$\boxed{\textbf{CIS 500 Software Foundations}}$

Homework Assignment 9

Object encodings; Featherweight Java

Due: Wednesday, December 7, 2005, by noon

Submission instructions:

You must submit your solutions electronically (in ascii, postscript, or PDF format). Electronic solutions should be submitted following the same instructions as last time; these can be found at `http://www.seas.upenn.edu/~cis500/homework.html`. Do **not** email your solutions to us.

The first three exercises consider an implementation of a class representing a set of integers, in the style of TAPL Section 18.11. With this encoding, the type of a set of integers is:

```
Set = {
  contains : Nat -> Bool,
  add : Nat -> Unit,
  add2 : Nat -> Nat -> Unit,
  remove : Nat -> Unit
}
```

This class contains a method for checking whether or not an element is in the set (`contains`) and methods for adding and removing elements (`add` and `remove`). The method `add2` simply adds both of its arguments to the set, naturally implemented by calling the `add` method twice. The type for the internal representation of sets will be:

```
SetRep = {x : Ref (Nat -> Bool)}
```

Following TAPL 18.11, we can implement the set class as below:

```
setClass =
  λr : SetRep.
    λthis : Unit -> Set.
      λ_ : Unit.
        { contains = λn:Nat. (!(r.x)) n,
          add = λn:Nat.
            let old = (!(r.x)) in
            r.x := λm:Nat. if m = n then true else old m,
          add2 =
            λm:Nat. λn:Nat.
            (this unit).add m;
            (this unit).add n,
          remove = λn:Nat.
            let old = (!(r.x)) in
            r.x := λm:Nat. if m = n then false else old m,
        }
```

Finally, a function to construct a new set might look like:

```
newEmptySet =
  λ_ : Unit. fix (setClass {x = ref (λn:Nat. false)}) unit
```

1 Exercise

Consider another definition for the constructor of an empty set:

```
newEmptySet' = fix (setClass {x = ref (λn:Nat. false)})
```

- Does `newEmptySet'` have the same type as `newEmptySet`?
- Does `newEmptySet'` have the same behavior as `newEmptySet` (think about the issue of divergence)?

2 Exercise Now consider a subclass of `Set` that has a method returning the size of the set. That is, we would like an object of type:

```
SizeSet = {
  contains : Nat -> Bool,
  add : Nat -> Unit,
  add2 : Nat -> Nat -> Unit,
  remove : Nat -> Unit,
  size : Unit -> Nat
}
```

- Define a type `SizeSetRep` for the internal representation used by this class. (It should be a subtype of `SetRep`, of course.)
- Using this representation, write an implementation of a `sizeSetClass` as a subclass of the `setClass`. In particular, you must define the behavior of the method `add2` with inheritance.
- Write a function to construct a new, empty `SizeSet`.

3 Exercise Consider a subclass of `Set` that adds a union operation:

```
UnionSet = {
  contains : Nat -> Bool,
  add : Nat -> Unit,
  add2 : Nat -> Nat -> Unit,
  remove : Nat -> Unit,
  union : Set -> Unit
}
```

The `union` operation will update the object by adding all of the elements in the set given as an argument. Are there any problems in implementing an object of this type (with the desired behavior for `union`)? You may assume that the internal representation could be entirely changed and that the language contains other data structures such as list and trees. Are there any problems implementing such an object in Java?

The following four exercises are based on an extension of Featherweight Java with exceptions (TAPL 19.4.4). They build upon each other.

4 Exercise Extend the syntax of Featherweight Java with `throw` and `try` forms.

`t`	::=	...	terms:
		`throw t`	throw an exception
		`try t` $\overline{\texttt{cl}}$	handle exceptions
`cl`	::=	`catch (C x) t`	exception handler

Some of the new operational semantics are:

- If the thrown exception is a subtype of the first clause in the exception handler, that code is executed.

$$\frac{\texttt{D <: C}}{\texttt{try (throw new D}(\overline{\texttt{v}})\texttt{) (catch (C x) t) }\overline{\texttt{cl}} \longrightarrow [\texttt{x} \mapsto \texttt{new D}(\overline{\texttt{v}})]\texttt{t}} \text{ E-CATCH}$$

- If the exception is the wrong type for the first handler, check the rest of the handlers.

