A brief introduction to static program analysis

David Schmidt
Kansas State University

www.cis.ksu.edu/~schmidt
**What is static program analysis?**

It is the extraction of a program’s properties in advance of the program’s execution.

Example properties:

- the program will not generate a run-time exception (error)
- the program will generate an output that has a desirable property
- the program’s internal statements have desirable properties that admit optimization

Standard techniques:

- type checking
- iterative dataflow analysis
- theorem proving
**An example Python program**

Let \( n \) be some input integer:

\[
\begin{align*}
p_0 &: i = n; \quad x = 0; \\
p_1 &: \text{while } i \neq 0 : \\
&\quad p_2 : x = x + 1; \quad i = i - 1 \\
p_3 &: \text{print } x
\end{align*}
\]

What properties can we extract?

♦ the program will not generate a type-mismatch exception

♦ the definitions (assignments) at point \( p_0 \) possibly reach \( p_3 \)

♦ the program satisfies the postcondition, \( x = n \)
Type checking the Python program

$p_0 : i = n; \quad x = 0;$
$p_1 : \text{while } i \neq 0 :$
\hspace{1cm} $p_2 : x = x + 1; \quad i = i - 1$
$p_3 : \text{print } x$

The program is well-typed: it won’t generate a mismatch exception.

Although it’s drawn as an (inverted) deduction, type checking is implemented as a traversal of the program’s parse tree:

\[
\begin{align*}
\Gamma \vdash 0 : \text{int} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 + e_2 : \text{int} & \\
\Gamma \vdash e_1 == e_2 : \text{bool} \\
\Gamma \vdash e : \tau & \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2 \\
\Gamma \vdash c_1 ; c_2 : \Gamma_2 & \\
\Gamma \vdash \text{while } e : c : \Gamma & \\
\end{align*}
\]

Let \( \Gamma_0 = [ x \rightarrow \text{int}, \ i \rightarrow \text{int}] \)
Reaching definitions calculated by dataflow analysis

Does the assignment at \( p_i \) reach point \( p_j \)?

\[ p_0 : \begin{align*}
  i &= n \\
  x &= 0
\end{align*} \]

\[ p_1 : \begin{align*}
  i \neq 0 &? \\
  x &= x + 1 \\
  i &= i - 1
\end{align*} \]

\[ p_2 : \begin{align*}
  x &= x + 1 \\
  i &= i - 1
\end{align*} \]

\[ p_3 : \text{print } x \]

For the example program, the equations for reaching definitions are solved iteratively as

\[
\begin{align*}
  \text{in}_{p_0} &= \emptyset \\
  \text{in}_{p_1} &= \{p_0, p_2\} \\
  \text{in}_{p_2} &= \{p_0, p_2\} \\
  \text{in}_{p_3} &= \{p_0, p_2\}
\end{align*}
\]

\[
\begin{align*}
  \text{out}_{p_i} &= \bigcup_{p' \in \text{pred} p_i} \text{out}_{p'} \\
  \text{for } p_i &\equiv x' = e'
\end{align*}
\]
Partial correctness proved within Hoare logic

\[ p_0 : i = n; \ x = 0; \]
\[ p_1 : \text{while } i \neq 0 : \]
\[ p_2 : x = x + 1; \ i = i - 1 \]
\[ p_3 : \text{print } x \]

\[ \{ [e/x]P \} \ x = e \ \{ P \} \]
\[ \{ P \} \ c_1 \ \{ Q \} \ c_2 \ \{ R \} \]
\[ \{ P \} \ c_1; c_2 \ \{ R \} \]
\[ \{ e \land P \} \ c \ \{ P \} \]
\[ \{ P \} \ \text{while } e : c \ \{ \neg e \land P \} \]

\[ \{ x + 1 = n - i + 1 \} \ x = x + 1 \ \{ x = n - i + 1 \} \ i = i - 1 \ \{ x = n - i \} \]

\[ \{ i \neq 0 \land x = n - i \} \ x = x + 1; \ i = i - 1 \ \{ x = n - i \} \]

\[ \{ x = n - i \} \ \text{while } i \neq 0 : \ x = x + 1; \ i = i - 1 \ \{ x = n \} \]

\[ \{ \text{true} \} \ i = n; \ x = 0 \ \{ x = n - i \} \ \text{while } i \neq 0 : \ x = x + 1; \ i = i - 1 \ \{ x = n \} \]

