CMSC 22600 **Compilers for Computer Languages** **Handout 2**
Autumn 2016 **October 19, 2016**

The SOOL Type System

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.

a	$\in$	VAR	Variable identifiers
v	$\in$	MEMBVAR	Class variable identifiers
f	$\in$	MEMBFUN	Class/interface function identifiers
x	$\in$	MEMB = MEMBVAR $\cup$ MEMBFUN	Class member identifiers
C	$\in$	CLS $\cup \{\mathbf{obj}\}$	Class types
I	$\in$	IFC $\cup \{\mathbf{objI}\}$	Interface types
n	$\in$	INT	Integer constants
r	$\in$	STR	String constants

Note that we add the distinguished identifiers **obj** and **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 `C` that is derived from a class `B`

  ```
  class C (···) extends B (···) { ··· }
  ```

  can only appear after the declaration of `B`. This restriction ensures that the class hierarchy will be acyclic. Also note that this restriction does *not* prevent class `B` from referencing class `C`.
- A declaration of an interface `J` that extends an interface `I`

$$
\begin{array}{rcl}
p & ::= & d \\[4pt]
d & ::= & \textbf{class}\ C'(a_1 : \tau_1, \ldots, a_n : \tau_n)\ \textbf{extends}\ C(e_1, \ldots, e_k)\ \{\ vd; fd\ \} \\
& | & \textbf{interface}\ I'\ \textbf{extends}\ I\ \{\ \mathit{fs}\ \} \\
& | & d_1\ d_2 \\[4pt]
vd & ::= & \textbf{var}\ v : \tau = e \\
& | & vd_1\ vd_2 \\[4pt]
fd & ::= & \textbf{meth}\ f\ (a_1 : \tau_1, \ldots, a_n : \tau_n) \to \theta\ \{\ s\ \} \\
& | & \textbf{override}\,\textbf{meth}\ f\ (a_1 : \tau_1, \ldots, a_n : \tau_n) \to \theta\ \{\ s\ \} \\
& | & fd_1\ fd_2 \\[4pt]
\mathit{fs} & ::= & \textbf{meth}\ f\ (\tau_1, \ldots, \tau_n) \to \theta \\
& | & \mathit{fs}_1\ \mathit{fs}_2 \\[4pt]
s & ::= & s_1\,;s_2 \\
& | & \textbf{var}\ x = e \\
& | & \textbf{while}\ e\ \{\ s\ \} \\
& | & \textbf{if}\ e\ \textbf{then}\ \{\ s_1\ \}\ \textbf{else}\ \{\ s_2\ \} \\
& | & \textbf{return}\ e \\
& | & \textbf{return} \\
& | & m := e_2 \\
& | & x := e \\
& | & m(e_1, \ldots, e_n) \\[8pt]
m & ::= & e.x \\
& | & e!x \\[8pt]
e & ::= & (e_1 \odot e_2) \\
& | & m \\
& | & e?x \\
& | & e! \\
& | & m(e_1, \ldots, e_n) \\
& | & e?x(e_1, \ldots, e_n) \\
& | & C(e_1, \ldots, e_n) \\
& | & a \\
& | & n \mid r \mid \textbf{true} \mid \textbf{false} \\
& | & \textbf{nil}\ T
\end{array}
$$

Figure 1: SOOL abstract syntax

```
    interface J extends I { ··· }
```

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:

$$\begin{array}{rcll} \mathcal{V} & \in & 2^{\textsc{MembVar}} & \text{Sets of variable identifiers} \\ \mathcal{F} & \in & 2^{\textsc{MembFun}} & \text{Sets of function identifiers} \end{array}$$

