Introduction

Goal: Use programmer’s design decisions with automatic checking to detect 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 program behavior expectations
- Use weakest precondition (postcondition) semantics
- Verify conditions using a theorem prover
ESC Structure

Annotated Program

Translator

Verification Condition

Theorem Prover

"Valid"

Counter Examples

Post-processor

Warning Messages
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

\[\langle \text{assert} \rangle \ ::= \langle \text{var} \rangle \rightarrow \langle \text{var} \rangle\]

\[\langle \exp \rangle \diamond \langle \exp \rangle\]

\[\neg \langle \text{assert} \rangle\]

\[\forall \alpha. \langle \text{assert} \rangle\]

\[\exists \alpha. \langle \text{assert} \rangle\]

\[\langle \text{assert} \rangle \land \langle \text{assert} \rangle\]

\[\langle \text{assert} \rangle \lor \langle \text{assert} \rangle\]

\[\langle \text{true} \rangle\]

\[\langle \text{false} \rangle\]

\[\langle \text{stmt} \rangle \ ::= \ldots\]

\[\text{assume} \langle \text{assert} \rangle\]

\[\text{verify} \langle \text{assert} \rangle\]

\[\text{invariant} \langle \text{assert} \rangle. \langle \text{for stmt} \rangle\]

\[\text{invariant} \langle \text{assert} \rangle. \langle \text{while stmt} \rangle\]
Statement typing and \texttt{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:

\begin{align*}
\Sigma; \Xi; \Delta; \Gamma \vdash s_1 & : P_1 \leadsto Q_1 \\
\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 \quad (\text{seq})
\end{align*}

- A \texttt{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:

\[
\Sigma \vdash (fn : \Lambda x : \tau. \{ P \} r : \tau_r \{ Q \})
\]

- Typing a function

\[
\Sigma; \Delta; l : \tau_l \vdash e : P \rightsquigarrow Q
\]

\[
\Sigma; \Delta \vdash \lambda x : \tau. \text{let } l : \tau_l \text{ in } e; \text{ return } v \text{ end} : \Lambda x : \tau. \{ P \} v : \tau\{ \exists l : \tau_l \setminus (v : \tau_r). Q \}
\]
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 \( \Lambda(x : \tau).\{P\}r : \tau_r\{Q\} \).
- The problem is how to unify \( R \) and \( P \).
  - Initialize the formal parameters in \( P \).
  - Using a unification algorithm, find a substitution \( \sigma \), for meta-variables in \( P \) such that \( R \implies \sigma(P \overline{a}) \).

\[
\Sigma \vdash f : \Lambda(x : \tau).\{P\}r : \tau_r\{Q\} \\
\sigma = \text{unify}(R, P \overline{a}) \\
\sigma' = \sigma \cup \{r \mapsto v\} \\
\Sigma; \Xi; \Delta; \Gamma \vdash v = f(\overline{a}) : R \rightsquigarrow \sigma'(Q \overline{a})
\]
Handling pointers

If the target of the pointer is known:

\[ P \implies p \mapsto a \]
\[ \Sigma; \Xi; \Delta; \Gamma \vdash *p = e : P \sim (\exists a'.[a'/a]P) \land a = e \]

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

\[ \Sigma; \Xi; \Delta; \Gamma \vdash p : \tau \quad P \implies p \mapsto \text{array} \]
\[ \Sigma; \Xi; \Delta; \Gamma \vdash *p = e : P \sim \pi_{\tau}^{\text{array}}(P) \]

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

\[ \Sigma; \Xi; \Delta; \Gamma \vdash p : \tau \]
\[ \Sigma; \Xi; \Delta; \Gamma \vdash *p = e : P \sim \pi_{\tau}^{-}(P) \]
Handling pointers (contd.)

Dereferencing a pointer:

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

\[
P \implies p \leftrightarrow a
\]

\[
\Sigma; \Xi; \Delta; \Gamma \vdash v = \star p : P \rightsquigarrow (\exists v'.[v'/v]P) \land v = a
\]

- otherwise:

\[
\Sigma; \Xi; \Delta; \Gamma \vdash v = \star p : P \rightsquigarrow \exists v'[v'/v]P
\]
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 ...
