CS541 Lecture on Typing Objects and Classes By: Hridesh Rajan

A. Domain of Type System

Type checking uses the type attributes defined in Figure 1. (These use some of the notation and ideas from Schmidt’s book [1].)

\[ \theta ::= \text{"type attributes"} \]
\[ \text{prog } t \quad \text{"program/top-level decl."} \]
\[ \text{OK in } c \quad \text{"methods"} \]
\[ | \text{var } t \quad \text{"var/formal/field"} \]
\[ | \text{exp } t \quad \text{"expression"} \]
\[ \tau ::= c \mid \top \mid \bot \quad \text{"class type exps"} \]
\[ \pi, \Pi ::= \{ I : \theta_I \} \in K, \quad \text{"type environments"} \]
where \( K \) is finite, \( K \subseteq (L \cup \{\text{this}\} \cup V) \)

Figure 1: Type attributes.

B. Subtyping

The notation \( \tau' \preceq \tau \) means \( \tau' \) is a subtype of \( \tau \). It is the reflexive-transitive closure of the declared subclass relationships with the added facts that \( \top \) is a supertype of all class type expressions, and that \( \bot \) is a subtype of all class type expressions. The type \( \bot \) is used as the type of exceptions. This is formalized in Figure 2.

\[
\begin{array}{c}
\text{(BASIS)} \\
\text{(class } c \text{ extends } d(\ldots) \text{) } \in CT \\
\hline
\text{c } \preceq d
\end{array} \quad
\begin{array}{c}
\text{(REF)} \\
\tau \preceq \tau \\
\hline
\tau_1 \preceq \tau_2 \quad \tau_2 \preceq \tau_3 \\
\tau_1 \preceq \tau_3
\end{array} \quad
\begin{array}{c}
\text{(TRANS)} \\
\text{isClass}(c) \\
\hline
\tau_1 \preceq \tau_2 \\
\tau_1 \preceq \tau_3 \\
\tau_2 \preceq \tau_3 \\
\text{c } \preceq \top \\
\bot \preceq c
\end{array}
\]

Figure 2: Subtyping rules

C. Type Checking Rules for Top-level Declarations

The type checking rules for top-level declarations such as the program, class declaration, and method declarations are shown in Figure 3. The type checking rules are stated using a fixed class table (list of declarations) \( CT \), which can be thought of as an implicit (hidden) inherited attribute. This class table is used implicitly by many of the auxiliary functions. For ease of presentation, we assume that the names declared at the top level of a program are distinct and that the extends relation on classes is acyclic.

The rule (CHECK PROGRAM) says that the entire program type checks, if all the declarations type check and the expression \( e \) has type \( \text{exp } t \).

A class type checks, expressed as rule (CHECK CLASS), if all of its fields and methods definitions are well typed and the class extends a valid class type. When type checking a class we don’t allow overriding of super class fields. Thus we check that all the newly declared fields are not fields of its super class. Its super class \( d \) is defined in the Class Table.

The rule (CHECK METHOD) says that a method declaration type checks, if all the following constraints are satisfied. The return type is a class type. If all the parameters have their corresponding declared type, the body of the method has type \( t' \). \( t' \) has to be a subtype of \( t \).
(CHECK PROGRAM)
\[ (\forall i \in \{1..n\} :: \vdash decl_i : \text{OK}) \quad \vdash e : \text{exp} \quad t \]
\[ \vdash decl_1 \ldots decl_n e : \text{prog} \quad t \]

(CHECK CLASS)
\[ \text{isClass}(d) \]
\[ (\forall j \in \{1..m\} :: \vdash \text{meth}_j : \text{OK} \text{ in } c) \quad (\forall i \in \{1..n\} :: \text{isClass}(t_i) \wedge f_i \notin \text{dom(fieldsOf}(d))) \]
\[ \vdash \text{class } c \text{ extends } d \{ t_1 f_1 ; \ldots t_n f_n ; \text{meth}_1 \ldots \text{meth}_m \} : \text{OK} \]

(CHECK METHOD)
\[ \text{isType}(t) \]
\[ (\forall i \in \{1..n\} :: \text{isType}(t_i)) \quad \{ \text{var}_1 : \text{var } t_1 , \ldots , \text{var}_n : \text{var } t_n , \text{this : var } c \} \vdash e : \text{exp} t' \quad t' \preceq t \quad \text{class } c \text{ extends } d \{ \ldots \} \in CT \quad \text{override}(m, d, t_1 \times \cdots \times t_n \rightarrow t) \]
\[ \vdash t m(t_1 \text{var}_1 , \ldots , t_n \text{var}_n \{ e \}) : \text{OK} \text{ in } c \]

Figure 3: Type-checking rules for Top-level Declarations.

D. Type Checking Rules for Object-oriented Expressions

The type system for the expressions are shown in figure 3. Most of the rules are straightforward.

The rule (T-NEW) for new expression has type of class being constructed, if the class c has been properly declared.

The rule (T-GET) for field access expression returns the type of the field of the class. The rule (T-SET) for field assignment expression type checks if the following constrains are satisfied. The receiver expression is of a class type. The assignment expression e’ has a type which is a subtype of the type of the field.

The rule (T-VAR) for variable read checks that the var was put in the environment. The rule (T-DEF) for local variable declaration type checks if the initial expression is of a subtype of the type of the new variable.

The rule (T-SEQUENCE) for sequence expressions states that the sequence expression has the type of second expression.

The rule (T-CAST) for cast expressions says that the cast type must be a class type.

The (T-NULL) null expression says that null could be treated as any type.

The rule (T-UNDER) for under expressions say that it has same type as the inner expression.

The rule (T-CALL) for method call expression says that this expression type checks if the following constraints are satisfied. First the method can be found in the class table and this method is declared either in its own class or its super class. Each actual argument expression is of subtype of corresponding parameter type. This method call expression will have the type as the return type of the method.

References

Figure 4: Type-checking rules for OO features.