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

$$\begin{array}{rcll} \iota & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{string} & \text{Primitive types} \\ T & ::= & C \mid I \mid \iota & \text{Type constructors} \\ \tau & ::= & T \mid T? & \text{Type expressions (}\textsc{Typ}\text{)} \\ \theta & ::= & \tau \mid \textbf{void} & \text{Return types} \\ \sigma & ::= & (\tau_1, \ldots, \tau_n) \to \theta & \text{Member-function signature (}\textsc{Spec}\text{)} \\ \mu & ::= & \textsc{Typ} \cup \textsc{Spec} & \text{Class/interface member type} \\ \Sigma & ::= & (\tau_1, \ldots, \tau_n)\{\!\!\{ C; v : {\tau_v}^{v \in \mathcal{V}}; f : {\sigma_f}^{f \in \mathcal{F}} \}\!\!\} & \text{Class signature (}\textsc{ClsSig}\text{)} \\ \Upsilon & ::= & \{\!\!\{ f : {\sigma_f}^{f \in \mathcal{F}} \}\!\!\} & \text{Interface signature (}\textsc{IfcSig}\text{)} \end{array}$$

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 τ and τ' are equal.[2]

The class signature

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

[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 τ_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 singleton finite map that maps x to y. If A is a finite map, then we write $\mathrm{dom}(A)$ 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

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

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

$$\begin{array}{rcll} \mathit{CE} & \in & \mathrm{CE{\small NV}} = \mathrm{C{\small LS}} \overset{\text{fin}}{\to} \mathrm{C{\small LS}S{\small IG}} & \text{Class environments} \\ \mathit{IE} & \in & \mathrm{IE{\small NV}} = \mathrm{I{\small FC}} \overset{\text{fin}}{\to} \mathrm{I{\small FC}S{\small IG}} & \text{Interface environments} \\ \mathit{VE} & \in & \mathrm{VE{\small NV}} = \mathrm{V{\small AR}} \overset{\text{fin}}{\to} \mathrm{T{\small YP}} & \text{Variable environments} \\ E = \langle \mathit{CE}, \mathit{IE}, \mathit{VE} \rangle & \in & \mathrm{E{\small NV}} = \mathrm{CE{\small NV}} \times \mathrm{IE{\small NV}} \times \mathrm{VE{\small NV}} & \text{Environments} \end{array}$$

If $E = \langle \mathit{CE}, \mathit{IE}, \mathit{VE} \rangle$, then the notation $(\mathit{CE}\ \mathit{of}\ E)$ 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 Additonal 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:

$$\hat{\mathcal{F}} = \left\{ f : \sigma_f{}^{f \in \mathcal{F}} \right\} \qquad\qquad \hat{\mathcal{V}} = \left\{ v : \tau_v{}^{v \in \mathcal{V}} \right\}$$

$\hat{\mathcal{F}}$ 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

$E \vdash p\ \mathbf{ok}$	Program typing (see Section 5.1)
$E \vdash d : E'$	Declaration typing (see Section 5.2)
$E, C \vdash vd : \hat{\mathcal{V}}$	Member-variable-declaration typing (see Section 5.3)
$E, C', C \vdash fd : \hat{\mathcal{F}}$	Member-function-declaration typing (see Section 5.4)
$E \vdash fs : \hat{\mathcal{F}}$	Member-function-specification typing (see Section 5.6)
$E, C, \theta \vdash s : E$	Statement typing (see Section 5.6)
$E, C \vdash m : \mu$	Required member selection (see Section 5.7)
$E, C, T \vdash x : \mu$	Member typing (see Section 5.8)
$E, C \vdash e \preceq \tau$	Expression subtyping (see Section 5.9)
$E, C \vdash e : \tau$	Expression typing (see Section 5.10)

Figure 2: Typing judgments for SOOL

follows:

