Outline

Specialized Frameworks

Practical altenatives to the aforementioned general, abstract interpretation framework

- for simple languages and properties,
- $\exists$frameworks that are simple yet powerful enough
- review of their limitations

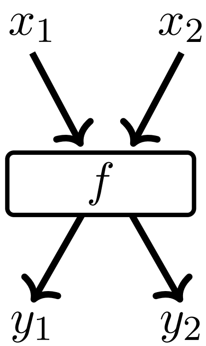
Three specialized frameworks:

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

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)

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

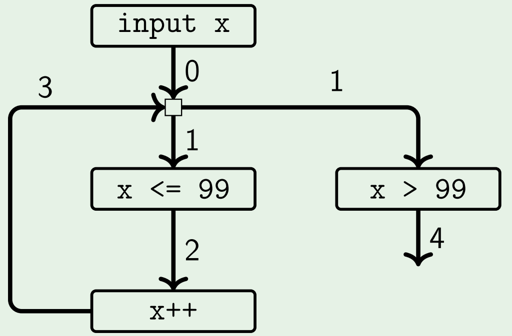


Figure: Control-flow graph

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

Figure: A set of equations for the program

Limitations

Not powerful enough for arbitrary languages

- control-flow before analysis?
  - control is also computed in modern languages
  - no: 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 data, 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

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 are 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

$$\bigcup_{i \geq 0} Y_i,$$

where

$$\begin{array}{rcl} Y_0 & = & X_0, \\ Y_{i+1} & = & Y \text{ such that } Y_i \vdash_R Y. \end{array}$$

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)

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

- Goal: estimate all “points-to” relations between variables that can occur during executions
- a $\rightarrow$ 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:

$$\frac{x \;:=\; \&y}{x \rightarrow y}$$

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

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

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

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

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:

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

- From $\texttt{y} \rightarrow \texttt{x}$ and the while-loop body, add

$$\texttt{x} \rightarrow \texttt{b}$$

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

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, but no systematic way for soundness proof

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)

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

- 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*

$$\tau ::= int \mid \tau \rightarrow \tau$$

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

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

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

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

is typed int because we can prove

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

as follows:

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

Example: Type Inference (4/4)

Algorithm

- given a program E, $V(\emptyset, E, \alpha)$ returns type equations.

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

- solving the equations is done by the *unification* procedure

Theorem (Correctness of the algorithm)

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

More precise analysis?

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

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.