Dataflow Analysis and Abstract Interpretation

Computer Science and Artificial Intelligence Laboratory
MIT

November 9, 2015

Recap

Last time we developed from first principles an algorithm to derive invariants.

Key idea:

- Define a lattice of possible invariants
- Define a fixpoint equation whose solution will give you the invariants

Today we follow a more historical development and will present a formalization that will allow us to better reason about this kind of analysis algorithms

Dataflow Analysis

First developed by Gary Kildall in 1973

- This was 4 years after Hoare presented axiomatic semantics in 1969, which itself was based on the work of Floyd in 1967
- The two approaches were not seen as being connected to each other

Framework defined in terms of “pools” of facts

- Observes that these pools of facts form a lattice, allowing for a simple fixpoint algorithm to find them.
- General framework defined in terms of facts that are created and destroyed at every program point.
- Meet operator is very natural as the intersection of facts coming from different edges.

Forward Dataflow Analysis

Simulates execution of program forward with flow of control

For each node n, have

- in_n – value at program point before n
- out_n – value at program point after n
- f_n – transfer function for n (given in_n, computes out_n)

Require that solution satisfy

- $\forall n.\ out_n = f_n(in_n)$
- $\forall n \neq n_0.\ in_n = \vee \{ out_m \ .\ m \text{ in } pred(n) \}$
- $in_{n0} = I$
- Where I summarizes information at start of program

Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

$$out_n := f_n(in_n)$$

$$in_n := \vee \{ out_m \,.\, m \text{ in } pred(n) \}$$

Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each n do $out_n := f_n(\bot)$
$in_{n0} := I$; $out_{n0} := f_{n0}(I)$
worklist := N - { $n_0$ } //N is the set of all nodes
while worklist $\neq \emptyset$ do
    remove a node n from worklist
    $in_n := \vee \{ out_m \mid m \text{ in } pred(n) \}$
    $out_n := f_n(in_n)$
    if $out_n$ changed then
        worklist := worklist $\cup$ succ(n)
```

Correctness Argument

Why result satisfies dataflow equations?

Whenever a node n is processed, $out_n := f_n(in_n)$

Algorithm ensures that $out_n = f_n(in_n)$

Whenever out_n changes, put succ(n) on worklist.

Consider any node m ∊ succ(n). When it comes off the worklist, the algorithm will set

$$in_n := \vee \{ out_m \ . \ m \text{ in } pred(n) \}$$

to ensure that $in_n = \vee \{ out_m \ . \ m \text{ in } pred(n) \}$

So final solution will satisfy dataflow equations

Termination Argument

Why does algorithm terminate?

Sequence of values taken on by in_n or out_n is a chain. If values stop increasing, worklist empties and algorithm terminates.

If lattice has finite chain property, algorithm terminates

- Algorithm terminates for finite lattices

Abstract Interpretation

History

POPL 77 paper by Patrick Cousot and Radhia Cousot

- Brings together ideas from the compiler optimization community with ideas in verification
- Provides a clean and general recipe for building analyses and reasoning about their correctness

Collecting Semantics

We are interested in the states a program may have at a given program point

- Can x ever be null at program point i
- Can n be greater than 1000 at point j

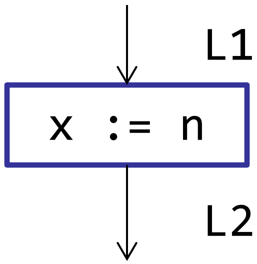
Given a labeling of program points, we are interested in a function

- $\mathcal{C}: Labels \rightarrow \mathcal{P}(\Sigma)$
- For each program label, we want to know the set of possible states the program may have at that point.

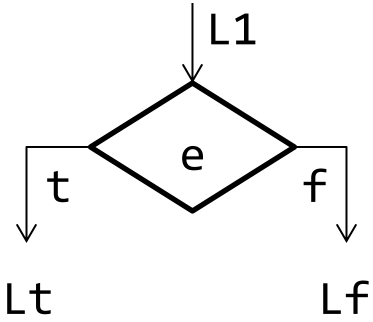
This is the collecting semantics

- Instead of defining the state of the program at a given point, define the set of *all states* up to that given point.

Defining the Collecting Semantics

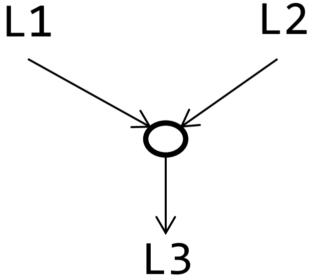


$$\mathcal{C}[L2] = \{\sigma[x \to n] \mid \sigma \in \mathcal{C}[L1]\}$$



$$\mathcal{C}[Lt] = \{\sigma \mid \sigma \in \mathcal{C}[L1], [\![e]\!]\sigma = true\}$$

$$\mathcal{C}[Lf] = \{\sigma \mid \sigma \in \mathcal{C}[L1], [\![e]\!]\sigma = false\}$$



$$\mathcal{C}[L3] = \mathcal{C}[L1] \cup \mathcal{C}[L2]$$

Computing the collecting semantics

Computing the collecting semantics is undecidable

- Just like computing weakest preconditions

