Semantics of Programming Languages
Exercise Sheet 8

Exercise 8.1  Type checker as recursive functions

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

\begin{align*}
\text{fun } atype &:: \text{"tyenv } \Rightarrow \text{aexp } \Rightarrow \text{ty option"} \\
\text{fun } bok &:: \text{"tyenv } \Rightarrow \text{bexp } \Rightarrow \text{bool"} \\
\text{fun } cok &:: \text{"tyenv } \Rightarrow \text{com } \Rightarrow \text{bool"}
\end{align*}

and prove

\begin{align*}
\text{lemma } atyping_{atype} &:: \text{"}(\Gamma \vdash a : \tau) = (atype \Gamma a = \text{Some } \tau)" \\
\text{lemma } btyping_{bok} &:: \text{"}(\Gamma \vdash b) = (bok \Gamma b)" \\
\text{lemma } ctyping_{cok} &:: \text{"}(\Gamma \vdash c) = (cok \Gamma c)"
\end{align*}

Exercise 8.2  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.

1. Modify, in the theory \emph{Types}, the inductive definitions of \emph{taval} and \emph{tbval} such that implicit coercions are applied where necessary.
2. Adapt all proofs in the theory \emph{Types} accordingly.

Hint: Isabelle already provides the coercion functions \emph{nat}, \emph{int}, and \emph{real}.

Homework 8.1  A Typed Language

\emph{Submission until Tuesday, December 20, 2016, 10:00am.}

Use the template file \texttt{hw08 tmpl.thy}.

We unify boolean expressions \emph{bexp} and arithmetic expressions \emph{aexp} into one expressions language \emph{exp}. We also define a datatype \emph{val} to represent either integers or booleans.
We then give a type system and small semantics. Your task is to show preservation and progress of the type system, i.e. replace all oops by valid proofs.

**Preparation 1**: We define unified values and expressions:

```plaintext
datatype val = Iv int | Bv bool
```

```plaintext
datatype exp =
  N int | V vname | Plus exp exp | Bc bool | Not exp | And exp exp | Less exp exp
```

Evaluation is now defined as an inductive predicate only working when the types of the values are correct:

```plaintext
inductive eval :: "exp \Rightarrow state \Rightarrow val \Rightarrow bool" where
  "eval (N i) s (Iv i)" |
  "eval (V x) s (s x)" |
  "eval a s (Iv i_1) \Rightarrow eval a_2 s (Iv i_2) \Rightarrow eval (Plus a_1 a_2) s (Iv (i_1 + i_2))" |
  "eval (Bc v) s (Bv v)" |
  "eval b s (Bv bv) \Rightarrow eval (Not b) s (Bv (¬ bv))" |
  "eval b_1 s (Bv bv_1) \Rightarrow eval b_2 s (Bv bv_2) \Rightarrow eval (And b_1 b_2) s (Bv (bv_1 ∧ bv_2))" |
  "eval a s (Iv i_1) \Rightarrow eval a_2 s (Iv i_2) \Rightarrow eval (Less a_1 a_2) s (Bv (i_1 < i_2))"
```

**Preparation 2**: The small-step semantics are as before, we just replaced `eval` and `beval` with `eval`.

```plaintext
inductive small_step :: "(com \times state) \Rightarrow (com \times state) \Rightarrow bool" (infix "\Rightarrow") where
  Assign: "eval a s v \Rightarrow (x := a, s) \Rightarrow (SKIP, s(x := v))" |
  IfTrue: "eval b s (Bv True) \Rightarrow (IF b THEN c_1 ELSE c_2, s) \Rightarrow (c_1,s)" |
  IfFalse: "eval b s (Bv False) \Rightarrow (IF b THEN c_1 ELSE c_2, s) \Rightarrow (c_2,s)"
```

...  

