Dataflow Analysis and Abstract Interpretation

Computer Science and Artificial Intelligence Laboratory
MIT

Nov 15, 2011
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

- \( \text{in}_n \) – value at program point before \( n \)
- \( \text{out}_n \) – value at program point after \( n \)
- \( f_n \) – transfer function for \( n \) (given \( \text{in}_n \), computes \( \text{out}_n \))

Require that solution satisfy

- \( \forall n. \, \text{out}_n = f_n(\text{in}_n) \)
- \( \forall n \neq n_0. \, \text{in}_n = \lor \{ \text{out}_m \, . \, m \in \text{pred}(n) \} \)
- \( \text{in}_{n_0} = I \)
- Where \( I \) summarizes information at start of program
Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

\[ \text{out}_n := f_n(\text{in}_n) \]
\[ \text{in}_n := \lor \{ \text{out}_m . m \text{ in pred}(n) \} \]

Conceptually separates analysis problem from program
for each n do outₙ := fₙ(⊥)
inₙ₀ := I; outₙ₀ := fₙ₀(I)
worklist := N - { n₀ }
while worklist ≠ ∅ do
    remove a node n from worklist
    inₙ := ∨ { outₘ | m in pred(n) }
    outₙ := fₙ(inₙ)
    if outₙ changed then
        worklist := worklist ∪ succ(n)
Correctness Argument

Why result satisfies dataflow equations?
Whenever process a node $n$, set $\text{out}_n := f_n(\text{in}_n)$
- Algorithm ensures that $\text{out}_n = f_n(\text{in}_n)$
Whenever $\text{out}_m$ changes, put $\text{succ}(m)$ on worklist.
- Consider any node $n \in \text{succ}(m)$. It will eventually come off worklist and algorithm will set
  $$\text{in}_n := \lor \{ \text{out}_m . m \in \text{pred}(n) \}$$
  to ensure that $\text{in}_n = \lor \{ \text{out}_m . m \in \text{pred}(n) \}$
So final solution will satisfy dataflow equations
Termination Argument

Why does algorithm terminate?
Sequence of values taken on by $\text{in}_n$ or $\text{out}_n$ 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
Available Expressions

\[ P = \text{powerset of set of all expressions in program (all subsets of set of expressions)} \]
\[ \lor = \cap \text{ (order is } \supseteq) \]
\[ \bot = P \]
\[ I = \text{in}_{n0} = \emptyset \]
\[ F = \text{all functions f of the form } f(x) = a \cup (x-b) \]
- \( b \) is set of expressions that node kills
- \( a \) is set of expressions that node generates

Another GEN/KILL analysis
Concept of Conservatism

Reaching definitions use $\cup$ as join
- Optimizations must take into account all definitions that reach along ANY path

Available expressions use $\cap$ as join
- Optimization requires expression to reach along ALL paths

Optimizations must conservatively take all possible executions into account. Structure of analysis varies according to way analysis used.
Backward Dataflow Analysis

Simulates execution of program backward against the flow of control

For each node \( n \), have

- \( \text{in}_n \) – value at program point before \( n \)
- \( \text{out}_n \) – value at program point after \( n \)
- \( f_n \) – transfer function for \( n \) (given \( \text{out}_n \), computes \( \text{in}_n \))

Require that solution satisfies

- \( \forall n. \text{in}_n = f_n(\text{out}_n) \)
- \( \forall n \notin N_{\text{final}}. \text{out}_n = \lor \{ \text{in}_m. m \in \text{succ}(n) \} \)
- \( \forall n \in N_{\text{final}} = \text{out}_n = O \)
- Where \( O \) summarizes information at end of program
Worklist Algorithm for Solving Backward Dataflow Equations

