This is different than the value currently at node 3, and when joined yields the value 3. This is set to node #3, and as it has changed it is put in the work list.



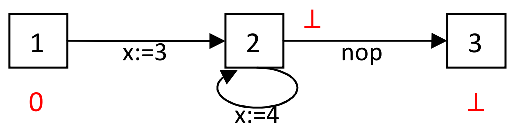
Node 3 is taken from the work list, and as its list of connected nodes is empty, it requires no action. The work list is now empty and the algorithm finishes with df={1:0, 2:3, 3:3}.

Let us quickly run a second, similar example:

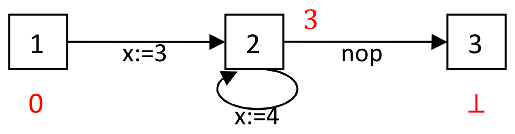


```
succ = {1:{2}, 2:{2,3}, 3:{}} # CFG edges
tr = {(1,2): lambda x: 3, (2, 2): lambda x: 4, (2, 3): lambda
x: x} # transfer function
tr_txt  = {(1,2): "x := 3", (2, 2): "x := 4", (2, 3): "nop"}
# for debugging


chaotic(succ, 1, 0, join, bottom, tr)
```

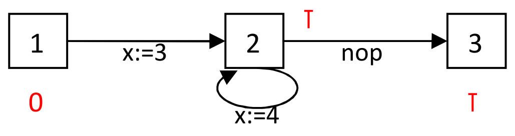
Here we again start with the work list containing node #1 and nodes 2 and 3 initialized to bottom.



When node #1 is pulled from the work list, its transfer function is executed as before, and as before yields a new value for node 2 which is the join of bottom and 3, or 3. As node 2 was modified, it is placed into the work list.



Node 2 is taken from the work list, and both its connected vertices are handled. For node #2, the transfer function for (2,2) is executed yielding a state that equals 4, and joined with the value existing in the state is equal to top. It is updated into the state, and two is put into the work list. Now the edge (2,3) is executed and the nop carries the value from node 2, which is now top, over to node 3. Since it was modified, it is also put into the work list.



Since the work list is sorted in descending order, node #3 is taken from it first. As it has no nodes in its list of successors, nothing is to be done. Next node #2 is taken from the work list, and its two successors (nodes 2 and 3) calculated.

The transfer function for the edge (2,2) again yields 4, but this time, when joined with top, it results in top and the value does not change. The transfer function for (2,3) yields top, which joined with the existing top remains the same. As no node was added to the work list, the algorithm finishes with df={1:0, 2:⊤, 3:⊤}.

The Abstract Interpretation Technique

This is the theoretical and mathematical foundation of program analysis, which defines what information is computed by the static tool. It also allows proving that the analysis is sound, not simply per program, but per programming language.

It is a tool for planning the analysis. The basis of the method is the relationship between the semantics of analysis and operational semantics: approximating what is executed concretely. The cases that are analyzed may include more than the concrete case – we might try cases that couldn't actually happen. This is due to loss of information in the abstraction.

Abstract Interpretation is based on the relationship between two lattices: the concrete, C, and the abstract, A. They are connected by two functions, α:C→A and γ:A→C. The function α, the abstraction, is not usually one-to-one, so multiple concrete values could be directed here. This will later help us partition the concrete realm into equivalence classes.

The value of an abstraction will lose information, whereas the value of a concretization will gain information. Running them in sequence will amplify this: taking an abstract value, and on its concretization running an abstraction will minimize the domain even further. An example from constant propagation will be the state $[x \mapsto \bot, y \mapsto 5]$ which is an abstract representation. The concretization function will yield ∅, since any mapping of ⊥ represents an unreachable state. Running an abstraction on ∅ will yield $[x \mapsto \bot, y \mapsto \bot]$, a more "accurate" abstract representation. In the same manner, a concrete state where $[x \mapsto 2 or x \mapsto 3]$, when applying the constant propagation abstraction, will yield [x↦⊤], the most general possible answer. Concretization of this abstraction will result in $[x \mapsto 0 or x \mapsto 1 or x \mapsto 2 or \ldots]$, a much wider range than we started with initially.

