CSE-321: FeatherWeight Java
(100 points)

1 Introduction

In this programming test, you will be implementing *Featherweight Java*, or FJ, in Objective CAML. FJ is compact yet retains all the core features of Java. First we will have an overview of FJ, and then present a rigorous description of the syntax, typing rules, and reduction rules of FJ. Given a parser for FJ, you will be writing a typechecker (from the typing rules) and an evaluator (from the reduction rules). Although you are given a complete parser for FJ, you will begin only with the signatures for the type checker and the evaluator. So you want to exploit your experience gained in the previous programming assignments so that your code remains compact and well organized.

We suggest that you fully understand all the details of the typing and reduction rules, design your code carefully by clearly stating the meaning of each function, and then begin to write the code. Since the typing rules and reduction rules of FJ are not so simple, we encourage you to spend enough time before actually writing code; both the amount of time it takes you to finish this programming test and the conciseness of your code will be heavily affected by the clarity of your overall design.

In the following section, we first introduce the main ideas of FJ informally, and then formally presents its syntax, typing rules, and reduction rules.

2 Featherweight Java

In FJ, a program consists of a collection of class declarations plus an expression to be evaluated. Here are some typical class declarations in FJ.

```
class A extends Object {
  A() { super(); }
}

class B extends Object {
  B() { super(); }
}

class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst = fst; this.snd = snd;
  }
```

```
    Pair setfst(Object newfst) {
      return new Pair(newfst, this.snd);
    }
  }
```

For the sake of syntactic regularity, we require that a class definition specify the supertype (even when it is `Object`) and that a field access or a method invocation specify the receiver (*e.g.*, receiver `this` in a field access `this.snd`). Constructors always take the same stylized form: there is one argument for each field which has the same name as the field itself; the `super` constructor is invoked on the fields of the supertype; the remaining fields are initialized to the corresponding arguments. In the example above, the supertype is always `Object`, which has no fields, so the invocations of `super` have no arguments. Constructors are the only place where `super` or = appears in a FJ program. Since FJ has no side effects, a method body always consists of `return` followed by an expression, as in the body of `setfst()`.

Under the above declaration of classes, the expression

```
new Pair(new A(), new B()).setfst(new B())
```

evaluates to the following expression:

```
new Pair(new B(), new B())
```

There are five forms of expression in FJ. Expressions such as `new A()`, `new B()`, and `new Pair(e1, e2)` are *object constructors*, and `e.setfst(e')` is an example of a *method invocation*. In the body of `setfst`, the expression `this.snd` is a *field access*, and the occurrences of `newfst` and `this` are *variables*. The syntax of FJ differs from Java in that `this` is a variable rather than a keyword.

The remaining form of expression is a *cast*. The expression

```
((Pair)new Pair(new Pair(new A(), new B()), new A()).fst).snd
```

evaluates to the expression `new B()`. Here, `((Pair)e)`, where `e` is `new Pair(...).fst`, is a cast. The cast is required because `e` is a field access to `fst`, which is declared to contain an `Object`, whereas the next field access, to `snd`, is only valid on a `Pair`. At run time, it is checked whether the `Object` stored in the `fst` field is a `Pair` (and the check succeeds in this case).

Since FJ has no side effects, evaluation can be formalized entirely within the syntax of FJ, with no additional mechanisms for modeling the heap. Moreover, in the absence of side effects, the order in which expressions are evaluated *does not affect the final outcome*, so we can define the operational semantics of FJ straightforwardly using a nondeterministic small-step reduction relation.

There are three basic reduction rules: one for field access, one for method invocation, and one for casts. The reduction rules assume that the object operated upon is first simplified to a `new` expression, like the call-by-value reduction strategy for the λ-calculus.

Here is an example of the rule for field access in action:

```
new Pair(new A(), new B()).snd → new B()
```

Because of the stylized form for object constructors, we know that the constructor has one argument for each field, in the same order that the fields are declared. Here the fields are `fst` and `snd`, and an access to the `snd` field selects the second argument `new B()`.