$$\begin{aligned}
\mathrm{FunsOf}_E(\mathbf{obj}) &= \emptyset \\
\mathrm{FunsOf}_E(C) &= \hat{\mathcal{F}} \text{ where } C \in \mathrm{dom}(\mathit{CE\ of\ E}) \text{ and} \\
&\qquad (\mathit{CE\ of\ E})(C) = (\tau_1, \ldots, \tau_n)\{\!\!\{ C'; \hat{\mathcal{V}}; \hat{\mathcal{F}} \}\!\!\} \\
\mathrm{FunsOf}_E(\mathbf{objI}) &= \emptyset \\
\mathrm{FunsOf}_E(I) &= \hat{\mathcal{F}} \text{ where } I \in \mathrm{dom}(\mathit{IE\ of\ E}) \text{ and} (\mathit{IE\ of\ E})(I) = \{\!\!\{ \hat{\mathcal{F}} \}\!\!\}
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{ParamsOf}_E(\mathbf{obj}) &= \mathit{empty} \\
\mathrm{ParamsOf}_E(C) &= \tau_1, \ldots, \tau_n \text{ where } C \in \mathrm{dom}(\mathit{CE\ of\ E}) \text{ and} \\
&\qquad (\mathit{CE\ of\ E})(C) = (\tau_1, \ldots, \tau_n)\{\!\!\{ C'; \hat{\mathcal{V}}; \hat{\mathcal{F}} \}\!\!\}
\end{aligned}$$

For primitive types $\iota \in \{\mathbf{bool}, \mathbf{int}, \mathbf{string}\}$, we write $\mathrm{InterfaceOf}(\iota)$ 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 "$\mathit{context} \vdash \mathit{term} : \mathit{property}$", which can be read as "term has $\mathit{property}$ in $\mathit{context}$."

4 Judgments about types

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

$E \vdash \tau\ \mathbf{ok}$	The type τ is *well-formed*
$\mathit{CE} \vdash C' \lessdot C$	Class C' *inherits* from C
$E \vdash C \prec I$	Class C *implements* interface I
$E \vdash \tau' \preceq \tau$	Type τ' is a *subtype* of τ

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 ("$\vdash$") 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 $\boxed{E \vdash \tau\ \mathbf{ok}}$

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.

$$\frac{}{E \vdash \iota\ \mathbf{ok}}$$

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

$$\frac{C \in \mathrm{dom}(\mathit{CE}\ \mathit{of}\ E)}{E \vdash C\ \mathbf{ok}}$$

Similarly for interface types:

$$\frac{I \in \mathrm{dom}(\mathit{IE}\ \mathit{of}\ E)}{E \vdash I\ \mathbf{ok}}$$

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

$$\frac{E \vdash T\ \mathbf{ok}}{E \vdash T?\ \mathbf{ok}}$$

We lift the well-formedness judgement to return types (θ) and member-function signatures (σ) in the obvious way.

4.2 The inherits relation $\boxed{\mathit{CE} \vdash C' \lessdot C}$

Classes inherit from their immediate superclass:

$$\frac{C' \in \mathrm{dom}(\mathit{CE}) \qquad \mathit{CE}(C') = (\tau_1, \ldots, \tau_n)\{\!|\, C; v : {\tau_v}^{v \in \mathcal{V}}; f : {\sigma_f}^{f \in \mathcal{F}} \,|\!\}}{\mathit{CE} \vdash C' \lessdot C}$$

and the inherits relation is transitive

$$\frac{\mathit{CE} \vdash C' \lessdot C'' \qquad \mathit{CE} \vdash C'' \lessdot C}{\mathit{CE} \vdash C' \lessdot C}$$

4.3 The implements relation $\boxed{E \vdash C \prec I}$

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

$$\frac{\begin{array}{c} C \in \mathrm{dom}(\mathit{CE}\ \mathit{of}\ E) \qquad (\mathit{CE}\ \mathit{of}\ E)(C) = (\tau_1, \ldots, \tau_n)\{\!|\, C; \hat{\mathcal{V}}; \hat{\mathcal{F}}_C \,|\!\} \\ I \in \mathrm{dom}(\mathit{IE}\ \mathit{of}\ E) \qquad (\mathit{IE}\ \mathit{of}\ E)(I) = \{\!|\hat{\mathcal{F}}_I|\!\} \\ \mathrm{dom}(\hat{\mathcal{F}}_I) \subseteq \mathrm{dom}(\hat{\mathcal{F}}_C) \qquad {\vdash \hat{\mathcal{F}}_I(f) = \hat{\mathcal{F}}_C(f)}^{f \in \mathrm{dom}(\hat{\mathcal{F}}_I)} \end{array}}{E \vdash C \prec I}$$

