Homework 5: Semantics of AspectJ

Due: Tuesday, November 18, 2003.

This homework can be done in groups or individually. Its purpose is to help you explore the semantics of AspectJ.

Don't hesitate to contact the staff if you are not clear about what to do.

See the syllabus for readings for this homework, which include the article by Kiczales, *et al.* we passed out, as well as the material on the `aspectj.org` website.

1. (50 points) Find and write down, as precisely as you can, AspectJ's type checking rules for the parts of the language having to do with advice declarations and pointcut declarations. (You can ignore features related to static introductions, aspects as a whole, and the "declare" primitive.) These are rules that AspectJ actually either documents or enforces.

   For maximum credit, you should use the type-inference rule style given in class for the simply-typed λ-calculus. (But you can use math with English explanations if necessary.)

   Please include an example, showing how the rule's violation gives a type error, for each rule you find.

   If you find more than 10 rules, or have more than 3 pages of text, you can stop, unless you are working in a group, in which case you should multiply these limits by the number of people in your group. Of course, if you have covered all of the rules for the relevant syntax, then you should also stop.

   For example, as we discussed in class, suppose that we use type checking judgements for statements, expressions, point cut descriptions, advice declarations, and subtyping judgments in the following forms:

$$\begin{aligned}
\mathit{Types}; TE &\vdash E : \mathsf{exp}\ JT \uparrow ES && (1)\\
\mathit{Types}; TE &\vdash B : \mathsf{comm}\ JT \uparrow ES && (2)\\
\mathit{Types}; TE &\vdash PC : \mathsf{pntct}\ TE' && (3)\\
\mathit{Types}; TE &\vdash AD : \mathsf{decl}\ \{\} && (4)\\
\mathit{Types} &\vdash ES' <: ES && (5)
\end{aligned}$$

   where the assumptions $\mathit{Types}; TE$ contains information about the program's types, in Types, a type environment in TE, and E is an expression, B a body (statement), PC a pointcut declaration, AD an advice declaration, JT a Java type (i.e., one in Types), and ES is a set of exception types. Thus form (1) says that with assumptions in $\mathit{Types}; TE$, the expression E type checks with type JT and may throw checked exceptions from the set ES. Similarly, form (2) says that with assumptions in $\mathit{Types}; TE$, the body B type checks, may return (via a `return` statement) type JT, and may throw checked exceptions from the set ES. Form (3) says that a pointcut typechecks and its type is $\mathsf{pntct}\ TE'$, where I'm thinking of TE' containing information about what "bindings" are made by `args`, `this`, and `target` pointcuts. Form (4) says that an advice declaration type checks okay under the given assumptions; its type contains an empty type environment, as it's always useful to have declaration types contain type environments that map the names declared to their types [2]. Finally, judgement (5) says that for each exception type JT' in ES', there is some exception type JT in ES such that $\mathit{Types} \vdash JT' <: JT$. I can't promise these judgements have all the information you might want in them.

   Then we might have something like the following, which says that a formal declared in a `before` advice is available in the pointcut and body of that advice with the same type, that the pointcut must "bind" the formal (since the resulting type of PC contains the miniature

environment $\{I : JT\}$), and that the body cannot use `return` to return a value, and that it may not throw any checked exceptions (since none were declared on the advice).

$$\frac{\begin{array}{l} \mathit{Types}; TE, I : JT \vdash PC : \mathsf{pntct}\ \{x : JT\}, \\ \mathit{Types}; TE, I : JT, \texttt{this} : JT_c \vdash B : \mathsf{comm}\ \texttt{void} \uparrow ES', \\ \mathit{Types} \vdash ES' <: \{\} \end{array}}{\mathit{Types}; TE \vdash \texttt{before(}JT\ I\texttt{)}\ :\ PC\ \texttt{\{}\ B\ \texttt{\}} : \mathsf{decl}\ \{\}} \quad \text{if } (\texttt{aspect} : JT_c) \in TE,$$

Note that we put `this` in the environment to allow the body to have the correct type for the `this` expression, and we got its type from the special variable `aspect`, which the rule for aspect declarations has to place in the environment TE. We would have to define subtyping on exception sets to make sense of the judgement $\mathit{Types} \vdash ES' <: ES$. Also, this rule is intended only as an example. It's certainly a special case. So you may need to change the types and typing rules for various parts of such a rule to fit in with what you understand.

An example that illustrates this rule is the following.

```
1  public aspect Before1Arg {
2      before (Float f)
3      : args(f) { // (binding for formal f, ok)
4          Float g = f; // (f has type Float, ok)
5          f = true;  // illegal, f has type Float
6      }
7
8      before (Float f)
9      : args(String) { // illegal, no binding for f
10         return 3.14159; // illegal, no result in before advice
11     }
12 }
```

The above example produces the following output.

```
Before1Arg.java:8 formal unbound in pointcut
Before1Arg.java:5 Type mismatch: cannot convert from boolean to Float
f = true;  // illegal, f has type Float
    ^^^^
Before1Arg.java:10 Void methods cannot return a value
return 3.14159; // illegal, no result in before advice
^^^^^^^^^^^^^^

3 errors
```

Note that you don't need to use all this (or any) formalism if it's too difficult for you. For example, you could state the above rule by simply writing some version of the English in the paragraph before it, and giving an example.

2. (30 points) Describe type checking rules related to the parts of AspectJ described in the previous problem that AspectJ does not currently document or enforce, but which it should. For each of these, give an example, and a description of why the language's users would benefit from your new rules.

   Again, describe your new rules as precisely as you can. You can stop if you find more than 3 rules or more than 2 pages of text, unless you are working in a group, in which case you should multiply these limits by the number of people in your group. Again, if you have achieved some sort of completeness, you can also stop.

3. (30 points) What kinds of advice in AspectJ can be considered syntactic sugars? For each of these, if any, describe how to desugar the advice to other kinds of advice. Give a working example, which runs in AspectJ, and an explanation of why it should work.

4. (50 points; extra credit) Describe type checking rules related to other parts of AspectJ that AspectJ does not currently document or enforce, but which it should.

5. (100 points; extra credit) Describe how AspectJ's point cuts could be made less dependent on implementation details, while retaining the language's "obliviousness" property.

References

[1] AspectJ Team. The AspectJ programming guide. Available from `http://eclipse.org/aspectj`, October 2003.

[2] David A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.