Here is an example of the rule for method invocation in action (/ denotes substitution):

```
new Pair(new A(), new B()).setfst(new B())
→ [          new B()/newfst,                ] new Pair(newfst, this.snd)
    new Pair(new A(), new B())/this
≡ new Pair(new B(), new Pair(new A(), new B()).snd)
```

The receiver of the invocation is the object `new Pair(new A(), new B())`, so we look up the `setfst` method in the `Pair` class, where we find that it has formal argument `newfst` and body `new Pair(newfst, this.snd)`. The invocation reduces to the body with the formal argument replaced by the actual argument (*i.e.*, `new B()/newfst`), and the special variable `this` replaced by the receiver object (*i.e.*, `new Pair(new A(), new B())/this`). Note that the class of the receiver determines where to look for the body (which means that FJ supports method override), and the substitution of the receiver for `this` (which means that FJ supports "recursion through self"). If a formal argument appears more than once in the body, this may lead to duplication of the actual argument, but fortunately this causes no problem because there are no side effects in FJ.

Here is the rule for casts in action:

```
(Pair)new Pair(new A(), new B()) → new Pair(new A(), new B())
```

Once the subject of the cast is reduced to an object, it is easy to check if the class of the constructor is a subclass of the target of the cast. If so, as in the case here, the reduction removes the cast. If not, as in the expression `(A)new B()`, no rule applies and the evaluation gets *stuck*, denoting a run time error.

There are three ways in which an evaluation may get stuck: an attempt to access a field not declared for the class, an attempt to invoke a method not declared for the class, or an attempt to cast to something other than a superclass of the class. We can prove that the first two of these never happen in well typed programs and that the third never happens in well typed programs that contain no downcasts and no stupid casts, both of which are explained in Section 3.2.

As usual, we allow reductions to apply to any subexpression of an expression. Here is an example of an evaluation, where the next subexpression to be reduced at each step is underlined.

```
   ((Pair)new Pair(new Pair(new A(), new B()), new A()).fst).snd
→ ((Pair)new Pair(new A(), new B())).snd
→ new Pair(new A(), new B()).snd
→ new B()
```

We can prove the type soundness of FJ: if an expression `e` reduces to expression `e'`, and if `e` is well typed, `e'` is also well typed and its type is a *subtype* of the type of `e`.

With this informal introduction, we now proceed to a formal definition of FJ.

3 Definition of FJ

3.1 Syntax

The syntax, typing rules, and computation rules for FJ are given in Figure 1, with a few auxiliary functions in Figure 2.

Metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over argument names; `d` and `e` range over expressions; `CL` ranges over class declarations; `K` ranges over constructor declarations; and `M` ranges over method declarations.

We write $\overline{\texttt{f}}$ as a shorthand for $\texttt{f}_1, \ldots, \texttt{f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as a shorthand for $\texttt{M}_1 \ \ldots \ \texttt{M}_n$ *with no commas*. We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\texttt{x}}$ is written $\#(\overline{\texttt{x}})$. We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$", and similarly "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1\texttt{; } \ldots \texttt{; C}_n\ \texttt{f}_n\texttt{;}$", and "$\texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{= f}_1\texttt{; } \ldots \texttt{; this.f}_n\texttt{= f}_n\texttt{;}$". Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

A class table CT is a mapping from class names C to class declarations CL. A program is a pair $(CT, \texttt{e})$ of a class table CT and an expression e. To lighten the notation, we always assume a fixed class table CT. That is, no new class declaration is generated when typechecking e.

The abstract syntax of FJ declarations, constructor declarations, method declarations, and expressions is given at the top left of Figure 1. As in Java, we assume that casts bind less tightly than other forms of expressions. We assume that the set of variables includes the special variable this, but that this is never used as the name of an argument to a method.

Every class has a superclass, declared with extends. This raises a question: what is the superclass of the Object class ? We take Object as a distinguished class name whose definition does not appear in the class table. The auxiliary functions that look up fields and method declarations in the class table are equipped with special cases for Object that return an empty sequence of fields and an empty set of methods.