4.4 The subtyping relation $\boxed{E \vdash \tau' \preceq \tau}$

A well-formed type τ is always a subtype of itself.

$$\frac{E \vdash \tau\ \mathbf{ok}}{E \vdash \tau \preceq \tau}$$

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

$$\frac{E \vdash Tu\ \mathbf{ok}}{E \vdash T \preceq T?}$$

Primitive types are subtypes of the interfaces that they implement.

$$\frac{I = \mathrm{InterfaceOf}(\iota)}{E \vdash \iota \preceq I}$$

Classes are subtypes of their super-classes.

$$\frac{(\mathit{CE\ of\ E}) \vdash C' \lessdot C}{E \vdash C' \preceq C}$$

Classes are also subtypes of the interfaces that they implement.

$$\frac{(\mathit{CE\ of\ E}) \vdash C \prec I}{E \vdash C \preceq I}$$

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

$$\frac{\mathrm{FunsOf}_E(I') = \hat{\mathcal{F}}' \qquad \mathrm{FunsOf}_E(I) = \hat{\mathcal{F}} \qquad \mathrm{dom}(\hat{\mathcal{F}}) \subseteq \mathrm{dom}(\hat{\mathcal{F}}') \qquad \vdash \hat{F}'(f) = \hat{F}(f)^{f \in \mathrm{dom}(\hat{\mathcal{F}})}}{E \vdash I' \preceq I}$$

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

$$\frac{E \vdash \tau' \preceq \tau'' \qquad E \vdash \tau'' \preceq \tau}{E \vdash \tau' \preceq \tau}$$

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 $\boxed{E \vdash p\ \mathbf{ok}}$

The typing rule for programs is deceptively simple.

$$\frac{E_0 \pitchfork E \qquad E_0 \cup E \vdash d : E}{E_0 \vdash d\ \mathbf{ok}}$$

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 $\mathbf{obj}$ and $\mathbf{objI}$ (see Section 6).

5.2 Declaration typing

$$\boxed{E \vdash d : E'}$$

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.

$$\frac{\begin{array}{c} E \vdash \tau_i\ \mathbf{ok}^{1\le i\le n} \qquad C \in \mathrm{dom}(\mathit{CE\ of\ E}) \qquad (\mathit{CE\ of\ E})(C) = (\tau'_1,\,\ldots,\,\tau'_k)\{\!\!\{\,C; \hat{\mathcal{V}}; \hat{\mathcal{F}}\,\}\!\!\} \\ E' = E\pm\{a_i \mapsto \tau_i | 1 \le i \le n\} \qquad E' \vdash e_i \preceq \tau_i'^{1\le i\le k} \qquad E' \vdash vd : \hat{\mathcal{V}}' \\ E\pm\{\mathbf{self} \mapsto C'\}, C \vdash \mathit{fd} : \hat{\mathcal{F}}' \\ \mathrm{dom}(\hat{\mathcal{V}}) \pitchfork \mathrm{dom}(\hat{\mathcal{V}}') \qquad \mathrm{dom}(\hat{\mathcal{F}}) \pitchfork \mathrm{dom}(\hat{\mathcal{F}}') \qquad \Sigma = (\tau_1,\,\ldots,\,\tau_n)\{\!\!\{\,C; \hat{\mathcal{V}} \cup \hat{\mathcal{V}}'; \hat{\mathcal{F}} \cup \hat{\mathcal{F}}'\,\}\!\!\} \end{array}}{E \vdash \mathbf{class}\ C'(a_1 : \tau_1,\,\ldots,\,a_n : \tau_n)\ \mathbf{extends}\ C(e_1,\,\ldots,\,e_k)\ \{\ vd;\ \mathit{fd}\ \} : \langle \{C' \mapsto \Sigma\}, \emptyset, \emptyset\rangle}$$

The typing rule for interface declarations is simpler, since we only need check the member-function specifications and ensure that the declaration does not redefine functions from the interface that it extends.

