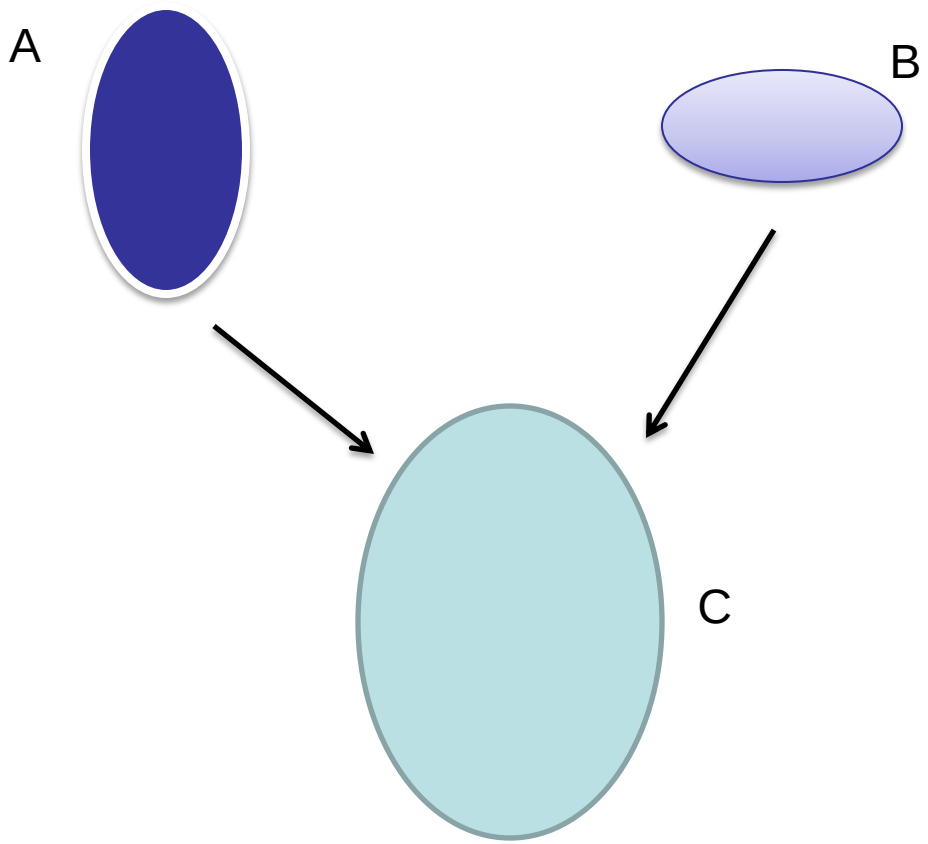
Abstract Interpretation: a survey on product operators

Combining domains

How to combine the analysis on an abstract domain A and the analysis on another abstract domain B?



Example

```
1. x = 7;
2. y = 14;
3. if ( x*y > 0 && y%2=0) then…
```

- By an analysis with Sign we now that the first condition of statement 3 is satisfied
- By an analysis with Parity we know that the second condition of statement 3 is satisfied
- We have to use them together!

Cartesian Product

- The elements of this domain are elements of the cartesian product R=AxB
- The partial order is defined componentwise: $(a,b) \leq_R (a',b')$ if $a \leq_A a'$ and $b \leq_B b'$
- Same for lub and glb
- The cartesian product forms a Galois connection with the concrete domain
- The semantics operator $\sigma_R[(a,b)]=(\sigma_A[a], \sigma_B[b])$
- Correctness: $\mu_C[\gamma_R(a,b)] \leq_C \gamma_R(\sigma_R[a,b])$

Cartesian Product

- The Cartesian product discovers in one shot the information found separately by the component analyses, but we do not learn more by performing all analyses simultaneously than by performing them one after another and finally taking their conjunctions.

- Consider R=Intervals x Parity

  ([2,4],Odd), ([2,3],Odd), ([3,4],Odd), ([3,3],Odd) are all mapped by γ_R to the set {3}.

  This result into inefficiencies of the analysis!