For two lattices A and C with two functions α:C→A and γ:A→C, the functions (α,γ) form a **Galois Connection** if one of the following, equivalent sets of conditions are satisfied:

- α and γ are monotone
- ∀a∈A, α(γ(a))⊑a

- ∀c∈C, c⊑γ(α(c))

And:

∀a∈A, ∀c∈C: α(c)⊑a ⟺ c⊑γ(a)

The first definition requires that both abstractions and concretizations are order preserving and that abstraction potentially loses information while concretization potentially gains information.

The second definition implies that comparing with an abstraction to its result is equivalent to comparing concrete values with concretizations.

A special case of Galois connection is Galois insertion when α(γ(a))=a

Let us prove the equivalence between the two sets of conditions.

Assuming the first set of conditions, let us prove α(c)⊑a ⟺ c⊑γ(a). Assuming α(c)⊑a, we can use the monotonous property of γ and apply it to both sides: γ(α(c))⊑γ(a). Using the third condition in the first set results in: c⊑ γ(α(c))⊑γ(a). Since ⊑ is transitive, yields c⊑γ(a).

The second direction is proved in the same manner: Assuming c⊑γ(a), we can use the monotonous property of α and apply it to both sides: α(c)⊑α(γ(a)). Using the second condition above, we receive α(c)⊑α(γ(a)) ⊑a. Since ⊑ is transitive, this yields α(c)⊑a.

Now, assuming the second set of conditions, let us prove the first. We will begin with α(c)⊑ α(c), which applying the iff in the assumed set of conditions, we define $a' = \alpha(c) \in A$ and since α(c)⊑ a' ⇒ c⊑ γ(a'), which, when expanding a' is c⊑ γ(α(c)). In the same way we start at γ(a)⊑ γ(a), and arrive at α(γ(a))⊑a.

To prove that α and γ are monotonous, given some $a_1 \sqsubseteq a_2$, using what was just proved above we may say that $\alpha(\gamma(a_1\ ))\sqsubseteq a_1 \sqsubseteq a_2$ which by transitivity is $\alpha(\gamma(a_1)) \sqsubseteq a_2$. We now apply the condition in the iff that is assumed, we get $\gamma(a_1) \sqsubseteq \gamma\ (\ a_2)$. The same is done for α staring with some $c_1 \sqsubseteq c_2$. Since all three conditions in the first set are proved using the second set, and the second set is proved using the first, we may say the two sets of conditions for Galois Connections are equivalent.

The Abstraction Function (Example from Constant Propagation)

To create the abstraction function, the collecting states need to be mapped into abstract constants. First we define the abstraction of an individual state: abstraction of a constant value σ is the constant value σ. Next we define the abstraction for a set of states: joining of the abstraction of each of the single abstracted values comprising it.

An abstraction function must be sound, meaning that no reachable concrete state will be unrepresented in the abstraction. This is implied by α being part of a Galois connection, so for any concrete state c we know that c⊑γ(α(c)). That means for any reachable state in the concrete domain, we will only enlarge our group of states. Therefore, for the reachable set of states, Reach(v), γ(α(Reach(v))) will include all the states in Reach(v). Having applied α on a group of states will only move up the lattice order when joining the abstract values for the states in the set. That means re-applying the concrete transformation will only include more states, or the same states, as in Reach(v), not less.

However, an abstraction function is usually not complete. A complete abstraction would include all concrete states and only them, which implies no loss of information in the abstraction function.

The Concretization Function (And more examples from Constant Propagation)

To create a concretization function, the abstract constants need to be expanded back into collecting states. This function is defined as a set, instead of the formula to calculate the abstraction. If β is the abstraction for a single constant value, then the concretization for an abstract value in the dataflow df would be $\gamma_{CP}(df) = \{\sigma \mid \beta_{CP}(\sigma) \sqsubseteq df\}$, which is any value that could work its way up the chain in the lattice that leads to df. Since order in a lattice doesn't exist between every two items, this can be more simply stated as $\gamma_{CP}(df) = \{\sigma \mid \sigma \sqsubseteq df\}$.

