Specialized Static Analysis Frameworks

Material covered in chapter 10 of
Introduction to Static Analysis: an Abstract Interpretation Perspective

Purpose of this lecture

In this lecture we

- review of specialized static analysis techniques that can be simple yet powerful enough for specific cases in hand
- discuss some limitations, if any, and how these techniques can be seen from the general abstract interpretation point of view
- intended more as a survey of the specialized frameworks than an in-depth coverage

We cover three specialized frameworks:

- static analysis by equations
- static analysis by monotonic closure
- static analysis by proof construction

Use of specialized frameworks

Practical altenatives to the aforementioned general, abstract interpretation frameworks

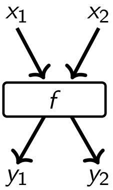
- analogous to domain-specific programming languages as opposed to general-purpose ones
- can be practical alternatives when the target languages and properties are good fits for them
  - for simple languages and properties,
  - $\exists$frameworks that are simple yet powerful enough
- the burden of soundness proof can be reduced and the special algorithms can outperform the general worklist-based fixpoint iteration algorithms

Outline

Static analysis by equations

- static analysis = equation setup and resolution
  - equations capture all the executions of the program
  - a solution of the equations is the analysis result
- represent programs by control-flow graphs
  - nodes for semantic functions (statements)
  - edges for control flow
- straightforward to set up sound equations

For each node



we set up equations

$$
\begin{aligned}
y_1 &= f(x_1 \sqcup x_2) \\
y_2 &= f(x_1 \sqcup x_2)
\end{aligned}
$$

Example: data-flow analysis for integer intervals

Example (Data-flow analysis)

Program

```
input (x);
while (x <= 99)
    x := x+1
```

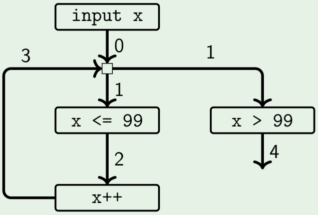


Figure: Control-flow graph

$$
\begin{aligned}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \sqcap [-\infty, 99] \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty]
\end{aligned}
$$

Figure: Equations for the program

Limitations

Not powerful enough for arbitrary languages

- control-flow before analysis?
  - control is also computed in modern languages
  - impossible: the dichotomy of control being fixed and data being dynamic
- sound transformation function?
  - error prone for complicated features of modern languages
  - e.g. function call/return, function as a value, dynamic method dispatch, exception, pointer manipulation, dynamic memory allocation, ...
- lacks a systematic approach
  - to prove the correctness of the analysis
  - to vary the accuracy of the analysis

Outline

Static analysis by monotonic closure (1/2)

- static analysis = setting up initial facts then collecting new facts by a kind of chain reaction
  - has rules for collecting initial facts
  - has rules for generating new facts from existing facts
- the initial facts immediate from the program text
- the chain reaction steps simulate the program semantics
- the universe of facts is finite for each program
- analysis accumulates facts until no more possible

Static analysis by monotonic closure (2/2)

- let R be the set of the chain-reaction rules
- let X_0 be the initial fact set
- let *Facts* be the set of all possible facts

Then, the analysis result is

$$\bigcup_{i\geq 0} Y_i, \quad \text{where} \quad Y_0 = X_0 \text{ and } Y_{i+1} = Y \text{ such that } Y_i \vdash_R Y.$$

Or, equivalently, the analysis result is the least fixpoint

$$\bigcup_{i\geq 0} \phi^i(\emptyset)$$

of monotonic function $\phi : \wp(\mathit{Facts}) \to \wp(\mathit{Facts})$:

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$

Example: pointer analysis (1/3)

$$
\begin{array}{rcll}
P & ::= & \mathtt{C} & \text{program} \\
\mathtt{C} & ::= & & \text{statement} \\
 & | & L := R & \text{assignment} \\
 & | & \mathtt{C} \; ; \; \mathtt{C} & \text{sequence} \\
 & | & \mathtt{while}\ B\ \mathtt{C} & \text{while-loop} \\
L & ::= & x \mid *x & \text{target to assign to} \\
R & ::= & n \mid x \mid *x \mid \&x & \text{value to assign} \\
B & & & \text{Boolean expression}
\end{array}
$$

- goal: estimate all "points-to" relations between variables that can occur during executions
- $\mathtt{a} \rightarrow \mathtt{b}$: variable `a` can point to (can have the address of) variable `b`

Example: pointer analysis (2/3)

