Program Analysis **November 29, 2015**

Lecture 6: Combining Abstract Interpreters

Lecturer: Mooly Sagiv *Scribe: Keren Zubovsky, Orit Sanandaji*

6.1 Introduction

When analyzing a program and trying to prove some property on it, the quality of the result is determined by the abstract domain choice. So far we handled programs over a single data type. Now we would like to analyze a program over different data types. In order to obtain this goal, we need to understand how to develop new abstract domain from old ones. Moreover, as there is always a trade-off between accuracy and efficiency of analysis, we would like to use the knowledge of combining simple abstract domains to create more precise ones.

6.1.1 Example

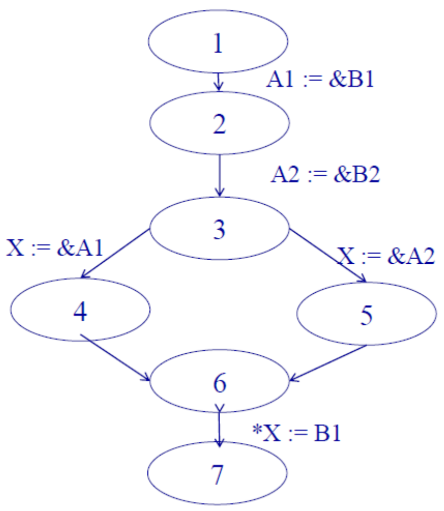
To demonstrate the need for combining Abstract interpreters, we will take a look at the following example.

We will use the pointer semantics, as shown below:

```
<com> ::= X := Y        X, Y ∈ Var
        | X := &Y
        | *X := Y
        | assume X = Y
        | assume *X = Y
        | assert X = Y
        | assert *X = Y
```

Now, let G(N,E,1) be a control flow graph where:

- N is a set of nodes
- $E \subseteq N \times N$ is a set of edges, annotated with commands
- 1 is the start node



We want to compute the abstract domain of X at the end of G.

We will traverse G, while keeping for each node a set of all the pointers and their possible values so far. We start at node 1, with $\emptyset$. Then, we execute the statements annotated on the edges by their order to get the following results:

- node 2: {A1 → B1}
- node 3: {A1 → B1, A2 → B2}
- node 4: {A1 → B1, A2 → B2, X → A1}
- node 5: {A1 → B1, A2 → B2, X → A2}
- node 6: {A1 → B1, A2 → B2, X → A1, X → A2}

Note that when we reach node 6, we apply the join operator on the results from nodes 4 and 5, meaning that we unite both of their sets. Then, when we reach node 7, we don't know which value of X is unnecessary so we keep both possibilities and therefore lose accuracy.

6.2 The loss of information in joins

The question then arises as to how to take an abstract domain which will stay accurate in join? And furthermore, given a Galois Connection, what does it **mean** to stay accurate in join? Suppose we have a domain D= $(C = 2^{\Sigma}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, and two functions:
- abstraction function α:C→A
- concrete function γ:A→C
Then we say that the domain stays accurate in join if γ is distributive with respect to join:
$\gamma(a_1 \sqcup a_2) = \gamma(a_1) \sqcup \gamma(a_2)$.

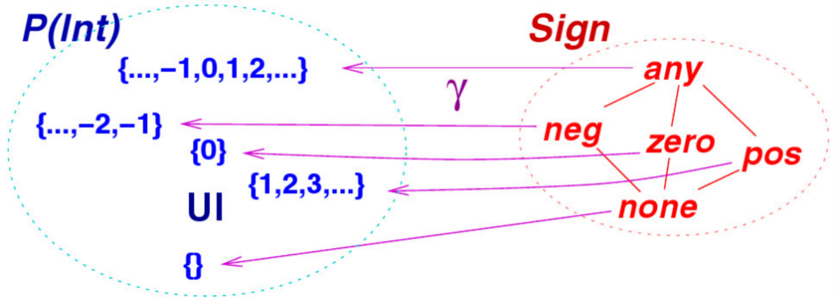
Notice that the distribution of γ over join is easier to achieve than the distribution of α.For example, when using the Interval domain, we get that α({1,2})=[1,2] and α({4,5})=[4,5], but α({1,2,4,5})=[1,5]≠[1,4]. Thus α is not distributive. The reason for this is that γ is applied on a group that takes all the related subgroups under consideration. α, on the other hand, does not maintain this quality, thus we can get side effects when applying it on a joined group, as we witnessed above.