Complexity and Implementation

- The complexity of the operators defined on C is exactly the sum of the complexity of the corresponding operators on A and B.
- The height of the lattice of R (that is important to estimate the complexity of computing a fixpoint using this domain) is the multiplication of the heights of A and B.
- Given the implementations of A and B, the implementation of the analysis on R=AxB is completely straightforward.

Reduced Product

- The reduced product of two domains A and B is the subset of AxB whose elements are
  $\{ \rho(a,b) : (a,b) \text{ is in AxB}\}$
  where the reduction operator ρ is defined by:

  $\rho(a,b) = \text{glb}\{ (a',b') : \gamma_R(a,b) \leq_C \gamma_R(a',b')\}$

- For instance, consider R=Intervals x Parity.
  ρ ([2,4],Odd) = ρ([2,3],Odd) = ρ([3,4],Odd) = ([3,3],Odd)

  ([3,3],Odd) is the glb of all the pairs in Intervals x Parity that over-approximate by γ_R the set {3}.

Reduction operator

- A reduction operator has to satisfy the following two properties:

1. $\rho(d) <= d$
   the result of its application is a more precise abstract element
2. $\gamma(\rho(d))=\gamma(d)$
   an abstract element and its reduction represent the same property

Example

- What is the element obtained by reduction from ([1,1],Even) when considering the reduced product of Intervals and Parity?

Complexity

- In addition to the complexity of the Cartesian product, the reduced product requires to compute the reduction operator.
- Therefore, the complexity of an operator of the reduced product is the sum of the complexity of the operators defined on A and B and of the reduction operator.
- The final cost of a generic operator could be rather expensive.
- The height of the lattice has as upper bound the multiplication of the heights of A and B

Implementation

- The implementation of the reduction operator has to be specific for the domains we are refining.
- Therefore, while the Cartesian product was completely automatic, the reduced product requires one to define and implement how two domains let the information flow among them.
- This means that each time we want to combine two domains in a reduced product we have to implement such operator.

Granger's product

- An elegant solution to compute an overapproximation of the reduction operator.
- Define two operators ρ_A from AxB to A
  and ρ_B from AxB to A
- Each operator refines one of the two domains

$$\rho_A(a,b) \leq_A a \text{ , and } \gamma_R(\rho_A(a,b), b) = \gamma_R(a,b)$$
$$\rho_B(a,b) \leq_B b \text{ , and } \gamma_R(a, \rho_B(a,b)) = \gamma_R(a,b)$$

- The reduction operator is defined as the fixpoint of the following decreasing iteration sequence:

$(a_0,b_0) = (a,b)$

$(a_{n+1},b_{n+1}) = (\rho_A(a_n,b_n) , \rho_B(a_n,b_n))$

Granger's product

- The Granger product has exactly the same complexity we discussed for the reduced product.
- The main practical advantage of the Granger product is that one only needs to define and implement ρ_A and ρ_B, that is, how the information flows from one domain to the other in one step.
- Then the reduction operator relying on the fixpoint computation comes for free.

ρ_{Intv}

$\rho_{Intv}([a,b],p) = ref([a',b'])$

$a' = \begin{cases} a & \text{if } (parity(a) = p) \\ a+1 & \text{otherwise} \end{cases}$

$b' = \begin{cases} b & \text{if } (parity(b) = p) \\ b-1 & \text{otherwise} \end{cases}$

$ref([a,b]) = \begin{cases} \perp_{\mathbf{Intv}} & \text{if } a>b, \\ [a,b] & \text{otherwise} \end{cases}$

$\rho_{Intv}([2,7],Even) = [2,6]$
$\rho_{Intv}([1,7],Even) = [2,6]$
$\rho_{Intv}([2,7],Odd) = [3,7]$

ρ_{Parity}

$\rho_{Parity}(I,p) = p'$

$p' = \quad \perp_{\mathbf{Parity}}$ if $I = \perp_{\mathbf{Intv}}$ or $I=[a,a]$ with parity(a) != p
p otherwise

$\rho_{Parity}([2,7],Even) = Even$
$\rho_{Parity}([1,7],Odd) = Odd$
$\rho_{Parity}([2,2],Odd) = \perp_{\mathbf{Parity}}$

Example

- What is the element obtained by Granger's product from ([1,1],Even) when considering Intervals and Parity?