$$\frac{\begin{array}{c} I \in \mathrm{dom}(\mathit{IE\ of\ E}) \qquad (\mathit{IE\ of\ E})(I) = \{\!\!\{\,\hat{\mathcal{F}}\,\}\!\!\} \qquad E \vdash \mathit{fs} : \hat{\mathcal{F}}' \\ \mathrm{dom}(\hat{\mathcal{F}}) \pitchfork \mathrm{dom}(\hat{\mathcal{F}}') \qquad \Upsilon = \{\!\!\{\,\hat{\mathcal{F}} \cup \hat{\mathcal{F}}'\,\}\!\!\} \end{array}}{E \vdash \mathbf{interface}\ I'\ \mathbf{extends}\ I\ \{\ \mathit{fs}\ \} : \langle \emptyset, \{I' \mapsto \Upsilon\}, \emptyset\rangle}$$

Sequences of declarations must define disjoint environments (note the variable environments will always be empty). We then join the environments.

$$\frac{\begin{array}{c} E \vdash d_1 : \langle \mathit{CE}_1, \mathit{IE}_1, \emptyset\rangle \\ E \vdash d_2 : \langle \mathit{CE}_2, \mathit{IE}_2, \emptyset\rangle \qquad \mathrm{dom}(\mathit{CE}_1) \pitchfork \mathrm{dom}(\mathit{CE}_2) \qquad \mathrm{dom}(\mathit{IE}_1) \pitchfork \mathrm{dom}(\mathit{IE}_2) \end{array}}{E \vdash d_1\ d_2 : \langle \mathit{CE}_1 \cup \mathit{CE}_2, \mathit{IE}_1 \cup \mathit{IE}_2, \emptyset\rangle}$$

5.3 Member-variable-declaration typing

$$\boxed{E, C \vdash vd : \hat{\mathcal{V}}}$$

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

$$\frac{E \vdash \tau_v\ \mathbf{ok} \qquad E, C \vdash e \preceq \tau_v}{E, C \vdash \mathbf{var}\ v : \tau_v = e : \left\{ v : {\tau_v}^{v \in \{v\}} \right\}}$$

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

$$\frac{E, C \vdash vd_1 : \hat{\mathcal{V}}_1 \qquad E, C \vdash vd_2 : \hat{\mathcal{V}}_2 \qquad \mathrm{dom}(\hat{\mathcal{V}}_1) \pitchfork \mathrm{dom}(\hat{\mathcal{V}}_2)}{E, C \vdash vd_1\ vd_2 : \hat{\mathcal{V}}_1 \cup \hat{\mathcal{V}}_2}$$

5.4 Member-function-declaration typing

$$\boxed{E, C', C \vdash \mathit{fd} : \hat{\mathcal{F}}}$$

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 $\mathbf{obj}$).

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.

$$\frac{f \notin \mathrm{dom}(\mathrm{FunsOf}_E(C)) \qquad E\pm\{a_i \mapsto \tau_i | 1 \le i \le n\}, C', \theta \vdash s : E' \qquad \sigma_f = (\tau_1,\,\ldots,\,\tau_n) \to \theta}{E, C', C \vdash \mathbf{meth}\ f\ (a_1 : \tau_1,\,\ldots,\,a_n : \tau_n) \to \theta\ \{\ s\ \} : \left\{ f : {\sigma_f}^{f \in \{f\}} \right\}}$$

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.

$$\frac{f \in \mathrm{dom}(\mathrm{FunsOf}_E(C)) \qquad \mathrm{FunsOf}_E(C)(f) = \sigma_f \qquad \vdash \sigma_f = (\tau_1, \ldots, \tau_n) \to \theta \\ E \pm \{a_i \mapsto \tau_i | 1 \leq i \leq n\}, C', \theta \vdash s : E'}{E, C', C \vdash \textbf{override}\,\textbf{meth}\ f\ (a_1 : \tau_1, \ldots, a_n : \tau_n) \to \theta\ \{\, s \,\} : \{\}}$$