6.3 Disjunctive Completion

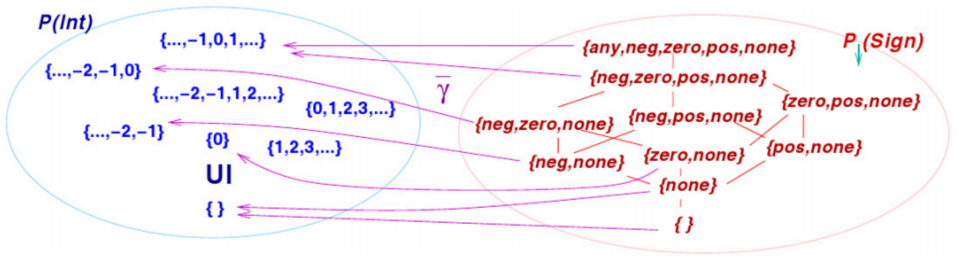
Definition 6.1 *The disjunctive completion of an abstract domain D is the smallest distributive abstract domain containing D. The disjunctive completion adds all the missing joins to the abstract domain.*

6.3.1 Example

Consider the following sign domain -



γ is not distributive; The union of the positive and the negative groups won't necessarily contain zero. However if we replace our sign domain by its power group we won't lose information by joining two subsets. Therefore, the completion of the sign domain will be:



This domain is more expressive, however exponentially larger than the starting point.

In other words, we achieve precision in the cost of complexity, which is realized by the hight of the lattice.

6.4 Cartesian Product

The elements of this domain are elements in the Cartesian product of two domains, and the operators are defined as the component-wise application of the operators of the two domains.
Formally, Given two domains: $D_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$ and $D_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$ we construct a domain D= $(D_1 \times D_2, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, so that:

- The partial order is defined as the conjunction of the partial orders of the two domain: $(a_1, b_1) \sqsubseteq (a_2, b_2) \iff a_1 \sqsubseteq_1 a_2 \wedge b_1 \sqsubseteq_2 b_2$.
- The least upper bound and the greatest lower bound operators are defined as the component-wise application of the operators of the two domains: $(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup_1 a_2, b_1 \sqcup_2 b_2)$ and $(a_1, b_1) \sqcap (a_2, b_2) = (a_1 \sqcap_1 a_2, b_1 \sqcap_2 b_2)$.

By this definition, the Cartesian product forms a lattice.

6.4.1 Widening

The same approach holds for the widening operator. Meaning that given widening operators ∇_1 and ∇_2 on domains D_1 and D_2, respectively, the widening operator ∇ on domain D $= D_1 \times D_2$ is defined as follows: $\nabla((a_1, b_1), (a_2, b_2)) = (a_1 \nabla_1 a_2, b_1 \nabla_2 b_2)$.

6.4.2 Galois Connection

Following previous definitions, we define

- The abstraction function as: α(c) = (α_1(c),α_2(c)), where α_1, α_2 are the abstraction functions from D_1 and D_2, respectively.
- The concretization function as: γ(a,b) = γ_1(a)$\sqcap\gamma_2$(b), where γ_1, γ_2 are the concretization functions of the two domains.

