Type Judgments

- **Static semantics** = formal notation which describes type judgments:
  \[ E : T \]
  means "E is a well-typed expression of type T"

- Type judgment examples:
  - `2 : int`
  - `2 * (3 + 4) : int`
  - `true : bool`
  - "Hello" : string

Type Judgments for Statements

- Statements may be expressions (i.e. represent values)
- Use type judgments for statements:
  \[
  \begin{align*}
  \text{if (b) then 2 else 3 : int} \\
  x = 10 : \text{bool} \\
  b = \text{true}, y = 2 : \text{int}
  \end{align*}
  \]
- For statements which are not expressions: use a special unit type (empty type):
  \[ S : \text{unit} \]
  means "S is a well-typed statement with no result type"

Deriving a Judgment

- Consider the judgment:
  \[
  \begin{align*}
  \text{if (b) then 2 else 3 : int}
  \end{align*}
  \]

- What do we need to decide that this is a well-typed expression of type int?
  - `b` must be a bool (`b : bool`)
  - `2` must be an int (`2 : int`)
  - `3` must be an int (`3 : int`)

Type Judgments

- Type judgment notation: \[ A \vdash E : T \]
  means "In the context A the expression E is a well-typed expression with the type T"

- Type context is a set of type bindings `id : T`
  \[ \text{(i.e. type context = symbol table)} \]
  - `b : bool, x : int \vdash b : bool`
  - `b : bool, x : int \vdash \text{if (b) then 2 else x : int}`
  - `\vdash 2 + 2 : \text{int}`
Deriving a Judgement

- To show:
  \( b: \text{bool}, x: \text{int} \vdash \text{if (b) then 2 else } x : \text{int} \)

- Need to show:
  \( b: \text{bool}, x: \text{int} \vdash b : \text{bool} \)
  \( b: \text{bool}, x: \text{int} \vdash 2 : \text{int} \)
  \( b: \text{bool}, x: \text{int} \vdash x : \text{int} \)

General Rule

- For any environment \( A \), expression \( E \), statements \( S_1 \) and \( S_2 \), the judgment
  \( A \vdash \text{if (E) then } S_1 \text{ else } S_2 : T \)

  is true if:
  \( A \vdash E : \text{bool} \)
  \( A \vdash S_1 : T \)
  \( A \vdash S_2 : T \)

Inference Rules

**Premises**

\[
A \vdash E : \text{bool} \quad A \vdash S_1 : T \quad A \vdash S_2 : T
\]

**Conclusion**

\[ A \vdash \text{if (E) then } S_1 \text{ else } S_2 : T \]  \[ \text{(if-rule)} \]

- Holds for any choice of \( E, S_1, S_2, T \)

Why Inference Rules?

- **Inference rules**: compact, precise language for specifying static semantics (can specify languages in ~20 pages vs. 100’s of pages of Java Language Specification)
- Inference rules correspond directly to recursive AST traversal that implements them
- **Type checking** is attempt to prove type judgments \( A \vdash E : T \) true by walking backward through rules

Meaning of Inference Rule

- Inference rule says:
  - given that antecedent judgments are true
  - with some substitution for \( A, E_1, E_2 \)
  - then, consequent judgment is true
- with a consistent substitution

\[
A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int} \quad A \vdash E + E_2 : \text{int} \quad (\text{+})
\]

Proof Tree

- Expression is well-typed if there exists a type derivation for a type judgment
- Type derivation is a proof tree
- Example: if \( A1 = b: \text{bool}, x: \text{int} \), then:

\[
A1 \vdash b : \text{bool} \quad A1 \vdash 2 : \text{int} \quad A1 \vdash 3 : \text{int} \\
A1 \vdash !b : \text{bool} \quad A1 \vdash 2+3 : \text{int} \quad A1 \vdash x : \text{int} \\
b : \text{bool}, x: \text{int} \vdash \text{if (!b) then } 2+3 \text{ else } x : \text{int}
\]
More about Inference Rules

- No premises = axiom
  \[ A \vdash \text{true} : \text{bool} \]

- A goal judgment may be proved in more than one way
  \[
  A \vdash E_1 : \text{float} \\
  A \vdash E_2 : \text{float} \\
  A \vdash E_3 + E_2 : \text{float}
  \]

- No need to search for rules to apply -- they correspond to nodes in the AST

While Statements

- Rule for while statements:
  \[
  A \vdash E : \text{bool} \\
  A \vdash S : T \\
  A \vdash \text{while} \ (E) \ S : \text{unit}
  \]

- Why use unit type for while statements?

