EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät

Programmiersprachen und Softwaretechnik

Prof. Klaus Ostermann

Übungsleitung
Julian Jabs
David Binder

Interactive Theorem Proving

Homework 9 – WS18/19

Tübingen, 18. Dezember 2018

This homework is due Monday, January 9th 23:59!

Task 1: Revision & GADTs

```
(* Task 1.1 : GADTs: Types as type parameters *)

Require Import Coq.Strings.String.

Inductive expr : Type :=
  | E_N_Lit : nat -> expr
  | E_S_Lit : string -> expr
  | E_Add : expr -> expr -> expr
  | E_Repeat : expr -> expr -> expr.

Inductive type : Type := T_Nat | T_String.

Inductive typechecks : expr -> type -> Prop :=
  | T_N_Lit : forall n, typechecks (E_N_Lit n) T_Nat
  | T_S_Lit : forall s, typechecks (E_S_Lit s) T_String
  | T_Add : forall e1 e2,
      typechecks e1 T_Nat ->
      typechecks e2 T_Nat ->
      typechecks (E_Add e1 e2) T_Nat
  | T_Repeat : forall e1 e2,
      typechecks e1 T_String ->
      typechecks e2 T_Nat ->
      typechecks (E_Repeat e1 e2) T_String.

(* Your task: Fill in the following two definitions such that the lemmas below hold,
   and prove the lemmas.
 *)
Inductive expr_t : type -> Type :=.

Fixpoint erase {t : type} (e : expr_t t) : expr. Admitted.

Lemma erased_expr_t_typechecks : forall (t : type) (e : expr_t t), typechecks (erase e) t.
Proof.
```

```
  admit.
Admitted.

(* Slightly advanced. *)
Lemma erase_surjective_over_tc : forall (t : type) (e : expr),
  typechecks e t ->
  exists (e' : expr_t t), erase e' = e.
Proof.
  admit.
Admitted.

(* Task 1.2 : Relations : values as type parameters *)

Require Import Coq.Lists.List.

Inductive sublist {A} (prefix : list A) : list A -> list A -> Prop :=
  | S_Nil : forall l, sublist prefix nil ((rev prefix) ++ l)
  | S_Cons : forall a l' l, sublist (a :: prefix) l' l -> sublist prefix (a :: l') l.

(* Your task: Prove the following two lemmas. *)
(* Hint: Try to first understand the definition of sublist.
   For instance, find out what the parameter prefix is used for.
 *)

(* Advanced *)
(* Hint: use app_assoc, app_eq_nil, and app_cons_not_nil *)
Lemma sublist_correct : forall {A} (prefix l' l : list A),
  sublist prefix l' l ->
  exists l0, l = (rev prefix) ++ l' ++ l0.
Proof.
  admit.
Admitted.

(* Slightly advanced *)
(* Hint: use rev_involutive *)
Lemma sublist_complete : forall {A} (prefix l l' l0 : list A),
  l = (rev prefix) ++ l' ++ l0 ->
  sublist prefix l' l.
Proof.
  admit.
Admitted.

(* Finally, here's how the two lemmas can be combined to show
   that our sublist relation indeed does exactly what we expect from it.
 *)

Lemma sublist_equivalent : forall {A} (l' l : list A),
  (exists prefix, sublist prefix l' l) <->
  exists l1 l0, l = l1 ++ l' ++ l0.
Proof with auto.
intros. split; intros.
- destruct H. exists (rev x). apply sublist_correct...
- destruct H as [l1 [l0 H]]. exists (rev l1).
  apply sublist_complete with (l3:=l0). rewrite rev_involutive...
Qed.
```

```
(* Task 1.3 : Option and partial maps *)

(* Consider again [erase] from task 1.
   Write a function that is the inverse of erase, but where you can select the type.
   Its result is of type option since not all expr are in the image of erase
   (and even less when considering only a specific type t).
 *)
Fixpoint unerase (t : type) (e : expr) : option (expr_t t). Admitted.

(* Finally, show that the functions and the relation behave as expected: *)

Lemma unerase_erase : forall (t : type) (e : expr_t t),
  unerase t (erase e) = Some e.
Proof.
  admit.
Admitted.
```

Task 2: Christmas Exercise: Combinatory Logic

In this exercise we consider a formal system of typed combinatory logic with arithmetic primitives. The types of that language are very simple. We just have the natural numbers and function types[1]:

Types, T	::=	$\mathbb{N}$	*Type of natural numbers*
	|	T $\Rightarrow$ T	*Function type*

Examples of types are $\mathbb{N}$, $\mathbb{N} \Rightarrow \mathbb{N}$, $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$, etc. The language has five constants, K, S, O, Succ and iterate. Here they are with their respective types:

K	$\alpha \Rightarrow \beta \Rightarrow \alpha$	*The K combinator*
S	$(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)$	*The S combinator*
O	$\mathbb{N}$	*Zero*
Succ	$\mathbb{N} \Rightarrow \mathbb{N}$	*The successor function*
iterate	$\mathbb{N} \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$	*The iteration function*

where α, β, γ are type variables which can be instantiated with any type.

From these constants we can build more complex terms by applying them to one another. This requires the types to match, e.g. we can form $Succ\, O$ but not $O\, Succ$.

The semantics of the language is given by the following rewriting rules: Expressions are evaluated by repeatedly replacing the left hand side of the following equations by their right hand side, until we reach a point where no further rules apply.

