1 Introduction

This document presents the formal specification of the typing rules for SOOL. It is a companion to the Project 2 description, which describes the project of implementing this formal specification in a type checker.

2 SOOL abstract syntax

The SOOL type system is defined in terms of an abstract syntax, which elides many of the syntactic details found in the concrete syntax. We start by defining conventions for the various kinds of identifiers in SOOL.

\[
\begin{align*}
\alpha & \in \text{VAR} & \text{Variable identifiers} \\
\nu & \in \text{MEMBVAR} & \text{Class variable identifiers} \\
f & \in \text{MEMBFUN} & \text{Class/interface function identifiers} \\
x & \in \text{MEMB} = \text{MEMBVAR} \cup \text{MEMBFUN} & \text{Class member identifiers} \\
C & \in \text{CLS} \cup \{\text{obj}\} & \text{Class types} \\
I & \in \text{IFC} \cup \{\text{objI}\} & \text{Interface types} \\
n & \in \text{INT} & \text{Integer constants} \\
r & \in \text{STR} & \text{String constants}
\end{align*}
\]

Note that we add the distinguished identifiers \texttt{obj} and \texttt{objI} to the sets of class and interface identifiers (respectively). Adding these special identifiers allows us to simplify the abstract syntax and typing rules to only deal with class/interface declarations that extend classes/interfaces. The abstract syntax of SOOL is given in Figure 1.

We assume that the abstract syntax satisfies certain syntactic restrictions.

- Class and interface names in a program are distinct.
- A declaration of subclass class \texttt{C} that is derived from a class \texttt{B}

\[
\text{class C (\ldots) extends B (\ldots) \{ \ldots \}}
\]

can only appear after the declaration of \texttt{B}. This restriction ensures that the class hierarchy will be acyclic. Also note that this restriction does not prevent class \texttt{B} from referencing class \texttt{C}.
- A declaration of an interface \texttt{J} that extends an interface \texttt{I}
\[ p ::= d \]
\[ d ::= \text{class } C'(a_1 : \tau_1, \ldots, a_n : \tau_n) \text{ extends } C(e_1, \ldots, e_k) \{ vd ; fd \} \]
\[ \text{interface } I' \text{ extends } I \{ fs \} \]
\[ d_1 d_2 \]
\[ vd ::= \text{var } v : \tau = e \]
\[ vd_1 vd_2 \]
\[ fd ::= \text{meth } f (a_1 : \tau_1, \ldots, a_n : \tau_n) \rightarrow \theta \{ s \} \]
\[ \text{override meth } f (a_1 : \tau_1, \ldots, a_n : \tau_n) \rightarrow \theta \{ s \} \]
\[ fd_1 fd_2 \]
\[ fs ::= \text{meth } f (\tau_1, \ldots, \tau_n) \rightarrow \theta \]
\[ fs_1 fs_2 \]
\[ s ::= s_1 ; s_2 \]
\[ \text{var } x = e \]
\[ \text{while } e \{ s \} \]
\[ \text{if } e \text{ then } \{ s_1 \} \text{ else } \{ s_2 \} \]
\[ \text{return } e \]
\[ \text{return} \]
\[ m ::= e_2 \]
\[ x ::= e \]
\[ m(e_1, \ldots, e_n) \]
\[ m ::= e.x \]
\[ e!x \]
\[ e ::= (e_1 \odot e_2) \]
\[ m \]
\[ e?x \]
\[ e! \]
\[ m(e_1, \ldots, e_n) \]
\[ e?x(e_1, \ldots, e_n) \]
\[ C(e_1, \ldots, e_n) \]
\[ a \]
\[ u \mid r \mid \text{true} \mid \text{false} \]
\[ \text{nil } T \]

Figure 1: SOOL abstract syntax
can only appear after the declaration of $I$.

- The parameter names of a class and of a member-function are distinct.
- The member-variable names of a class are distinct.
- The member-function names of a class or interface definition are distinct.
- Integer literals are restricted to the range $-2^{62}..2^{62} - 1$ (i.e., representable as a 63-bit 2’s-complement integer).
- Functions with non-void return type must have a `return` statement on every possible execution path.\(^1\)

### 3 Definitions

This document uses a fair number of definitions and notational conventions. We describe these in this section.