for each n do
    \( \text{in}_n := f_n(\bot) \)
for each \( n \in N_{\text{final}} \) do
    \( \text{out}_n := 0; \text{in}_n := f_n(0) \)
worklist := \( N - N_{\text{final}} \)
while worklist \( \neq \emptyset \) do
    remove a node n from worklist
    \( \text{out}_n := \lor \{ \text{in}_m \cdot m \text{ in succ}(n) \} \)
    \( \text{in}_n := f_n(\text{out}_n) \)
    if \( \text{in}_n \) changed then
        worklist := worklist \( \cup \) pred(n)
Live Variables

\[ P = \text{powerset of set of all variables in program (all subsets of set of variables in program)} \]
\[ \lor = \cup \text{ (order is } \subseteq \text{)} \]
\[ \bot = \emptyset \]
\[ O = \emptyset \]
\[ F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b) \]
- \( b \) is set of variables that node kills
- \( a \) is set of variables that node reads
Meaning of Dataflow Results

Concept of program state $s$ for control-flow graphs

- Program point $n$ where execution located ($n$ is node that will execute next)
- Values of variables in program

Each execution generates a trajectory of states:

- $s_0; s_1; \ldots; s_k$, where each $s_i \in ST$
- $s_{i+1}$ generated from $s_i$ by executing basic block to
  - Update variable values
  - Obtain new program point $n$
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

- $C: \text{Labels} \rightarrow \mathcal{P}(\Sigma)$
- 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 states that can reach a given point.
Defining the Collecting Semantics

\[ C[L_2] = \{ \sigma[x \to n] | \sigma \in C[L_1], \llbracket e \rrbracket \sigma = n \} \]

\[ C[Lt] = \{ \sigma | \sigma \in C[L:] , \llbracket e \rrbracket \sigma = true \} \]

\[ C[Lf] = \{ \sigma | \sigma \in C[L-] , \llbracket e \rrbracket \sigma = false \} \]

\[ C[L \ ] = C[L1] \cup C[L2] \]
Example

L0
x = _input();
x = x * 2;
y = 0;

L1
while(x < 16){
    x = x - y;
    y = 2 + x;
}

L2
X<16

L3
x = x - y;
y = 2 + x;

L4
end
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 $\mathcal{A}[Li] \supset C[Li]$
Abstract Domain

An abstract domain is a lattice
- Although 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**: $\beta: \mathcal{V} \rightarrow \text{Abs}$
  - Maps a value in the program to the “best” abstract value
- **Concretization Function**: $\gamma: \text{Abs} \rightarrow \mathcal{P}(\mathcal{V})$
  - Maps an abstract value to a set of values in the program

Example:
- Parity Lattice
Correctness Conditions

What is the relationship between
\[ \gamma(a_1 \text{ op } a) \subseteq \gamma(a_1) \text{ op } \gamma(a_2) \]

Abstraction Function:
- \( \alpha: \mathcal{P}(\mathcal{V}) \rightarrow \text{Abs} \)
- \( \alpha(S) = \sqcup_{s \in S} \beta(s) \)

We can define
- \( (a_1 \text{ op } a_2) = \alpha(\gamma(a_1) \text{ op } \gamma(a_2)) \)
Gallois Connections

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

- In general define relationship between two complete lattices

Gallois Connection: A pair of functions $\alpha: \mathcal{P}(\mathcal{V}) \rightarrow \text{Abs}$ and $\gamma: \text{Abs} \rightarrow \mathcal{P}(\mathcal{V})$ such that

- Both are monotonic
- $\alpha(\gamma(a)) = a$
- $\gamma(\alpha(V)) \supseteq V$ for $V \in \mathcal{P}(\mathcal{V})$
Abstract Interpretation

Simple recipe for arguing correctness of an analysis
- Define an abstract domain
- Define $\alpha$ and $\gamma$ and show they form a Gallois Connection
- Define the semantics of program constructs for the abstract domain and show that they are correct:
  $$(\alpha \ op \ a) = \alpha(\gamma(a^1) \ op \ \gamma(a^2))$$

Sometimes you may want to abstract the whole state rather than each value independently
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: \( \triangledown : \text{Abs} \times \text{Abs} \rightarrow \text{Abs} \)
- \( a \triangledown a2 \equiv a , a \)