Type Inference Systems

• Type inference systems are a declarative formal system used to define typings for legal programs in a language

• Type inference systems are to type-checking:
  – As regular expressions are to lexical analysis
  – As context-free grammars are to syntax analysis

• Type inference systems are examples of the more general notion: natural semantics
Type Judgments

• The type judgment:
  \[ |- E : T \]
  is read:
  "E is a well-typed construct of type T"

• Type checking program P is demonstrating the validity of the type judgment \[ |- P : T \] for some type T

• Sample valid type judgments for program fragments:
  \[ |- 2 : \text{int} \]
  \[ |- 2 \ast (3 + 4) : \text{int} \]
  \[ |- \text{true} : \text{bool} \]
  \[ |- (\text{true} \ ? \ 2 : 3) : \text{int} \]
Deriving a Type Judgment

• Consider the judgment:

   \[ \vdash (b \ ? \ 2 \ : \ 3) : \text{int} \]

• What do we need in order to decide that this is a valid type judgment?

  • b must be a bool (\[ \vdash b : \text{bool} \])
  • 2 must be an int (\[ \vdash 2 : \text{int} \])
  • 3 must be an int (\[ \vdash 3 : \text{int} \])
Hypothetical Type Judgments

- The hypothetical type judgment
  \[ A |- E : T \]
  is read:
  “In type context A expression E is well-typed with type T”

- A type context is a mapping of identifiers to types (i.e., a symbol table)

- Sample valid hypothetical type judgments:
  \[ b: \text{bool} \mid |- b : \text{bool} \]
  \[ |- 2 + 2 : \text{int} \]
  \[ b: \text{bool}, x: \text{int} \mid |- (b \ ? \ 2 : x) : \text{int} \]
  \[ b: \text{bool}, x: \text{int} \mid |- b : \text{bool} \]
  \[ b: \text{bool}, x: \text{int} \mid |- 2 + 2 : \text{int} \]

- Type checking program P is demonstrating the validity of \[ A |- P : T \]
  for some type T and the language’s standard environment A
Deriving a Type Judgment

• To show:

\[
\begin{align*}
\text{b: bool, x: int} & \vdash (\text{b ? 2 : x}) : \text{int} \\
\end{align*}
\]

• Need to show:

\[
\begin{align*}
\text{b: bool, x: int} & \vdash \text{b} : \text{bool} \\
\text{b: bool, x: int} & \vdash 2 : \text{int} \\
\text{b: bool, x: int} & \vdash x : \text{int} \\
\end{align*}
\]
General Rule

• For any type environment A, expressions E, E₁ and E₂, the judgment

\[ A \vdash (E \ ? \ E₁ \ : \ E₂) : T \]

is valid if:

\[ A \vdash E : \text{bool} \]
\[ A \vdash E₁ : T \]
\[ A \vdash E₂ : T \]
Inference Rule Schema

Premises (a.k.a., antecedent)

\[
A |- E: \text{bool} \quad A |- E_1: T \quad A |- E_2: T
\]

Conclusion (a.k.a., consequent)

\[
A |- (E \ ? \ E_1 : E_2) : T
\]

- Holds for any choice of \( A, E, E_1, E_2, \) and \( T \)
- An inference rule schema defines an infinite number of inference rules
Axioms

• An axiom is an inference rule (schema) with no premises

\[
A \vdash \text{true} : \text{bool}
\]
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** are to type inference systems as productions are to context-free grammars

• **Type judgments** are to type inference systems as nonterminals are to context-free grammars

• **Type checking** is an attempt to prove that a type judgment is \( A \mid - E : T \) is valid
Meaning of Inference Rule

- Inference rule says:
  given that the antecedent judgments are derivable
  - with a uniform substitution for meta-variables (i.e., A, E₁, E₂)
  then the consequent judgment is derivable
  - with the same uniform substitution for the meta-variables

\[
\begin{align*}
A \mid- E₁ & : \text{int} \\
A \mid- E₂ & : \text{int} \\
\hline
A \mid- E₁ + E₂ & : \text{int}
\end{align*}
\]
Proof Tree

- A construct is well-typed if there exists a type derivation for a type judgment for the construct.

- **Type derivation** is a proof tree where all the leaves are axioms.

