Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Sascha Böhme, Alexander Krauss

WS 2010/11
13. 12. 2010

Semantics of Programming Languages

Exercise Sheet 8

Exercise 8.1 Type coercions

Adding and comparing integers and reals can be allowed by introducing implicit conversions: Adding an integer and a real results in a real value, comparing an integer and a real can be done by first converting the integer into a real. Implicit conversions like this are called *coercions*.

(a) Modify, in the theory *Types*, the inductive definitions of *taval* and *tbval* such that implicit coercions are applied where necessary.

(b) Extend the datatype *com* by a loop construct *DO a TIMES c* which executes the command c exactly a times, where a is an arbitrary arithmetic expression of integer type.

(c) Adapt all proofs in the theory *Types* accordingly.

Hint: Isabelle already provides the coercion functions *nat*, *int*, and *real*.

Homework 8 Type Inference

Submission until Wednesday, January 12, 2011, 12:00 (noon).

This is a programming exercise. You are not asked to prove anything. Use the template file `Type_Inference_Template.thy`

You will implement *type inference* for our simple type system. A type inference is an algorithm that takes a program c and computes an environment Γ such that $\Gamma \vdash c$. In general there are multiple type environments for a program. For example the program $x ::= V\ y$ can be typed in any Γ where $\Gamma\ x = \Gamma\ y$.

To express multiple solutions, we introduce type variables. There is one type variable $TVar\ x$ for every program variable x, which stands for "the type assigned to program variable x". For simplicity the type *tvar* also includes constants:

datatype *tvar* = *TVar name* | *Type ty*

fun *type_of* :: *"tyenv ⇒ tvar ⇒ ty"*
where
  "type_of Γ (*TVar v*) = Γ(*v*)*"*
| *"type_of* Γ (*Type t*) = *t"*

Type inference is best implemented in two parts. The first part traverses the program and collects constraints that must hold to make the program type correct. Our constraints are simply equalities between type variables, modelled as pairs:

```
types
  constraint = "tvar × tvar"
  constraints = "constraint list"

fun constraint_holds :: "tyenv ⇒ constraint ⇒ bool" (infix "⊨" 50)
where "Γ ⊨ (v, v') ⟷ type_of Γ v = type_of Γ v'"
```

(a) Implement a function *ccollect* :: *com* ⇒ *constraints* that collects constraints from a program. Your function should have the following property, which you can test using the *quickcheck* counterexample generator.

```
lemma ccollect_sound_and_complete:
"Γ ⊢ c ⟷ (∀ C ∈ set (ccollect c). Γ ⊨ C)"
```

In the second part we must solve the constraints. Instead of a set of solutions (which could become very large) we compute a "most general typing" of which every valid typing will be an instance. It is expressed as an association list $M :: (name \times tvar)\ list$ that associates each variable occurring in the program to a type or type variable.

The following function instantiates a solution to a concrete type environment, given an instantiation *I*. Here the predefined *map_of* :: $('a \times 'b)\ list \Rightarrow 'a \Rightarrow 'b\ option$ implements the lookup operation in an association list.

```
definition instantiate :: "(name ⇒ ty) ⇒ (name × tvar) list ⇒ tyenv"
where
  "instantiate I M =
    (λx. case map_of M x of
      None ⇒ I x
    | Some (Type T) ⇒ T
    | Some (TVar y) ⇒ I y)"
```

Conversely, the following function checks if a concrete environment Γ is an instance of a solution:

```
fun is_instance :: "tyenv ⇒ (name × tvar) list ⇒ bool" (infix "<:" 50)
where
  "Γ <: [] ⟷ True"
| "Γ <: ((x,Type t)#M) ⟷ (Γ(x) = t ∧ Γ <: M)"
| "Γ <: ((x,TVar y)#M) ⟷ (Γ(x) = Γ(y) ∧ Γ <: M)"
```

(b) Implement a constraint solving algorithm as a function *solve* :: *constraints* ⇒ $(name \times tvar)\ list \Rightarrow (name \times tvar)\ list\ option$ that refines a (partial) solution using the given constraints. If a contradiction is detected, the function returns *None*.

(c) Combine the parts to a type inference algorithm. Use quickcheck to test the properties specified in the template file.