This means that an example abstract state of [x↦⊤,y↦5] would yield a host of concrete functions: $\begin{cases} [x \mapsto 1, y \mapsto 5] \\ [x \mapsto 2, y \mapsto 5] \\ [x \mapsto 3, y \mapsto 5] \\ \vdots \end{cases}$ but no state in which y has a value other than 5, as that has no order relationship directly with a state in which y is mapped to 5.

In an abstract state of [x↦⊤,y↦⊤], the concretization must go through two phases (since the lattice here is a Cartesian product of two lattices: CPxCP, which means the height of the lattice increases accordingly). This state is the join of this set of states:

$$\begin{cases} [x \mapsto 1, y \mapsto \top] \\ [x \mapsto 2, y \mapsto \top] \\ [x \mapsto 3, y \mapsto \top] \\ \vdots \end{cases}$$

each state of which is in itself a join of smaller concrete states, as in the example above for y↦5. And each of the resulting states is itself ⊑[x↦⊤,y↦⊤].

In regard to soundness, we can see that any abstract state represents more concrete states than could actually be reached in the course of the program.

To show completeness, we need only show that α and γ satisfy the conditions for a Galois Connection.

Another example: Parity

We can create a more extreme abstraction for the parity property. The abstract domain would look like this:

$$\begin{tikzcd} & \top \arrow[dl, no head] \arrow[dr, no head] & \\ O \arrow[dr, no head] & & E \arrow[dl, no head] \\ & \bot & \end{tikzcd}$$

As we did in CP, we first define the abstraction for a single state:

$$\beta_{parity}(2\sigma) = e$$
$$\beta_{parity}(2\sigma + 1) = o$$

With this definition we can create a general abstraction function in the same manner:

$$\alpha_{parity}(CS) = \sqcup\{\beta_{parity}(c) \mid c \in CS\}$$

The definition for the concretization function would also be similar to the function in CP:

$$\gamma_{parity}(df) = \{\sigma \mid \beta_{parity}(\sigma) \sqsubseteq df\}$$

Here, the effect of the transition back and forth is much more drastic. Suppose we begin with the state s=[x↦2]. α(s) would be [x↦*e*], which after re-concretization would be [x↦0,2,4,6,…], so the power of α to partition the concrete realm is again displayed.

Also, the α function is, while still monotone, not so fast to climb up. Two different concrete values could have the same abstract value, which in CP they could not, and so their join would remain the same.

Upper Closure

Upper closure is a partitioning within the concrete realm, denoting two members that could not be told apart by the abstraction: they are unified by "what the abstraction does not know about them". This is the aforementioned use of the abstraction function to create equivalence classes.

For example, in Constant Propagation, the following two sets of states could not be told apart by the analysis:

$$\left\{\begin{array}{l}[x \mapsto 5]\\ [x \mapsto 7]\end{array}\right. \quad \left\{\begin{array}{c} x \mapsto 1\\ x \mapsto 2\\ \vdots\\ x \mapsto 20\end{array}\right.$$

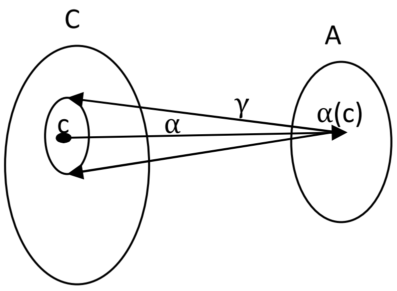
The abstraction of both would be [x↦⊤], and would be treated the same.

In the parity example, any two even numbers could not be told apart by the analysis, nor could any two odd numbers. This is more acute than in constant propagation, when *groups* of states couldn't be told apart.

We would like upper closure to be defined thus:

$$
\begin{aligned}
&up : C \to C \\
&up(c) \sqsupseteq c \\
&up(up(c)) = up(c)
\end{aligned}
$$

We can see that it can be defined as $up(c) = \gamma(\alpha(c))$.



The first condition for upper closure is satisfied simply by α and γ forming a Galois Connection, and the second is correct because of the definition of γ – it has given us values all concrete values that map to α(c), and calling α on it again would simply return us to α(c), from which the second call to γ would be identical.

Proving the soundness of an analysis

We don't want to prove the soundness of the analysis on a specific program, but of all programs. For that, we want to define a **collecting semantics**, prove its soundness for the atomic statements (**Local Correctness**) and conclude that the entire analysis is sound (**Global Correctness**).