#### 3.1 Semantic objects

We use the following notation for sets of member variables:

$$
\begin{align*}
\mathcal{V} & \in 2^{\text{MEMBVAR}} & \text{Sets of variable identifiers} \\
\mathcal{F} & \in 2^{\text{MEMBFUN}} & \text{Sets of function identifiers}
\end{align*}
$$

The following grammar defines the abstract syntax of the various kinds of “types” that we use to define the static semantics of SOOL:

$$
\begin{align*}
\iota & ::= \text{bool} \mid \text{int} \mid \text{string} & \text{Primitive types} \\
T & ::= C \mid I \mid \iota & \text{Type constructors} \\
\tau & ::= T \mid T? & \text{Type expressions (Typ)} \\
\theta & ::= \tau \mid \text{void} & \text{Return types} \\
\sigma & ::= (\tau_1, \ldots, \tau_n) \rightarrow \theta & \text{Member-function signature (Spec)} \\
\mu & ::= \text{TYP} \cup \text{SPEC} & \text{Class/interface member type} \\
\Sigma & ::= (\tau_1, \ldots, \tau_n) \{ C; v : \tau_v \in \mathcal{V}; f : \sigma_f \in \mathcal{F} \} & \text{Class signature (CLSSIG)} \\
\Upsilon & ::= \{ f : \sigma_f \in \mathcal{F} \} & \text{Interface signature (IFCSIG)}
\end{align*}
$$

Because SOOL does not have type variables or type renaming, equality between types is purely structural; *i.e.*, two types are equal if they are syntactically equal. We write $\vdash \tau = \tau'$ (likewise, $\vdash \sigma = \sigma'$, etc.) to assert that $\tau$ and $\tau'$ are equal.\(^2\)

The class signature

$$
(\tau_1, \ldots, \tau_n) \{ C; v : \tau_v \in \mathcal{V}; f : \sigma_f \in \mathcal{F} \}
$$

\(^1\)This condition was omitted from the original problem description, so you are not responsible for checking this property in Project 2.

\(^2\)If you see “$=$” used without the turnstile, then it is a definition.
describes a class that is derived from class \( C \). The \( \tau_1, \ldots, \tau_n \) are the types of the class parameters; \( \mathcal{V} \) is the set of member-variable names in the class (including inherited variables), and \( \mathcal{F} \) is the set of member functions in the class (including inherited functions). We use the notation \( v : \tau_v^v \in \mathcal{V} \) to mean that for each class variable \( v \in \mathcal{V} \), the variable \( v \) has the type \( \tau_v \). We use similar notation for the member functions of a class or interface, and we use similar notation to describe the typing of a sequence of argument expressions in the rules below.

Environments are finite maps from identifiers to types. We write \( \{ x \mapsto y \} \) to denote the single-тон finite map that maps \( x \) to \( y \). If \( A \) is a finite map, then we write \( \text{dom}(A) \) for the domain of \( A \) and we write \( A(x) \) for the application of \( A \) to \( x \). If \( A \) and \( B \) are finite maps, then we write \( A \pm B \) for the extension of \( A \) by \( B \), which denotes a finite map with the following behavior:

\[
(A \pm B)(x) = \begin{cases} B(x) & \text{if } x \in \text{dom}(B) \\ A(x) & \text{otherwise} \end{cases}
\]

We define three kinds of environments for classes, interfaces, and local variables. We also define notation for the triple of all three environments.

\[
CE \in \text{CENV} = \text{CLS} \xrightarrow{\text{fn}} \text{CLS} \text{SIG} \quad \text{Class environments}
\]

\[
IE \in \text{IEENV} = \text{IFC} \xrightarrow{\text{fn}} \text{IFC} \text{SIG} \quad \text{Interface environments}
\]

\[
VE \in \text{VENV} = \text{VAR} \xrightarrow{\text{fn}} \text{TYP} \quad \text{Variable environments}
\]

\[
E = \langle CE, IE, VE \rangle \in \text{ENV} = \text{CENV} \times \text{IEENV} \times \text{VENV} \quad \text{Environments}
\]

If \( E = \langle CE, IE, VE \rangle \), then the notation \( \text{CE of } E \) denotes \( CE \), and similarly for the \( IE \) and \( VE \) components of \( E \). We will also write \( E \pm A \) for the extension of one of the components of \( E \), where the context makes it clear which component is involved (the other components are not extended).