And then, the Cartesian product forms a Galois connection with the concrete domain, as proved bellow:
$\Rightarrow$: Assume α(c)$\sqsubseteq$(a,b). By definition we get that (α_1(c),α_2(c))$\sqsubseteq$(a,b), meaning that α_1(c)$\sqsubseteq$a and α_2(c)$\sqsubseteq$b. Since (α_1, γ_1) and (α_2, γ_2) form Galois connections we get that c$\sqsubseteq\gamma_1$(a) and c$\sqsubseteq\gamma_2$(b). Then, c$\sqsubseteq\gamma_1$(a)$\sqcap\gamma_2$(b)=γ(a,b).
$\Leftarrow$: Assume c$\sqsubseteq\gamma$(a,b). By definition we get that c$\sqsubseteq\gamma_1$(a)$\sqcap\gamma_2$(b), meaning that c$\sqsubseteq\gamma_1$(a) and c$\sqsubseteq\gamma_2$(b). As mentioned, (α_1, γ_1) and (α_2, γ_2) form Galois connections, so α_1(c)$\sqsubseteq$a and α_2(c)$\sqsubseteq$b. By the Cartesian product definition, we conclude that α(c) = (α_1(c),α_2(c))$\sqsubseteq$(a,b).

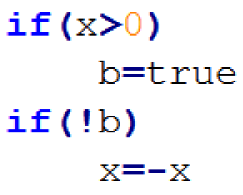
6.4.3 Examples

First, we will take a look at the following program that initializes to 0 all the elements in an array. We want

```
for (i=0; i < arr.length ; i++)
   arr[i] = 0
```

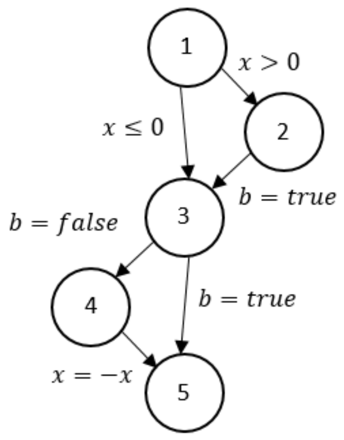
to prove that the given array is not accessed out of its bounds. In other words, we want to show that i≥0 and i<arr.length whenever the statement arr[i]=0 is executed.
If we run the analysis using only Intervals, we obtain that i≥0. However it does not guarantee that i<arr.length, since there is no upper bound for arr.length.
On the other hand, if we run the analysis using only relational domain, we can prove the second property, but not the first one.
While none of the domains alone can prove both properties, the Cartesian product can. It runs the two analyses in parallel and then takes the most precise result regarding the property to verify. This way, the whole property is proved to hold.

Another important example is given by the following program:



```
if(x>0)
    b=true
if(!b)
    x=-x
```

The program consists of two variables: boolean variable b and an integer variable x. Suppose we know that b=false at the beginning of the program, we would like to prove that x is positive at the end of it. We construct a matching CFG:



Guided by intuition, we will probably choose to run the analysis using Sign domain. However, when we execute the statements annotated on the edges of the CFG we get the following results:

- node 1: x $\rightarrow \top$
- node 2: x $\rightarrow +$
- node 3: x $\rightarrow \top$
- node 4: x $\rightarrow \top$
- node 5: x $\rightarrow \top$

Thus, we cannot determine whether x is positive at the end of the program. The same happens when we try to use Booleans. We cannot determine the value of b, and therefore we cannot decide if x is positive. However, when combining both domains into one Cartesian domain, we achieve our goal. This example demonstrates how Cartesian product can influence the precision of the analysis.

6.4.4 Conclusion

The Cartesian product is a useful tool that helps us combine different analyses into one piece of information. Moreover, in terms of complexity, the height of D=$D_1 \times D_2$ is the multiplication of the heights of D_1 and D_2, meaning that the analysis does not get expensive.

