# 50.054 Static Semantics

ISTD, SUTD

## Learning Outcomes

1. Explain what static semantics is.
2. Apply type checking rules to verify the type correctness property of a SIMP program.
3. Explain the relation between type system and operational semantics.
4. Apply type inference rules and unification to determine the most general type environment given a SIMP program

## Static Semantics

Static semantics defines the compile-time properties of the given program.

For example, a *statically correct* program, must satisfy

1. all uses of variables in it must be defined somewhere earlier.
2. all the use of variables, the types must be matching with the expected type in the context.
3. . . .

## Static Semantics

Statically correct

```
x = 0;
y = input;
if y > x {
    y = 0;
}
return y;
```

Statically incorrect

```
x = 0;
y = input;
if y + x { // type error
    x = z; // the use of an
           // undefined variable z
}
return x;
```

## Static type checking a chore or a core?

- The rise of dynamic typing languages such as Python, JS
  - Why?
- When the unicorn is turning into a gigantic behemoth.
  - https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python

## SIMP with Types

$$
\begin{array}{rcll}
\texttt{(Statement)} & S & ::= & X = E; \mid \mathit{return}\ X; \mid \mathit{nop}; \mid \mathit{if}\ \ E\ \{\overline{S}\}\ \mathit{else}\ \{\overline{S}\} \mid \mathit{while}\ E\ \{\overline{S}\} \\
\texttt{(Expression)} & E & ::= & E\ \ OP\ \ E \mid X \mid C \mid (E) \\
\texttt{(Statements)} & \overline{S} & ::= & S \mid S\ \overline{S} \\
\texttt{(Operator)} & OP & ::= & + \mid - \mid * \mid < \mid == \\
\texttt{(Constant)} & C & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \mathit{true} \mid \mathit{false} \\
\texttt{(Variable)} & X & ::= & a \mid b \mid c \mid d \mid \ldots \\
\texttt{(Types)} & T & ::= & \mathit{int} \mid \mathit{bool} \\
\texttt{(Type Environments)} & \Gamma & \subseteq & (X \times T)
\end{array}
$$

- $\Gamma$ is a mapping from variables to types.
- There should be maiximum one entry for the each variable in $\Gamma$.
- Examples of type environment $\Gamma = \{(x, \mathit{int}), (y, \mathit{bool})\}$.

## Type Checking

Given a type environment $\Gamma$ and a SIMP program $\overline{S}$, we would like to check whether $\overline{S}$ is having valid type assignment. Sometimes, we refer a program $\overline{S}$ that has valid type assignment is *typeable*, or *well-typed*.

There are two types of rules

- $\Gamma \vdash \overline{S}$, checks whether $\overline{s}$ is well-typed.
- $\Gamma \vdash E : T$, checks whether an expression $E$ is having type $T$.

## Type Checking Expressions

$$(\texttt{tVar}) \qquad \frac{(X, T) \in \Gamma}{\Gamma \vdash X : T}$$

$$(\texttt{tInt}) \qquad \frac{C \texttt{ is an integer}}{\Gamma \vdash C : \mathit{int}}$$

$$(\texttt{tBool}) \qquad \frac{C \in \{\mathit{true}, \mathit{false}\}}{\Gamma \vdash C : \mathit{bool}}$$

$$(\texttt{tOp1}) \qquad \frac{\Gamma \vdash E_1 : \mathit{int} \quad \Gamma \vdash E_2 : \mathit{int} \quad \mathit{OP} \in \{+, -, *\}}{\Gamma \vdash E_1 \ \mathit{OP} \ E_2 : \mathit{int}}$$

$$(\texttt{tOp2}) \qquad \frac{\Gamma \vdash E_1 : \mathit{int} \quad \Gamma \vdash E_2 : \mathit{int} \quad \mathit{OP} \in \{==, <\}}{\Gamma \vdash E_1 \ \mathit{OP} \ E_2 : \mathit{bool}}$$

