Introduction

Goal: Use programmer's design decisions with automatic checking todetect potential errors.

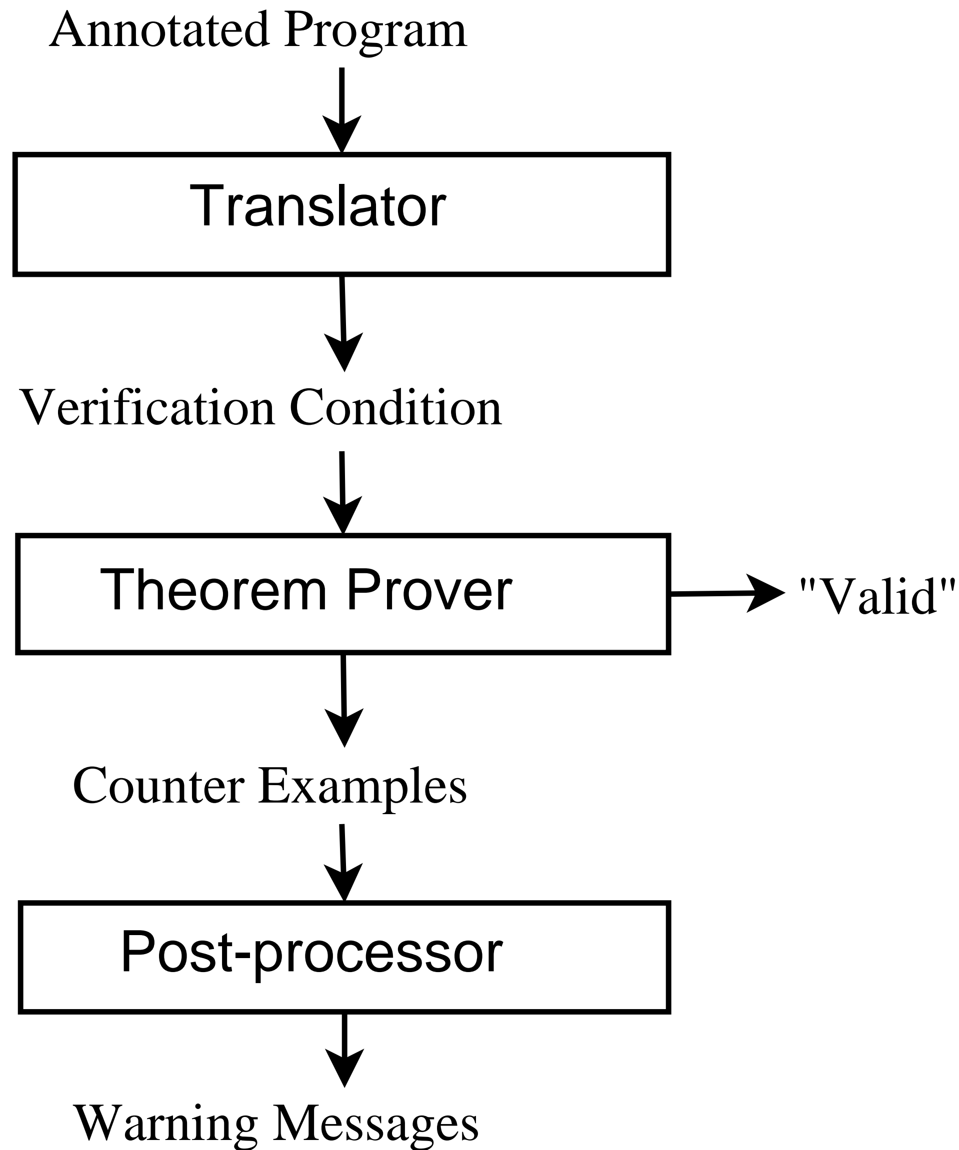
Extended Static Checking (ESC)

- tries to prove correctness at compile-time
- helps finding run-time exceptions (e.g., array exceptions)

Run a program with specifications through a checker to detect errors

- Annotate source with progam behavior expectations
- Use weakest precondition (postcondition) semantics
- Verify conditions using a theorem prover

ESC Structure



ESC in action

Annotate the source code with pre-conditions (and post-conditions)

```
//@ some PRE-condition
//@ some POST-condition
```

func **foobar()**

Generate verification conditions (VC)

```
PRE => WP(POST)
```

Check if the VC is valid (TRUE) in all states

- If VC is valid, then all executions of the function `foobar()` from **PRE** state is guaranteed to terminate only in the **POST** state(s).
- Use theorem prover (Simplify) to check VC

L3 Assertion Language

$$
\begin{array}{lcl}
\langle \text{assert} \rangle & ::= & \langle \text{var} \rangle \mapsto \langle \text{var} \rangle \\
& | & \langle \text{exp} \rangle \diamond \langle \text{exp} \rangle \\
& | & \neg \langle \text{assert} \rangle \\
& | & \forall \alpha.\ \langle \text{assert} \rangle \\
& | & \exists \alpha.\ \langle \text{assert} \rangle \\
& | & \langle \text{assert} \rangle \wedge \langle \text{assert} \rangle \\
& | & \langle \text{assert} \rangle \vee \langle \text{assert} \rangle \\
& | & \textbf{true} \\
& | & \textbf{false}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \text{exp} \rangle & ::= & \langle \text{var} \rangle \\
& | & \langle \text{exp} \rangle \oplus \langle \text{exp} \rangle \\
& | & \text{Integer} \\
& | & \text{Boolean} \\
\langle \text{var} \rangle & ::= & \langle \text{pvar} \rangle \\
& | & \alpha
\end{array}
$$