$$\frac{\texttt{D} \not<: \texttt{C}}{\texttt{try (throw new D}(\overline{\texttt{v}})\texttt{) (catch (C x) t) }\overline{\texttt{cl}} \longrightarrow \texttt{try (throw new D}(\overline{\texttt{v}})\texttt{) }\overline{\texttt{cl}}} \text{ E-NEXT}$$

- If there are no handlers left, the exception continues to propagate.

$$\frac{}{\texttt{try (throw v)} \longrightarrow \texttt{throw v}} \text{ E-THROW-TRY}$$

- For method calls, if we throw an exception while evaluating the receiver of a method invocation, we propagate it.

$$\frac{}{\texttt{(throw v).m}(\overline{\texttt{t}}) \longrightarrow \texttt{throw v}} \text{ E-THROW-RECV}$$

- If we throw an exception while evaluating one of the arguments of a method call, we propagate it.

$$\frac{}{\texttt{v}_0\texttt{.m}(\overline{\texttt{v}}, \texttt{throw v}, \overline{\texttt{t}}) \longrightarrow \texttt{throw v}} \text{ E-THROW-ARG}$$

Question: What other rules should we add to the operational semantics?

5 Exercise Another feature of Java is that method types keep track of what exceptions could be thrown during evaluation of that method. We'll do the something similar here by adding a `throws` clause to method declarations. Thus, we have:

$$\texttt{M} \quad ::= \quad \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}}) \texttt{ throws } \widehat{\texttt{E}} \ \{ \texttt{ return t; } \} \qquad \text{method declaration}$$

Here, $\widehat{\texttt{E}}$ denotes a *set* of class names; this is in contrast to $\overline{\texttt{E}}$ which denotes a list (or sequence) of class names. Many other things in Featherweight Java need to be modified in order to accommodate this change.

- To make sure that this `throws` declaration is correct, the typing judgment must also calculate the set of exceptions that *might* be thrown when a term is evaluated. Thus, they now take the following form:

$$\Gamma \vdash \texttt{t} : \texttt{D throws } \widehat{\texttt{E}}$$

  When $\widehat{\texttt{E}}$ is empty, we abbreviate this judgment to $\Gamma \vdash \texttt{t} : \texttt{D}$. In general, we'll calculate $\widehat{\texttt{E}}$ in a conservative manner; that is, $\widehat{\texttt{E}}$ may sometimes contain types (i.e., class names) that `t` cannot possibly throw or it may contain "redundant" class names.

- We "know" that a `throw` expression will throw an exception of type `C`. Like `raise` in Chapter 14, the `throw` expression itself can have any type.

$$\frac{\Gamma \vdash \texttt{t} : \texttt{C throws } \widehat{\texttt{E}}}{\Gamma \vdash \texttt{throw t} : \texttt{D throws } \{\texttt{C}\} \cup \widehat{\texttt{E}}} \text{ T-THROW}$$

  Note that this rule conservatively calculates the set of exceptions that can be thrown by an expression. For example, the following derivation is valid, assuming `A` is a class with a zero-argument constructor.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{new A()} : \texttt{A}}\ \text{T-New}}{\Gamma \vdash \texttt{throw new A()} : \texttt{B throws } \{\texttt{A}\}}\ \text{T-Throw}}{\Gamma \vdash \texttt{throw (throw new A())} : \texttt{C throws } \{\texttt{A}, \texttt{B}\}}\ \text{T-Throw}$$

However, it's clear that `throw (throw new A())` throws only an exception of type `A`.

- In a `try` expression, we need to figure out what sorts of exceptions could be thrown in the "body" and what sorts can be handled by the exception handlers.

$$\dfrac{\Gamma \vdash \texttt{t} : \texttt{A}_0 \texttt{ throws } \widehat{\texttt{C}} \qquad \Gamma \vdash \overline{\texttt{cl}} : \texttt{A throws } \widehat{\texttt{D}} \texttt{ handles } \widehat{\texttt{E}} \qquad \texttt{A}_0 \texttt{ <: A}}{\Gamma \vdash \texttt{try t } \overline{\texttt{cl}} : \texttt{A throws } (\widehat{\texttt{C}} - \widehat{\texttt{E}}) \cup \widehat{\texttt{D}}}\ \text{T-Try}$$