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

$$\frac{E, C', C \vdash \mathit{fd}_1 : \hat{\mathcal{F}}_1 \qquad E, C', C \vdash \mathit{fd}_2 : \hat{\mathcal{F}}_2 \qquad \mathrm{dom}(\hat{\mathcal{F}}_1) \pitchfork \mathrm{dom}(\hat{\mathcal{F}}_2)}{E, C', C \vdash \mathit{fd}_1\ \mathit{fd}_2 : \hat{\mathcal{F}}_1 \cup \hat{\mathcal{F}}_2}$$

5.5 Member-function-specification typing

$\boxed{E \vdash \mathit{fs} : \hat{\mathcal{F}}}$

The rule for typing a member-function specification requires that the function signature σ_f be well formed.

$$\frac{\sigma_f = (\tau_1, \ldots, \tau_n) \to \theta \qquad E \vdash \sigma_f\ \textbf{ok}}{E \vdash \textbf{meth}\ f\ (\tau_1, \ldots, \tau_n) \to \theta : \left\{ f : \sigma_f{}^{f \in \{f\}} \right\}}$$

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

$$\frac{E \vdash \mathit{fs}_1 : \hat{\mathcal{F}}_1 \qquad E \vdash \mathit{fs}_2 : \hat{\mathcal{F}}_2 \qquad \mathrm{dom}(\hat{\mathcal{F}}_1) \pitchfork \mathrm{dom}(\hat{\mathcal{F}}_2)}{E \vdash \mathit{fs}_1\ \mathit{fs}_2 : \hat{\mathcal{F}}_1 \cup \hat{\mathcal{F}}_2}$$

5.6 Statement typing

$\boxed{E, C, \theta \vdash s : E}$

Statements are type checked in a context that includes the enclosing class C and the declared return type θ 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.

$$\frac{E, C, \theta \vdash s_1 : E_1 \qquad E_1, C, \theta \vdash s_2 : E_2}{E, C, \theta \vdash s_1 \,;\, s_2 : E_2}$$

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

$$\frac{E, C \vdash e : \tau}{E, C, \theta \vdash \textbf{var}\ x = e : E \pm \{x \mapsto \tau\}}$$

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.

$$\frac{E, C \vdash e : \textbf{bool} \qquad E, C, \theta \vdash s : E'}{E, C, \theta \vdash \textbf{while}\ e\ \{\, s \,\} : E}$$

Conditional statements also require a boolean predicate.

$$\frac{E, C \vdash e : \textbf{bool} \qquad E, C, \theta \vdash s_1 : E_1 \qquad E, C, \theta \vdash s_2 : E_2}{E, C, \theta \vdash \textbf{if}\ e\ \textbf{then}\ \{\, s_1 \,\}\ \textbf{else}\ \{\, s_2 \,\} : E}$$

A **return** statement without an argument expression is valid if the enclosing method's return type is **void**.

$$\frac{}{E, C, \textbf{void} \vdash \textbf{return} : E}$$

A **return** statement with an argument expression is valid if the type if e is a subtype of the enclosing method's return type.

$$\frac{E, C \vdash e \preceq \tau}{E, C, \tau \vdash \textbf{return}\ e : E}$$

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

$$\frac{E, C \vdash m : \tau \qquad E, C \vdash e \preceq \tau}{E, C, \theta \vdash m := e : E}$$

$$\frac{x \in \mathrm{dom}(\mathit{VE\ of\ E}) \qquad (\mathit{VE\ of\ E})(x) = \tau \qquad E \vdash e \preceq \tau}{E, C, \theta \vdash x := e : E}$$

$$\frac{E, C \vdash m : (\tau_1, \ldots, \tau_n) \to \textbf{void} \qquad E, C \vdash e_i \preceq \tau_i{}^{1 \leq i \leq n}}{E, C, \theta \vdash m(e_1, \ldots, e_n) : E}$$

5.7 Required member selection $\boxed{E, C \vdash m : \mu}$

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

$$\frac{E, C \vdash e \preceq T \qquad E, C, T \vdash x : \mu}{E, C \vdash e.x : \mu}$$

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

$$\frac{E \vdash e \preceq T? \qquad E, C, T \vdash x : \mu}{E, C \vdash e!x : \mu}$$

