CS 510 Programming Languages
Assignment #6

Out: Monday, Dec. 2nd (12:00 Noon)
Due: Monday, Dec. 9th (11:00 AM)

In this, your last assignment, you will get acquainted with a minimal Java-like language called Featherweight Java (or FJ for short). The language is defined in the lecture notes, and slides. Please make sure to read that material before attacking the problems below.

1 Subtyping Terminology and Concepts

Provide English definitions and/or explanations of the following terms and concepts. Be as clear, precise and concise as possible.

(a) τ_1 is a subtype of τ_2.

(b) *Contravariant* subtyping.

(c) The algorithmic subtyping rules for MinML (with subtyping) are *sound* with respect to the declarative subtyping rules.

(d) The algorithmic subtyping rules for MinML (with subtyping) are *complete* with respect to the declarative subtyping rules.

(e) Deductive system A is *sound* with respect to deductive system B.

(f) Deductive system A is *complete* with respect to deductive system B.

(g) *Syntax-directed* inference rules.

(h) A *join*.

(i) A *meet*.

(j) *Coherence* of a translation.

2 Exceptions

Extend the syntax of FJ so that programmers can raise an exception and catch an exception. You don't need to make your exceptions carry values. Re-define the operational semantics of FJ in the style of the C-machines from Robert Harper's manuscript (section 11.1 and also see the lecture notes on C-machines and exceptions) and add exceptions. Be sure to define stacks and stack frames precisely as well as the judgments that use them.

3 Interfaces

Consider the following extended version of FJ:

Classes	C	$::=$	$\texttt{class}\,c\,\texttt{extends}\,c\,\{\overline{\tau}\,\overline{f}\texttt{;}\ k\,\overline{d}\}$
		$\mid$	$\texttt{class}\,c\,\texttt{extends}\,c\,\texttt{implements}\,\overline{i}\,\{\overline{\tau}\,\overline{f}\texttt{;}\ k\,\overline{d}\}$
Interfaces	I	$::=$	$\texttt{interface}\,i\,\{\overline{a}\}$
Constructors	k	$::=$	$c\,\texttt{(}\overline{\tau}\,\overline{x}\texttt{)}\,\{\,\texttt{super(}\overline{x}\texttt{);}\,\texttt{this.}\overline{f}\texttt{=}\overline{x}\texttt{;}\,\}$
Methods	d	$::=$	$\tau\,m\,\texttt{(}\overline{\tau}\,\overline{x}\texttt{)}\,\{\,\texttt{return}\,e\texttt{;}\,\}$
Abstract Methods	a	$::=$	$\tau\,m\,\texttt{(}\overline{\tau}\,\overline{x}\texttt{);}$
Types	τ	$::=$	$c \mid i$
Expressions	e	$::=$	$x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new}\,c\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}\tau\texttt{)}\,e$

We have added a limited interface feature. As in Java, a class can implement several interfaces. An interface i specifies a set of methods. Any class that implements i is required to provide the methods specified by i (we do not allow abstract classes). Anywhere an interface i is expected, we can provide a class c that implements i. Thus, implementation implies subtyping. For simplicity, we do not allow interfaces to extend other interfaces, which otherwise is possible in Java. Note that we have replaced c with τ in the syntax above everywhere a type can be an interface. This holds for argument types, return types, field types, and casting.

(a) Extend the static semantics for FJ to accommodate the interface feature described above.

(b) Do we need to modify the dynamic semantics as well? (Explain your answer.)

(c) Assume that you have already proven progress and preservation for the core Featherweight Java language. Give an argument for why preservation and progress still holds with your modified static and dynamic semantics.

Hint: Let the interfaces reside in the class table.

4 Privacy

Java provides mechanisms for access control. A method or field of a class can be declared *public*, *protected*, or *private*. We modify the syntax of FJ as follows:

$$
\begin{array}{llcl}
\textit{Classes} & C & ::= & \texttt{class}\, c\, \texttt{extends}\, c\, \{\, \overline{p}\,\overline{c}\,\overline{f}\texttt{;}\ k\,\overline{d}\,\} \\
\textit{Methods} & d & ::= & p\, c\, m\, \texttt{(}\overline{c}\,\overline{x}\texttt{)}\ \{\, \texttt{return}\, e\texttt{;}\ \} \\
\textit{Privacy Modifiers} & p & ::= & \texttt{public} \mid \texttt{protected} \mid \texttt{private}
\end{array}
$$

In the absence of packages the privacy modifiers get the following semantics:

- If a method or field of class c is declared `public`, then access is permitted from anywhere.
- If a method or field of class c is declared `protected`, then access is only permitted from methods of c and subclasses of c.
- If a method or field of class c is declared `private`, then access is only permitted from methods of c.

(a) **Extend the static semantics so that code containing privacy violations, according to the semantics given above, is ruled out. Be clear about which rules you keep, which are substituted for new rules, and which are removed from the original static semantics.**

Hint: When checking if a method in class c is well-formed, you need to check that the expression in the body of the method does not refer to fields or methods in another class c' that are private (or protected if $c \not<: c'$). You may need to pass more than just a typing context Γ down to the typing judgement for expressions.