$$(\texttt{tOp3}) \qquad \frac{\Gamma \vdash E_1 : \mathit{bool} \quad \Gamma \vdash E_2 : \mathit{bool} \quad \mathit{OP} \in \{==, <\}}{\Gamma \vdash E_1 \ \mathit{OP} \ E_2 : \mathit{bool}}$$

$$(\texttt{tParen}) \qquad \frac{\Gamma \vdash E : T}{\Gamma \vdash (E) : T}$$

## Type Checking Statements

$$(\texttt{tSeq}) \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash \overline{S}}{\Gamma \vdash S\overline{S}}$$

$$(\texttt{tAssign}) \qquad \frac{\Gamma \vdash E : T \quad \Gamma \vdash X : T}{\Gamma \vdash X = E}$$

$$(\texttt{tReturn}) \qquad \frac{\Gamma \vdash X : T}{\Gamma \vdash \mathit{return}\ X}$$

$$(\texttt{tNop}) \qquad \Gamma \vdash \mathit{nop}$$

$$(\texttt{tIf}) \qquad \frac{\Gamma \vdash E : \mathit{bool} \quad \Gamma \vdash \overline{S_1} \quad \Gamma \vdash \overline{S_2}}{\Gamma \vdash \mathit{if}\ \ E\ \{\overline{S_1}\}\ \mathit{else}\ \{\overline{S_2}\}}$$

$$(\texttt{tWhile}) \qquad \frac{\Gamma \vdash E : \mathit{bool} \quad \Gamma \vdash \overline{S}}{\Gamma \vdash \mathit{while}\ E\ \{\overline{S}\}}$$

# Type Checking Example

Let $\Gamma = \{(input, int), (x, int)\}$, type check

```
x = input;
x = x + 1;
return x;
```

should succeed

## Type Checking Example

Let $\Gamma = \{(input, int), (x, int)\}$, type check

```
x = input;
if (x + 1) {
    x = x + 1;
} else { nop; }
return x;
```

should fail

## Type Checking (Static Semantics) and Dynamic Semantics

- Definition 1 - Type and Value Environments Consistency

We say $\Gamma \vdash \Delta$ iff for all $(X, C) \in \Delta$ we have $(X, T) \in \Gamma$ and $\Gamma \vdash C : T$.

For example, $\Gamma = \{(x, int), (y, bool)\}$ and $\Delta = \{(x, 1)\}$, we have $\Gamma \vdash \Delta$.

## Progress - a well typed SIMP program must not be stuck

Let $\overline{S}$ be a SIMP statement sequence. Let $\Gamma$ be a type environment such that $\Gamma \vdash \overline{S}$. Then $\overline{S}$ is either

1. a return statement, or
2. a sequence of statements, and there exist $\Delta$, $\Delta'$ and $\overline{S'}$ such that $\Gamma \vdash \Delta$ and $(\Delta, \overline{S}) \longrightarrow (\Delta', \overline{S'})$.

It can be proven by mutual induction.

## Preservation - evaluation does not change the typability

Let $\Delta$, $\Delta'$ be value environments. Let $\overline{S}$ and $\overline{S'}$ be SIMP statement sequences such that $(\Delta, \overline{S}) \longrightarrow (\Delta', \overline{S'})$. Let $\Gamma$ be a type environment such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \overline{S}$. Then $\Gamma \vdash \Delta'$ and $\Gamma \vdash \overline{S'}$.

It can be proven by mutual induction.

## Type Inference

Given a SIMP program $\overline{S}$ find the best $\Gamma$ such that $\Gamma \vdash \overline{S}$.

- What is "the best"?
  - the smallest possible $\Gamma$ that make $\overline{S}$ typeable.

For example

```
x = 1;
return x;
```

The above program can be type-checked under both $\Gamma_1 = \{(x, int)\}$ and $\Gamma_2 = \{(x, int), (y, int)\}$.

$\Gamma_1$ should be favored since it is smaller.

