

Course Overview

CS357 Lecture 1

Decision Procedures

- This course is about decision procedures
- Also about how decision procedures are used
 - In static analysis

Prerequisites

- Undergraduate CS theory
 - Particularly basic logic
- Knowledge of compilers helpful

Course Administration

- Workload
 - Homeworks
 - Project
- There is a newsgroup
 - su.class.1112-cs-357

Definitions

- Program analysis
 - Discovering facts about programs.*
- Static analysis
 - Program analysis without running the program.*

History

- Static analysis is nearly as old as programming.
- First used in compilers
 - For program optimization
 - Starting with FORTRAN (1954)



So What?

- It's a big field
 - With different approaches
 - And applications
 - And lots of terms
- This lecture aims to sketch basic
 - concepts
 - techniques
 - terminology

Laundry List Outline

- Analysis paradigms
 - Type systems
 - Dataflow analysis
 - Model checking
- Terminology
 - Abstract values
 - Flow insensitive
 - Flow sensitive
 - Path sensitive
 - Local vs. global analysis
 - Verification vs. bug-finding
 - Soundness
 - False positives and false negatives
 - ...

Type Systems

A Notation for Describing Static Analyses

Type Systems

- Types are the most widely used static analysis
- Part of nearly all mainstream languages
 - Widely recognized as important for quality
- Type notation is useful for discussing concepts
 - We use type notation to discuss type checking, dataflow analysis, and model checking

What is a Type?

- Consensus
 - A set of values
- Examples
 - `Int` is the set of all integers
 - `Float` is the set of all floats
 - `Bool` is the set `{true, false}`

More Examples

- `List(Int)` is the set of all lists of integers
 - `List` is a *type constructor*
 - A function from types to types
- `Foo`, in Java, is the set of all objects of class `Foo`
- `Int → Int` is the set of functions mapping an integer to an integer
 - E.g., increment, decrement, and many others



What is a Type?

- Consensus
 - A set of values
- A type is an example of an *abstract* value
 - Represents a set of *concrete* values
 - Every static analysis has abstract values
- In type systems,
 - Every concrete value is an element of some abstract value
 - i.e., every concrete value has a type

Abstraction

- All static analyses use abstraction
 - Represent sets of values as abstract values
- Why?
 - Because we can't reason directly about the infinite set of possible concrete values
 - For performance (even just termination), must make such approximations
- In type systems, the approximations are called types

The Next Step

- Now we need to compute with types
 - Actually analyze programs
- Type systems have a well-developed notation for expressing such computations

Inference Rules

- By tradition inference rules are written
$$\frac{\vdash \text{Hypothesis}_1 \dots \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$
- Type rules have hypotheses and conclusions
$$\vdash e : T$$
- \vdash means "it is provable that ..."

Two Rules

$$\frac{i \text{ is an integer}}{\vdash i : \text{Int}} \quad [\text{Int}]$$

$$\frac{\begin{array}{l} \vdash e_1 : \text{Int} \\ \vdash e_2 : \text{Int} \end{array}}{\vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions
- Note that
 - Hypotheses prove facts about subexpressions
 - Conclusions prove facts about the entire expression



Example: 1 + 2

$$\frac{\frac{1 \text{ is an integer}}{\vdash 1: \text{Int}} \quad \frac{2 \text{ is an integer}}{\vdash 2: \text{Int}}}{\vdash 1 + 2: \text{Int}}$$

A Problem

- What is the type of a variable reference?

$$\frac{x \text{ is a variable}}{\vdash x: ?} \quad [\text{Var}]$$

- The rule does not carry enough information to give x a type.

A Solution

- Do what logicians do
 - Put more information in the rules
- An *environment* gives types for *free* variables
 - An environment is a function from variables to types
 - For other static analyses the environment may map variables to other abstract values
 - A variable is free in an expression if it is not defined within the expression

Type Environments

Let A be a function from *Variables* to *Types*

The sentence $A \vdash e : T$ is read:

Under the assumption that variables have the types given by A , it is provable that the expression e has the type T

Modified Rules

The type environment is added to all rules:

$$\frac{A \vdash e_1 : \text{Int} \quad A \vdash e_2 : \text{Int}}{A \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