By looking at the class table, we can read off the subtype relation between classes. We write $\texttt{C} \texttt{ <: } \texttt{D}$ when C is a subtype of D, that is, subtyping is the reflexive and transitive closure of the immediate subclass relation given by the extends clauses in CT. It is defined formally in the middle of the left column of Figure 1.

The given class table is assumed to satisfy some sanity conditions ($dom(?)$ returns the domain of ?): (1) $CT(\texttt{C}) = \texttt{class C } \ldots$ for every $\texttt{C} \in dom(CT)$; (2) $\texttt{Object} \notin dom(CT)$; (3) for every class name C except Object appearing somewhere in CT, we have $\texttt{C} \in dom(CT)$; and (4) there are no cycles in the subtype relation induced by CT, that is, the <: relation is antisymmetric.

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 2. The fields of a class C, written $fields(\texttt{C})$, is a sequence $\overline{\texttt{C}}\ \overline{\texttt{f}}$ pairing the class of a field with its name, for all the fields declared in class C and all of its superclasses. The type of the method m in class C, written $mtype(\texttt{m}, \texttt{C})$, is a pair, written $\overline{\texttt{B}} \to \texttt{B}$, of a sequence of argument types $\overline{\texttt{B}}$ and a result type B. Similarly, the body of the method m in class C, written $mbody(\texttt{m}, \texttt{C})$, is a pair, written $(\overline{\texttt{x}}\texttt{, e})$, of a sequence of parameters $\overline{\texttt{x}}$ and an expression e. The predicate $override(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{C}_0)$ judges if a method m with argument types $\overline{\texttt{C}}$ and a result type $\texttt{C}_0$ may be defined in a subclass of D. In case of overriding, if a method with the same name is declared in the superclass, it must have the same type.

3.2 Typing

The typing rules for expressions, method declarations, and class declarations are in the right column of Figure 1. An environment Γ is a finite mapping from variables to types, written $\overline{\texttt{x}} : \overline{\texttt{C}}$.

The typing judgment for expressions has the form $\Gamma \vdash \texttt{e} \in \texttt{C}$, read "in the environment Γ, expression e has type C." The typing rules are syntax directed, with one rule for each form of expression, save that there are three rules for casts. The typing rules for constructors and method invocations check that each actual argument has a type that is a subtype of the corresponding formal argument. We abbreviate typing judgments on sequences in the obvious

way, writing $\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}}$ as a shorthand for $\Gamma \vdash \mathtt{e_1} \in \mathtt{C_1}, \cdots, \Gamma \vdash \mathtt{e_n} \in \mathtt{C_n}$ and writing $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ as a shorthand for $\mathtt{C_1} <: \mathtt{D_1}, \cdots, \mathtt{C_n} <: \mathtt{D_n}$.

There are three rules for type casts: in an *upcast* `(C)e`, the subject `e` is a subclass of the target `C`; in a *downcast* `(C)e`, the target `C` is a subclass of the subject `e`; in a *stupid* cast, the target is unrelated to the subject. Although the Java compiler rejects an expression containing a stupid cast as ill-typed, we allow stupid casts in FJ. This is because a sensible expression may be reduced to one containing a stupid cast. For example, consider the following expression which uses classes `A` and `B` as defined as in the previous section:

```
(A)(Object)new B() → (A)new B()
```

We indicate the special nature of stupid casts by including the hypothesis *stupid warning* in the typing rule for stupid casts (T-SCAST); an FJ typing corresponds to a legal Java typing only if it does not contain this rule.[1]

The typing judgment for method declarations has the form `M OK IN C`, read "method declaration `M` is okay if it occurs in class `C`." It uses the expression typing judgment on the body of a method, where the free variables are the augments of the method with their declared types, plus the special variable `this` with type `C`.

The typing judgment for class declarations has the form `CL OK`, read "class declaration `CL` is okay." It checks that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is okay.