The initial facts that are obvious from the program text are collected by this rule:

$$\frac{x \ := \ \&y}{x \to y}$$

The chain-reaction rules are as follows for other cases of assignments:

$$\frac{x \ := \ y \quad y \to z}{x \to z} \qquad \frac{x \ := \ *y \quad y \to z \quad z \to w}{x \to w}$$

$$\frac{*x \ := \ y \quad x \to w \quad y \to z}{w \to z} \qquad \frac{*x \ := \ *y \quad x \to w \quad y \to z \quad z \to v}{w \to v}$$

$$\frac{*x \ := \ \&y \quad x \to w}{w \to y}$$

Example: pointer analysis (3/3)

Example (Pointer analysis steps)

```
x := &a ; y := &x ;
while B
    *y := &b ;
*x := *y
```

- initial facts are from the first two assignments:

$$\mathtt{x} \rightarrow \mathtt{a}, \quad \mathtt{y} \rightarrow \mathtt{x}$$

- from $\mathtt{y} \rightarrow \mathtt{x}$ and the while-loop body, add

$$\mathtt{x} \rightarrow \mathtt{b}$$

- from the last assignment:
  - from $\mathtt{x} \rightarrow \mathtt{a}$ and $\mathtt{y} \rightarrow \mathtt{x}$, add $\mathtt{a} \rightarrow \mathtt{a}$
  - from $\mathtt{x} \rightarrow \mathtt{b}$ and $\mathtt{y} \rightarrow \mathtt{x}$, add $\mathtt{b} \rightarrow \mathtt{b}$
  - from $\mathtt{x} \rightarrow \mathtt{a}$, $\mathtt{y} \rightarrow \mathtt{x}$, and $\mathtt{x} \rightarrow \mathtt{b}$, add $\mathtt{a} \rightarrow \mathtt{b}$
  - froom $\mathtt{x} \rightarrow \mathtt{b}$, $\mathtt{y} \rightarrow \mathtt{x}$, and $\mathtt{x} \rightarrow \mathtt{a}$, add $\mathtt{b} \rightarrow \mathtt{a}$

Limitations

Not powerful enough for arbitrary language

- sound rules?
  - error prone for complicated features of modern languages
  - e.g. function call/return, function as a data, dynamic method dispatch, exception, pointer manipulation, dynamic memory allocation, ...
- accuracy problem
  - consider program a set of statements, with no order between them
  - rules do not consider the control flow
  - the analysis blindly collects every possible facts when rules hold
  - accuracy improvement by more elaborate rules, yet no systematic way for soundness proof

Example: higher-order control-flow analysis (1/4)

Consider the following higher-order call-by-value functional language. Each subexpression of the program is uniquely labeled:

$$
\begin{array}{rcll}
P & ::= & F & \text{program} \\
F & ::= & & \text{expression} \\
 & | & \mathtt{x} & \text{variable} \\
 & | & \lambda\mathtt{x}.E & \text{a function with argument } x \text{ and body } E \\
 & | & E\ E & \text{function application} \\
E & ::= & F_l & \text{expression } F \text{ with label } l
\end{array}
$$

- a program is an expression without a free variable
- program execution is defined by the *beta reduction* ($\to$) sequence in the call-by-value order: $(\lambda\mathtt{x}.e)\ e' \to \{e'/\mathtt{x}\}e$, where $\{e'/\mathtt{x}\}e$ denotes the expression obtained by replacing $\mathtt{x}$ by e' in e. We assume that, during execution, every function's argument is uniquely renamed.

Example: higher-order control-flow analysis (2/4)

Control-flow is determined by which functions are called for each application expression. We need to collect which lambda expression can be bound to which argument during execution.

- for example, the program

$$(\lambda \mathrm{x}.(\mathrm{x}(\lambda \mathrm{y}.\mathrm{y})))(\lambda \mathrm{z}.\mathrm{z})$$

  runs as follows:

$$\begin{array}{rc} & (\lambda \mathrm{x}.(\mathrm{x}(\lambda \mathrm{y}.\mathrm{y})))(\lambda \mathrm{z}.\mathrm{z}) \\ \rightarrow & (\lambda \mathrm{z}.\mathrm{z})(\lambda \mathrm{y}.\mathrm{y}) \\ \rightarrow & \lambda \mathrm{y}.\mathrm{y} \end{array}$$

- during the execution, the first step binds x to $\lambda \mathrm{z}.\mathrm{z}$ and the second step binds z to $\lambda \mathrm{y}.\mathrm{y}$

Example: higher-order control-flow analysis (3/4)

We let an analysis collect facts about which lambda expression "$\lambda \mathrm{x}.e$" a sub-expression may evaluate to. Hence, we represent each fact by a pair $L \ni R$, meaning "L can have value R".

$$
\begin{array}{rcll}
L & ::= & l \mid \mathrm{x} & \text{expression label or variable} \\
R & ::= & l \mid \mathrm{x} \mid v & \\
v & ::= & \lambda \mathrm{x}.E & \text{value}
\end{array}
$$

Note that set of facts $L \ni R$ for a program is finite.