### 3.2 Additional notation

To simplify the rules, we define the notation \( \tau? \) as follows:

\[
\tau? = \begin{cases} \tau & \text{if } \tau \text{ is } T? \text{ (for some } T) \\ T? & \text{if } \tau \text{ is } T \text{ (for some } T) \end{cases}
\]

\((i.e., T?? = T?)\).

In order to specify the typing rules for class and interface bodies, we need to collect together member function and member variable signatures. We use the following notation for these collections:

\[
\hat{\mathcal{F}} = \{ f : \sigma_f \hat{\in} \mathcal{F} \} \quad \hat{\mathcal{V}} = \{ v : \tau_v^v \in \mathcal{V} \}
\]

\(\hat{\mathcal{F}}\) and \(\hat{\mathcal{V}}\) are finite maps (or environments) with domains \( \mathcal{F} \) and \( \mathcal{V} \) respectively. We also use \( \hat{\mathcal{F}} \) and \( \hat{\mathcal{V}} \) to represent the members of a class signature \( \Sigma = (\tau_1, \ldots, \tau_n) \{ C; \hat{\mathcal{V}}; \hat{\mathcal{F}} \} \) or interface signature \( \Upsilon = \{ \hat{\mathcal{F}} \} \). We define the methods of a class (or interface) with respect to an environment as
Program typing (see Section 5.1)

Declarations typing (see Section 5.2)

Member-variable declaration typing (see Section 5.3)

Member-function declaration typing (see Section 5.4)

Member-function specification typing (see Section 5.6)

Statement typing (see Section 5.6)

Required member selection (see Section 5.7)

Member typing (see Section 5.8)

Expression subtyping (see Section 5.9)

Expression typing (see Section 5.10)

Figure 2: Typing judgments for SOOL

follows:

\[
\begin{align*}
Funs_{\text{obj}}(E) & = \emptyset \\
Funs_{C}(E) & = \hat{\mathcal{F}} \text{ where } C \in \text{dom}(CE of E) \text{ and } \text{(CE of E)}(C) = (\tau_1, \ldots, \tau_n) \{ \hat{C'}; \hat{\mathcal{V}}; \hat{\mathcal{F}} \} \\
Funs_{\text{objI}}(E) & = \emptyset \\
Funs_{I}(E) & = \hat{\mathcal{F}} \text{ where } I \in \text{dom}(IE of E) \text{ and } \text{(IE of E)}(I) = \{ \hat{\mathcal{F}} \}
\end{align*}
\]

We also define the parameter signature of a class with respect to an environment as follows:

\[
\begin{align*}
\text{Params}_{\text{obj}}(E) & = \text{empty} \\
\text{Params}_{C}(E) & = \tau_1, \ldots, \tau_n \text{ where } C \in \text{dom}(CE of E) \text{ and } \text{(CE of E)}(C) = (\tau_1, \ldots, \tau_n) \{ \hat{C'}; \hat{\mathcal{V}}; \hat{\mathcal{F}} \}
\end{align*}
\]

For primitive types \(\iota \in \{\text{bool, int, string}\}\), we write \(\text{InterfaceOf}(\iota)\) to denote the corresponding interface type as defined in Section 3.3 of the Project Overview.

3.3 Judgment forms

The SOOL type system is defined by a collection of various judgement forms, which are summarized in Figure 2. The general form of a judgment is “\(\text{context} \vdash \text{term} : \text{property}\)”, which can be read as “\(\text{term}\) has \text{property} in \text{context}.”

4 Judgments about types

Before presenting the typing rules for the abstract syntax, we define the following judgments about types:

\[
\begin{align*}
E \vdash \tau \text{ ok} & \quad \text{The type } \tau \text{ is well-formed} \\
CE \vdash C' \prec C & \quad \text{Class } C' \text{ inherits from } C \\
E \vdash C \prec I & \quad \text{Class } C \text{ implements interface } I \\
E \vdash \tau' \preceq \tau & \quad \text{Type } \tau' \text{ is a subtype of } \tau
\end{align*}
\]
These have a similar form to the typing judgments for terms that are shown in Figure 2, but they describe properties of types. As above, the environment on the left-hand-side of the turnstile ("⊢") provides the context for making the judgment. These judgments are defined by inference rules in the remainder of this section.

