Last Time

- Major result from this class: Abstract fixpoint approximates concrete fixpoint
- And requirements this imposes on…
  - Domain structure
  - Galois connection between domains
  - Transfer functions between elements of the abstract domain
- Partitioning
- Example domains
- Widening

- Homework 4 due today
- Homework 5 assigned

Today

- Combining abstract domains

Widening

- Problem: Abstract domains with large or infinite height result in analysis that is slow or non-terminating
- I.e. for some domains we can't (efficiently) compute:

$$fix = \bigsqcup F^i(\bot, \ldots, \bot)$$

- Solution: Introduce a widening operator w that composes with the transfer function like this:

$$fixw = \bigsqcup (F \circ w)^i(\bot, \ldots, \bot)$$

Widening

- As long as w is monotonic, the analysis remains sound
- Example widening function for intervals:
  - For some fixed set of constants, find the smallest interval with bounds chosen from this set that contains starting interval
  - For example, for the set {-∞, -10, -1,1, 10, ∞}
    - [0..1] goes to [-1..1]
    - [-5..0] goes to [-10..1]
    - [-15..11] goes to [-∞.. ∞]
  - Set could be found by looking for constants in the source code

---

Widening Example

- Let's say we want to analyze this code using the interval domain

```
y=0;
while (opaque) {
  x=7;
  x++;
  y++;
}
```

- Interval analysis does not terminate (quickly)
  - Result is x=[8..8], y=[0..∞]

Widening Example

```
y=0;
while (opaque) {
  x=7;
  x=x+1;
  y=y+1;
}
```

- Widening with the set of constants {-∞, 0, 1, 7, ∞}
  - Result is x=[7..∞], y=[0..∞]

Widening Discussion

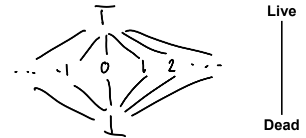
- Why use widening when we could just define a rapidly-terminating lattice ahead of time?
  - Widening effectively permits a new sub-lattice to be picked for each program analyzed
  - Avoids messing with the original, clean domain
  - Narrowing – can potentially recover some precision by running the original transfer functions after widening has overshot the fixpoint

Combining Abstract Domains

- Domains can be combined in different ways
- First we look at why, then how

- Motivating example program:

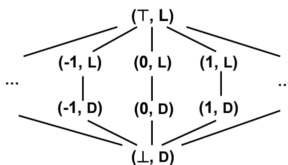
```
x = 1;
while (opaque) {
  if (x != 1) {
    x = 2;
  }
}
```



- First analyze using constant propagation
- Second perform liveness analysis
- Iterate if needed

Direct Product of Domains

- Compute the direct product of the constant and liveness lattices
  - Lattice is cross product
  - Transfer functions are created by applying original functions element-wise
- For programs with one variable:



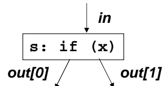
Direct Product

- Now analyze using the combined domain

```
x = 1;
while (opaque) {
  if (x != 1) {
    x = 2;
  }
}
```

- What happens?
- What's going on here?

Direct Product



For constants:

out [0] = *in* ∧
out [1] = *in*

For liveness:

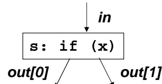
out [0] = ⊥ if *x* ≠ *0* or *in* = ⊥
∧
out [1] = ⊥ if *x* = *0* or *in* = ⊥

Here 0 on the right side is literal zero

Reduced Product of Domains

- **Problem: Domains failed to interact**
  - **Direct product is not any better than running analyses one after the other**
- **Solution: Make the domains interact**
  - **Requires some new transfer functions!**
  - **The most precise combination of domains is called the reduced product**

Reduced Product



For constants:

out [0] = *in* ∧
out [1] = *in*

For liveness:

out [0] = ⊥ if *x* ≠ *0* or *in* = ⊥
∧
out [1] = ⊥ if *x* = *0* or *in* = ⊥

Here 0 on the right side is an element of the constant lattice

Back to Example

```
x = 1;
while (opaque) {
  if (x != 1) {
    x = 2;
  }
}
```

- **Finally we get the desired result**
  - **This analysis is called "conditional constant propagation"**
  - **The original paper on CCP is a classic**
  - **CCP is used in real compilers, such as gcc**

Reduced Product Background

- **Recall the requirement for local transfer function F_A:**
  - $\alpha \circ F_C \circ \gamma \sqsubseteq_A F_A$
- **In other words, the optimal transfer function is obtained by:**
  - **Concretizing the abstract value(s)**
  - **Applying the concrete transfer function**
  - **Abstracting the resulting set of concrete values**

Reduced Product

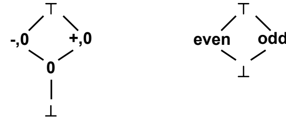
- **A reduced product transfer function is defined as (and may be computed by):**
  - **Concretizing the abstract values in all domains**
  - **Applying the concrete transfer functions**
  - **Intersecting the resulting sets of concrete values**
  - **Abstracting the intersection**
- **Not generally computable**
  - **Just like good transfer functions in the first place**
  - **Efficient reduced product implementations require hard work**

Another way to look at it

- Individual dataflow analyses can be internally optimistic
  - This means: Abstract states start at the bottom
- However, dataflow results must be pessimistic
- Lack of optimism across different analyses results in missed opportunities
- Reduced product means maximal sharing of optimism across domains

Another Example

- Consider lattices of signs, parity, and constants



- Exercise: Give a nontrivial transfer function from the reduced product domain
  - (+,0, odd, ⊤) - (⊤ , ⊤ , 1) =
  - (+,0, even, ⊤)

Another Example

```
a = 5;
b = 10;
p = &a;
while (*p > 5) {
  p = &b;
  b = 20;
}
```

- Does CCP do the trick here?
- What needs to be added?

Domain Products

- Direct product is easy but does not win
- Reduced product is the most precise but is not computable
- Products between these exist
  - One is Granger's product

Granger's Product

- Let $\sigma_1 : D_1 \times D_2 \rightarrow D_1$ such that
  - $\sigma_1(d_1,d_2) \leq d_1$
  - $\gamma(\langle \sigma_1(d_1,d_2) , d_2 \rangle) = \gamma(\langle d_1,d_2 \rangle)$
- Let $\sigma_2 : D_1 \times D_2 \rightarrow D_2$ such that
  - $\sigma_2(d_1,d_2) \leq d_2$
  - $\gamma(\langle d_1 , \sigma_2(d_1,d_2) \rangle) = \gamma(\langle d_1,d_2 \rangle)$
- To use these, apply the element-wise transfer functions to (d_1,d_2) as in the direct product, then repeatedly reduce these abstract values using σ_1 and σ_2 until a fixpoint is reached
- Simple and efficient
  - But cannot capture some interactions
  - Reduction functions need to be designed by hand

- Abstract interpretations can be compared using a lattice
  - Partial order is "at least as precise as"
- Bottom of the lattice is the collecting semantics
- Top is the trivial domain with one element
- Direct product is LUB of its component domains
  - That is, the least precise of all the domains that are at least as precise as either component
  - Reduced product is always at least as precise as direct

Summary

- Different domains discover different information about a program
- Combining domains permits them to "learn" from each other
- Combining abstract domains provides a powerful and systematic way to design program analyses
  - I.e., start with many simple domains and compose them into useful composite domains
  - Con: Domain combination isn't easy
  - Pro: May still be easier than designing the composite domain from scratch