- initial fact setup rules:

$$
\frac{(\lambda \mathrm{x}.E)_l}{l \ni \lambda \mathrm{x}.E} \qquad \frac{(\mathrm{x})_l}{l \ni \mathrm{x}}
$$

- the propagation rules:

$$
\frac{(E_{l_1}\ E_{l_2})_l \quad l_1 \ni \lambda \mathrm{x}.E_{l_3} \quad l_2 \ni v}{l \ni l_3 \quad \mathrm{x} \ni v} \qquad \frac{l_1 \ni l_2 \quad l_2 \ni v}{l_1 \ni v}
$$

Example: higher-order control-flow analysis (4/4)

Example (Control-flow analysis)

$$\underbrace{\underbrace{(\lambda \mathrm{x}.\underbrace{(\mathrm{x}_5(\overbrace{\lambda \mathrm{y}.\mathrm{y}_6}^{3}))}_{2})}_{1}(\overbrace{\lambda \mathrm{z}.\mathrm{z}_7}^{4})}_{0}$$

The initial facts are collected from the lambda expressions 1, 3, and 4 and variable expressions 5, 6, and 7:

$$\{1 \ni \lambda \mathrm{x}.(\mathrm{x}(\lambda \mathrm{y}.\mathrm{y})), \quad 3 \ni \lambda \mathrm{y}.\mathrm{y}, \quad 4 \ni \lambda \mathrm{z}.\mathrm{z}, \quad 5 \ni \mathrm{x}, \quad 6 \ni \mathrm{y}, \quad 7 \ni \mathrm{z}\}$$

- from expression 0, we add $\mathrm{x} \ni 4$ (parameter binding) and $0 \ni 2$ (application result)
- then by the last propagation rule from $\mathrm{x} \ni 4$ and $4 \ni \lambda \mathrm{z}.\mathrm{z}$, we add $\mathrm{x} \ni \lambda \mathrm{z}.\mathrm{z}$, and then from $5 \ni \mathrm{x}$, we add $5 \ni \lambda \mathrm{z}.\mathrm{z}$
- then from application expression 2, we add $\mathrm{z} \ni 3$ (parameter binding) and $2 \ni 7$ (application result)
- then by the last propagation rule, we add $\mathrm{z} \ni \lambda \mathrm{y}.\mathrm{y}$; then, from $7 \ni \mathrm{z}$, we add $7 \ni \lambda \mathrm{y}.\mathrm{y}$, then $2 \ni \lambda \mathrm{y}.\mathrm{y}$, and then $0 \ni \lambda \mathrm{y}.\mathrm{y}$.

Limitations

- the above analysis uses a crude abstraction for the function values. It collects only the function code part (lambda expressions), with no distinction for the values of the function's free variables.
  - a function value in the concrete semantics is a pair of the function code and a table (called *environment*) that determines the values of the function's free variables. The above analysis completely abstracts away the environment part.
  - during a program's execution, a function expression ($\lambda x.E$) may evaluate into distinct values at different contexts (i.e., when the function's free variable is the parameter of a function that is multiply called with different values).
- for more elaborate abstraction for function values, the above analysis needs an overhaul, whose design and soundness assurance will be facilitated by general semantic frameworks of chapters 3 and 4.

Outline

Static analysis by proof construction

- static analysis = proof construction in a finite proof system
- finite proof system = a finite set of inference rules for a predefined set of judgments
- the soundness corresponds to the soundness of the proof system.
  - the input program is provable $\Rightarrow$ the program satisfies the proven judgment.

Example: type inference (1/4)

$$
\begin{array}{lcll}
P & ::= & E & \text{program} \\
E & ::= & & \text{expression} \\
 & | & n & \text{integer} \\
 & | & \mathtt{x} & \text{variable} \\
 & | & \lambda \mathtt{x}.E & \text{function} \\
 & | & E\ E & \text{function application}
\end{array}
$$

- judgment that says expression E has type τ is written as

$$\Gamma \vdash E : \tau$$

- Γ is a set of type assumptions for the free variables in E.