New Rules

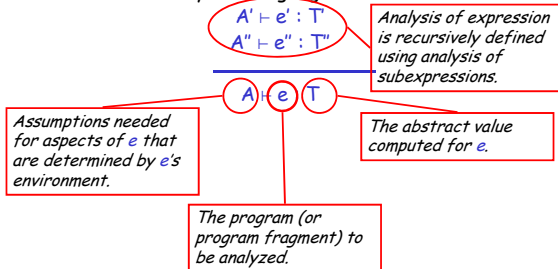
And we can write new rules:

$$\frac{A(x) = T}{A \vdash x : T} \quad [\text{Var}]$$



Summary

Describe static analyses using logics of the form:



An Example

The Rule of Signs

- We want to estimate a computation's sign
- Example: $-3 * 4 = -12$
- Abstraction: $- * + = -$

Abstract Values

- $+$ = {set of positive integers}
- 0 = { 0 }
- $-$ = {set of negative integers}
- Environment: Variables \rightarrow {+, 0, - }

Example Rules

$$\frac{A \vdash e_0 : + \quad A \vdash e_1 : -}{A \vdash e_0 * e_1 : -}$$

$$\frac{A \vdash e_0 : + \quad A \vdash e_1 : +}{A \vdash e_0 * e_1 : +}$$

$$\frac{A \vdash e_0 : 0 \quad A \vdash e_1 : x}{A \vdash e_0 * e_1 : 0}$$

$$\frac{A \vdash e_0 : x \quad A \vdash e_1 : 0}{A \vdash e_0 * e_1 : 0}$$

A Problem

$$\frac{A \vdash e_0 : + \quad A \vdash e_1 : -}{A \vdash e_0 + e_1 : ?}$$

Solution:

Add abstract values to ensure analysis is closed under all operations:

We don't have an abstract value that covers this case!

- $+$ = { positive integers }
- 0 = { 0 }
- $-$ = { negative integers }
- τ = { all integers }
- \perp = { }



More Example Rules

$$\frac{A \vdash e_0 : + \quad A \vdash e_1 : -}{A \vdash e_0 + e_1 : \top}$$

$$\frac{A \vdash e_0 : + \quad A \vdash e_1 : +}{A \vdash e_0 + e_1 : +}$$

$$\frac{A \vdash e_0 : 0 \quad A \vdash e_1 : +}{A \vdash e_0 / e_1 : 0}$$

$$\frac{A \vdash e_0 : \top \quad A \vdash e_1 : 0}{A \vdash e_0 / e_1 : \perp}$$

Back To The Main Story . . .

A More Complex Rule

$$\frac{A \vdash e_0 : \text{Bool} \quad A \vdash e_1 : T_1 \quad A \vdash e_2 : T_2 \quad T_1 = T_2}{A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1} \text{ [If-Then-Else]}$$

We'll use this rule to illustrate several ideas . . .

Soundness

$$\frac{A \vdash e_0 : \text{Bool} \quad A \vdash e_1 : T_1 \quad A \vdash e_2 : T_2 \quad T_1 = T_2}{A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1} \text{ [If-Then-Else]}$$

(Note: In the original image, $A \vdash e_0 : \text{Bool}$ is circled in red and annotated with "e₀ is guaranteed to be a Bool". The entire rule is circled in purple and annotated with "e₁ and e₂ are guaranteed to be of the same type".)

- An analysis is *sound* if
 - Whenever $A \vdash e : T$
 - And if $A(x) = T$ then x has a value $v \in T$
 - Then e evaluates to a value $v \in T$

Comments on Soundness

- Sound analyses are *extremely* useful
 - If a program has no errors according to a sound system, then no errors of that kind can arise at runtime
 - We have *verified* the absence of a class of errors
- Verification is a very strong guarantee
 - The property verified always holds

Comments on Soundness

- But soundness has a price in many applications
 - Spurious errors/warnings due to abstraction
 - These are *false positives*
- Alternative is to use *unsound* analyses
 - Allows some control of false positives
 - Introduces possibility of *false negatives*
 - Undetected errors
- Type systems are sound
 - But most analyses used for detecting bugs are not sound
 - These are called *bug finding* analyses



Constraints

$A \vdash e_0 : \text{Bool}$

$A \vdash e_1 : T_1$

$A \vdash e_2 : T_2$

$T_1 = T_2$ Side constraints must be solved [If-Then-Else]