### 4.1 Well-formedness of types

Informally, a type is well formed with respect to an environment $E$ if the class and interface names in the type are defined in the environment. The following rules make this definition precise. The primitive types are well formed.

- $E \vdash \iota \text{ ok}$

Class types are well formed if they are in the class environment:

- $C \in \text{dom}(CE \text{ of } E)$
  
  $E \vdash C \text{ ok}$

Similarly for interface types:

- $I \in \text{dom}(IE \text{ of } E)$
  
  $E \vdash I \text{ ok}$

Lastly, option types are well formed if their type constructor is well formed.

- $E \vdash T \text{ ok}$
- $E \vdash T? \text{ ok}$

We lift the well-formedness judgement to return types ($\theta$) and member-function signatures ($\sigma$) in the obvious way.

### 4.2 The inherits relation

Classes inherit from their immediate superclass:

- $C' \in \text{dom}(CE)$
  
  $CE(C') = (\tau_1, \ldots, \tau_n)\{C; \forall v \in \hat{V}; f : \sigma f \in \hat{F}\}$

and the inherits relation is transitive

- $CE \vdash C' \ll C''$
- $CE \vdash C'' \ll C$

$CE \vdash C' \ll C''$

### 4.3 The implements relation

A class $C$ implements an interface $I$ if it provides all of the member functions declared in the interface at the same types.

- $C \in \text{dom}(CE \text{ of } E)$
  
  $CE(C) = (\tau_1, \ldots, \tau_n)\{C; \forall \hat{V}; \hat{F}_C\}$

- $I \in \text{dom}(IE \text{ of } E)$
  
  $IE(I) = \{\hat{F}_I\}$

- $\text{dom}(\hat{F}_I) \subseteq \text{dom}(\hat{F}_C)$
  
  $E \vdash C \prec I$
4.4 The subtyping relation

A well-formed type $\tau$ is always a subtype of itself.

\[
\frac{\vdash \tau}{\vdash \tau \preceq \tau}
\]

and a well-formed type is always a subtype of its option type.

\[
\frac{\vdash T u \ ok}{\vdash T \preceq T ?}
\]

Primitive types are subtypes of the interfaces that they implement.

\[
\frac{I = \text{InterfaceOf}(\iota)}{\vdash \iota \preceq I}
\]

Classes are subtypes of their super-classes.

\[
\frac{(CE \ of \ E) \vdash C' \triangleleft C}{\vdash C' \preceq C}
\]

Classes are also subtypes of the interfaces that they implement.

\[
\frac{(CE \ of \ E) \vdash C < I}{\vdash C \preceq I}
\]

Extending an interface with additional member-function specifications creates a subtype.

\[
\frac{\text{FunsOf}_E(I') = \hat{F}' \quad \text{FunsOf}_E(I) = \hat{F} \quad \text{dom}(\hat{F}) \subseteq \text{dom}(\hat{F}')}{\vdash I' \preceq I}
\]

Note that the two interfaces agree on the types of their common member functions; this form of subtyping is called width subtyping, since it does not involve subtyping on the types of the member functions. Lastly, the subtyping relation is transitive

\[
\frac{\vdash \tau' \preceq \tau'' \quad \vdash \tau'' \preceq \tau}{\vdash \tau' \preceq \tau}
\]

5 Typing rules

The judgments are defined in a syntax directed set of typing rules, which means that each syntactic form has an associated typing rule.