Collecting Semantics

Using Collecting Semantics we can capture captures how instructions of a programming language operate rather than captures how a given program runs. We can use fixed points to capture the semantics of a program.

Collecting semantics get rid of the assumption that the input is known at compile time. They then "collect" all the states for any possible input. Since Tarski's theorem is also usable for an infinite set of states, it can be applied here as shown later.

Collecting semantics compute set of all program states. The following example shows collecting semantics for the given program.

```
z = 3       {[x↦0, y↦0, z↦0]}

x = 1    {[x↦0, y↦0, z↦3]}

while (x > 0) (                  {[x↦1, y↦0, z↦3]}

      if (x = 1) then y = 7

                  else y = z + 4

       x = 3          {[x↦1, y↦7, z↦3], [x↦3, y↦7, z↦3]}

       print y        {[x↦3, y↦7, z↦3]}

  )                  {[x↦3, y↦7, z↦3]}
```

An “Iterative” Definition of Collecting Semantics:

Generate a system of monotone equations. The least solution is well defined. The least solution is the collecting interpretation. It may be incomputable.

We need to define the behavior for each of the atomic commands. If we examine constant propagation on the While language:

When CS is the collected set of states

- [skip] $CS_{exit}(l)=CS_{entry}(l)$
- [b] $CS_{exit}(l)=\{\sigma:\sigma\in CS_{entry}(l),[b]\sigma=tt\}$ (Or: the collected states are the states out of the set upon entry where the condition evaluates as true. In a control statement such as while or if, each branch would get the relevant part of the entry set of states)
- [x:=a] $CS_{exit}(l)=\{s[x\mapsto A[a]s \mid s\in CS_{entry}(l)\}$ (Or: apply the assignment to each of the states in the set)

The collection iteration is similar to chaotic iteration, when its intention is to collect all possible reachable states, to ensure the soundness of the analysis – that no reachable state might be left out and leave an error undetected. This collection of states could result in an infinite set, for example the collection for while(tt) x:= x+1.

Even when collecting the states in the abstract domain, the height of the abstract lattice could be infinite. But, according to Tarski’s theorem there wil still be a least fixed point, and reaching it will give us a collection of states in which every reachable code path is included.

The general idea would be to use an abstract domain in order to "compute" approximation for the least solution. This way we refer to two least fixed points (lfp's), one in the concrete domain and the other in the abstract domain.

By the global soundness theorem we show that the least solution from the abstract domain is a sound approximation of the concrete domain. I.e., the solution may only over approximate the concrete domain.