- Example: if $A_1 = b: \text{bool}$, $x: \text{int}$, then:

\[
\begin{align*}
A_1 & \vdash b: \text{bool} & A_1 & \vdash 2: \text{int} & A_1 & \vdash 3: \text{int} \\
\hline
A_1 & \vdash \neg b: \text{bool} & A_1 & \vdash 2+3: \text{int} & A_1 & \vdash x: \text{int} \\
\hline
A_1 & \vdash (\neg b \ ? \ 2+3 \ : \ x) : \text{int}
\end{align*}
\]
Proof Tree, cont.

• Axioms are analogous to production with epsilon on the right hand side

• A complete proof of $A \vdash E : T$ is like a derivation of epsilon from $A \vdash E : T$
Type Judgments for Statements

• Statements that have no value are said to have type `void`, i.e., judgment

  \[ \vdash S : \text{void} \]

  means “\(S\) is a well-typed statement with no result type”

• ML uses `unit` instead of `void`
While Statements

• Rule for while statements:

\[
\begin{align*}
A \vdash E : \text{bool} \\
A \vdash S : T \\
\frac{}{A \vdash \text{while} (E) S : \text{void}} \quad \text{(while)}
\end{align*}
\]
Assignment (Expression) Statements

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

\[ \frac{A \vdash E_3 : T}{A \vdash E_2 : \text{int}} \quad \frac{A \vdash E_1 : \text{array}[T]}{A \vdash E_1[E_2] = E_3 : T} \quad \text{(array-assign)} \]
Sequence Statements

• Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:

\[
egin{align*}
A |- S_1 : T_1 \\
A |- (S_2 ; \ldots ; S_n) : T_n \\
\hline
A |- (S_1 ; S_2 ; \ldots ; S_n) : T_n
\end{align*}
\]

(sequence)
Identifier Declaration List

• What about variable declarations (with initialization)?
• Declarations add entries to the type environment in which the scope of the declared variable must type check

\[
\begin{align*}
A |- E : T \\
A, id : T |- (S_2 ; \ldots ; S_n) : T' \\
\hline \\
A |- (id : T = E ; S_2 ; \ldots ; S_n) : T'
\end{align*}
\]
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 \( E(E_1, \ldots, E_n) \)?

\[
\begin{align*}
A & \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \\
A & \vdash E_i : T_i \quad (i \in 1..n) \\
\hline
A & \vdash E(E_1, \ldots, E_n) : T_r
\end{align*}
\]
Function Declarations

• Consider a function declaration of the form

\[
T_r \ f (T_1 a_1, \ldots, T_n a_n) \ \{ \ E; \ \}
\]

• The body of the function must type check in an environment containing the type bindings for the formal parameters

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

\[
A \vdash T_r \ f (T_1 a_1, \ldots, T_n a_n) \ \{ \ E; \ \} : \text{void}
\]
But what about recursion?

• Example:

```c
int fact(int x) {
    if (x==0) return 1;
    else return x * fact(x - 1);
}
```

• Need to prove: $A \vdash x * \text{fact}(x-1) : \text{int}$

where: $A = \{ \text{fact: int} \to \text{int}, x : \text{int} \}$
And mutual recursion?

• Example:
  ```c
  int f(int x) { return g(x) + 1; }
  int g(int x) { 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 needed:
  
  - First pass: collect all function signatures into a type environment \( A \)
  
  - Second pass: type-check each function declaration using this global environment \( A \)
  
  - How to express this with type inference schema is left as an exercise
How to Check Return?

A \vdash E : T \quad (\text{return1})
A \vdash \text{return} E : \text{void}

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

- Add a special entry \{ return\_fun : T \} when we start checking the function “f”, look up this entry when we hit a return statement.

- To check \( T_r f (T_1 a_1, \ldots, T_n a_n) \) \{ return S; \} in environment A, need to check:

\[
A, a_1 : T_1, \ldots, a_n : T_n, \text{return\_f} : T_r \vdash E : T_r
\]

A \vdash T_r f (T_1 a_1, \ldots, T_n a_n) \{ E; \} : \text{void}

A, \text{return\_f} : T \vdash E : T

A, \text{return\_f} : T \vdash \text{return} E : \text{void}
Static Semantics Summary

• Type inference system = formal specification of typing 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