5.8 Member typing $\boxed{E, C, T \vdash x : \mu}$

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:

$$\mathrm{VarsOf}_{E,C'}(C) = \begin{cases} \hat{\mathcal{V}} & \text{if } C' = C \text{ or } E \vdash C' \lessdot C \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{where } C \in \mathrm{dom}(\mathit{CE\ of\ E}) \text{ and}$$
$$(\mathit{CE\ of\ E})(C) = (\tau_1, \ldots, \tau_n)\{\!|\, C''; \hat{\mathcal{V}}; \hat{\mathcal{F}} \,|\!\}$$

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

$$\frac{v \in \mathrm{dom}(\mathrm{VarsOf}_{E,C'}(C)) \qquad \tau_v = \mathrm{VarsOf}_{E,C'}(C)(v)}{E, C', C \vdash v : \tau_v}$$

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

$$\frac{f \in \mathrm{dom}(\mathrm{FunsOf}_E(T)) \qquad \sigma_f = \mathrm{FunsOf}_E(T)(f)}{E, C, T \vdash f : \sigma_f}$$

5.9 Expression subtyping $\boxed{E, C \vdash e \preceq \tau}$

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.

$$\frac{E, C \vdash e : \tau' \qquad E \vdash \tau' \preceq \tau}{E, C \vdash e \preceq \tau}$$

5.10 Expression typing $\boxed{E, C \vdash e : \tau}$

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:

$$\frac{\odot : (\iota_1, \iota_2) \to \iota_3 \qquad E, C \vdash e_1 : \iota_1 \qquad E, C \vdash e_2 : \iota_2}{E, C \vdash (e_1 \odot e_2) : \iota_3}$$

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

$$\frac{\odot \in \{\texttt{==}, \texttt{!=}\} \qquad E, C \vdash e_1 : \tau_1 \qquad E, C \vdash e_2 : \tau_2 \qquad \text{either } E \vdash \tau_1 \preceq \tau_2 \text{ or } E \vdash \tau_2 \preceq \tau_1}{E, C \vdash (e_1 \odot e_2) : \mathbf{bool}}$$

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

$$\frac{E, C \vdash m : \tau}{E, C \vdash m : \tau}$$

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

$$\frac{E, C \vdash e : T? \qquad E, C, T \vdash x : \tau}{E, C \vdash e?x : \tau?}$$

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

$$\frac{E, C \vdash e : T?}{E, C \vdash e! : T}$$

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.

$$\frac{E, C \vdash m : (\tau_1, \ldots, \tau_n) \to \tau \qquad E, C \vdash e_i \preceq \tau_i{}^{1 \leq i \leq n}}{E, C \vdash m(e_1, \ldots, e_n) : \tau}$$

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

$$\frac{E, C \vdash e : T? \qquad E, C, T \vdash x : (\tau_1, \ldots, \tau_n) \to \tau \qquad E, C \vdash e_i \preceq \tau_i{}^{1 \leq i \leq n}}{E, C \vdash e?x(e_1, \ldots, e_n) : \tau?}$$

The rule for constructing new objects of a class C

$$\frac{C' \in \mathrm{dom}(\mathit{CE\ of\ E}) \qquad \tau_1, \ldots, \tau_n = \mathrm{ParamsOf}_E(C') \qquad E, C \vdash e_i \preceq \tau_i{}^{1 \leq i \leq n}}{E, C \vdash C'(e_1, \ldots, e_n) : C'}$$

The rule for local variable reference checks that the variable is defined in the variable environment.

$$\frac{x \in \mathrm{dom}(\mathit{VE}\ \mathit{of}\ E) \qquad (\mathit{VE}\ \mathit{of}\ E)(x) = \tau}{E, C \vdash x : \tau}$$

Literals are given their corresponding primitive types.

$$\frac{}{E, C \vdash \textbf{true} : \textbf{bool}} \quad \frac{}{E, C \vdash \textbf{false} : \textbf{bool}} \quad \frac{}{E, C \vdash n : \textbf{int}} \quad \frac{}{E, C \vdash r : \textbf{string}}$$