## Meta Symbols for Type Inference

$$\begin{array}{rccl}
(\texttt{Extended Types}) & \hat{T} & ::= & \alpha \mid T \\
(\texttt{Constraints}) & \kappa & \subseteq & (\hat{T} \times \hat{T}) \\
(\texttt{Type Substitution}) & \Psi & ::= & [\hat{T}/\alpha] \mid [] \mid \Psi \circ \Psi
\end{array}$$

Where $\alpha$ denotes a type variable. $\kappa$ define a set of pairs of extended types that are supposed to be equal, e.g. $\kappa = \{(\alpha, \beta), (\beta, \mathit{int})\}$ means $\alpha = \beta \wedge \beta = \mathit{int}$.

## Type Substitution

$$\begin{array}{rrcl} (\texttt{Extended Types}) & \hat{T} & ::= & \alpha \mid T \\ (\texttt{Constraints}) & \kappa & \subseteq & (\hat{T} \times \hat{T}) \\ (\texttt{Type Substitution}) & \Psi & ::= & [\hat{T}/\alpha] \mid [] \mid \Psi \circ \Psi \end{array}$$

Type substititution replace type variable to some other type.

$$\begin{array}{rcll} []\hat{T} & = & \hat{T} & \\ [\hat{T}/\alpha]\alpha & = & \hat{T} & \\ [\hat{T}/\alpha]\beta & = & \beta & \quad \textit{if } \alpha \neq \beta \\ [\hat{T}/\alpha]T & = & T & \\ (\Psi_1 \circ \Psi_2)\hat{T} & = & \Psi_1(\Psi_2(\hat{T})) & \end{array}$$

## Type Inference Rules for SIMP

- For statements $S \vDash \kappa$, $\overline{S} \vDash \kappa$
- For expression $E \vDash \hat{T}, \kappa$

# Type Inference for SIMP expressions

(tiInt) $$\frac{C \texttt{ is an integer}}{C \vDash int, \{\}}$$

(tiBool) $$\frac{C \in \{true, false\}}{C \vDash bool, \{\}}$$

(tiVar) $$X \vDash \alpha_X, \{\}$$

(tiOp1) $$\frac{OP \in \{+, -, *\} \quad E_1 \vDash \hat{T}_1, \kappa_1 \quad E_2 \vDash \hat{T}_2, \kappa_2}{E_1\ OP\ E_2 \vDash int, \{(\hat{T}_1, int), (\hat{T}_2, int)\} \cup \kappa_1 \cup \kappa_2}$$

(tiOp2) $$\frac{OP \in \{<, ==\} \quad E_1 \vDash \hat{T}_1, \kappa_1 \quad E_2 \vDash \hat{T}_2, \kappa_2}{E_1\ OP\ E_2 \vDash bool, \{(\hat{T}_1, \hat{T}_2)\} \cup \kappa_1 \cup \kappa_2}$$

(tiParen) $$\frac{E \vDash \hat{T}, \kappa}{(E) \vDash \hat{T}, \kappa}$$

$\alpha_X$ is *skolem* variable, which is

- universally quantified, can be any of the ground type $T$.
- reserved specifically for some purpose, i.e. it is meant for variable $X$.

## Type Inference for SIMP Statements

$$(\texttt{tiNOP}) \quad nop \vDash \{\}$$

$$(\texttt{tiReturn}) \quad return\ X \vDash \{\}$$

$$(\texttt{tiSeq}) \quad \frac{S \vDash \kappa_1 \qquad \overline{S} \vDash \kappa_2}{S\overline{S} \vDash \kappa_1 \cup \kappa_2}$$

$$(\texttt{tiAssign}) \quad \frac{E \vDash \hat{T}, \kappa}{X = E \vDash \{(\alpha_X, \hat{T})\} \cup \kappa}$$

$$(\texttt{tiIf}) \quad \frac{E \vDash \hat{T}_1, \kappa_1 \qquad \overline{S_2} \vDash \kappa_2 \qquad \overline{S_3} \vDash \kappa_3}{if\ \ E\ \{\overline{S_2}\}\ else\{\overline{S_3}\} \vDash \{(\hat{T}_1, bool)\} \cup \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

$$(\texttt{tiWhile}) \quad \frac{E \vDash \hat{T}_1, \kappa_1 \qquad \overline{S_2} \vDash \kappa_2}{while\ E\ \{\overline{S_2}\} \vDash \{(\hat{T}_1, bool)\} \cup \kappa_1 \cup \kappa_2}$$