K x y	=	x
S x y z	=	x z (y z)
iterate O f x	=	x
iterate (Succ n) f x	=	iterate n f (f x)

Examples of reduction sequences are:

$$K(KSS)(SKK) \to KSS \to S$$
$$iterate(Succ(Succ(Succ 0)))(K0)S \to \ldots \to (K0)((K0)((K0)S)) \to 0$$

```
(* We have the type of natural numbers, and function types.
For function types we add the special notation "==>" which associates to the right. *)
Inductive Types :=
| Nat : Types
```

[1] Function arrows are right associative.

```
| Fun : Types -> Types -> Types.
Notation "t1 ==> t2" := (Fun t1 t2) (at level 80, right associativity).

Example example_type1 : Types := Nat ==> Nat ==> Nat.
Example example_type2 : Types := (Nat ==> Nat) ==> Nat.

(* Expressions are build up from the five constants, and are combined with App. Expressions are
   inherently typed, so it is not possible to form ill-typed expressions.
   Note that the type arguments are marked as implicit, so Coq tries to infer them automatically,
   For application we use the special notation "@" which associates to the left. *)

(* Task 1: Extend the following inductive datatype with the constructors for K and O. *)
Inductive Expr : Types ->  Type :=
(*| K : ... *)
| S : forall {x y z : Types}, Expr ((x ==> (y ==> z)) ==> (x ==> y) ==> (x ==> z))
(* | O : ... *)
| Succ : Expr (Nat ==> Nat)
| iterate : forall {x : Types}, Expr (Nat ==> (x ==> x) ==> x ==> x)
| App : forall {x y : Types}, Expr (x ==> y) -> Expr x -> Expr y.
Notation "e1 @ e2" := (App e1 e2) (at level 81, left associativity).

Example one : Expr Nat := Succ @ O.
Example two : Expr Nat := Succ @ (Succ @ O).
Example three : Expr Nat := Succ @ (Succ @ (Succ @ O)).
Example four : Expr Nat := Succ @ (Succ @ (Succ @ (Succ @ O))).

(* As an additional example, consider the defined term I = SKK with reduction behaviour "I x = x".
   Note that we have to annotate some term (in this case, K) to get the type inference of
   Coq to work. *)
Example I :forall (t : Types), Expr (t ==> t) := fun t =>  S @ K @ (@K t t).

(* The reduction semantics is given by the following inductive relation. Observe that the
  type of the relation guarantees that the reduction does not change the type of
  the expression. *)

(* Task 2: Extend the evaluate relation with the missing cases for S and iterate succ *)
Inductive evaluate : forall (x : Types), Expr x -> Expr x -> Prop :=
| eval_cong_left : forall (t1 t2 : Types) (e1 e1': Expr (t1 ==> t2)) (e2 : Expr t1),
    evaluate (t1 ==> t2) e1 e1' ->
    evaluate t2 (e1 @ e2) (e1' @ e2)
| eval_cong_right : forall (t1 t2 : Types) (e1 : Expr (t1 ==> t2)) (e2 e2': Expr t1),
    evaluate t1 e2 e2' ->
    evaluate t2 (e1 @ e2) (e1 @ e2')
| eval_K : forall (t1 t2 : Types) (e1 : Expr t1) (e2 : Expr t2),
    evaluate t1
             (K @ e1 @ e2)
             e1
(* | eval_S : ... *)
(* | eval_iterate_succ : .... *)
| eval_iterate_zero : forall (t : Types) (f : Expr ( t ==> t)) (ex : Expr t),
    evaluate t
             (iterate @ O @  f @ ex)
             ex.

(* evaluate_tr is the transitive-reflexive closure of evaluate *)
Inductive evaluate_tr : forall (x : Types), Expr x -> Expr x -> Prop :=
  eval_refl : forall t e, evaluate_tr t e e
| eval_step : forall t e1 e2,
    evaluate t e1 e2 -> evaluate_tr t e1 e2
| eval_trans : forall t e1 e2 e3,
    evaluate t e1 e2 ->
    evaluate_tr t e2 e3 ->
    evaluate_tr t e1 e3.

(* Task 3: Prove that I works, i.e. that (I e) evaluates to e *)
Lemma I_works : forall (t : Types) (e : Expr t), evaluate_tr t ((@I t) @ e) e.
Proof.
  admit.
Admitted.


(* Task 4a: The constant K takes two arguments, and ignores its second argument.
   Write an expression K' which takes two arguments and ignores its first argument.
   Hint: You should solve this exercise on paper first. This one is a puzzle ;) *)
Definition K' : forall {t1 t2 : Types},  Expr (t1 ==> t2 ==> t2) := (* ... *).

(* Task 4b: Prove that your construction for K' works. *)
Lemma K'_works : forall (t1 t2 : Types) (e1 : Expr t1) (e2 : Expr t2),
    evaluate_tr t2 (K' @ e1 @ e2) e2.
Proof.
  admit.
Admitted.

(* Task 5a: Define an expression of type "Nat ==> Nat" which applied to a
   natural number evaluates to that number + 2. *)
Definition plus_two : Expr (Nat ==> Nat) := (* ... *).

(* Task 5b: Prove that your construction works. *)
Lemma plus_two_works : evaluate_tr Nat (plus_two @ O) two.
Proof.
  admit.
Admitted.


(* Task 6a: Define an expression of type "Nat ==> Nat" which applied to a
   natural number doubles that number. *)
Definition double : Expr (Nat ==> Nat) := (* ... *).

(* Task 6b: Prove your construction correct. *)
Lemma double_works : evaluate_tr Nat (double @ one) two.
Proof.
  admit.
Admitted.
```