5.1 Program typing

The typing rule for programs is deceptively simple.

\[
\frac{E_0 \vdash E \quad E_0 \cup E \vdash d : E}{E_0 \vdash d \ ok}
\]

The appearance of $E$ on both the left-hand and right-hand sides of the turnstile may seem strange, but it accounts for the fact that declarations in a SOOL program are allowed to be mutually recursive. The environment $E_0$ is the SOOL Initial Basis extended with bindings for $\text{obj}$ and $\text{objI}$ (see Section 6).
5.2 Declaration typing

The typing rule for class declarations is the most complicated rule in the system, since there are many pieces that have to be specified. We need to check that the superclass is defined and that the superclass-initialization arguments have the correct type. We also need to derive the types of the member declarations and check that they do not conflict with the superclass’s members.

\[ E \vdash \tau_i \text{ ok}_1 \leq i \leq n \quad C \in \text{dom}(CE \text{ of } E) \quad (CE \text{ of } E)(C) = (\tau_1', \ldots, \tau_k') \{ C; \hat{V}; \hat{F} \} \]

\[ E' = E \pm \{ a_i \mapsto \tau_i | 1 \leq i \leq n \} \quad E' \vdash e_i \leq \tau_i^{1 \leq i \leq k} \quad E' \vdash \text{vd} : \hat{V}' \]

\[ E \vdash \text{self} \mapsto C', C \vdash \text{fd} : \hat{F}' \]

\[ \text{dom}(\hat{V}) \cap \text{dom}(\hat{V}') \quad \text{dom}(\hat{F}) \cap \text{dom}(\hat{F}') \quad \Sigma = (\tau_1, \ldots, \tau_n) \{ C; \hat{V} \cup \hat{V}'; \hat{F} \cup \hat{F}' \} \]

\[ E \vdash \text{class } C'(a_1 : \tau_1, \ldots, a_n : \tau_n) \text{ extends } C(e_1, \ldots, e_k) \{ \text{vd}; \text{fd} \} : \{ \{ C' \Rightarrow \Sigma \}, \emptyset, \emptyset \} \]

5.3 Member-variable-declaration typing

A member-variable definition must be checked for well-formedness of the declared type \( \tau \) and that the right-hand-side expression is a subtype of \( \tau \).

\[ E \vdash \tau \text{ ok} \quad E, C \vdash e \leq \tau \]

\[ E, C \vdash \text{var } v : \tau = e : \{ v : \tau_v | v \in \{ \} \} \]

The environments from sequences of variable declarations must be disjoint and are combined.

\[ E, C \vdash \text{vd}_1 : \hat{V}_1 \quad E, C \vdash \text{vd}_2 : \hat{V}_2 \quad \text{dom}(\hat{V}_1) \cap \text{dom}(\hat{V}_2) \]

\[ E, C \vdash \text{vd}_1 \text{ vd}_2 : \hat{V}_1 \cup \hat{V}_2 \]

5.4 Member-function-declaration typing

The rules for typing member-function declarations use a context that includes the environment \( E \), the enclosing class \( C' \), and the superclass of the enclosing class \( C \) (which may be \text{obj}).

For a non-overriding function definition, we check that the function was not defined by its superclass and that the function’s body type checks.

\[ f \notin \text{dom}(\text{FunsOf}(E)(C)) \quad E \pm \{ a_i \mapsto \tau_i | 1 \leq i \leq n \}, C', \theta \vdash s : E' \quad \sigma_f = (\tau_1, \ldots, \tau_n) \Rightarrow \theta \]

\[ E, C', C \vdash \text{meth } f (a_1 : \tau_1, \ldots, a_n : \tau_n) \Rightarrow \theta \{ s : \{ f : \sigma_f^{\in \{ f \}} \} \}
\]
For overriding definitions, we need to check that the superclass defines the function and that the types agree. The resulting environment is empty, since the function’s signature is already inherited from the superclass.

\[
f \in \text{dom}(\text{FunsOf}_E(C)) \quad \text{FunsOf}_E(C)(f) = \sigma_f \quad \vdash \sigma_f = (\tau_1, \ldots, \tau_n) \to \theta
\]

\[
E \vdash \text{override meth } f (a_1 : \tau_1, \ldots, a_n : \tau_n) \to \theta \{ s \} : \{ \}
\]

The environments from sequences of function declarations must be disjoint and are combined.

