

Theory exercises

Program verification with types and logic (NWI-IMC060)

Week 5

1. Consider the syntax of a tiny typed programming language in Coq:

```
Inductive expr :=
  | EVar (x : string) : expr
  | ENat (n : nat) : expr
  | EBool (b : bool) : expr
  | ELet (x : string) (e1 e2 : expr) : expr
  | EIf (e1 e2 e3 : expr) : expr.
```

```
Inductive ty := TNat | TBool.
Notation ctx := (stringmap ty).
```

- (a) Implement a function that computes the type of a given expression:

```
Fixpoint infer (Gamma : ctx) (e : expr) : option ty :=
  (* FILL OUT HERE *)
```

You may assume you have the following helper functions at your disposal:

```
Definition type_eq_dec (ty1 ty2 : ty) : { ty1 = ty2 } + { ty1 <> ty2 }.
```

- (b) When one defines a programming language formally, one usually defines a typing judgment $\Gamma \vdash e : A$, which would be modeled in Coq as:

```
Inductive typed : ctx -> expr -> ty -> Prop := (* ... *).
```

Explain the difference between the typing judgment $\Gamma \vdash e : A$ (*i.e.*, the inductively defined predicate `typed`) and the function `infer`.

- (c) State a Coq lemma that fully captures the desired relationship between the typing judgment $\Gamma \vdash e : A$ (*i.e.*, `typed`) and the function `infer`.

2. This question is about the type preservation property.

- (a) Most typed languages satisfy the *type preservation* property, which is formally stated as:

If $\Gamma \vdash e : A$ and $e \Rightarrow e'$ then $\Gamma \vdash e' : A$

Explain why this property is considered important for programming languages.

- (b) Consider the reverse direction of the type preservation property:

If $\Gamma \vdash e : A$ and $e' \Rightarrow e$ then $\Gamma \vdash e' : A$

Explain if this property holds for functional programming languages with at least functions, Booleans, arithmetic operations, and an if-then-else statement. If the property does not hold, you should provide a concrete counterexample.

3. Consider the expressions and types of the simply-typed lambda calculus with natural numbers in Coq:

```
Inductive expr :=
  | EVal (v : val) : expr
  | EVar (x : string) : expr
  | EApp (e1 e2 : expr) : expr
  | ELam (x : string) (e : expr) : expr
with val :=
  | VNat (n : nat) : val
  | VLam (x : string) (e : expr) : val.

Inductive ty :=
  | TNat : ty
  | TFun : ty -> ty -> ty.

Notation ctx := (stringmap ty).
```

Furthermore, assume that we have a small-step semantics and typing judgment:

```
Inductive small_step : expr -> expr -> Prop := (* ... *).
Inductive typed : ctx -> expr -> ty -> Prop := (* ... *).
```

- (a) Define an inductive predicate `terminates : expr -> Prop` so that `terminates e` holds if there exists a reduction sequence from `e` to a value (*i.e.*, a lambda expression or a natural number).
- (b) For the simply-typed lambda calculus it is true that if an expression is well-typed in the empty context, then it terminates. That is, the following lemma holds:

```
Lemma typed_terminates e A :
  typed StringMap.empty e A ->
  terminates e.
```

We can try to do the proof by direct induction on the typing derivation. In Coq, this means starting our proof script with `intros H; induction H`. This approach does not work. Explain where we get stuck in the `EApp` case if we try to prove this statement directly by induction on the typing derivation.

- (c) The method of semantic typing addresses these issues. We define the `wp` predicate transformer as follows:

```
Inductive wp : expr -> (val -> Prop) -> Prop :=
  | wp_val (Phi : val -> Prop) v :
    Phi v -> wp (EVal v) Phi
  | wp_step (Phi : val -> Prop) e e' :
    small_step e e' -> wp e' Phi -> wp e Phi.
```

Explain intuitively what `wp e P` means.

- (d) To formulate a lemma that gives us a stronger induction hypothesis, we define a semantic interpretation of types `sem : ty -> (val -> Prop)` as predicates over values. The strengthened lemma will then be the following:

```
Lemma typed_sem e A :
  typed StringMap.empty e A ->
  wp e (sem A).
```

Give a definition for `sem` and explain how your definition solves the problem in the `EApp` case of a proof by induction.

- (e) Although the lemma `typed_sem` addresses the problem with `EApp`, it causes a new problem with the `ELam` case. More precisely, if we again start our proof with `intros H; induction H` we now get stuck in the `ELam` case. Explain why, and explain how we need to further strengthen the lemma to get an induction hypothesis that also works for the `ELam` case.
- (f) Consider the following lemma:

```
Lemma foo e A :
  wp e (sem A) ->
  typed StringMap.empty e A.
```

Does this lemma hold? If yes, explain why. If not, give a counterexample.

4. Assume that we have a pure, statically-typed language with a small-step operational semantics. Formally, assume that we have a set of expressions `Expr`, a set of values `Val` \subseteq `Expr`, a set of types `Type`, a small-step operational semantics $e \rightsquigarrow e'$, and a typing judgment $\vdash e : A$.

This language enjoys *type safety* if for all expressions $e \in \text{Expr}$ and types $A \in \text{Type}$ such that $\vdash e : A$, we have:

$$\forall e' \in \text{Expr}. (e \rightsquigarrow^* e') \rightarrow (e' \in \text{Val}) \vee (\exists e''. e' \rightsquigarrow e'')$$

Here, \rightsquigarrow^* is reflexive-transitive closure of the relation \rightsquigarrow .

- (a) Type safety is often proved via the method of progress and preservation. Give the statements of progress and preservation and explain why together they imply type safety.
- (b) Explain whether or not the reverse implications hold. That is:
- Does type safety imply preservation?
 - Does type safety imply progress?

In each case, either explain why the implication holds or explain with a counterexample of a language that the implication does not hold. Your counterexample should involve a definition of a set of expressions `Expr`, a set of values `Val` \subseteq `Expr`, a set of types `Type`, a small-step operational semantics $e \rightsquigarrow e'$, and a typing judgment $\vdash e : A$.

- (c) Consider what happens to type safety, preservation, and progress when we change the language. To change the language, we can add or remove typing rules, add or remove step rules from the small-step operational semantics, and add or remove elements from the set of values. Make a table containing for each possibility whether it is possible that such a modification **destroys/invalidates** the property of type safety, preservation, and progress. You do not need to explain your answer. You get $+\frac{1}{2}$ point for each correct answer and $-\frac{1}{2}$ point for each wrong answer. You may leave blank cells.

	Typing rule		Step rule		Set of values	
	Add	Remove	Add	Remove	Add	Remove
Type safety						
Preservation						
Progress						