The type of an expression may depend on the types of methods that it invokes, and the type of a method depends on the type of an expression, namely its body. So it is necessary to check that there is no ill-defined circularity here. It turns out that there is no such ill-defined circularity: the circularity is broken because the type of each method is explicitly declared. It is possible to load and use the class table before all the classes are checked as long as each class is eventually typechecked.

3.3 Evaluation

The reduction judgment is of the form $\mathtt{e} \to \mathtt{e}'$, read "expression `e` reduces to expression $\mathtt{e}'$ in one step." We write $\to^*$ for the reflexive and transitive closure of $\to$.

The reduction rules are given in the bottom left column of Figure 1. There are three reduction rules, (R-FIELD) for field access, (R-INVK) for method invocation, and (R-CAST) for casting. These rules were already explained in Section 1. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the result of replacing $\mathtt{x}_1$ by $\mathtt{d}_1$, $\mathtt{x}_2$ by $\mathtt{d}_2$, $\cdots$, $\mathtt{x}_n$ by $\mathtt{d}_n$, and `y` by `e` in expression $\mathtt{e}_0$.

The remaining reduction rules in Figure 1 specify a nondeterministic reduction strategy for FJ under which the reduction rules may be applied at any subexpression. For example, the rule (RC-INVK-ARG) does not specify whether $\mathtt{e}_0$ is reducible or not, or whether $\mathtt{e}_j$ with $j < i$ is reducible or not. To be strict, therefore, an expression can be reduced to a unique expression in several different ways and there is no single correct order of reduction in FJ.

For this programming test, we will choose a specific deterministic reduction strategy that can be thought of as an analogue of the call-by-value reduction strategy for the λ-calculus. This assumption will simplify the overall implementation of the evaluator and save you a lot of time!

- In the rules (R-FIELD), (R-INVK), and (R-CAST), `new C(`$\overline{\mathtt{e}}$`)` should not be reducible.
- In the rule (RC-INVK-ARG), $\mathtt{e}_0$ should not be reducible.

[1] Stupid casts were omitted from Java, causing its published proof of type soundness to be incorrect.

- In the rule (RC-INVK-ARG), $\texttt{e}_j$ with $j < i$ should not be reducible.
- In the rule (RC-NEW-ARG), $\texttt{e}_j$ with $j < i$ should not be reducible.

Then a value (which is not further reducible) must always be of the form `new C(`$\overline{\texttt{v}}$`)` where each v is also a value. (Why?)

4 What to hand in

Download `hw9.zip` and unzip it on your working directory. You will implement two functions `typeOf` and `step` in `typingEval.ml`:

```
(* computes the type of a program, or raises TypeError if no type exists *)
val typeOf : Fjava.program -> Fjava.typ

(* one-step evaluation, raises Stuck if impossible *)
val step : Fjava.program -> Fjava.program
```

Given a program of type `FJava.program` which consists of class declarations and an expression, your implementation needs to first verify that each class declaration is okay (according to the rule T-CLASS) and then compute the type of the expression. If some class declaration is not okay or the expression has no type, `TypeError` should be raised. When a stupid cast warning is issued by the rule (T-SCAST), your implementation should print `Stupid Warning`. Just print a string `"Stupid Warning\n"`.

To test your implementation, first open the structure `Loop` and then use the function `loopFile` as follows:

```
# loopFile "turing.fj" (step (wait showType));
...
```

The stub file includes five FJ files: `bool.fj`, `test_downcast.fj`, `test_override.fj`, `test_stupidcast.fj`, and `turing.fj`. (Use the included Java program `src.java` to see how `turing.fj` works.)

Strive to make your code as concise as possible. In the sample solution, `typingEval.ml` is about 200 lines long, including comments.

Submission instruction

Please submit your implementation `typingEval.ml` to your hand-in directory. If your Hemos ID is `foo`, your hand-in directory is:

```
/home/class/cs321/handin/foo/
```

Please double-check that your code compiles. You get no credit for the code that fails to compile!

Good luck !