\[
E, C', C \vdash \text{fd}_1 : \hat{F}_1 \quad E, C', C \vdash \text{fd}_2 : \hat{F}_2 \quad \text{dom}(\hat{F}_1) \cap \text{dom}(\hat{F}_2) = \emptyset
\]

\[
E, C', C \vdash \text{fd}_1  \text{ fd}_2 : \hat{F}_1 \cup \hat{F}_2
\]

### 5.5 Member-function-specification typing

The rule for typing a member-function specification requires that the function signature \( \sigma_f \) be well formed.

\[
\sigma_f = (\tau_1, \ldots, \tau_n) \to \theta \quad E \vdash \sigma_f \text{ ok}
\]

\[
E \vdash \text{meth } f (\tau_1, \ldots, \tau_n) \to \theta : \{ f : \sigma_f \in (f) \}
\]

The environments from sequences of function specifications must be disjoint and are combined.

\[
E \vdash \text{fs}_1 : \hat{F}_1 \quad E \vdash \text{fs}_2 : \hat{F}_2 \quad \text{dom}(\hat{F}_1) \cap \text{dom}(\hat{F}_2) = \emptyset
\]

\[
E \vdash \text{fs}_1 \text{ fs}_2 : \hat{F}_1 \cup \hat{F}_2
\]

### 5.6 Statement typing

Statements are type checked in a context that includes the enclosing class \( C \) and the declared return type \( \theta \) of the enclosing method. The rules for typing statements yield an environment enriched by any local variable declarations.

The rule for statements in sequence uses the environment produced by the first statement as the environment for the second one.

\[
E, C, \theta \vdash s_1 : E_1 \quad E_1, C, \theta \vdash s_2 : E_2
\]

\[
E, C, \theta \vdash s_1 : E_2
\]

The rule for variable declarations extends the environment with a binding for the variable.

\[
E, C \vdash e : \tau
\]

\[
E, C, \theta \vdash \text{var } x = e : E \pm \{ x \mapsto \tau \}
\]

The expression of a while loop must have type bool. Note that the variables declared in the body of the loop are not visible outside of the loop.

\[
E, C \vdash e : \text{bool} \quad E, C, \theta \vdash s : E'
\]

\[
E, C, \theta \vdash \text{while } e \{ s \} : E
\]

Conditional statements also require a boolean predicate.

\[
E, C \vdash e : \text{bool} \quad E, C, \theta \vdash s_1 : E_1 \quad E, C, \theta \vdash s_2 : E_2
\]

\[
E, C, \theta \vdash \text{if } e \text{ then } \{ s_1 \} \text{ else } \{ s_2 \} : E
\]
A return statement without an argument expression is valid if the enclosing method’s return type is void.

\[ E, C, \text{void} \vdash \text{return : } E \]

A return statement with an argument expression is valid if the type if \( e \) is a subtype of the enclosing method’s return type.

\[ E, C \vdash e \preceq \tau \]

\[ E, C, \tau \vdash \text{return } e : E \]

Assigning to a member variable requires that the right-hand-side expression \( e \) be a subtype of the left-hand-side member variable.

\[ E, C \vdash m : \tau \quad E, C \vdash e \preceq \tau \]

\[ E, C, \theta \vdash m := e : E \]

\[ x \in \text{dom}(\text{VE of } E) \quad (\text{VE of } E)(x) = \tau \quad E \vdash e \preceq \tau \]

\[ E, C, \theta \vdash x := e : E \]

\[ E, C \vdash m : (\tau_1, \ldots, \tau_n) \rightarrow \text{void} \quad E, C \vdash e_i \preceq \tau_{i}^{1 \leq i \leq n} \]

\[ E, C, \theta \vdash m(e_1, \ldots, e_n) : E \]

5.7 Required member selection

Selecting a member requires that the expression have a subtype of some type constructor \( T \), such that \( T \) has the member \( x \) with type \( \mu \) (we use subtyping here in case \( e \) has a primitive type).

\[ E, C \vdash e \preceq T \quad E, C, T \vdash x : \mu \]

\[ E, C \vdash e.x : \mu \]

We have a similar rule for when the expression has an optional type and we require that it be non-nil.

\[ E \vdash e \preceq T? \quad E, C, T \vdash x : \mu \]

\[ E, C \vdash e!x : \mu \]