ρ_{Intv} ([1,1],Even) = ref([2,0]) = $\bot_{\mathbf{Intv}} \leq_{Intv}$ [1,1]
ρ_{Parity} ([1,1],Even) = $\bot_{\mathbf{Parity}} \leq_{Parity}$ Even

ρ_{Intv} ([$\bot_{Intv}$, $\bot_{Parity}$) = $\bot_{\mathbf{Intv}}$
ρ_{Parity} ($\bot_{Intv}$, $\bot_{Parity}$) = $\bot_{\mathbf{Parity}}$

Then a fixpoint is reached and the result is ($\bot_{intv}$, $\bot_{Parity}$)

Reduced Cardinal Power

- The main feature of the cardinal power is that it allows one to track disjunctive information over the abstract values of the analysis.
- For instance, given the Interval and the Parity domain, one could track information like “when x is odd, y is in [0..10]”.
- Given two abstract domains, A and B, the cardinal power $R = B^A$ with base B and exponent A is the set of all isotone maps ϕ from A to B.
- The combination of two abstract domains in B^A means that a state in A implies the abstract state of B it is in relation with.

Example

y= 27
if (x<0) then y=0

α_A: Int -> Sign $\alpha_A(x)$= sign(x)

α_B: Int -> Boolean $\alpha_B(x)$= true if x>0, false otherwise

The variable y will be assigned the a value in $\text{Sign}^{\text{Boolean}}$ such that:

true(x) is mapped to +(y)
false(x) is mapped to 0(y)

Reduced Cardinal Power

- The partial ordering on the reduced cardinal power $R = B^A$ is defined by:
  $f \leq_R g$ if and only if
  for all x in A: $f(x) \leq_B g(x)$ in B.
- The abstraction function a can be defined by:

$$\alpha_R(c) = \lambda x.\ \alpha_B(\ \mathrm{glb}_C\ (c(y) \mid y \text{ in } \gamma_A(x))$$

Example

```
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, while the base is the Sign domain for variable x tracking also values 0+ and 0- (values greater/lower or equal to 0).
- $\text{Sign}^{\text{Boolean}}$ tracks that when variable b has a particular Boolean value, then the variable x has a particular sign.

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

- Before entering the while loop, we know that:
  b=true implies x=+,
  b=false implies x= ⊥, since if the loop guard holds, the only possible value of b is true.
- After the application of the semantics of statement 3 (first iteration), we will have that:
  b = true implies x = 0+,
  b = false implies x = ⊥.