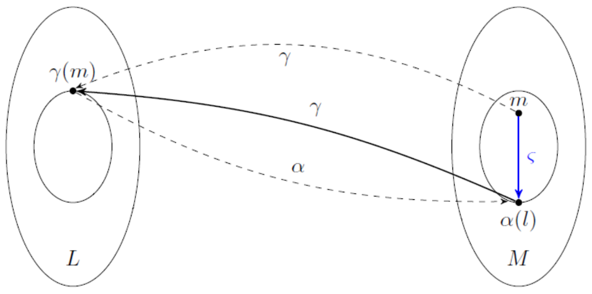
6.5 Reduced Product

As mentioned above, the Cartesian product is quite an effective way to combine two domains, however it does not utilize the interaction between domains (with respect to atomic operations). We want to find a way so that each analysis in the abstract composition benefits from the information brought by the other analyses.
One solution is to employ semantic reduction. In particular, given an abstract state that is non-minimal we can take the smallest element which represents the same information by reducing it.The semantic reduction improves the precision of the analysis without effecting its concrete meaning.
Formally, let (a_1,a_2) be of a reduced product, where $a_1 \in D_1$ and $a_2 \in D_2$ and D_1, D_2 are domains. Let c_1 and c_2 be the set of concrete values associated to a_1 and a_2, respectively. Then (a_1, a_2) represents the set of concrete elements $c_1 \bigcap c_2$. The reduction tries to find the minimal element (a_1',a_2') such that concretizations of a_1' and a_2' are subsets of a_1 and a_2 but their intersection remains the same as the original one.

6.5.1 The Reduction Operator

Definition 6.2 *Let (L,α,γ,M) be a Galois connection. Then, the reduction operator ζ:M→M is defined by ζ(m)=$\sqcap${m' |γ(m)=γ(m')}*

The construction of ζ is described bellow:



By this definition, ζ[M]=({ζ(m)|m∈M}, $\sqsubseteq_M$) is a complete lattice and (L,α, γ, ζ[M]) is a Galois insertion.

In general, the reduction operator has to satisfy the following properties:

1. ∀m∈M ζ(m)⊑m
2. $\gamma(\zeta(\mathrm{m})) = \gamma(\mathrm{m})$

6.5.2 Example

Consider the product of the following domains: D_1=Intervals and D_2=Parity.
A simple reduction operator may increase by one the lower bound and decrease by one the upper bound of the interval if the bounds does not respect the Parity information. For instance, the reduction of ([12,18],O) yields the abstract value ([13,17],O).
However, the reduction operator does not always obtain minimal results. According to our example, ([3,3],E) yields (⊥,E), but it could further be reduced to (⊥, ⊥).

As useful as it may sound in theory, semantic reduction is not used in practice because it is not computable or too expensive. Therefore, the reduction operator is usually computed to a certain point by repetitively updating the lower and the upper bounds, as used by the generic system TVLA.

6.5.3 Granger product

An elegant solution to compute an approximation of the semantic reduction, suggested by Philippe Granger. Granger based his new product on the definition of two operators: $\rho_1 : D_1 \times D_2 \to D_1$ and $\rho_2 : D_1 \times D_2 \to D_2$, such that:

- $\rho_1(d_1, d_2) \sqsubseteq^1 d_1$ and $\gamma(\rho_1(d_1, d_2), d_2) = \gamma(d_1, d_2)$
- $\rho_2(d_1, d_2) \sqsubseteq^2 d_2$ and $\gamma(d_1, \rho_2(d_1, d_2)) = \gamma(d_1, d_2)$

The idea is that each operator refines one of the two domains involved in the product. Dealing with only one flow of information at a time is simpler. Each of the two operators ρ_1,ρ_2 tries to descend in one of the lattices: given the abstract element made by the pair (d_1, d_2), the ρ_1 operator uses the information from d_2 to go down the lattice of D_1, while ρ_2 uses the information from d_1 to go down the lattice of D_2.
The semantic reduction is obtained by iteratively applying ρ_1 and ρ_2, until the operators cannot recover any more precision and we reach the fixpoint. this is defined by the sequence $(a_n, b_n)_{n\in\mathbb{N}}$ as follows:
$(a_0, b_0) = (a, b)$
$(a_{n+1}, b_{n+1}) = (\rho_1(a_n, b_n), \rho_2(a_n, b_n))$

6.5.4 Example 1 - Intervals and Inequalities

Consider the following program which receives as input an integer variable I, and creates an array with one element if $I \leq 0$, otherwise an array with I elements. Eventually it initializes the first I elements to zero.

```
if (I <= 0)
  arr = new Int[1]
else
  arr = new Int[I]
for (i=0; i < I ; i++)
   arr[i] = 0;
```

As before, we want to prove that when we preform $arr[i] = 0$ we know that $i \geq 0$ and $i < arr.length$.