The **`nil`** expression has type $T?$ for a well-formed type constant T.

$$\frac{E \vdash T\ \textbf{ok}}{E, C \vdash \textbf{nil}\ T : T?}$$

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

$$E_0 = \langle \mathit{CE}_0, \mathit{IE}_0, \mathit{VE}_0 \rangle$$

where the individual environments are defined below. The initial class environment CE_0 is empty except for a binding for **obj**:

$$\mathit{CE}_0 = \{\textbf{obj} \mapsto ()\{\!|\, \textbf{obj}; \emptyset; \emptyset \,|\!\}\}$$

The initial interface environment defines bindings for **objI** and the predefined interfaces of the basis.

$$\mathit{IE}_0 = \left\{ \begin{array}{l} \textbf{objI} \mapsto \{\!|\, \emptyset \,|\!\} \\ \texttt{toStringI} \mapsto \Upsilon_{\texttt{toStringI}}, \\ \texttt{boolI} \mapsto \Upsilon_{\texttt{boolI}}, \\ \texttt{intI} \mapsto \Upsilon_{\texttt{intI}}, \\ \texttt{stringI} \mapsto \Upsilon_{\texttt{stringI}}, \\ \texttt{systemI} \mapsto \Upsilon_{\texttt{systemI}} \end{array} \right\}$$

where the predefined interface signatures are as follows:

$$\Upsilon_{\texttt{toStringI}} = \{\!|\, \texttt{toString} : () \to \textbf{string} \,|\!\}$$

$$\Upsilon_{\texttt{boolI}} = \left\{\!\left|\begin{array}{l} \texttt{toString} : () \to \textbf{string} \\ \texttt{not} : () \to \textbf{bool} \end{array}\right|\!\right\}$$

$$\Upsilon_{\texttt{intI}} = \left\{\!\left|\begin{array}{l} \texttt{toString} : () \to \textbf{string} \\ \texttt{char} : () \to \textbf{string} \end{array}\right|\!\right\}$$

$$\Upsilon_{\texttt{stringI}} = \left\{\!\left|\begin{array}{l} \texttt{toString} : () \to \textbf{string} \\ \texttt{length} : () \to \textbf{int} \\ \texttt{substring} : (\textbf{int}, \textbf{int}) \to \textbf{string} \\ \texttt{charAt} : (\textbf{int}) \to \textbf{int} \\ \texttt{toInt} : () \to \textbf{int}? \end{array}\right|\!\right\}$$

$$\Upsilon_{\texttt{systemI}} = \left\{\!\left|\begin{array}{l} \texttt{print} : (\texttt{toStringI}) \to \textbf{void} \\ \texttt{input} : () \to \textbf{string}? \\ \texttt{exit} : () \to \textbf{void} \\ \texttt{fail} : (\textbf{string}) \to \textbf{void} \end{array}\right|\!\right\}$$

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

$$VE_0 = \{\texttt{system} \mapsto \texttt{systemI}\}$$

We also give types for the builtin infix operators.

$$\begin{array}{rcl} \texttt{||} & : & (\textbf{bool}, \textbf{bool}) \to \textbf{bool} \\ \texttt{\&\&} & : & (\textbf{bool}, \textbf{bool}) \to \textbf{bool} \\ \texttt{<} & : & (\textbf{int}, \textbf{int}) \to \textbf{bool} \\ \texttt{<=} & : & (\textbf{int}, \textbf{int}) \to \textbf{bool} \\ \texttt{@} & : & (\textbf{string}, \textbf{string}) \to \textbf{string} \\ \texttt{+} & : & (\textbf{int}, \textbf{int}) \to \textbf{int} \\ \texttt{-} & : & (\textbf{int}, \textbf{int}) \to \textbf{int} \\ \texttt{*} & : & (\textbf{int}, \textbf{int}) \to \textbf{int} \\ \texttt{/} & : & (\textbf{int}, \textbf{int}) \to \textbf{int} \end{array}$$

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.