5.8 Member typing

The member variables of a class are only visible in its member functions and in the member functions of its subclasses. This restriction is captured in the following definition:

\[ \text{VarsOf}_{E,C'}(C) = \begin{cases} \hat{V} & \text{if } C' = C \text{ or } E \vdash C' \ll C \\ \emptyset & \text{otherwise} \end{cases} \]

where \( C \in \text{dom}(\text{CE of } E) \) and

\[ (\text{CE of } E)(C) = (\tau_1, \ldots, \tau_n) \{ C^m; \hat{V}; \hat{F} \} \]

We use this definition in the following rule for typing member-variable references:

\[ v \in \text{dom}(\text{VarsOf}_{E,C'}(C)) \quad \tau_v = \text{VarsOf}_{E,C'}(C)(v) \]

\[ E, C', C \vdash v : \tau_v \]

Class functions, on the other hand, are always visible:

\[ f \in \text{dom}(\text{FunsOf}_{E}(T)) \quad \sigma_f = \text{FunsOf}_{E}(T)(f) \]

\[ E, C, T \vdash f : \sigma_f \]
5.9 Expression subtyping

The “expression subtyping” judgment is syntactic sugar for the common case where we allow an expression to be viewed as having a super type of its type.

\[
E, C \vdash e : \tau' \quad E \vdash \tau' \preceq \tau \\
\hline
E, C \vdash e \preceq \tau
\]

5.10 Expression typing

The types of the primitive infix operators (other than the equality tests) are given in Section 6. These operators are defined on the primitive types, as reflected in the following rule:

\[
\odot : (\iota_1, \iota_2) \to \iota_3 \\
E, C \vdash e_1 : \iota_1 \\
E, C \vdash e_2 : \iota_2 \\
\hline
E, C \vdash (e_1 \odot e_2) : \iota_3
\]

The arguments to an equality operator can have any type as long as one is a subtype of the other.

\[
\odot \in \{==, !=\} \\
E, C \vdash e_1 : \tau_1 \\
E, C \vdash e_2 : \tau_2 \\
\hline
E, C \vdash (e_1 \odot e_2) : \text{bool}
\]

The rule for accessing a member variable just lifts the member typing rule for variables to expressions.

\[
E, C \vdash m : \tau \\
\hline
E, C \vdash m : \tau
\]

The rule for accessing a member variable from an optional object propagates the option annotation to expressions (recall that if \( \tau \) is itself an optional type, then \( \tau? = \tau \)).

\[
E, C \vdash e : T? \\
E, C, T \vdash x : \tau \\
\hline
E, C \vdash e?x : \tau?
\]

The rule for requiring a non-optional value from an expression \( e \) strips off the option annotation from the expression’s type.

\[
E, C \vdash e : T? \\
\hline
E, C \vdash e! : T
\]

Invoking an object’s member function (or method) requires checking the member-function access and then checking that the argument expressions are subtypes of the function’s parameter types.

\[
E, C \vdash m : (\tau_1, \ldots, \tau_n) \to \tau \\
E, C \vdash e_i \preceq \tau_i \leq \tau \leq \tau_n \\
\hline
E, C \vdash m(e_1, \ldots, e_n) : \tau
\]

Invoking a member function for an optional object is similar, but we also propagate the option annotation.

\[
E, C \vdash e : T? \\
E, C, T \vdash x : (\tau_1, \ldots, \tau_n) \to \tau \\
E, C \vdash e_i \preceq \tau_i \leq \tau \leq \tau_n \\
\hline
E, C \vdash e?x(e_1, \ldots, e_n) : \tau?
\]

The rule for constructing new objects of a class \( C \)

\[
C' \in \text{dom}(CE of E) \\
\tau_1, \ldots, \tau_n = \text{ParamsOf}_E(C') \\
E, C \vdash e_i \preceq \tau_i \leq \tau \leq \tau_n \\
\hline
E, C \vdash C'(e_1, \ldots, e_n) : C'
\]
The rule for local variable reference checks that the variable is defined in the variable environment.

\[ x \in \text{dom}(\text{VE} \text{ of } E) \quad (\text{VE} \text{ of } E)(x) = \tau \]

\[ E, C \vdash x : \tau \]

Literals are given their corresponding primitive types.

\[ E, C \vdash \text{true} : \text{bool} \quad E, C \vdash \text{false} : \text{bool} \quad E, C \vdash n : \text{int} \quad E, C \vdash r : \text{string} \]

The \text{nil} expression has type \( T? \) for a well-formed type constant \( T \).

\[ E \vdash \text{ok} \quad E, C \vdash \text{nil} T : T? \]

6 The SOOL Basis

The SOOL Basis environment was described in Section 3 of the Project Overview document; here we formalize that discussion using the notation of the formal type system.