## Type Inference Example

Consider $\overline{S}$ as

```
x = input;
y = 0;
while (y < x) {
    y = y + 1;
}
```

$\overline{S} \models \{(\alpha_x, \alpha_{input}), (\alpha_y, int), (\alpha_y, \alpha_x)\}$

- We work out the details in Jamboard.
- We need to solve the above type constraints.

## Unification

$$\begin{array}{rcl} mgu(int, int) & = & [] \\ mgu(bool, bool) & = & [] \\ mgu(\alpha, \hat{T}) & = & [\hat{T}/\alpha] \\ mgu(\hat{T}, \alpha) & = & [\hat{T}/\alpha] \end{array}$$

$mgu(\cdot, \cdot)$ function generates a type substitution from two extended types.

We overload it to handle a set of type tuples.

$$\begin{array}{rcl} mgu(\{\}) & = & [] \\ mgu(\{(\hat{T}_1, \hat{T}_2)\} \cup \kappa) & = & let\ \Psi_1 = mgu(\hat{T}_1, \hat{T}_2) \\ & & \qquad\ \kappa' = \Psi_1(\kappa) \\ & & \qquad\ \Psi_2 = mgu(\kappa') \\ & & in\ \Psi_2 \circ \Psi_1 \end{array}$$

## Unification Example

Apply *mgu* to $\{(\alpha_x, \alpha_{input}), (\alpha_y, int), (\alpha_y, \alpha_x)\}$

$$
\begin{array}{ll}
mgu(\{\underline{(\alpha_x, \alpha_{input})}, (\alpha_y, int), (\alpha_y, \alpha_x)\}) & \longrightarrow \\
let\ \Psi_1 = mgu(\alpha_x, \alpha_{input}) & \\
\qquad \kappa_1 = \Psi_1\{(\alpha_y, int), (\alpha_y, \alpha_x)\} & \\
\qquad \Psi_2 = mgu(\kappa_1) & \\
in\ \Psi_2 \circ \Psi_1 & \longrightarrow \\
let\ \Psi_1 = [\alpha_{input}/\alpha_x] & \\
\qquad \kappa_1 = \Psi_1\{(\alpha_y, int), (\alpha_y, \alpha_x)\} & \\
\qquad \Psi_2 = mgu(\kappa_1) & \\
in\ \Psi_2 \circ \Psi_1 & \longrightarrow \\
let\ \Psi_1 = [\alpha_{input}/\alpha_x] & \\
\qquad \kappa_1 = \{(\alpha_y, int), (\alpha_y, \alpha_{input})\} & \\
\qquad \Psi_2 = mgu(\kappa_1) & \\
in\ \Psi_2 \circ \Psi_1 & \longrightarrow \\
[int/\alpha_{input}] \circ [int/\alpha_y] \circ [\alpha_{input}/\alpha_x] &
\end{array}
$$

Apply *mgu* to $\kappa_1$

$$
\begin{array}{ll}
mgu(\{\underline{(\alpha_y, int)}, (\alpha_y, \alpha_{input})\}) & \longrightarrow \\
let\ \Psi_{21} = mgu(\alpha_y, int) & \\
\qquad \kappa_2 = \Psi_{21}\{(\alpha_y, \alpha_{input})\} & \\
\qquad \Psi_{22} = mgu(\kappa_2) & \\
in\ \Psi_{22} \circ \Psi_{21} & \longrightarrow \\
let\ \Psi_{21} = [int/\alpha_y] & \\
\qquad \kappa_2 = \Psi_{21}\{(\alpha_y, \alpha_{input})\} & \\
\qquad \Psi_{22} = mgu(\kappa_2) & \\
in\ \Psi_{22} \circ \Psi_{21} & \longrightarrow \\
let\ \Psi_{21} = [int/\alpha_y] & \\
\qquad \kappa_2 = \{(int, \alpha_{input})\} & \\
\qquad \Psi_{22} = mgu(\kappa_2) & \\
in\ \Psi_{22} \circ \Psi_{21} & \longrightarrow \\
let\ \Psi_{21} = [int/\alpha_y] & \\
\qquad \kappa_2 = \{(int, \alpha_{input})\} & \\
\qquad \Psi_{22} = [int/\alpha_{input}] & \\
in\ \Psi_{22} \circ \Psi_{21} & \longrightarrow \\
[int/\alpha_{input}] \circ [int/\alpha_y] &
\end{array}
$$

