CSC 548: Lecture 2

Overview

Type checking

Symbol tables

Source-to-source transforms

Type inference as a source-to-source transform

Issues in typechecking

Homework

Type checking

What is type checking? Type inference?

When does type checking happen?

Why bother?

How can we implement a type checker?

Type checking

What is a symbol table (aka type context)? Why is one needed?

Try typechecking this:

```
let flag:Boolean = True:Boolean;
if (flag:Boolean) { return 0; }
else { return 1; }
```

What about this:

```
let flag:Boolean = "Hello";
if (flag:Boolean) { return 0; }
else { return 1; }
```

Or this:

```
let flag:String = "Hello";
if (flag:Boolean) { return 0; }
else { return 1; }
```

Type checking

```
void BlockLet.typecheck (TypeContext ctxt, Type type) {
    Var var = getVar ();
    Exp exp = getExp ();
    Block rest = getRest ();
    ... what goes here...?
}
```

Uses:

```
Type Var.getType ();
TypeContext TypeContext.addVar (Var);
```

and added by aspect TypeCheckAspect:

```
Type Exp.getType (TypeContext);
boolean Type.isSubType (TypeContext, Type);
void Block.typecheck (TypeContext, Type);
```

What about if statements?

Type checking

```
Type Var.getType (TypeContext ctxt) {
  ... what goes here...?
}
```

Uses:

```
Type Var.getType ();
Var TypeContext.getVar (String);
```

and added by aspect TypeCheckAspect:

```
boolean Type.isSubType (TypeContext, Type);
```

Check: let flag:String = "Hello"; if (flag:Boolean)....

Full code is in hobbes/statics/TypeCheck.java.

Symbol tables

Two implementation strategies: mutable vs immutable symbol tables. What are these?

What changes to the code for type checking `let` and `if` statements do we need to make for mutable symbol tables?

What are the trade-offs?

Full code is in `hobbes/statics/TypeContext.java`.

Source-to-source transforms

What is a source-to-source transform?

A sample source-to-source transform is TransformMethodIds.

Input:

```
let x:Integer = y:Integer.infix+ (z);
```

Output:

```
let x:Integer = y:Integer.infix_plus (z);
```

Why do this transform?

How can we perform this transform?

Full code is in hobbes/transform/TransformMethodIds.java.

Type inference as a source-to-source transform

What is type inference?

How can we implement type inference as a source-to-source transform?

Full code is in hobbes/transform/TransformTypeInfer.java.

Issues in typechecking: recursion

How can we deal with code like this:

```
class Foo {
  method b () : Bar { ... }
}
class Bar {
  method f () : Foo { ... }
}
```

Issues in typechecking: subtyping

Imagine we added Java-like interfaces to Hobbes:

```
interface Foo { method getA () : Integer; }
interface Bar extends Foo { method getB () : Integer; }
class BarImpl implements Bar { ... }
```

What changes would we need to make to the typechecker?

Note: subtyping is tricky to get right! (Java's type system has holes caused by subtyping...)

More on this when we cover OO...

Issues in typechecking: effects

Some langauges support other kinds of typechecking, not just return types. For example:

```
method foo () : Integer throws MyExn { throw new MyExn {}; }
```

These kinds of additional information are called *effects*.

What changes would we need to make to the typechecker?

Homework

Finish the Hobbes typechecker!

Summary

Type checking is a recursive descent through the AST, ensuring that all code has appropriate type.

Type checking uses a symbol table (or type context) to keep track of types of free variables.

Type inference is a source-to-source transform, which annotates all variables with type information.

Type-checking is related to recursion, subtyping and effect-checking.

Reading: Appel, Ch. 6, 11.