$A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$

• Many analyses have side conditions

- Often constraints to be solved
- All constraints must be satisfied
- A separate algorithmic problem

Another Example

• Consider a recursive function

$f(x) = \dots f(e) \dots$

- If $x: T_1$ and $e: T_2$ then $T_2 = T_1$
 - Can be relaxed to $T_2 \subseteq T_1$
- Recursive functions yield recursive constraints
 - Same with loops
 - How hard constraints are to solve depends on constraint language, details of application

Algorithm

$A \vdash e_0 : \text{Bool}$ ²

$A \vdash e_1 : T_1$ ³

$A \vdash e_2 : T_2$ ³

$T_1 = T_2$ ⁴

Algorithm: ¹ $A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$ ⁵ [If-Then-Else]

1. Input: If-expression and A
2. Analyze e_0 , check it is of type Bool
3. Analyze e_1 and e_2 , giving types T_1 and T_2
4. Solve $T_1 = T_2$
5. Return T_1

Global Analysis

$A \vdash e_0 : \text{Bool}$ ²

$A \vdash e_1 : T_1$ ³

$A \vdash e_2 : T_2$ ³

$T_1 = T_2$ ⁴

¹ $A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$ ⁵ [If-Then-Else]

The first step requires the overall environment A

- Only then can we analyze the subexpressions

This is *global* analysis

- Requires the entire program
- Or we must somehow construct a model of the environment

Local Analysis

Algorithm:

$A_1 \vdash e_0 : \text{Bool}$ ¹

$A_2 \vdash e_1 : T_1$ ²

$A_3 \vdash e_2 : T_2$ ²

$T_1 = T_2$ ³

$A_1 = A_2 = A_3$

⁴ $A_1 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$ ⁴ [If-Then-Else]

1. Analyze e_0 , inferring environment A_1
check type is Bool
2. Analyze e_1 and e_2 , giving types T_1 and T_2
and environments A_2 and A_3
3. Solve $T_1 = T_2$ and $A_1 = A_2 = A_3$
4. Return T_1 and A_1

Local Analysis

$A_1 \vdash e_0 : \text{Bool}$ ¹

$A_2 \vdash e_1 : T_1$ ²

$A_3 \vdash e_2 : T_2$ ²

$T_1 = T_2$ ³

$A_1 = A_2 = A_3$

⁴ $A_1 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$ ⁴ [If-Then-Else]

In this approach, we first analyze the subexpressions, inferring the needed environment from the subexpression itself. Because the separately computed environments might not agree, they need to be constrained to be equal to get a valid analysis for the entire expression.

Local vs. Global Analysis

- Global analysis
 - Is usually simpler than local analysis
 - But may require a lot of extra engineering to construct models of the environment for partial programs
- Local analysis
 - Allows analysis of, e.g., a library without the client
 - Technically more difficult
 - Requires allowing unknown parameters in environments, which can be solved for later

Flow Insensitivity

$A \vdash e_0 : \text{Bool}$
 $A \vdash e_1 : T_1$
 $A \vdash e_2 : T_2$

Subexpressions are independent of each other

$T_1 = T_2$

[If-Then-Else]

$A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$

- No ordering required in the analysis of subexpressions, then analysis is *flow insensitive*
- Implies statements can be permuted and analysis is unaffected
- Type systems are generally flow-insensitive

Comments on Flow Insensitivity

- Flow insensitive analyses are often very efficient and scalable
- No need for modeling a separate state for each subexpression

Flow Insensitivity (Again)

$A \vdash e_0 : \text{Bool}$
 $A \vdash e_1 : T_1$
 $A \vdash e_2 : T_2$

Subexpressions are independent of each other

$T_1 = T_2$

[If-Then-Else]

$A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1$

Flow Sensitivity

$A_0 \vdash e_0 : \text{Bool}, A_1$
 $A_1 \vdash e_1 : T_1, A_2$
 $A_1 \vdash e_2 : T_2, A_3$

Rules produce new environments and analysis of a subexpression cannot take place until its environment is available.