One must discover the loop invariant, \( x = n - i \), to accomplish the proof.
Three axes of static analyses

**ALGORITHM**
- human interaction
- multipass (repeat till convergence)
- 1 pass
- syntax directed

**PRESENTATION**
- graph (whole program)
- inductive (modular)

**LOGIC (abstract domain)**
- propositional
- modal/predicate
## Some standard static analyses

<table>
<thead>
<tr>
<th>analysis</th>
<th>logic</th>
<th>algorithm</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type checking</td>
<td>propositional logic: int, int → bool</td>
<td>one pass (traverse syntax tree)</td>
<td>syntax-directed</td>
</tr>
<tr>
<td>ML type inference</td>
<td>shallow logic: ∀ α. α → α</td>
<td>one pass + unification</td>
<td>syntax-directed</td>
</tr>
<tr>
<td>a.i.-based dataflow analysis</td>
<td>propositional logic (token sets)</td>
<td>iterate until convergence</td>
<td>graph-based</td>
</tr>
<tr>
<td>model checking</td>
<td>LTL, ACTL (modal-like logics)</td>
<td>iterate forever (!)</td>
<td>graph-based</td>
</tr>
<tr>
<td>theorem proving, LF</td>
<td>predicate logics</td>
<td>human interaction</td>
<td>inductive, stated axiomatically</td>
</tr>
</tbody>
</table>
A “hybrid” analysis: predicate abstraction

We wish to prove that \( z \geq x \land z \geq y \) at \( p_3 \):

\[
\begin{align*}
p_0 & : \text{ if } x < y \\
p_1 & : \text{ then } z = y \\
p_2 & : \text{ else } z = x \\
p_3 & : \text{ exit}
\end{align*}
\]

We choose three predicates, \( \phi_1 = x < y \), \( \phi_2 = z \geq x \), and \( \phi_2 = z \geq y \) and compute their values at the program’s points. The predicates’ values come from the domain, \{t, f, ?\}. (Read ? as \( t \lor f \).)

At all occurrences of \( p_3 \) in the abstract trace, \( \phi_2 \land \phi_3 \) holds.
When a goal is undecided, refinement is necessary

Prove $\phi_0 \equiv x \geq y$ at $p_4$:

$p_0 : \text{if } \neg (x \geq y)$
$p_1 : \text{then } \{ i = x;$
$p_2 : \quad x = y;$
$p_3 : \quad y = i;$
$p_4 : \}$

To decide the goal, we must refine the state by adding a needed auxiliary predicate: $wp(y = i, x \geq y) = (x \geq i) \equiv \phi_1$. 

because $x \not\geq y$ and $x \geq i$ imply $y > i$ implies $x_{new} \geq i$
But incremental predicate refinement cannot synthesize many interesting loop invariants. For this example:

\[
\begin{align*}
  p_0 : & \quad i = n; \quad x = 0; \\
  p_1 : & \quad \text{while } i \neq 0 \{ \\
  p_2 : & \quad x = x + 1; \quad i = i - 1; \\
  \} \\
  p_3 : & \quad \text{goal: } x = n
\end{align*}
\]

We find that the initial predicate set, \( P_0 \equiv \{i = 0, x = n\} \), does not validate the loop body.

The first refinement suggests we add \( P_1 \equiv \{i = 1, x = n - 1\} \) to the program state, but this fails to validate a loop that iterates more than once.

Refinement stage \( j \) adds predicates \( P_j \equiv \{i = j, x = n - j\} \); the refinement process continues forever!

*The loop invariant is* \( x = n - i \quad :)\)
Mr. Gedell will present interesting combinations and variations of the standard analyses....
References


   http://santos.cis.ksu.edu/schmidt/Escuela03/home.html