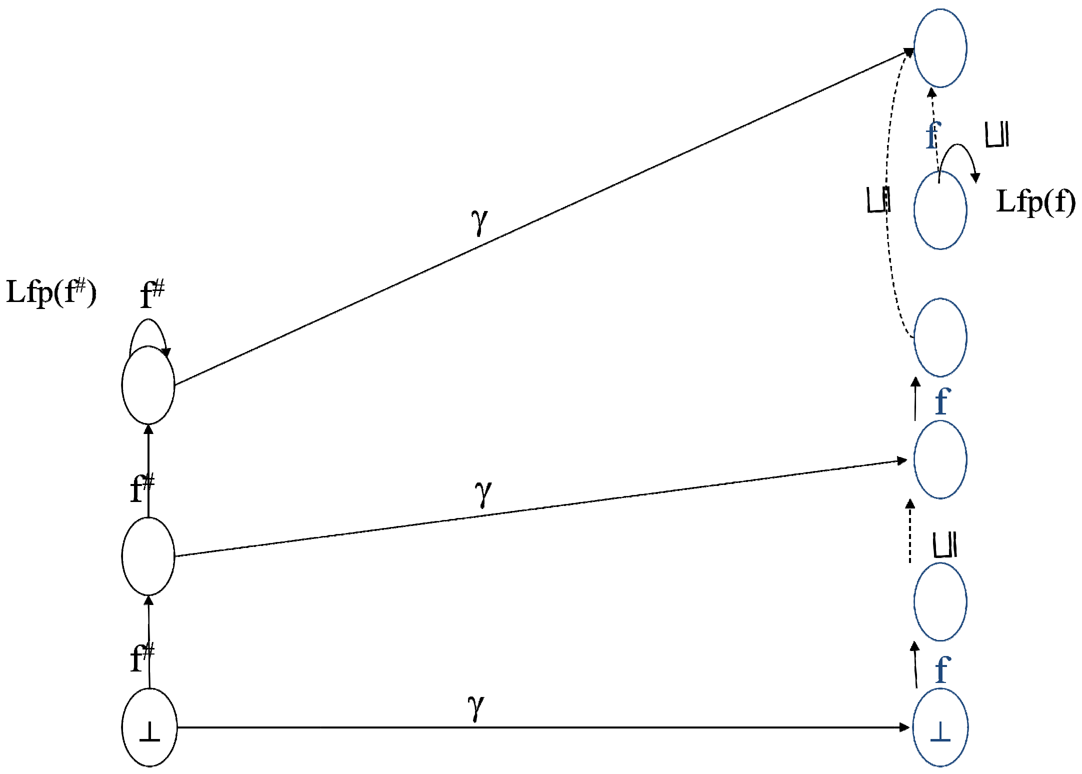
Soundness Theorem

In most general terms, abstract interpretation is a theory of fixed point approximation. Hence we would like to show that the abstract fixed point approximates the concrete fixed point.

Assume that the following takes place:

1. Let (α, γ) form Galois connection from C to A.
2. f: C→C be a monotone function. (The transfer function on the concrete domain)
3. $f^{\#}$: A→A be a monotone function. (The transfer function on the abstract domain).

The least fixed point is found by iterating the transfer function in the concrete and abstract domains. The finite height case is demonstrated by the following figure:



The least fixed point also exists in the infinite case according to Tarski's theorem.

The properties of global soundness (as shown in class) are described as follows:

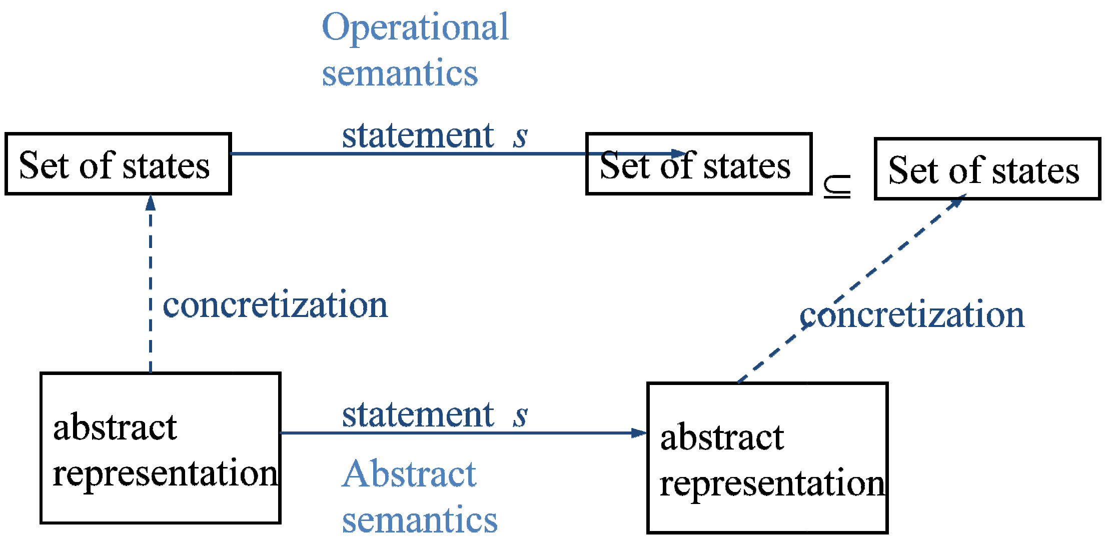
(1) $\mathbf{lfp(f)} \sqsubseteq \gamma(\mathbf{lfp(f^{\#})})$ - *the concretized lfp (least fixed point) of the abstract domain is always less accurate than the lfp of the concrete domain.*

(2) $\alpha(\mathbf{lfp(f)}) \sqsubseteq \mathbf{lfp(f^{\#})}$ – *the lfp of the abstract domain is always less accurate than the abstraction of the lfp of the concrete domain.*

The above conditions are global: they talk about the fixed point computation. In order to perform the global computation we need some local conditions on f and $f^{\#}$.