The initial basis used to give a typing to programs in Section 5.1 is defined to be

\[ E_0 = (C_{E_0}, I_{E_0}, V_{E_0}) \]

where the individual environments are defined below. The initial class environment \( C_{E_0} \) is empty except for a binding for \text{obj}:

\[ C_{E_0} = \{ \text{obj} \mapsto () \} \]

The initial interface environment defines bindings for \text{objI} and the predefined interfaces of the basis.

\[ I_{E_0} = \begin{cases} \text{objI} \mapsto \{ 0 \} \\ \text{toStringI} \mapsto \Upsilon_{\text{toStringI}} \\ \text{boolI} \mapsto \Upsilon_{\text{boolI}} \\ \text{intI} \mapsto \Upsilon_{\text{intI}} \\ \text{stringI} \mapsto \Upsilon_{\text{stringI}} \\ \text{systemI} \mapsto \Upsilon_{\text{systemI}} \end{cases} \]
where the predefined interface signatures are as follows:

\[
\begin{align*}
\Upsilon_{\text{toStringI}} &= \{ \text{toString: () } \rightarrow \text{ string} \} \\
\Upsilon_{\text{boolI}} &= \{ \text{toString: () } \rightarrow \text{ string} \\
&\quad \text{not: () } \rightarrow \text{ bool} \} \\
\Upsilon_{\text{intI}} &= \{ \text{toString: () } \rightarrow \text{ string} \\
&\quad \text{char: () } \rightarrow \text{ string} \} \\
\Upsilon_{\text{stringI}} &= \{ \text{toString: () } \rightarrow \text{ string} \\
&\quad \text{length: () } \rightarrow \text{ int} \\
&\quad \text{substring: (int, int) } \rightarrow \text{ string} \\
&\quad \text{charAt: (int) } \rightarrow \text{ int} \\
&\quad \text{toInt: () } \rightarrow \text{ int?} \} \\
\Upsilon_{\text{systemI}} &= \{ \text{print: (toStringI) } \rightarrow \text{ void} \\
&\quad \text{input: () } \rightarrow \text{ string?} \\
&\quad \text{exit: () } \rightarrow \text{ void} \\
&\quad \text{fail: (string) } \rightarrow \text{ void} \} 
\end{align*}
\]

Lastly, the initial variable environment contains just the definition for the global system variable

\[
VE_0 = \{ \text{system} \mapsto \text{ systemI} \}
\]

We also give types for the builtin infix operators.

\[
\begin{align*}
\| & : \text{ (bool, bool) } \rightarrow \text{ bool} \\
\&\& & : \text{ (bool, bool) } \rightarrow \text{ bool} \\
< & : \text{ (int, int) } \rightarrow \text{ bool} \\
\leq & : \text{ (int, int) } \rightarrow \text{ bool} \\
@ & : \text{ (string, string) } \rightarrow \text{ string} \\
+ & : \text{ (int, int) } \rightarrow \text{ int} \\
- & : \text{ (int, int) } \rightarrow \text{ int} \\
* & : \text{ (int, int) } \rightarrow \text{ int} \\
/ & : \text{ (int, int) } \rightarrow \text{ int} 
\end{align*}
\]

We treat unary negation “\(- e\)” as “\(0 - e\)” for type checking purposes.

7 Document history

November 3, 2016 Added missing requirement for return statements in non-void functions (note, this requirement was added for completeness, you do not have to implement it).

November 1, 2016 Fix typos.

October 23, 2016 Make the typing rule for equality a bit more flexible.

October 20, 2016 Fix typos; use \(r\) for string constants (instead of \(s\), which is used for statements).

October 19, 2016 Original version.