The first property has already been proved by Intervals as explained in the previous example, so let us focus on the second property $i < arr.length$.
If we analyze this code with the Cartesian product we obtain that:

- $(\{arr.length \mapsto [1...1], I \mapsto [-\infty...0]\}, \phi)$ in the *then* branch
- $(\{arr.length \mapsto [1...+\infty], I \mapsto [1...+\infty]\}, \{arr.length \leq I, I \leq arr.length\})$ in the *else* branch

Then when we compute the upper bound of this two states after the if statement, we get the result $(\{arr.length \mapsto [1...+\infty], I \mapsto [-\infty...+\infty]\}, \phi)$. This leads to infer that inside the for loop $(\{arr.length \mapsto [1...+\infty], I \mapsto [-\infty...+\infty], i \mapsto [0...+\infty]\}, \{i < I\})$, but this cannot prove that $i < arr.length$.

When we use the reduced product we define a specific reduction operator that refines the information tracked by the Inequalities with Intervals: if Intervals track that $x \mapsto [a...b], y \mapsto [c...d]$ when $b \neq +\infty$ and $c \neq -\infty$, then we will add the inequality relation $x < y + b - c + 1$.
We will infer that in the *then* branch it's $(\{arr.length \mapsto [1...1], I \mapsto [-\infty...0]\}, \{I < arr.length + 0\})$ and when we join the two states after the if statement we obtain that $(\{arr.length \mapsto [1...+\infty], I \mapsto [-\infty...+\infty]\}, \{I < arr.length + 1\})$.
Eventually inside the for loop we get $(\{arr.length \mapsto [1...+\infty], I \mapsto [-\infty...+\infty], i \mapsto [0...+\infty]\}, \{I < arr.length + 1, i < I + 0\})$, and this proves that $i < arr.length$.

6.5.5 Example 2 - Intervals and Inequalities

This example shows that in general, even the reduced product cannot achieve the most precise result.

```
if (I <= 2)
  arr = new Int[1]
else
  arr = new Int[I]
for (i=3; i < I ; i++)
   arr[i] = 0;
```

Same example as before, but now the condition inside the if statement is $I \leq 2$, and the array initialization starts from the third element.
When using the same reduction operator we previously saw, the abstract state associated to the *then* branch will be $(\{I \mapsto [-\infty ... 2], arr.length \mapsto [1...1]\}, \{I < arr.length + 2\})$, while in the *else* branch we obtain $(\{I \mapsto [3... + \infty], arr.length \mapsto [3... + \infty]\}, \{arr.length \leq I, I \leq arr.length\})$. The join of this two states will be $(\{I \mapsto [-\infty ... + \infty], arr.length \mapsto [1... + \infty]\}, \{I < arr.length + 2\})$.
Inside the for loop we know that $i \mapsto [3... + \infty]$ and $i < I$. From $i < I$ and $I < arr.length + 2$ we infer that $i < arr.length + 1$, but that is weaker than the property $i < arr.length$.
We need a better solution.

6.6 Reduced cardinal power

The main feature of the cardinal power is that it allows to track disjunctive information over the abstract values of the analysis. It combines the two domains in a way which keeps correlations between the individual elements. In this product the element $a \rightarrow b$ means that if the state obeys a, it also obeys b.

As before, given the abstract domains D_1 and D_2, and their properties, we construct a new domain: $D = (D_1 \rightarrow D_2, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$. The cardinal power $D = {D_2}^{D_1}$, with base D_2 and exponent D_1, is the set of all isotone maps $D : D_1 \rightarrow D_2$. Roughly, the combination of two abstract domains in ${D_2}^{D_1}$ means that a state in D_1 implies the abstract state of D_2 it is in relation with.