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & ::= & \ldots \\
& | & \texttt{assume}\ \langle \text{assert} \rangle \\
& | & \texttt{verify}\ \langle \text{assert} \rangle \\
& | & \texttt{invariant}\ \langle \text{assert} \rangle.\ \langle \text{for stmt} \rangle \\
& | & \texttt{invariant}\ \langle \text{assert} \rangle.\ \langle \text{while stmt} \rangle
\end{array}
$$

Statement typing and `verify` annotation

- Partial correctness specification as statement type:

$$\Sigma; \Xi; \Delta; \Gamma \vdash s \ : \ P \leadsto Q$$

- For a sequential composition, the post-condition of the first statement becomes the precondition of the latter:

$$\frac{\Sigma; \Xi; \Delta; \Gamma \vdash s_1 \ : \ P_1 \leadsto Q_1 \qquad \Sigma; \Xi; \Delta; \Gamma \vdash s_2 \ : \ Q_1 \leadsto Q_2}{\Sigma; \Xi; \Delta; \Gamma \vdash s_1; s_2 \ : \ P_1 \leadsto Q_2}(\texttt{seq})$$

- A `verify` statement acts as a compiler directive to type-check function body.

Typing a function

- Type the function body by propagating the precondition for the first statement down to the last statement.
- Existentially quantify, over local variables, the post-condition after the last statement.
- Typing judgment:

$$\Sigma \vdash (\textsf{fn} : \Lambda \overrightarrow{x : \tau}.\ \{P\} r : \tau_r \{Q\})$$

- Typing a function

$$\frac{\Sigma; \Delta; \overrightarrow{l : \tau_l} \vdash e \ : \ P \rightsquigarrow Q}{\Sigma; \Delta \vdash \lambda \overrightarrow{x : \tau}.\ \textbf{let}\ \overrightarrow{l : \tau_l}\ \textbf{in}\ e;\ \textsf{return}\ v\ \textbf{end} \ : \ \Lambda \overrightarrow{x : \tau}.\{P\} v : \tau \{\exists \overrightarrow{l : \tau_l} \setminus (v : \tau_r).Q\}}$$

Typing a function call

- Let the function call be: $v = f(a)$, and the precondition be R.
- Let the type of the function f be $\Lambda(\overrightarrow{x : \tau}).\{P\}r : \tau_r\{Q\}$.
- The problem is how to unify R and P.
  - Initialize the formal parameters in P.
  - Using a unification algorithm, find a substitution σ, for meta-variables in P such that $R \implies \sigma(P\,\overrightarrow{a})$.

$$\frac{\begin{array}{c}\Sigma \vdash f \;:\; \Lambda(\overrightarrow{x : \tau}).\{P\}r : \tau_r\{Q\} \\ \sigma \;=\; \text{unify}(R, P\,\overrightarrow{a}) \qquad \sigma' \;=\; \sigma \cup \{r \mapsto v\}\end{array}}{\Sigma; \Xi; \Delta; \Gamma \vdash v = f(\overrightarrow{a}) \;:\; R \rightsquigarrow \sigma'(Q\,\overrightarrow{a})}$$

Handling pointers

- If the target of the pointer is known:

$$\frac{P \implies p \mapsto a}{\Sigma; \Xi; \Delta; \Gamma \vdash *p \;=\; e \;:\; P \;\rightsquigarrow\; (\exists a'.[a'/a]P) \wedge a = e} (a' \text{fresh})$$

- If the pointer points inside an array, the projection function takes into account that memory outside array cannot be modifi ed.

$$\frac{\Sigma; \Xi; \Delta; \Gamma \vdash p \;:\; \tau \qquad P \implies p \mapsto \texttt{array}}{\Sigma; \Xi; \Delta; \Gamma \vdash *p \;=\; e \;:\; P \rightsquigarrow \pi_{\tau}^{array}(P)}$$

- If nothing is known about pointer, retain only that part of the predicate that is not affected by the update:

$$\frac{\Sigma; \Xi; \Delta; \Gamma \vdash p \;:\; \tau}{\Sigma; \Xi; \Delta; \Gamma \vdash *p \;=\; e \;:\; P \rightsquigarrow \pi_{\tau}^{-}(P)}$$

Handling pointers (contd.)

Dereferencing a pointer:

- If it is known what variable the pointer points to:

$$\frac{P \implies p \mapsto a}{\Sigma;\Xi;\Delta;\Gamma \vdash v \;=\; *p \;:\; P \;\leadsto\; (\exists v'.[v'/v]P) \wedge v = a}(v'\text{fresh})$$

- otherwise:

$$\frac{}{\Sigma;\Xi;\Delta;\Gamma \vdash v \;=\; *p \;:\; P \leadsto \exists v'[v'/v]P}$$

Open Questions

Goal: Same denotational semantics before and after annotations.

- How should the `assume` statement be interpreted by the compiler?
- How to ensure the correctness of annotations ?
  - May be the code checks the assumption at runtime ...
- Unification Algorithm to determine typing a function call

Further reading ...

David L. Detlefs, K. Rustan M. Leino, Greg Nelson, James B. Saxe. **”Extended Static Checking”**. *Compaq Systems Research Center (SRC) Report 159. December, 1998*

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe and Rymie Stata. **”Extended Static Checking for JAVA”**. *Proceedings of Programming Language Design and Implementation (PLDI) 2002.*