Semantics of Programming Languages
Exercise Sheet 7

Exercise 7.1 Type checker as recursive functions

Reformulate the inductive predicates $\Gamma \vdash a : \tau$, $\Gamma \vdash b$ and $\Gamma \vdash c$ as three recursive functions

fun atype :: "tyenv ⇒ aexp ⇒ ty option"
fun bok :: "tyenv ⇒ bexp ⇒ bool"
fun cok :: "tyenv ⇒ com ⇒ bool"

and prove

lemma atyping_atype: "(Γ ⊢ a : τ) = (atype Γ a = Some τ)"
lemma btyping_bok: "(Γ ⊢ b) = bok Γ b"
lemma ctyping_cok: "(Γ ⊢ c) = cok Γ c"

Exercise 7.2 Compiler optimization

A common programming idiom is IF b THEN c, i.e., the else-branch consists of a single SKIP command.

1. Look at how the program IF Less (V "x") (N 5) THEN "y" := N 3 ELSE SKIP is compiled by ccomp and identify a possible compiler optimization.
2. Implement an optimized compiler (by modifying ccomp) which reduces the number of instructions for programs of the form IF b THEN c.
3. Extend the proof of ccomp_bigstep to your modified compiler.

Homework 7 Type Inference

Submission until Tuesday, Dec 1, 10:00am.
Specifying the types of variables is annoying, in particular, as they are mostly clear from the program anyway.
In this exercise, you shall implement and prove correct a type inference scheme. The type inference goes through the program similar to atyping, btyping, ctyping. But instead
of only checking whether the specified types match the program, it computes matching types.

For this purpose, we extend types by an unknown value, which means that we do not yet know the type of that variable. If the type inference encounters a program part that determines the type of a variable typed with unknown, it will update the type environment accordingly. If type inference encounters a program part that does not match the already determined typing, it fails.

```
type_synonym ety = "ty option"
type_synonym etyenv = "vname => ety"
```

For efficiency (and simplicity) we want a one-pass type inference, i.e., we want to visit each part of the program only once. However, this causes a problem: Consider the possible types for expression \((x + y) + (x + 2.3)\). Clearly, we have that both \(x\) and \(y\) must be reals. However, when type inference is done in a top-down fashion, it will see \(x + y\) first, and infer \(x\) and \(y\) to be undetermined. Only later, if it sees the second term, it has to somehow go back and set \(y\) to be real too, although \(y\) does not occur in the second term.

To avoid this effect, we will assume that variables that we see in expressions have already a determined type, and let type inference fail otherwise. This means, that input variables of the program still need to be explicitly typed.

Define the following predicates, which determine the type of an arithmetic/Boolean expression. A type should only be returned if the types of all variables occurring in the expression are determined.

```
inductive infer_aty :: "etyenv => aexp => ty => bool"
inductive infer_bty :: "etyenv => bexp => bool"
```

A type environment is an instance of an extended type environment, if the two match on all variables with determined types:

```
definition is_inst :: "tyenv => etyenv => bool"
  where "is_inst Γ e e' ≡ ∀ x τ. e Γ x = Some τ → e' Γ x = τ"
```

Show that type inference infers a valid typing, i.e., all instances of the inferred typing are valid:

```
lemma ainfer: assumes "infer_aty e Γ a T" and "is_inst Γ e Γ" shows "atyping Γ a T"
lemma binfer: assumes "infer_bty e Γ b" and "is_inst Γ e Γ" shows "btyping Γ b"
```

Next, write a predicate that extends a typing according to a command. On an assignment, the type of the assigned variable is determined to have the type of the right hand side expression. If the assigned variable is already determined to have a different type, no typing for the program should be inferred.

On an if-statement, the inferred types for the then and else part must be combined. If combination is not possible, because a variable is determined to have two different types in the then and else part, no typing for the program should be inferred. This is expressed by the following predicate:
**definition** combine :: "etyenv ⇒ etyenv ⇒ etyenv ⇒ bool" where
"combine eΓ1 eΓ2 eΓ ≡
eΓ=eΓ1++eΓ2
∧ (∀x τ1 τ2. eΓ1 x = Some τ1 ∧ eΓ2 x = Some τ2 → τ1=τ2)"

**inductive** infer_cty :: "etyenv ⇒ com ⇒ etyenv ⇒ bool"

As a test, show that your type inference works for the following program

**definition** "test c ≡
"′′x′′ ::= Ic 0;;
(IF Less (V "x") (Ic 2) THEN SKIP ELSE "′′y′′ ::= Rc 1.0));;
"′′y′′ ::= Plus (V "y") (Rc 3.1)"

**lemma** "∃ eΓ′. infer_cty (λ . None) test c eΓ′"

As sketched below, a safe way to prove such a lemma is to apply the introduction rules manually. Of course, you may also try to automate this proof. Note that you probably have to adjust the applied introduction rules to your solution!

**unfolding** test.c.def
**apply** (rule exI)

**apply** (rule infer_cty.intros(4)) **apply** (rule infer_cty.intros(4)) **apply** (rule infer_cty.intros(2))
**apply** (rule infer_aty.intros) **apply** simp — and so on ...

Finally, prove the following lemma:

**lemma assumes** "infer_cty eΓ c eΓ′" and "is_inst Γ eΓ′" shows "ctyping Γ c"

Hint: You will need some auxiliary lemmas. The main idea is that infer_cty only determines more types, but does not change already determined ones, and that if type inference for aexp and bexp works on a type environment, it also works on a more determined type environment. You may use op ⊆_m (map_le, look it up using find_theorems!) to express that a type environment is less determined than another one. Moreover, it may be advantageous to prove some auxiliary lemmas about op ⊆_m, is_inst, combine and the relation of these concepts, rather than proving these things in the main proof.

Note: If you feel more comfortable with this, you may also develop a functional solution, in the spirit of exercise 7.1.