

**CS321 F'04 Lecture Notes**  
**Lecture 14**

## Syntax-Directed Type Checking

Consider a simple language of declarations, statements, and expressions.

$$P \rightarrow D ; S \quad \{ S.env = D.env; \}$$

Actions for declarations synthesize environment attributes:

$$\begin{array}{ll} D \rightarrow \epsilon & \{ D.env := empty \} \\ D \rightarrow id : T_1 ; D_1 & \{ D.env := \\ & \quad extend(D_1.env, binding(id, T_1.type)) \} \\ T \rightarrow bool & \{ T.type := boolean \} \\ T \rightarrow int & \{ T.type := integer \} \\ T \rightarrow array[num] of T_1 & \{ T.type := array(num.val, T_1.type) \} \\ T \rightarrow T_1^* & \{ T.type := pointer(T_1.type) \} \end{array}$$

Actions for expressions **check** for compatible operands and **synthesize** attribute type:

$$\begin{array}{ll} E \rightarrow num & \{ E.type := integer \} \\ E \rightarrow id & \{ E.type := lookup(E.env, id) \} \\ E \rightarrow E_1 \text{ div } E_2 & \{ E_1.env = E.env; E_2.env = E.env; \\ & \quad \text{if } E_1.type = integer \\ & \quad \quad \text{and } E_2.type = integer \text{ then} \\ & \quad \quad E.type := integer \\ & \quad \text{else} \\ & \quad \text{issue type error} \} \end{array}$$

## More Expressions

$$\begin{aligned}
 E \rightarrow E_1 [ E_2 ] & \{ E_1.env = E.env; E_2.env = E.env; \\
 & \quad \text{if } E_1.type = \text{array}(I,T) \\
 & \quad \quad \text{and } E_2.type = \text{integer} \text{ then} \\
 & \quad \quad \quad E.type := T \\
 & \quad \quad \text{else issue type error} \} \\
 E \rightarrow *E_1 & \{ E_1.env = E.env; \\
 & \quad \text{if } E_1.type = \text{pointer}(T) \text{ then} \\
 & \quad \quad E.type = T \\
 & \quad \text{else issue type error} \} \\
 E \rightarrow E_1 < E_2 & \{ E_1.env = E.env; E_2.env = E.env; \\
 & \quad \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \text{ then} \\
 & \quad \quad E.type := \text{boolean} \\
 & \quad \quad \text{else issue type error} \} \\
 E \rightarrow E_1 \text{ or } E_2 & \{ E_1.env = E.env; E_2.env = E.env; \\
 & \quad \text{if } E_1.type = \text{boolean} \\
 & \quad \quad \text{and } E_2.type = \text{boolean} \text{ then} \\
 & \quad \quad \quad E.type := \text{boolean} \\
 & \quad \quad \text{else issue type error} \} \\
 E \rightarrow E_1 = E_2 & \{ E_1.env = E.env; E_2.env = E.env; \\
 & \quad \text{if } (E_1.type = \text{boolean} \text{ or } E_1.type = \text{integer}) \\
 & \quad \quad \text{and } E_1.type = E_2.type \text{ then} \\
 & \quad \quad \quad E.type := \text{boolean} \\
 & \quad \quad \text{else issue type error} \}
 \end{aligned}$$

As shown, these actions abort immediately if there is an error. Alternatively, can continue to look for more errors; in this case, synthesized value should be chosen to avoid cascades of messages from a single mistake (e.g., last three always return *boolean*).

## Checking Statements

In most languages, statements don't have a type, so no point in synthesizing an attribute. Actions just check component types:

$$S \rightarrow \text{id} := E_1 \quad \left\{ \begin{array}{l} E_1.\text{env} = S.\text{env}; \\ \text{if } E_1.\text{type} \neq \text{lookup}(S.\text{env}, \text{id}) \text{ then} \\ \text{issue type error} \end{array} \right\}$$

*(Must also check that  $\text{id}$  is an l-value that can be assigned into.)*

$$S \rightarrow \text{if } E_1 \text{ then } S_1 \quad \left\{ \begin{array}{l} E_1.\text{env} = S.\text{env}; S_1.\text{env} = S.\text{env}; \\ \text{if } E_1.\text{type} \neq \text{boolean} \text{ then} \\ \text{issue type error} \end{array} \right\}$$

$$S \rightarrow S_1 ; S_2 \quad \left\{ S_1.\text{env} = S.\text{env}; S_2.\text{env} = S.\text{env}; \right\}$$

## Procedure/Function Definitions and Calls

Can describe type of function as  $type_1 \times type_2 \times \dots \times type_n \rightarrow type$

$$D \rightarrow id ( F_1 ) : T_1 ; D_1 \{ D.env := extend(D_1.env, \\ binding(id, F_1.type \rightarrow T_1.type)) \}$$

$$F \rightarrow id : T_1 \{ F.type := T_1.type \}$$

$$F \rightarrow id : T_1 , F_1 \{ F.type := T_1.type \times F_1.type \}$$

$$E \rightarrow id ( A_1 ) \{ A_1.env = E.env; \\ \text{if } lookup(E.env, id) = T_1 \rightarrow T_2 \text{ then} \\ \quad \text{if } A_1.type = T_1 \text{ then} \\ \quad \quad E.type := T_2 \\ \quad \text{else issue type error} \\ \text{else issue type error} \}$$

$$A \rightarrow E_1 \{ E_1.env = A.env; \\ A.type := E_1.type \}$$

$$A \rightarrow E_1 , A_1 \{ E_1.env = A.env; \\ A.type := E_1.type \times A_1.type \}$$

## Type Conversions

**Implicit** conversions (or “**coercions**”) occur as a result of applying semantic rules of the language, e.g., perhaps evaluating

$$r + i$$

where  $r$  is a real and  $i$  is an integer, causes implicit conversion of the fetched value of  $i$  to a real before the addition. Note that there's no effect on the permanent contents of  $i$ .

Implicit conversions complicate type-checking (as well as code generation), e.g.:

$$E \rightarrow E_1 + E_2 \quad \left\{ \begin{array}{l} E_1.env = E.env; E_2.env = E.env; \\ \textit{case } (E_1.type, E_2.type) \textit{ of} \\ \quad (integer, integer): E.type := integer \\ \quad (integer, real): \\ \quad (real, integer): \\ \quad (real, real): E.type := real \\ \quad \textit{otherwise: issue type error} \end{array} \right\}$$

**Explicit** conversions (programmer-specified) include operators like `ord` and `chr` in Pascal or casts in C. They usually appear as expression nodes in the AST, and naturally have their own type-checking rules.

It can be convenient to convert implicit coercions in the source program into explicit coercions in the AST.