However, we can compute an *approximation* $\mathcal{A}$

- Approximation is *sound* as long as $\mathcal{C}[Li] \subset \mathcal{A}[Li]$.

Abstract Domain

An abstract domain is a lattice

*Some analysis relax this restriction.

- Elements in the lattice are called *Abstract Values*

Need to relate elements in the lattice with states in the program

- **Abstraction Function**: $\alpha : \mathcal{P}(\mathcal{V}) \rightarrow Abs$
  - Maps a value in the program to the “best” abstract value
- **Concretization Function:** $\gamma : Abs \rightarrow \mathcal{P}(\mathcal{V})$
  - Maps an abstract value to a set of values in the program

Example:

- Parity Lattice

Galois Connections

Defines the relationship between $\mathcal{P}(\mathcal{V})$ and Abs

- In general define relationship between two complete lattices

Galois Connection: A pair of functions

(Abstraction) $\alpha \colon \mathcal{P}(\mathcal{V}) \rightarrow Abs$

and

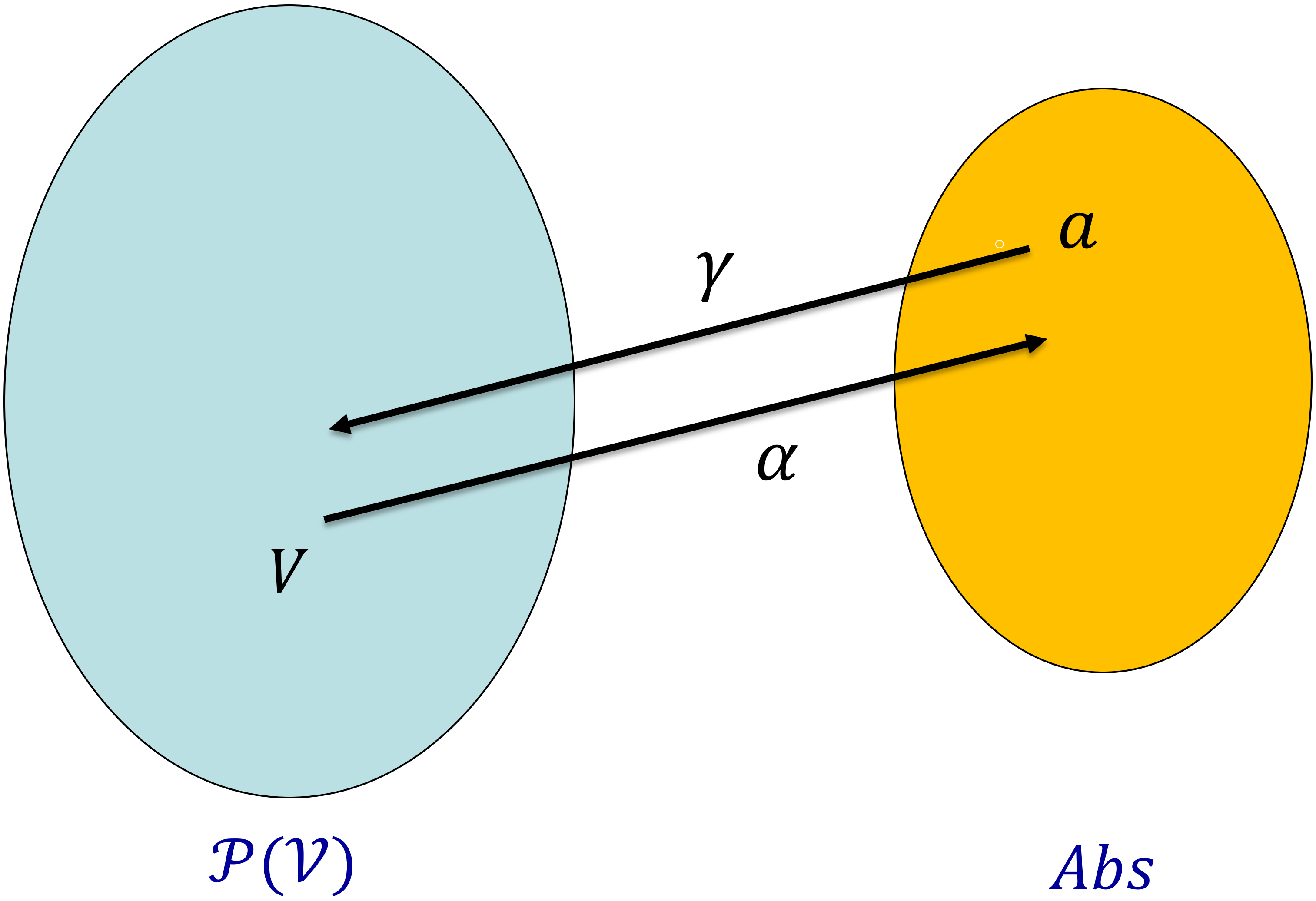
(Concretization) $\gamma \colon Abs \rightarrow \mathcal{P}(\mathcal{V})$

such that

$\forall a \in Abs, \forall V \in P(v).$

$$V \subseteq \gamma(a) \Leftrightarrow \alpha(V) \subseteq a$$

Galois Connections



Galois Connections: Properties

Both abstraction and concretization functions are monotonic.

$$V \subseteq V' \implies \alpha(V) \subseteq \alpha(V')$$

$$a \subseteq a' \implies \gamma(a) \subseteq \gamma(a')$$

Lemma:

$$\alpha(\gamma(a)) \subseteq a$$

Correctness Conditions

What is the relationship between

$$\gamma(a1\ op\ a2) \qquad \supseteq \qquad \gamma(a1) op\ \gamma(a2)$$

Abstraction Function:

- $\alpha\colon \mathcal{P}(\mathcal{V}) \to Abs, \alpha(S) = \sqcup_{s \in S} \beta(s)$

We can define

- $(a1\ op\ a2) = \alpha(\gamma(a1) op\ \gamma(a2))$

Abstract Domains: Examples

- Constant domain
- Sign domain
- Interval domain

Abstract Interpretation

Simple recipe for arguing correctness of an analysis

- Define an abstract domain *Abs*
- Define α and γ and show they form a Gallois Connection
- Define the semantics of program constructs for the abstract domain and show that they are correct

Some useful domains

Ranges

- Useful for detecting out-of-bounds errors, potential overflows

Linear relationships between variables

- $a_1x_1 + a_2x_2 + \cdots + a_kx_k \geq c$

Problem: Both of these domains have infinite chains!

Widening

Key idea:

- You have been running your analysis for a while
- A value keeps getting "bigger" and "bigger" but refuses to converge
- Just declare it to be $\top$ (or some other big value)

This loses precision

- but it's always sound

Widening operator: $\nabla : Abs \times Abs \rightarrow Abs$

- $a1 \nabla a2 \sqsupseteq a1, a2$

6.820 Fundamentals of Program Analysis
Fall 2015