IFDS, Conclusion.
Abstract Interpretation

Announcements

- HW3: XTA Analysis
  - Due Friday February 23rd
  - Submitty for HW3 open
- Quiz 3

Outline
- IFDS and CFL-reachability, conclusion
- Abstract interpretation
- Semantics
- Notion of abstraction

IFDS Context Sensitivity
- Interprocedural, Finite, Distributive, Subset (IFDS) problems
- Allows for efficient computation of summary transfer functions. Converts problem into Context-Free-Language (CFL)-Reachability
  - Can reduce monotone problem into the IFDS problem, but with loss of precisions
- Reading: Thomas Reps, Susan Horwitz and Mooly Sagiv, “Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL’95

Efficient Encoding of Transfer Functions
- Finite set of dataflow facts \( D \)
  - E.g., all variables \( \{x, y, z\} \)
- Transfer functions \( f: 2^D \rightarrow 2^D \)
  
Edge \( \Lambda \rightarrow d \) means \( d \in f(\Lambda) \)
  - I.e., \( d \) is generated
  
Edge \( d1 \rightarrow d2 \) means \( d2 \notin f(\Lambda) \) and \( d2 \in f(S) \) if \( d1 \in S \)
  - I.e., \( d1 \) in \( S \) leads to \( d2 \) in \( f(S) \)
  
Edge \( \Lambda \rightarrow \Lambda \) always there

What Can Be Encoded. Taint Analysis

1. \( z = 5 \)
2. \( y = \text{“tainted” value} \)
3. \( x = y + z \)

The paths from top \( \Lambda \) to \( x \) and to \( y \) entail that \( x \) and \( y \) are tainted at exit from 3.
Efficient Computation of Function Composition!

What Can Be Encoded.
All Bit-Vector Problems!
1. $x = a \cdot b$
2. $a = a - 1$

- Add edges from $\Lambda$ to facts being generated (e.g., $a \cdot b$)
- Add in-out edges to facts being preserved (e.g., $a - 1$)

What Cannot Be Encoded
- Monotone functions cannot be encoded
  - E.g., constant propagation, points-to analysis
- Points-to analysis, distributive subset?
  - $f_{p \cdot q}: p \rightarrow x$ in $f_{p \cdot q}(S)$ if $q \rightarrow y$ in $S$ AND $y \rightarrow x$ in $S$
  - Can encode disjunctions but not conjunctions
- Large class of problems falls under IFDS
- Monotone problems can be reduced into IFDS with loss of precision

Big Picture, Why Does It Matter
- We can compose transfer functions within a procedure $p$ and compute the summary transfer function $\Phi_p$!
- Precisely: Computes the MORP solution!
- Efficiently: $O(ED^3)$
  - $E$ is the number of intraprocedural edges across all procedures in ICFG

Exploded Supergraph $G^#$
- Let $G^*$ be the ICFG, which Reps et al. call the supergraph
- First, define the nodes of $G^#$
- For each node $j \in G^*$ there is node $< j, \Lambda > \in G^#$
- For each node $j \in G^*$ and $d \in D$ there is node $< j, d > \in G^#$
- $\Lambda$: Represents the $\text{in}(j)$

Exploded Supergraph $G^#$
- Next, add edges to $G^#$
- For each $k$ in successors of $j$
  - Add edge $< j, \Lambda > \rightarrow < k, \Lambda >$ to $G^#$
  - Add edge $< j, \Lambda > \rightarrow < k, d >$ if $d \in f_j(\emptyset)$
  - Add edge $< d_1, j > \rightarrow < d_2, k >$ if $d_2 \notin f_j(\emptyset)$ and $d_2 \in f_j(in(j))$ if $d_1 \in in(j)$
- $\text{in}(j)$: Represent (encode) transfer function $f_j$
### Exploded Supergraph

- One can think about IFDS in terms of Sharir and Pnueli’s functional approach
- … or in terms of graph reachability: IFDS reduces the standard dataflow problem to a reachability problem in G#
  - Path from $<1, \Lambda>$ to $<j, d>$ means that $d$ reaches $j$
  - More precisely, it is a CFL-reachability (Context-Free-Language reachability) problem: “Is there a path from $<1, \Lambda>$ to $<j, d>$ whose edges form a string in the language of realizable paths?”
  - Gives rise to on-demand approaches

### IFDS Conclusion

- Key idea is encoding of transfer functions $f_j$
  - Allows for efficient computation of summary transfer functions $\Phi_p$
  - Reduces to CFL-reachability problem on G#
- IFDS is defined for forward may-problems. Forward must-problems can be expressed as complement
- Real-world analysis problems
  - Soot has a built-in IFDS framework
  - Some taint analyses for Android use IFDS

### Outline

- IFDS and CFL-reachability, conclusion
- Abstract interpretation
  - Semantics
  - Notion of abstraction

### Abstract Interpretation

- Patrick Cousot and Radhia Cousot, POPL’77
  - A general framework
    - Building static analyses
    - Reasoning about correctness of static analysis
    - Comparing static analyses
    - Combines ideas from dataflow analysis (monotone frameworks) and formal verification (semantics)
Reading