```
1: x := 100; b := true;      b=true implies x=+ ;  b=false implies x= ⊥
2: while b do {
3:   x := x-1;               b=true implies x=0+ ;  b=false implies x= ⊥
4:   b := (x > 0);
5: }
```

- After line 4, we obtain that:
  b = true implies x = +, and
  b = false implies x = 0, since the new condition x > 0 is assigned to b.
- The fixpoint computation over the while loop stabilizes immediately. At the end of the program we have that:
  b = true implies x = ⊥, and b = false implies x = 0, since we have to assume the negation of b to terminate the execution of the while loop.

Complexity

- Each time a lattice or semantics operator has to be applied to the abstract state, the cardinal power requires to apply it to all the elements of the base.
- Let n be the number of states of A, *a* the cost of an operator on A and *b* the cost on B. Then the overall cost over B^A is n*(*a*+*b*), since, for any element in A we have to apply the operator both on A and B.
- Let h_A and h_B be the height of the lattice of A and B, respectively. Then the height of B^A is $h_B{}^{h_A}$

Implementation

- The implementation can be rather simple when using a programming language providing functional constructs.
- In fact, the most part of the cardinal power (namely, elements of the domain, and lattice and semantics operators) can be defined as the functional point-wise application of the operators on the base and the exponent.
- Instead, the implementation of the cardinal power may be more verbose using an imperative programming language, but we do not expect it represents a significant challenge.

A more complex example

- The two abstract domains we will combine are the domain of Intervals and a relational domain that tracks constraints of the form x < y+c.
- As a first example, we consider a quite standard program that initializes to 0 all the elements of a given array:

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

- We want to prove that the array accesses are safe, that is, i=0+ and i < arr.length, in particular when we perform arr[i] = 0.

Cartesian product

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

- If we run the analysis using only Intervals, we obtain that i = [0,+inf]. In fact, this domain cannot infer any information from the loop guard i < arr.length since it does not have any upper bound for arr.length.
- On the other hand, if we run the analysis with the relational domain, we obtain that i < arr.length+0 when we analyze the statement arr[i] = 0 thanks to the loop guard.
- Therefore, the two domains alone cannot prove the property of interest, while the Cartesian product can.

Reduced Product

- Let us introduce a more complex example. It receives as input an integer variable l, and it creates an array with one element if l 0, and of l elements otherwise. Then it initializes the first l elements to zero.

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

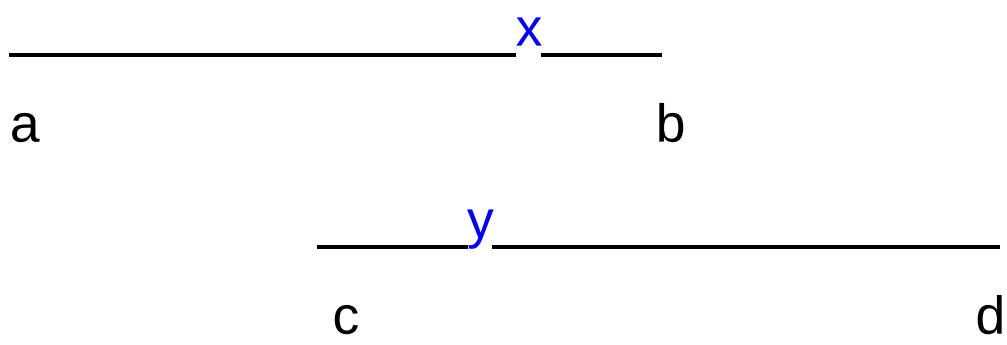
- As before, we want to prove that when we perform arr[i] = 0 we have that i=0+ and i < arr.length.

… the cartesian product is not enough

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

- If we analyze this example with the Cartesian product, we obtain that:
  ((arr.length = [1,1]; l =[-inf,0]), --) in the then branch, and
  ((arr.length = [1,+inf]; l =[1,+inf]), (arr.length < l+1; l<arr.length+1)) in the else branch.
- When we compute the upper bound of these two states, we obtain:
  ((arr.length = [1,+inf]; l =[-inf,+inf]), --)
- And so, inside the loop, we get:
  ((arr.length = [1,+inf]; l =[-inf,+inf]), i=[0,+inf]), (i < l+0)) but this cannot prove that i < arr.length.

- We now define a specific reduction operator that refines the information tracked by the relational domain with Intervals.
- In particular, if Intervals track that x = [a,b], y=[c,d] and we have that b!= +inf and c != -inf, then in the relational domain we introduce the constraint x < y + k where k = b-c+1.



- If Intervals track that x = [a,b], y=[c,d] and we have that b!= +inf and c != -inf, then in the relational domain we introduce the constraint x < y + k where k = b-c+1.

- Thanks to this reduction operator, we infer that, in the “then” branch, ((arr.length = [1,1]; l =[-inf,0]), l < arr.length+0).
- So, when we join the two abstract states after the if/else statement we obtain that:
  ((arr.length = [1,+inf]; l =[-inf,+inf]), l < arr.length+1)
- And so, inside the loop, we get:
  ((arr.length = [1,+inf]; l =[-inf,+inf]), i=[0,+inf]),
  (l < arr.length+1 , i < l+0))
  and this allows to prove that i<arr.length.

Reduced Cardinal Power

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

Also in this case we want to show that i <arr.length when executing arr[i] = 0. But this cannot be obtained neither by cartesian nor by reduced product.

It can be reached by using the reduced cardinal power $T^{Intervals}$, where T is the relational domain used before.

just consider the two cases :

l =[-inf,2] implies $\perp$ as in this case we don't get into the for loop

l =[3,+inf] implies array.lenght=l and i<l, i.e. i <arr.length