If Statements

- If statement as an expression (e.g., in ML): its value is the value of the branch that is executed
  \[
  A \vdash E : \text{bool} \\
  A \vdash S_1 : T \\
  A \vdash S_2 : T \\
  A \vdash \text{if} \ (E) \ 	ext{then} \ S_1 \ 	ext{else} \ S_2 : T \quad \text{(if-then-else)}
  \]

- If no else clause, no value (why?)
  \[
  A \vdash E : \text{bool} \\
  A \vdash S : T \\
  A \vdash \text{if} \ (E) \ S : \text{unit} \quad \text{(if-then)}
  \]

Assignment Statements

- id : T ∈ A
  \[
  A \vdash E : T
  \]

- (variable-assign)
  \[
  A \vdash \text{id} = E : T
  \]

- A \vdash E_3 : T

- int
  \[
  A \vdash E_2 : \text{int}
  \]

- (array-assign)
  \[
  A \vdash E_1 : \text{array}[T] \\
  A \vdash E_1[E_2] = E_3 : T
  \]

Sequence Statements

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:
  \[
  A \vdash S_1 : T_1 \\
  A \vdash (S_2 ; \ldots ; S_n) : T_n \\
  \]

- What about variable declarations?

Declarations

- id : T [ = E ] : T
  \[
  A, \text{id} : T \vdash (S_2 ; \ldots ; S_n) : T_n \\
  \]

- (declaration)
  \[
  A \vdash (\text{id} : T [ = E ]); S_2 ; \ldots ; S_n : T_n
  \]

- Declarations add entries to the environment (in the symbol table)
Function Calls

- If expression $E$ is a function value, it has a type $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- $T_i$ are argument types; $T_r$ is return type
- How to type-check function call $E(E_1, \ldots, E_n)$?

\[
A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r
\]

\[
A \vdash E_i : T_i \quad (i = 1 \ldots n)
\]

\[
A \vdash E(E_1, \ldots, E_n) : T_r
\]

(function-call)

Function Declarations

- Consider a function declaration of the form
  
  \[ T_r \text{ fun } (T_1 \ a_1, \ldots, \ T_n \ a_n) \ { \text{return } E; } \]

- Type of function body $S$ must match declared return type of function, i.e. $E : T_r$
- ... but in what type context?

Add Arguments to Environment!

- Let $A$ be the context surrounding the function declaration. Function declaration:
  
  \[ T_r \text{ fun } (T_1 \ a_1, \ldots, \ T_n \ a_n) \ { \text{return } E; } \]

  is well-formed if

  \[ A, a_1 : T_1, \ldots, a_n : T_n \vdash E : T_r \]

- ...what about recursion?
  
  Need: \[ \text{fun: } T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \in A \]

Recursive Function Example

- Factorial:
  
  \[
  \text{int fact(int x)} \{ \\
  \quad \text{if (x==0) return 1;} \\
  \quad \text{else return x * fact(x - 1);} \\
  \}
  \]

- Prove: $A \vdash x \ * \ \text{fact(x-1)} : \text{int}$
  
  Where: $A = \{ \text{fact: int\rightarrow int, x : int} \}$

Mutual Recursion

- Example:
  
  \[
  \text{int f(int x)} \{ \text{return g(x) + 1;} \} \\
  \text{int g(int x)} \{ \text{return f(x) - 1;} \}
  \]

- Need environment containing at least
  
  \[
  f: \text{int} \rightarrow \text{int}, g: \text{int} \rightarrow \text{int}
  \]

  when checking both $f$ and $g$

- Two-pass approach:
  
  \[
  \text{Scan top level of AST picking up all function signatures} \\
  \text{and creating an environment binding all global identifiers} \\
  \text{Type-check each function individually using this global environment}
  \]

How to Check Return?

\[
A \vdash E : T
\]

\[
A \vdash \text{return } E : \text{unit}
\]

- A return statement produces no value for its containing context to use
- Does not return control to containing context
- Suppose we use type unit...
- ...then how to make sure the return type of the current function is $T$?
Put Return in the Symbol Table

- Add a special entry \{ return\_fun : T \} when we start checking the function "fun", look up this entry when we hit a return statement.
- To check $T$, `fun (T_1, a_1, ..., T_n, a_n) { return S; }` in environment $A$, need to check:

$$A, a_i : T_i, ..., a_n : T_n, \text{return\_fun : T} \vdash S : T, \text{(return)}$$

$$A \vdash E : T \quad \text{return\_fun : T} \in A \quad \text{(return)}$$

$$\quad A \vdash \text{return} E : \text{unit}$$

Static Semantics Summary

- Static semantics = formal specification of type-checking rules
- Concise form of static semantics: typing rules expressed as inference rules
- Expression and statements are well-formed (or well-typed) if a typing derivation (proof tree) can be constructed using the inference rules