Programming Languages — Homework 9
Objects

Due: Friday, 31 May 2013, 23:55

1. [3 pts] **Subtyping in Dart**

   In the new language Dart, programs are dynamically typed, but the language features an *optional* static type system that can be used to detect type errors in programs before execution. The static type system is rather unusual in that it does not detect all type errors statically even where it is used.

   In Java, one can only assign a subtype into a supertype variable, but Dart is more permissive. In Java we'd write the typing rule:

$$\frac{x : T_1 \in \Gamma \qquad \Gamma \vdash e : T_2 \qquad T_2 <: T_1}{\Gamma \vdash x = e : T_1}$$

   In this problem, you'll write the correponding rule(s) for assignment in Dart.

   Visit `http://dartlang.org/`, download the Dart Editor, and try to determine the static semantics of local variable assignment. Try creating some classes with different subtyping relationships and writing assignments statements. When the optional type system rejects an assignment, the editor will report a warning and underline the offending expression with a yellow squiggly line.

   Be sure static checking is enabled in the editor (it should by enabled by default). Write one or more inference rules to specify typing of assignment expressions $x = e$. Typing judgments should be of the form:

$$\Gamma \vdash e : T$$

   where Γ is a typing environment mapping variables to types. Environments Γ are sets of type assignments drawn from the following grammar:

$$\Gamma ::= \cdot \mid \Gamma, x : T$$

   The only expressions of interest are assignments and variables:

$$e ::= x \mid x = e$$

   As usual, variables are typed as by looking them up the in typing environment.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

   Subtyping judgments should be of the form:

$$T_1 <: T_2$$

   Subclasses in Dart are subtypes. Do not define the subtyping relation formally, just the typing rule for assignment. Justify your rules with example Dart programs.

   Write a short example program in Dart that compiles without warnings but where a type error occurs at runtime. The error might look something like:

```
Exception: type 'A' is not a subtype of type 'B' of 'b'.
```

2. [2 pts] **Generics in Dart**

   Consider the following Dart code:

```
class A { int a; }
class B extends A { int b; }
class C<T> { /* THIS */ }

void main() {
  C<B> ob = new C<B>();
  C<A> oa = ob; /* BAD */
  test(oa, ob);
}

void test(C<A> oa, C<B> ob) {
  /* THAT */
}
```

   This program should compile in the Dart editor without any warnings or errors.

   The assignment in `main` labeled `BAD` is allowed in Dart, In this example, `C<B>` is a subtype of `C<A>` because `B` is a subtype of `A`. This *covariant* subtyping relationship would *not* hold in Java.

   Allowing covariant subtyping can lead to type errors. Add fields or methods to `C` at `THIS` and add code to the `test` method at `THAT` that will cause the code to compile without additional warnings or errors but produce a run-time type error when executed. The error should occur somewhere in `THIS` or `THAT`. Use only the classes `A`, `B`, and `C`; do not introduce unrelated code (e.g., the code you wrote for part 1) to force a type error.

3. [2 pts] **Super**

   In Java, a method can override a method in its superclass that has the same signature. The overriding method in the subclass can call the superclass method through the special `super` receiver.

```
class Window {
  void paint() {
    ... /* paint the background */
  }
}
class Button extends Window {
  void paint() {
    super.paint(); /* paint the background */
    ... /* paint the button label */
  }
}
```

   The language β inverts this design. Rather than having subclasses invoke superclass methods via the `super` keyword; instead, superclasses can invoke subclass methods via the `sub` keyword. Rewriting the example in β (with Java-like syntax):

```
class Window {
  void paint() {
    ... /* paint the background */
    sub.paint(); /* paint the rest of the window */
  }
}
class Button extends Window {
  void paint() {
    ... /* paint the button label */
  }
}
```

When `Window.paint` invokes the `paint` method on `sub`, `Button.paint` is invoked if `this` is indeed a `Button`.

(a) What are some advantages of the two approaches? Disadvantages?

(b) How could `sub` be simulated in Java? Rewrite the code above in legal Java so that `Window.paint` calls `Button.paint`.

(c) Note that the `paint` method is `void`. Does it make sense to invoke a non-`void` method using `sub`? Discuss any problems that might occur if this were allowed and suggest a solution.

4. [3 pts] **Type checking**

In this problem, you'll implements a type-checker for a small Java-like language called Nespresso.[1] Nespresso has classes, interfaces, methods, fields, constructors, and not much else. Programs consist of a sequence of class and interface declarations and a single "main" expression.

As in Java, classes are singly inherited and interfaces may be multiply inherited. Each class contains—in the following order—zero or more field declarations (just a type and a name, no initializer), a single constructor, and then zero or more methods. All fields are final and must be initialized in the constructor. To prevent accesses to uninitialized fields, Expressions that occur super-constructor calls and the right-hand-side of field initializers cannot mention `this`. All methods must have a body consisting of a single return statement. Interfaces contain only abstract methods.

Like in Java, method return types are covariant. Unlike Java, method argument types are contravariant. Method overloading is not supported, so if two methods in an inheritance hierarchy have the same name, one overrides the other.

`Nespresso.hs` contains an interpreter for Nespresso. The interpreter will fail at run-time if there is a type error. Some of the type-checking code is already implemented—checking if methods are overridden correctly, type-checking of constructor bodies, etc. You will finish the implementation. A `TODO` comment is left at each place you should write code.

Specifically, you should:

(a) Extend `subtype` to handle interfaces.

(b) Extend `checkClass` to check that if a class extends an interface, all methods of that interface are either implemented or are inherited from another class.

(c) Extend `compatibleSignature` to check that a method that overrides another method has covariant return types and contravariant parameter types.

All type-checking functions return a `TypeCheck a`, which is either a value of type `a` or is an error to be printed. Many of the functions only check for errors and don't return any value. These return a `TypeCheck ()`.

Submission

1. Complete the survey linked from the course web page after completing this assignment.
2. Submit your code and solutions on Moodle by 23:55 on Wednesday, 31 May 2013. Include your name in each file you submit.

[1] Not endorsed by George Clooney.