**Preparation 3**: We introduce the type system.

```plaintext
datatype ty = Ity | Bty
```

```plaintext
type_synonym tyenv = "vname \Rightarrow ty"
```

```plaintext
inductive etyping :: "tyenv \Rightarrow exp \Rightarrow ty \Rightarrow bool" where
  "(\"{1, \vdash / (\vdash : / \vdash)\") \)"
```

```plaintext
where
  "\Γ \vdash N i : Ity" |
  "\Γ \vdash V x : \Gamma x" |
  "\Γ \vdash a_1 : Ity \Rightarrow \Gamma \vdash a_2 : Ity \Rightarrow \Gamma \vdash Plus a_1 a_2 : Ity" |
  "\Γ \vdash Bc v : Bty" |
  "\Γ \vdash b : Bty \Rightarrow \Gamma \vdash Not b : Bty" |
  "\Γ \vdash b_1 : Bty \Rightarrow \Gamma \vdash b_2 : Bty \Rightarrow \Gamma \vdash And b_1 b_2 : Bty" |
  "\Γ \vdash a_1 : Ity \Rightarrow \Gamma \vdash a_2 : Ity \Rightarrow \Gamma \vdash Less a_1 a_2 : Bty"
```

```plaintext
inductive ctyping :: "tyenv \Rightarrow com \Rightarrow bool" (infix "\vdash") where
  Skip_ty: "\Γ \vdash SKIP" |
```
Assign ty: “Γ ⊢ a : ω ⇒ Γ ⊢ x := a” |
Seq ty: “Γ ⊢ c_1 ⇒ Γ ⊢ c_2 ⇒ Γ ⊢ c_1;c_2” |
If ty: “Γ ⊢ b : Bty ⇒ Γ ⊢ c_1 ⇒ Γ ⊢ c_2 ⇒ Γ ⊢ IF b THEN c_1 ELSE c_2” |
While ty: “Γ ⊢ b : Bty ⇒ Γ ⊢ c ⇒ Γ ⊢ WHILE b DO c” |

We define a state typing styping to describe the type context of a state.

fun type :: “val ⇒ ty” where
type (Iv i) = Ity |
type (Bv r) = Bty

definition styping :: “tyenv ⇒ state ⇒ bool” (infix “⊢” 50) where
“Γ ⊢ s ←→ (∀ x. type (s x) = Γ x)”

Task 1: Show preservation and progress on expressions:

lemma epreservation: “Γ ⊢ a : τ ⇒ eval a s v ⇒ Γ ⊢ s ⇒ type v = τ”
lemma eprogress: “Γ ⊢ a : τ ⇒ Γ ⊢ s ⇒ ∃ v. eval a s v”

Task 2: Show progress and preservation on commands:

theorem progress: “Γ ⊢ c ⇒ Γ ⊢ s ⇒ c ≠ SKIP ⇒ ∃ cs’. (c, s) → cs’”
theorem styping_preservation: “(c, s) → (c’, s’) ⇒ Γ ⊢ c ⇒ Γ ⊢ s ⇒ Γ ⊢ s’”
theorem ctyping_preservation: “(c, s) → (c’, s’) ⇒ Γ ⊢ c ⇒ Γ ⊢ c’”
theorem type_sound:
“(c, s) →* (c’, s’) ⇒ Γ ⊢ c ⇒ Γ ⊢ s ⇒ c’ ≠ SKIP ⇒ ∃ cs”. (c’, s’) → cs”

Hint: For most of the proof work, you should be able to closely follow the proofs in the original IMP theory.
Homework 8.2 A Type System for Physical Units

Submission until Tuesday, December 20, 2016, 10:00am.

Start with a fresh copy of Types.thy. We will define a language that only computes on real values but attaches a physical unit to every constant. The binary operators are addition and multiplication (op * in Isabelle/HOL). The semantics shall get stuck if trying to add or compare values with different physical units.

Define a type system that uses physical units as types. Well-typed programs must not add or compare values with different physical units. Adapt the theory up to the type_sound-theorem, i.e., show that in a well-typed program, every reachable non-skip state can make another step. Some steps of this development are detailed below.

Note: Please turn in two separate files for the two homework exercises.

A unit is either an elementary unit (Newton or Meters), or a product of units.

```plaintext
datatype unit = N | M | Prod unit unit
```

We only consider real values but attach units to values:

```plaintext
type_synonym val = "real × unit"
```

```plaintext
datatype aexp = Pc val | V vname | Plus aexp aexp | Mult aexp aexp
```

You will need to define an equality predicate `unit_eq :: unit ⇒ unit ⇒ bool` on units. Note that e.g. `Prod N M` should be the same as `Prod M N`.

The types are simply all possible units:

```plaintext
type_synonym ty = unit
```

It is easy to read types from values in our setting: they are already attached to them. Thus a well-typed state is expressed as follows:

```plaintext
definition styping :: "tyenv ⇒ state ⇒ bool" (infix "⊢" 50)
where "Γ ⊢ s ←→ (∀ x. unit_eq (snd (s x)) (Γ x))"
```

Note: Please turn in two separate files for the two homework exercises.