- “Principles of Program Analysis” by Nielsen, Nielsen and Hankin, Chapter 3
  - Alex Salcianu’s friendlier account of Chapter 3: https://web.eecs.umich.edu/~bchandra/courses/papers/Salcianu_AbstractInterpretation.pdf
- Lecture notes by Xavier Rival, ENS

Overview

Program Execution: Points-to Analysis (PTA):

\[ x \rightarrow h_i: A \]

\[ x = y.n(z) \]

passes value of \( z \) to parameter \( p \)

\[ x \rightarrow o_i: A \]

\[ x = y.n(z) \]

\[ \text{pts}(z) \subseteq \text{pts}(p) \]

Points-to analysis is an abstraction.
Abstrats infinitely many heap objects \( h_i \) created at site \( i \) into a single \( o_i \).

PTA:

\[ x = y.n(z) \]

\[ \text{pts}(z) \subseteq \text{pts}(p) \]

XTA:

\[ x = y.n(z) \]

\[ \text{SubTypes} \left( \text{StaticType}(p) \right) \]

\[ S_n \cap S_m \]

RTA is an abstraction of XTA.
Abstracts \( S_m \) into \( I \), the set of instantiated types across _all_ of the program.

XTA is an abstraction of PTA.
Abstracts \( o_i: A \) into just the type \( A \).
Abstracts variable \( x \) into \( S_m \) where \( m \) is the enclosing method of \( x \).

Small-step Operational Semantics

- Also called _trace semantics_, or _concrete semantics_, models program execution
- _Memory state_ maps variables \( (V) \) to values \( (Z) \):
  \[ \sigma : V \rightarrow Z \]
- _Control state_ describes where we are
  - label \( \ell \) (note: we used the term program point)
  - Describes transition \( (\ell_1, \sigma_1) \rightarrow (\ell_2, \sigma_2) \)
  (read: program executes statement at label \( \ell_1 \) on current state \( \sigma_1 \) transitioning to label \( \ell_2 \) in state \( \sigma_2 \))
A Simple Imperative Language: Syntax (We’ve Seen This Before!)

- **E ::= x | n**  
  simple expression

- **S ::= x = E | x = E Op E**  
  assignment

- **while ( b ) Seq**  
  loop

- **if ( b ) Seq else Seq**  
  conditional

- **Seq ::= { S; ... S; }**  
  sequence

- **V** is the set of program variables, **x ∈ V**
- **Z** is the set of values variables take, **n ∈ Z**

Operational semantics of expressions:

- **|[n]|(σ) = n**; if constant **n** evaluates to **n**
- **|x|(σ) = σ(x)**; if variable **x** evaluates to the value **n** maps to in **σ**

Assignment:

- **l_j**: x = E; l_i: ...

  For a label **l_j**, if **b** evaluates to True, then the state transitions to **l_i**; otherwise, it transitions to **l_k**.

Loop:

- **l_j**: while ( b ) { l_i; ...

Conditional:

- **l_j**: if ( b ) { l_T; ... } else { l_F; ...

Sequence:

- **{ l_0; l_1; ... }**  
  Transition defined by individual transition relations

Operational semantics of expressions:

- **|n|(σ) = n**; if constant **n** evaluates to **n**
- **|x|(σ) = σ(x)**; if variable **x** evaluates to the value **n** maps to in **σ**

Assignment:

- **l_j**: x = E; l_i: ...

  For a label **l_j**, if **b** evaluates to True, then the state transitions to **l_i**; otherwise, it transitions to **l_k**.

Example

- **σ**: empty map

  1. **b = 0;**
  2. if (b≥0)
  3. **x=1**
  4. **y=2**
  5. **x=2**
  6. **y=1**
  7. **x+y**
  8. **x=10*z**

Collecting Semantics

- **C**: Labels → 2^Σ
  - Collects all states a program can have at a given label (i.e., program point)
  - E.g., **x** cannot ever be null at **j**
  - **n** is always greater than 100 at **j**

Given a label, we are interested in a function

- **C**: Labels → 2^Σ
  - The set of all states a program can have at **l_j**

We’ve Seen This Before…

- **Operational semantics**
  - Defines “concrete transfer functions”
  - Works on values drawn from **Z** (concrete state)

- **A static analysis, e.g., Constant Propagation**
  - Defines “transfer functions”
  - Work on values from the flat lattice (abstract state)
Collecting Semantics

- "Ground truth"
- We base reasoning about correctness (soundness) of static analysis off of it
- Undecidable
- Relation to MOP/MORP solution?
- Define abstraction of state and semantics
- Goal: show that abstraction "properly represents" all values computed by collecting semantics

Outline

- IFDS and CFL-reachability, conclusion
- Abstract interpretation
- Semantics
- Notion of abstraction

Abstraction Example 1: signs

Concrete values: sets of integers
- Abstract values: signs

Lattice of signs:

- $\bot$ represents the empty set
- $+$ represents any set of positive integers
- $0$ represents set of non-negative integers
- $-$ represents any set of negative integers
- $T$ represents any set of integers

Abstraction relation relates concrete elements to abstract ones:
- $c \rightarrow_{a} a$ when $a$ represents $c$

Concrete space: A lattice
- Abstract space: A lattice

Abstraction Example 1: signs
Abstraction Example 1: signs

- We use the abstraction relation to define abstract semantics, i.e., the execution of program statements over abstract elements.