  Note that $\widehat{\texttt{C}} - \widehat{\texttt{E}}$ is defined as

$$\{\texttt{A} \in \widehat{\texttt{C}} \mid \texttt{A} \text{ is not a subtype of any type in } \{\texttt{E}_1, \ldots, \texttt{E}_n\}\}\,.$$

  This rule relies on a new judgment that type checks the `catch` clauses of a `try` expression. This judgment, which takes the form $\Gamma \vdash \overline{\texttt{cl}} : \texttt{A throws } \widehat{\texttt{D}} \texttt{ handles } \widehat{\texttt{E}}$, determines the type of each exception handling clause, the set of exceptions that could be thrown when evaluating the handler, and the exceptions that the clauses handle. We define this judgment with the single rule:

$$\dfrac{\begin{array}{c}\Gamma, \texttt{x} : \overline{\texttt{C}} \vdash \overline{\texttt{t}} : \overline{\texttt{A}} \texttt{ throws } \overline{\widehat{\texttt{B}}} \\ \texttt{A}_\texttt{i} <: \texttt{A} \quad \widehat{\texttt{B}} = \bigcup \widehat{\texttt{B}_\texttt{i}} \quad \widehat{\texttt{C}} = \bigcup \{\texttt{C}_\texttt{i}\}\end{array}}{\Gamma \vdash \overline{\texttt{catch (C x) t}} : \texttt{A throws } \widehat{\texttt{B}} \texttt{ handles } \widehat{\texttt{C}}}\ \text{T-Handler}$$

  Note that $\Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{A}} \texttt{ throws } \overline{\widehat{\texttt{B}}}$ is short-hand for $\Gamma \vdash \texttt{t}_1 : \texttt{A}_1 \texttt{ throws } \widehat{\texttt{B}_1}$, $\Gamma \vdash \texttt{t}_2 : \texttt{A}_2 \texttt{ throws } \widehat{\texttt{B}_2}$, ..., $\Gamma \vdash \texttt{t}_\texttt{n} : \texttt{A}_\texttt{n} \texttt{ throws } \widehat{\texttt{B}_\texttt{n}}$.

  For example, the following should be derivable (note again that our analysis is conservative here):

  - $\Gamma \vdash \texttt{try (throw new A()) (catch (A x) (throw new A()))} : \texttt{C throws } \{\texttt{A}\}$
  - $\Gamma \vdash \texttt{try (throw new A()) (catch (B x) (throw new C()))} : \texttt{D throws } \{\texttt{A}, \texttt{C}\}$ when $\texttt{A} \not<: \texttt{B}$.
  - $\Gamma \vdash \texttt{try (throw new A()) (catch (B x) (throw new C()))} : \texttt{D throws } \{\texttt{C}\}$ when $\texttt{A} <: \texttt{B}$.

- Now consider type checking a method call. The exceptions that could be thrown in this case include those that could be thrown when computing the receiver of the method invocation, computing the arguments, or executing the actual method call.

$$\dfrac{\begin{array}{ll}\Gamma \vdash \texttt{t}_0 : \texttt{C}_0 \texttt{ throws } \widehat{\texttt{A}} & \mathit{mtype}(\texttt{m}, \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{C throws } \widehat{\texttt{E}} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{C}} \texttt{ throws } \overline{\widehat{\texttt{B}}} & \overline{\texttt{C}} <: \overline{\texttt{D}}\end{array}}{\Gamma \vdash \texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) : \texttt{C throws } \widehat{\texttt{E}} \cup \widehat{\texttt{A}} \cup \bigcup \widehat{\texttt{B}_\texttt{i}}}\ \text{T-Invk}$$

  This is where the `throws` annotation on method declarations comes in. We must modify *mtype* so that it returns this set, as in:

$$\mathit{mtype}(\texttt{m}, \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{C throws } \widehat{\texttt{E}}$$

- We also have to modify the definitions of *mbody* and *override*. For example, the rules for *mbody* become:

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{mbody(\texttt{m}, \texttt{C}) = mbody(\texttt{m}, \texttt{D})}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}}) \texttt{ throws } \widehat{\texttt{E}}\ \{\texttt{return t};\} \in \overline{\texttt{M}} \end{array}}{mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$$