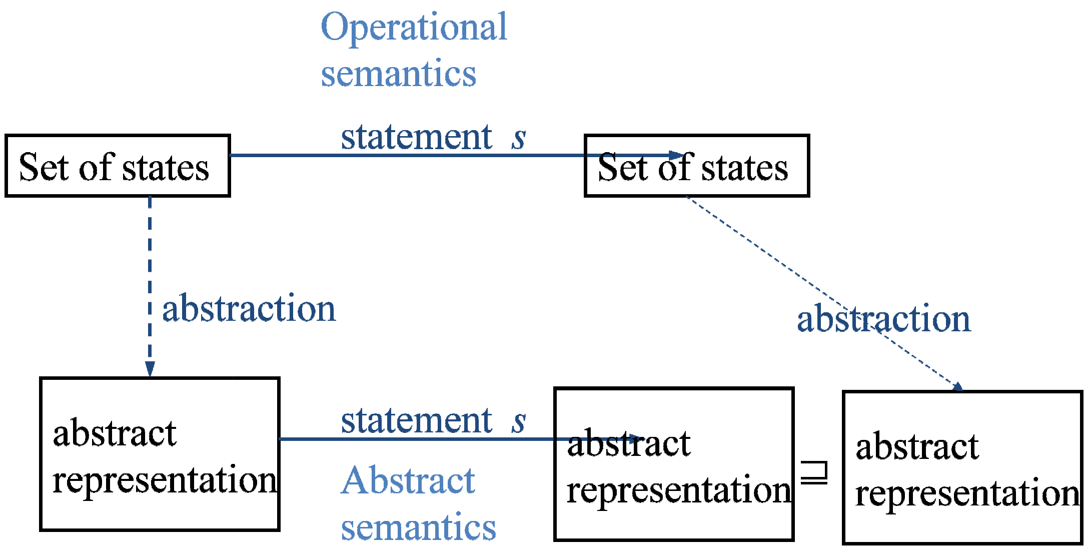
Cousot and Cousot has shown that we can use local conditions iteratively in order to get the global approximation sound. Specifically they have shown that the following **local** conditions are sufficient for proving the **global** (1) and (2):

1. $f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a))$



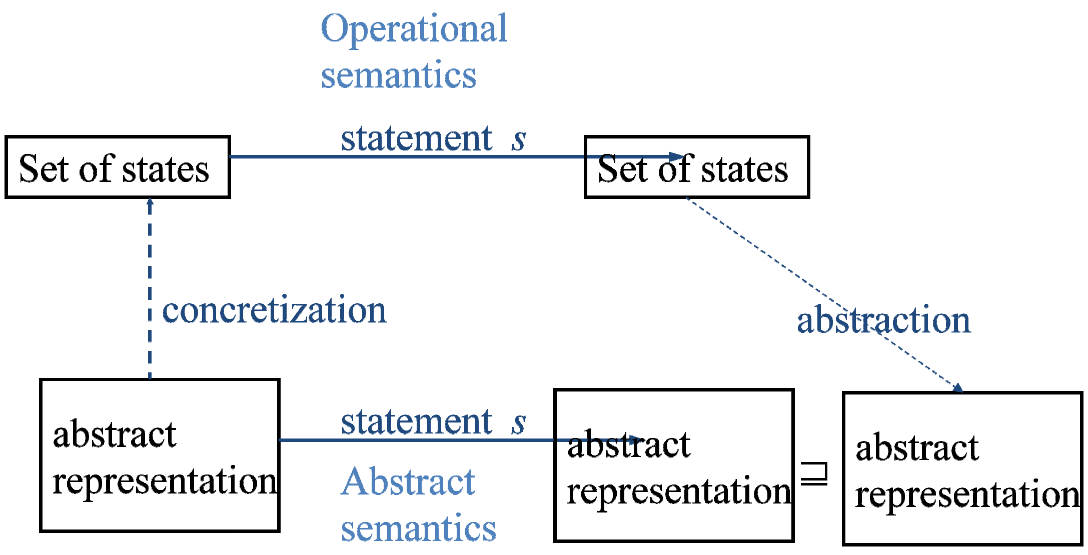
This condition means that given an abstract representation *a*, applying one step of *f* on the concrete domain on *γ(a)* results in a more accurate set of states then applying one step of $f^{\#}$ on the abstract domain on *a* and then concretize the result.