- If \( x_1 \) is \( + \) and \( x_2 \) is \( + \) then \( x_1 + x_2 \) is \( + \)

(Since \( + \) represents only sets of positive integers, then \( x_1, x_2 \) are positive integers, and therefore \( x_1 + x_2 \) is a positive integer.)

Abstraction Example 1: signs

- If \( x_1 \) is \( + \) and \( x_2 \) is \( + \) then \( x_1 + x_2 \) is \( + \)

Analysis computes over abstract elements

Correctness conclusion, informally: if analysis (works on abstract elements \( a \)) determines that at label \( l \) \( x \) is \( a \), then \( a \) represents the set of concrete values \( c \) collected by the collecting semantics for \( x \) at \( l \)

Abstraction Example 1: signs

- We can also use \( U \) and \( \cap \)

if \( x_1 \) is \( + \) and \( x_2 \) is \( + \) then \( x_1 \cup x_2 \) is \( + \)

How about if \( x_1 \) is \( + \) and \( x_2 \) is \( 0 \)?

then \( x_1 \cup x_2 \) is \( T \)

because only \( \{0,1,2,3,\ldots\} \) \( \cap \) \( T \) holds

No other relation holds

In the abstract, we include negative integers in \( x_1 \cup x_2 \) (we lose precision!)

Abstraction Example 2: constants

- Concrete elements: elements of concrete lattice, \( c \in 2^\mathbb{Z} \)

- Abstract elements: \( \bot, T, n \), where \( n \in \mathbb{Z} \)
  - Flat lattice:

- Abstraction relation:
  - \( c \models n \) if and only if \( c = \{ n \} \)
  - \( \bot \) represents the empty set
  - \( T \) represents any set of integers

Abstraction Example 2: constants

- Abstract semantics, works on abstract elements (the elements of the flat lattice)

  - If \( x_1 \) is \( n_1 \) and \( x_2 \) is \( n_2 \) then \( x_1 + x_2 \) is \( n_1 + n_2 \)

(Since \( n_1 \) represents exactly integer \( n_1 \), and \( n_2 \) represents \( n_2 \), then \( x_1 + x_2 \) is \( n_1 + n_2 \) which represents \( n_1 + n_2 \).)

  - If \( x_1 \) is \( n_1 \) and \( x_2 \) is \( T \), then \( x_1 + x_2 \) is \( T \)

(Since \( T \) represents any set, we cannot do better but abstract \( x_1 + x_2 \) by \( T \)
Abstraction Example 3: intervals

Concrete elements: elements of \(2^\mathbb{Z}\)

Abstract elements: \(\bot, \top, \text{intervals } [a,b] \) where \(a \in \mathbb{Z}\cup\{-\infty\}\) and \(b \in \mathbb{Z}\cup\{\infty\}\) and \(a \leq b\)

Is it a lattice?
Yes!

Abstraction relation:
\(S \vdash T\)
\(S \vdash [a,b]\) iff for every \(n \in S\), \(a \leq n \leq b\)

Spring 18 CSCI 4450/6450, A Milanova

Abstraction Example 4: heap

We need to expand our syntax and operational semantics to model the heap

Realistic analyses of imperative languages do need to model heap

“Giant array” view (C)

“Collection of objects” view (Java):

\[s \in S : \text{Id} \rightarrow \text{Addr} \quad /\!\!/ \text{Stack}\]
\[h \in H : \text{Addr} \times \text{Id} \rightarrow \text{Addr} \quad /\!\!/ \text{Heap}\]

\(h(j,f_1) \leftarrow [x_1](s), \ldots h(j,f_k) \leftarrow [x_k](s)\)

\((i)\) is the unique static allocation site identifier;

\(j\) is the next available address in \(h\)

(read: allocate new object at address \(j\) on heap \(h\), then initialize values of fields \(f_1,\ldots,f_k\) of \(j\) with respective addresses resulting from evaluation of \(x_1,\ldots,x_k\) in \(s\))

Abstraction Example 4: heap

Concrete elements:
All heap objects \(h_i\)

Abstract elements:
Abstract heap objects \(o_i\)

Abstraction relation:
Objects \(h_i\) allocated at static allocation site \(i\), are represented by \(o_i\)

Abstract semantics:
If \(x\) points to \(o_i\), after \(y = x\), \(y\) points to \(o_i\)

Spring 18 CSCI 4450/6450, A Milanova

Outline

IFDS and CFL-reachability, conclusion

Abstract interpretation
Semantics
Notion of abstraction
Concretization and abstraction functions

Spring 18 CSCI 4450/6450, A Milanova