The partial ordering $\sqsubseteq$ is defined by: $f \sqsubseteq g \leftrightarrow \forall d_1 \in D_1 : f(d_1) \sqsubseteq_2 g(d_1)$

Similarly, the least upper bound and the greatest lower bound operators are defined as the pointwise application of the operators of D_2:

- $\sqcup : f \sqcup g = \lambda x. f(x) \sqcup g(x)$
- $\sqcap : f \sqcap g = \lambda x. f(x) \sqcap g(x)$

This way $(D_1 \rightarrow D_2, \sqsubseteq, \sqcup, \sqcap)$ forms a lattice.
Likewise, the bottom and top are defined as:

- $\perp : \lambda d.\perp$
- $\top : \lambda d.\top$

The Galois connection, assuming that
$\alpha_1 : D \to D_1, \gamma_1 D_1 \to D$
$\alpha_2 : D \to D_2, \gamma_2 : D_2 \to D$:

- $\forall y_1 \in D_1 : \alpha(x) = \lambda y_1.\alpha_2(x \sqcap \gamma_1(y_1))$
- $\gamma(f) = \sqcup\{x \in D | \forall y_1 \in D_1, \gamma_1(y_1) \sqcap \gamma_2(f(y_1))\}$

Back to Example 6.5.4: now we consider the reduced cardinal power of the Intervals on I as exponent and the relational domain as base. The *then* branch is associated to the abstract state $[-\infty...2] \to \{I < arr.length + 2\}$, and the *else* branch to $[3... + \infty] \to \{arr.length \leq I, I \leq arr.length\}$. The join between these two states creates a new abstract state which contains both informations, that is $[-\infty...2] \to \{I < arr.length + 2\}$ and $[3... + \infty] \to \{arr.length \leq I, I \leq arr.length\}$. When we enter the for loop, we need to consider the two cases separately:

- In the first case, $I = [-\infty...2]$. Then, the loop condition $i < I$ is surely evaluated to false. The loop is not executed, so we don't need to verify the property about array accesses. Therefor, when $i < I$ holds, we have that $[-\infty...2] \to \perp$. This way, when we analyze $arr[i] = 0$, we can discard this case.
- In the second case, we know that $I = [3... + \infty]$ and $i < I$. Then, from the abstract state obtained after the if statement and assuming the loop condition, we know that $[3... + \infty] \to \{arr.length \leq I, I \leq arr.length\}$. By combining $I \leq arr.length$ and $i < I$, we obtain that $i < arr.length$, which is the property that we wanted to prove.

6.6.1 Example - Booleans and Signs

```
1: x := 100;  b := true;
2: while b do {
3:        x := x – 1;
4:        b := (x > 0);
5: }
```

The exponent of the cardinal power we use to analyze this example is the Boolean domain for variable b, and the base is the Sign domain for variable x tracking values +,-,0. This way we track that when b has a particular Boolean value, then variable x has a particular Sign.
Before entering the while loop, we know that $b = true \to x = \{+\}$ while $b = false \to x = \{\perp\}$. After the application of statement 3, we will have that $b = true \to x = \{0, +\}$ while $b = false \to x = \{\perp\}$, because the value of b is unchanged while the value of x has been decreased by one (so it could become equal to 0 or stay a positive integer). After line 4, we obtain that $b = true \to x = \{+\}$ and $b = false \to x = \{0\}$ since we know that $x \geq 0$ and the condition $x > 0$ is assigned to b.
The fixpoint computation over the while loop stabilizes immediately (because if we enter the loop again we know that b is true and as consequence x is positive, thus returning to the same conditions of the first iteration), and so we obtain that at the end of the program we have that $b = true \to x = \perp$ and $b = false \to x = \{0\}$, since we have to assume the negation of b to terminate the execution of the while loop.

6.6.2 Practical Applications of Reduced Cardinal Power

- **Astree Branch Correlations -** Astree is a static program analyzer aiming at proving the absence of Run Time Errors (RTE) in programs written in the C programming language. Astree exploits the Boolean relation domain, which applies the cardinal power using the values of some particular Boolean program variables as exponent. In this way, the analysis tracks precise disjunctive information w.r.t these variables.
- **Interprocedural analysis**
- **Shape Analysis**