Example: type inference (2/4)

Consider *simple types*

$$\tau ::= \mathit{int} \mid \tau \rightarrow \tau$$

$$\frac{}{\Gamma \vdash n : \mathit{int}} \qquad \frac{\mathtt{x} : \tau \ \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}$$

$$\frac{\Gamma + \mathtt{x} : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda \mathtt{x}.E : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1\ E_2 : \tau_2}$$

Figure: Proof rules of simple types

Theorem (Soundness of the proof rules)

Let E be a program, an expression without free variables. If $\emptyset \vdash E : \tau$, then the program runs without a type error and returns a value of type τ if it terminates.

Example: type inference (3/4)

Program

$$(\lambda \mathrm{x}.\mathrm{x}\ 1)(\lambda \mathrm{y}.\mathrm{y})$$

is typed *int* because we can prove

$$\emptyset \vdash (\lambda \mathrm{x}.\mathrm{x}\ 1)(\lambda \mathrm{y}.\mathrm{y}) : \mathit{int}$$

as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{x} : \mathit{int} \to \mathit{int} \ \in \{\mathrm{x} : \mathit{int} \to \mathit{int}\}}{\{\mathrm{x} : \mathit{int} \to \mathit{int}\} \vdash \mathrm{x} : \mathit{int} \to \mathit{int}} \quad \dfrac{}{\{\mathrm{x} : \mathit{int} \to \mathit{int}\} \vdash 1 : \mathit{int}}}{\{\mathrm{x} : \mathit{int} \to \mathit{int}\} \vdash \mathrm{x}\ 1 : \mathit{int}}}{\emptyset \vdash \lambda \mathrm{x}.\mathrm{x}\ 1 : (\mathit{int} \to \mathit{int}) \to \mathit{int}} \qquad \dfrac{\dfrac{\mathrm{y} : \mathit{int} \ \in \{\mathrm{y} : \mathit{int}\}}{\{\mathrm{y} : \mathit{int}\} \vdash \mathrm{y} : \mathit{int}}}{\emptyset \vdash \lambda \mathrm{y}.\mathrm{y} : \mathit{int} \to \mathit{int}}}{\emptyset \vdash (\lambda \mathrm{x}.\mathrm{x}\ 1)(\lambda \mathrm{y}.\mathrm{y}) : \mathit{int}}$$

Example: type inference (4/4)

Algorithm

- given a program E, $V(\emptyset, E, \alpha)$ (new type variable α) returns a set of type equations of $\tau \doteq \tau'$. Here, $\tau ::= \alpha \mid int \mid \tau \to \tau$.

$$
\begin{array}{rcll}
V(\Gamma, n, \tau) & = & \{\tau \doteq int\} & \\
V(\Gamma, \mathrm{x}, \tau) & = & \{\tau \doteq \Gamma(\mathrm{x})\} & \\
V(\Gamma, \lambda \mathrm{x}.E, \tau) & = & \{\tau \doteq \alpha_1 \to \alpha_2\} \cup V(\Gamma + \mathrm{x} : \alpha_1, E, \alpha_2) & (\text{new } \alpha_i) \\
V(\Gamma, E_1\ E_2, \tau) & = & V(\Gamma, E_1, \alpha \to \tau) \cup V(\Gamma, E_2, \alpha) & (\text{new } \alpha)
\end{array}
$$

- solving the equations $V(\emptyset, E, \alpha)$ is done by the *unification* procedure
- the *unification* procedure finds the most general (least) solution

Theorem (Correctness of the algorithm)

Solving the equations $\equiv$ proving in the simple type system

More precise analysis? (i.e., for more programs that run without a type error to be provable)

- need new proof rules (e.g.,*polymorphic type systems*) and algorithms

Limitations

- for target languages that lack a sound static type system, we have to invent it:
  - design a finite proof system
  - prove the soundness of the proof system
  - design its algorithm that automates proving
  - prove the correctness of the algorithm
- what if the unification procedure is not enough?
  - for some properties, the algorithm can generate constraints that are unsolvable by the unification procedure
- for some conventional imperative languages, sound and precise-enough static type systems are elusive

Outline

Summary

Sketched three specialized framework and their limitations

- static analysis by equations (a.k.a "data-flow analysis")
- static analysis by monotonic closure
- static analysis by proof construction (a.k.a "static type system")

A reminder

- for specific languages and semantic properties, they can be powerful enough, yet
- weak in handling arbitrary languages and properites
- not general enough as the semantics-based frameworks (chapter 3 and chapter 4)