2. $\alpha(f(c)) \sqsubseteq f^{\#}(\alpha(c))$



This condition means that given a concrete representation *c*, applying one step of *f* on the concrete domain on *c* and then applying α results in a more accurate abstract representation then performing first abstraction on c using α and then applying one step of $f^{\#}$.

3. $\alpha(f(\gamma(a))) \sqsubseteq f^{\#}(a)$
   Called also the least transformer. Tells us what the most accurate abstraction that can be computed is. In other words, the best approximation that we can get in the abstract world is by concretizing *a* using γ, then perform one step of *f* in the concrete domain and then perform abstraction on the resulting set of states using α.



We note that the above three properties are equivalent.

Now we would like to show the connection between the local and the global properties. First we show that if *a* is a prefix point in the abstract domain then ***γ(a)*** is a prefix point in the concrete domain.

Proof:

- We start with $f^{\#}(a) \sqsubseteq a$ which is a prefix point on the abstract domain (since applying on step/statement on *a* does not result in a less accurate abstract representation).

- We then apply the concretization function γ. Since γ is monotone the ⊑ relation is kept, hence $\gamma(f^{\#}(a)) \sqsubseteq \gamma(a)$.

- Observing local property 1: $\forall a \in A: f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a))$, we can deduce that: $\mathbf{f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a)) \sqsubseteq \gamma(a)}$.

- From the transitive property of $\sqsubseteq$ we get that $\mathbf{f(\gamma(a)) \sqsubseteq \gamma (a)}$. Hence, $\boldsymbol{\gamma(a)}$ is a prefix point in the concrete domain.

■

The lfp (least fixed point) is the minimal among the prefix points. We have just shown that by using the above local properties we get prefix points. Therefore, we can use those properties over all fixed points according to Tarski's Fixed Point Theorem in order to get the lfp.

Therefore the equivalent global properties are:

$$(1)\ \forall a \in A\!: f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a))$$
$$(2)\ \forall c \in C\!: \alpha(f(c)) \sqsubseteq f^{\#}(\alpha(c))$$
$$(3)\ \forall a \in A\!: \alpha(f(\gamma(a))) \sqsubseteq f^{\#}(a)$$

Now we prove the correctness of the properties of global soundness:

$$(1)\ \mathbf{lfp(f) \sqsubseteq \gamma(lfp(f^{\#}))}$$
$$(2)\ \mathbf{\alpha(lfp(f)) \sqsubseteq lfp(f^{\#})}$$

Let a be the lfp of $f^{\#}$. By using property (1) and using the knowledge that a is a fixed point we get that $\boldsymbol{f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a)) \sqsubseteq \gamma(a)}$. Hence $\boldsymbol{\gamma(a)} \in Red(f)$. Using Tarski's theorem twice we have that $lfp(f^{\#}) = \sqcap Red(f^{\#})$ and $lfp(f) = \sqcap Red(f)$. So it follows that $\mathbf{lfp(f) \sqsubseteq \gamma(lfp(f^{\#}))}$ as required.

Property (2) correctness follows because from this is a Galois connection and it maintains that:

$\alpha(c) \sqsubseteq a \Leftrightarrow c \sqsubseteq \gamma(a)$.

Proof of soundness (Summary)

We have shown a global scheme for construction of globally sound abstract interpretations:

1. Define an “appropriate” structural operational semantics.

2. Define “collecting” structural operational semantics. Those help us to gather collecting states on the concrete domain.

3. Establish a Galois connection between collecting states on the concrete domain and reaching definitions on the abstract domain.

4. (Local correctness) Show that the abstract interpretation of every atomic statement is sound with respect to the collecting semantics.

5. (Global correctness) Conclude that the analysis is sound.