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, 20. Dezember 2018

**This homework is due Monday, January 7th 23:59!**

## Task 1: GADTs

In this exercise we consider a simple language for expressions which can denote both natural numbers and strings. This language has literals for natural numbers and strings, an operation which adds two numbers, and an operation which repeats a string a given number of times.

Compared to arithmetic expressions which we considered before, we now have to take care of expressions which don't typecheck. For example, there is no way to evaluate the following expression:

```
(E_Add (E_N_Lit 2) (E_S_Lit ``Hello''))
```

We consider two ways to deal with this problem. The first way is to allow such expressions, and to define a property „typechecks“ which holds for well-typed expressions. The second way is to introduce a new data-type „`expr_t`“ which does not permit the formation of ill-formed expressions. Your task is to relate these two approaches by some functions and lemmas.

```
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.
```

```
(* Task 1: Give the missing constructors of expr_t. *)
Inductive expr_t : type -> Type :=
  (* | Et_N_Lit : .... *)
  (* | Et_S_Lit : ... *)
  (* | Et_Add :  ... *)
  | Et_Repeat : expr_t T_String -> expr_t T_Nat -> expr_t T_String.

(* Task 2: Implement the function erase which maps the typed expressions to their untyped corresponding
Fixpoint erase {t : type} (e : expr_t t) : expr := (* TODO *).


(* Task 3: Proof that a typed expression whose types have been erased is still typeable. *)
Lemma erased_expr_t_typechecks : forall (t : type) (e : expr_t t), typechecks (erase e) t.
Proof.
  admit.
Admitted.

(* Task 4: (Slightly advanced) Show that every typeable term is in the image of erase. *)
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 2: Stack Machine

In this exercise we consider stack machines as another alternative for giving semantics to an expression language.

```
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Import ListNotations.

(* Arithmetic expressions and string expressions. *)

(* Arithmetic expressions which evaluate to natural numbers *)
Inductive aexpr : Type :=
  AE_Lit : forall (n : nat), aexpr
| AE_Add : aexpr -> aexpr -> aexpr
| AE_Mult : aexpr -> aexpr -> aexpr.

(* Task 1: Write a function which evaluates an aexpr to a natural number. *)
Fixpoint eval_aexpr (ae : aexpr) : nat. Admitted.

(* Expressions which evaluate to strings. *)
Inductive strexpr : Type :=
  SE_Lit : forall (s : string), strexpr
| SE_Append : strexpr -> strexpr -> strexpr
| SE_Repeat : aexpr -> strexpr -> strexpr.

(* Helper function: Repeat a string n times. *)
Fixpoint repeat_str (num : nat) (str : string) : string :=
  match num with
```

```
  | O => ""%string
  | S n' => String.append str (repeat_str n' str)
  end.

(* Task 2: Write a function which evaluates a strexpr to a string. *)
Fixpoint eval_strexpr (se : strexpr) : string. Admitted.

(* The Stack Machine *)

(* We formalize a model of a simple stack machine. The stack itself is a list of strings and nats.
   The following instructions for our stack machine are available:
   - Push_nat : Push a nat on the stack.
   - Push_string : Push a string on the stack.
   - Add  : Pop two nats from the stack, add them and push the result on the stack.
   - Mult : Pop two nats from the stack, multiply them and push the result on the stack.
   - Append : Pop two strings from the stack, append them and push the result on the stack.
   - Repeat : Pop a string and a nat from the stack, repeat the string n times and push
              the result on the stack.

  Note that the behaviour of the stack machine is undefined if the required operands of an
  operation cannot be popped from the stack.
 *)

Inductive instruction : Type :=
| Push_nat : forall (n : nat), instruction
| Push_string : forall (s : string), instruction
| Add : instruction
| Mult : instruction
| Append : instruction
| Repeat : instruction.

Definition stack : Type := list (nat + string).
Definition program : Type := list instruction.
Definition state : Type := stack * program.

(* Execute one instruction from the program *)
(* Task 3: Fill in the missing cases *)
Inductive execute_step : state -> state -> Prop :=
| exec_push_nat : forall (n : nat)(st : stack)(pr : program),
    execute_step (st, Push_nat n :: pr) (inl n :: st, pr)
| exec_add : forall (n m : nat) (st : stack) (pr : program),
    execute_step (inl n :: inl m :: st, Add :: pr) (inl (n + m) :: st, pr).

Example push_nat_example : execute_step ([], [Push_nat 3]) ([inl 3],[]).
Proof.
  apply exec_push_nat.
Qed.

(* Execute all the instructions from the program. This results with the state
   of the stack at the end of execution. *)
Inductive execute : state -> stack -> Prop :=
  execute_nop : forall (st : stack), execute (st, []) st
| execute_cons : forall (s s' : state) (st : stack),
    execute_step s s' ->
    execute s' st ->
    execute s st.
```

```
(* Task 4: Give a proof of the following example. *)
Example addition_example : execute ([], [Push_nat 2; Push_nat 4; Add]) [inl 6].
Proof.
  admit.
Admitted.

(* Compiling expressions into stack programs. *)

(* We now have expressions for nats and strings, a stack machine, and want to relate them by
   a compilation step. We want to be able to compile expressions into programs for our
   stack machine. *)

(* Task 5: Define the following two compilation functions. *)
Fixpoint compile_aexpr (ae : aexpr) : program. Admitted.

Fixpoint compile_strexpr (se : strexpr) : program. Admitted.

Definition example_prog : program :=
  (compile_strexpr (SE_Repeat (AE_Add (AE_Lit 1) (AE_Lit 1)) (SE_Append (SE_Lit "Hello ") (SE_Lit "Worl


(* Task 6: Define the tactic "execute_tac" which can solve the goal below.
   You should use "match goal" for this. Alternatively, you can solve this goal by hand. *)
Ltac execute_tac := idtac.

Lemma example_prog_execute : execute ([], example_prog) [inr "Hello World Hello World "%string].
Proof.
  unfold example_prog. simpl.
  repeat execute_tac.
  admit.
Admitted.
```

## Task 3: Christmas Puzzle: Combinatory Logic

This task is purely voluntary.

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 $\alpha, \beta, \gamma$ 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“.

[1] Function arrows are right associative.

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.

$$\begin{array}{rcl} \text{K x y} & = & \text{x} \\ \text{S x y z} & = & \text{x z (y z)} \\ \text{iterate O f x} & = & \text{x} \\ \text{iterate (Succ n) f x} & = & \text{iterate n f (f x)} \end{array}$$

Examples of reduction sequences are:

$$\text{K(KSS)(SKK)} \to \text{KSS} \to \text{S}$$

$$\text{iterate (Succ (Succ (Succ 0))) (K 0) S} \to \ldots \to \text{(K0) ((K0) ((K0) S))} \to 0$$

```
(* 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
| 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. *)
Definition K' : forall {t1 t2 : Types},  Expr (t1 ==> t2 ==> t2) := (* TODO *)

(* 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) := (* TODO *).

(* 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) := (* TODO *).
```

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