## Derive Type Environment from Type Substition

1. Collect all the variables in the source program $\overline{S}$
2. Apply type substitution to each $\alpha_X$.

For example

$$\begin{array}{rl}
([int/\alpha_{input}] \circ [int/\alpha_y] \circ [\alpha_{input}/\alpha_x])\alpha_{input} & = \\
([int/\alpha_{input}] \circ [int/\alpha_y])\alpha_{input} & = \\
[int/\alpha_{input}]\alpha_{input} & = \\
int &
\end{array}$$

$$\begin{array}{rl}
([int/\alpha_{input}] \circ [int/\alpha_y] \circ [\alpha_{input}/\alpha_x])\alpha_x & = \\
([int/\alpha_{input}] \circ [int/\alpha_y])\alpha_{input} & = \\
[int/\alpha_{input}]\alpha_{input} & = \\
int &
\end{array}$$

$$\begin{array}{rl}
([int/\alpha_{input}] \circ [int/\alpha_y] \circ [\alpha_{input}/\alpha_x])\alpha_y & = \\
([int/\alpha_{input}] \circ [int/\alpha_y])\alpha_y & = \\
[int/\alpha_{input}]int & = \\
int &
\end{array}$$

Hence

$$\Gamma = \{(input, int), (x, int), (y, int)\}$$

## Type Inference with Unification

- The order of choosing the type tuples from $\kappa$ does not affect the final result.
- Refer to the notes for the other derivations.

## Type Inference Formal Results

### Soundness

Let $\overline{S}$ be a SIMP program and $\Gamma$ is a type environment inferred using the described inference algorithm. Then $\Gamma \vdash \overline{S}$.

### Principality

Let $\overline{S}$ be a SIMP program and $\Gamma$ is a type environment inferred using the described inference algorithm. Then $\Gamma$ is the most general type environment that can type-check $\overline{S}$.

## Limitation of the Type Inference algorithm - unspecified input type

```
x = input;
y = 0;
while (y < 3) {
    y = y + 1;
}
```

1. $\kappa = \{(\alpha_x, \alpha_{input}), (\alpha_y, int)\}$
2. $\Psi = [\alpha_{input}/\alpha_x] \circ [int/\alpha_y]$
3. It fails to ground $\alpha_x$ and $\alpha_{input}$.

we can force the programmers to fix the input type.

## Limitation of the Type Inference algorithm - uninitialized variable

```
x = z;
y = 0;
while (y < 3) {
    y = y + 1;
}
```

1. $\kappa = \{(\alpha_x, \alpha_z), (\alpha_y, \mathit{int})\}$
2. $\Psi = [\alpha_z / \alpha_x] \circ [\mathit{int} / \alpha_y]$
3. It fails to ground $\alpha_x$ and $\alpha_z$.

we run name analysis to rule out this kind of programs in the first place.

## Limitation of the Type Inference algorithm - control flow insensitivity

```
x = 0;
x = true;
```

1. $\kappa = \{(\alpha_x, \mathit{int}), (\alpha_x, \mathit{bool})\}$
2. unification fails.

Not an issue most of the time. If required, we use flow-sensitive analysis or SSA.

## Summary

- Type Checking for SIMP
  - Preservation and Progress
- Type Inference for SIMP
  - Soundness and Principality
- Limitation

Languages without type specifier
Writing in C++ 11 using "auto" all the time