$T_1 = T_2 \quad A_2 = A_3$

[If-Then-Else]

$A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1, A_2$

- Analysis of subexpressions is ordered by environments: *flow sensitive*
- *Dataflow analysis* is flow sensitive analysis
- The order of statements matters

Comments on Flow Sensitivity

- Example
 - Rule of signs extended with assignment statements

$$\frac{A \vdash e : +, A}{A \vdash x := e, A[x \leftarrow +]}$$

- $A[x \leftarrow +]$ means A modified so that $A(x) = +$

- Flow sensitive analysis can be expensive
 - Each statement has its own model of the state
 - Polynomial increase in cost over flow-insensitive



Path Sensitivity

$$\begin{array}{l}
 P, A_0 \vdash e_0 : \text{Bool}, A_1 \\
 P \wedge e_0, A_1 \vdash e_1 : T_1, A_2 \\
 P \wedge \neg e_0, A_1 \vdash e_2 : T_2, A_3 \\
 \hline
 T_1 = T_2 \quad \text{[If-Then-Else]} \\
 \hline
 P, A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1, e_0 ? A_2; A_3
 \end{array}$$

Predicate is refined at decision points (e.g., if's)

Part of the environment is a predicate saying under what conditions this expression is executed.

At points where control paths merge, still keep different paths separate in the final environment

Comments on Path Sensitivity

- *Model checking* is flow- and path-sensitive analysis
 - In practice, path sensitive analyses are also flow sensitive
 - Although in theory they are independent ideas
- Path sensitivity can be very expensive
 - Exponential number of paths to track
 - But appears to be necessary in many applications
- Often implemented using backtracking instead of explicit merges of environment
 - Explore one path
 - Backtrack to a decision point, explore another path

Summary

- A very rough taxonomy
 - Type systems = flow-insensitive
 - Dataflow analysis = flow-sensitive
 - Model checking = flow- and path-sensitive
- But note
 - These lines have been blurred
 - E.g., lots of flow-sensitive type systems recently

Lambda Calculus

Lambda calculus

$$\begin{array}{l}
 e ::= \lambda x.e \\
 \quad | e e \\
 \quad | x
 \end{array}$$

Two rules for computation

$$\begin{array}{l}
 (\lambda x.e) e' \rightarrow e[e'/x] \quad \text{if } x \text{ not free in } e' \\
 \lambda x.e \rightarrow \lambda y.e[y/x]
 \end{array}$$

Simply Typed Lambda Calculus

Types: $\tau ::= \alpha \mid \tau \rightarrow \tau$

Examples:

$$\begin{array}{l}
 \lambda x.x : \alpha \rightarrow \alpha \\
 \lambda x.\lambda y.x : \alpha \rightarrow (\beta \rightarrow \alpha)
 \end{array}$$

Type System #1

$$A, x : T \vdash x : T$$

$$\frac{A, x : T \vdash e : T'}{A \vdash \lambda x.e : T \rightarrow T'}$$

$$A \vdash e : T \rightarrow T'$$

$$\frac{A \vdash e' : T}{A \vdash e e' : T'}$$

It's not clear how to use these rules. What assumptions do we make for variables? What is the algorithm?

Type System #2

$A, x: a_x \vdash x: a_x$

Note: Assume all lambda-bound variables are distinct

$$\frac{A, x: a_x \vdash e: \tau}{A \vdash \lambda x. e: a_x \rightarrow \tau}$$

This system is *syntax-directed*.

$A \vdash e: \tau \rightarrow \tau'$

$A \vdash e': \tau''$

$\tau'' = \tau$

$A \vdash e e': \tau'$

Proof is valid for any solution of the constraints = unification of type terms.
Almost linear time.

Type System #3 \subseteq

$A, x: a_x \vdash x: a_x$

Note: Assume all lambda-bound variables are distinct

$$\frac{A, x: a_x \vdash e: \tau}{A \vdash \lambda x. e: a_x \rightarrow \tau}$$

Proof is valid for any solution of the set constraints.

$A \vdash e: \tau \rightarrow \tau'$

$A \vdash e': \tau''$

$\tau'' \subseteq \tau$

$A \vdash e e': \tau'$

Includes all of the unification solutions, plus more solutions. Cubic time.

Decision Procedures

- Mentioned a few
 - Unification of terms
 - Set constraints
 - Lattice constraints
- Others we'll see
 - SAT
 - BDDs
 - Integer constraints
 - Theory of Arrays
- Can often use same analysis algorithm, different decision procedure
- Or different analysis algorithm, same decision procedure