- We need to modify the rule for type-checking methods in order to take into account `throws` clauses.

$$\frac{\begin{array}{c} \overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{t}_0 : \texttt{E}_0 \texttt{ throws } \widehat{\texttt{A}} \\ \texttt{E}_0 \texttt{ <: } \texttt{C}_0 \\ CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\} \\ override(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{C}_0 \texttt{ throws } \widehat{\texttt{E}}) \\ \text{each type in } \widehat{\texttt{A}} \text{ is a subtype of some type in } \widehat{\texttt{E}} \end{array}}{\texttt{C}_0\ \texttt{m}(\overline{\texttt{C}}\ \overline{\texttt{x}}) \texttt{ throws } \widehat{\texttt{E}}\ \{\texttt{return t}_0;\} \texttt{ OK in C}}$$

Some questions now:

1. What are the rule(s) for defining *mtype*?
2. What are the rule(s) for defining *override*?
3. How should we state the rules T-Var, T-Field, T-New, T-UCast, T-DCast, and T-SCast so that they also calculate the set of exceptions that could be thrown? (While we introduce `ClassCastException` in a later exercise, you do *not* need to worry about it here since none of our evaluation rules mention it yet.)

6 Exercise Prove the preservation lemma for this extension by induction on the given typing derivation. The preservation lemma states:

> If $\Gamma \vdash \texttt{t} : \texttt{C} \texttt{ throws } \widehat{\texttt{E}}$ and $\texttt{t} \longrightarrow \texttt{t}'$ then $\Gamma \vdash \texttt{t}' : \texttt{D} \texttt{ throws } \widehat{\texttt{F}}$ where $\texttt{D} \texttt{ <: } \texttt{C}$ and each class in $\widehat{\texttt{F}}$ is a subtype of some class in $\widehat{\texttt{E}}$.

You only need to show the cases for T-Try and T-Invk. You may use the following lemmas without proof:

- If $\textit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{f}}$ and $\texttt{C} \texttt{ <: } \texttt{D}$, then $\textit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}}, \overline{\texttt{C}}\ \overline{\texttt{g}}$ for some $\overline{\texttt{C}}$ and $\overline{\texttt{g}}$.
- If $\Gamma \vdash \texttt{v} : \texttt{C} \texttt{ throws } \widehat{\texttt{E}}$, then $\widehat{\texttt{E}} = \emptyset$.
- If $\Gamma, \overline{\texttt{x}} : \overline{\texttt{B}} \vdash \texttt{t} : \texttt{D} \texttt{ throws } \widehat{\texttt{E}}$ and $\Gamma \vdash \overline{\texttt{v}} : \overline{\texttt{A}}$ where $\overline{\texttt{A}} \texttt{ <: } \overline{\texttt{B}}$, then $\Gamma \vdash [\overline{\texttt{x}} \mapsto \overline{\texttt{v}}]\texttt{t} : \texttt{C} \texttt{ throws } \widehat{\texttt{E}'}$ for some $\texttt{C} \texttt{ <: } \texttt{D}$ and where each type in $\widehat{\texttt{E}'}$ is a subtype of some type in $\widehat{\texttt{E}}$. (Note that we're substituting in *values* here!)
- Any inversion lemmas or canonical forms lemmas (make sure to at least state the ones you use, though).

7 Exercise Suppose we add the following evaluation rule

$$\frac{\texttt{C} \not\texttt{<:} \texttt{ D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{v}})) \longrightarrow \texttt{throw new ClassCastException}()}$$

where `ClassCastException` is a class with no fields or methods, and a zero argument constructor. How should we state the progress lemma now? Keep in mind that we haven't changed any typing rules from previous exercises.

8 Exercise Why does Java separate `RuntimeException`s from other `Exception`s? Note that `RuntimeException`s are exempt from compile-time checks: they do not need to appear in the `throws` clause of a method declaration, nor do they need to be handled by a `catch` clause. This is not the case for other `Exception`s.

9 Debriefing

1. How many hours did each person in your group spend on this assignment, including time taken to read TAPL?
2. Would you rate it as easy, moderate, or difficult?
3. Did everyone in your study group participate?
4. How deeply do you feel you understand the material it covers (0%–100%)?

If you have any other comments, we would like to hear them; please send them to `cis500@cis.upenn.edu`.