COMP 321: Principles of Programming Languages, Fall 2021
Homework 3: Gödel's T

To hand in, please upload

```
hw03/check.sml
hw03/step.sml
hw03/arith.t
hw03/hw03-written.pdf
```

to your Google Drive handin folder (note the sub-folder this week).

1 Implementing T

In Chapter 9 of PFPL, you can find the typing and operational semantics for the language with function and natural number types, which is called Gödel's T. In this task, you will implement a type-checker and evaluator for this language—the same task as from HW2, but for a richer language.

For this week, you are given an implementation of expressions in `syntax.sml`. This implementation includes `subst` for substitution, which should be analogous to your solution from last week's homework.

Task 1 (20%). Program a typechecker for Gödel's T. We have provided a stub for you in `check.sml`.

Task 2 (25%). Implement the operational semantics for T. Once again, we have provided stubs for you in `step.sml`.

In the support code, we have supplied a parser for a concrete syntax as well as several functions to help you test your code (see the `Top` module, and note that a couple of these have changed from last week):

```
signature TOP =
sig

    (* interactive loop *)

    (* just print the same expression back *)
    val loop_print : unit -> unit

    (* just type check *)
    val loop_type  : unit -> unit
```

```
    (* type check the input program, then,
       if it was well-typed,
       try to evaluate the program to a value, and then
       show the final value and its type (which is
       determined by re-typechecking the value).
       *)
    val loop_eval  : unit -> unit

    (* Starting with the initial program,
       show each step of evaluation with its type
       (which is determined by re-typechecking after each step).
       Stops when the result of a step is ill-typed.

       By preservation, these types *should* all be the same,
       but if they're not, it can be helpful for finding bugs.
       *)
    val loop_step  : unit -> unit

    (* *Ignoring the type checker*,
       try to evaluate the program to a value,
       and then show the final value.

       You can use this to test your operational semantics
       if you want to work on that first or
       if your type checker isn't working.
       *)
    val loop_eval_no_typechcker : unit -> unit

    (* *Ignoring the type checker*, show each step of evaluation.

       You can use this to test your operational semantics
       if you want to work on that first or
       if your type checker isn't working.
       *)
    val loop_step_no_typechecker : unit -> unit

    (* same as above but read an EXP source file *)
    val file_print : string -> unit
    val file_type  : string -> unit
    val file_eval  : string -> unit
    val file_step  : string -> unit
    val file_eval_no_typechecker  : string -> unit
    val file_step_no_typechecker  : string -> unit

end;  (* signature TOP *)
```

Some examples are in the file `arith.t`. For example:

```
let double be lam x:nat.
               rec x { z => z
                     | s _ with r => s s r }
in
   double @ 5
end;
```

In the concrete syntax, functions are written `lam x:t.e`. Application is written with an infix `@` sign; application is left-associative, so `f @ x @ y` means `(f @ x) @ y`. Recursion is written `rec e { z => e0 | s x with y => e1}`. `let` is the same as last week.

2 Definability

Next, you will write a few programs using your implementation. In the file `arith.t`:

Task 1 (5%). Define a function `add : (nat -> (nat -> nat))` such that `add @ x @ y` computes the sum of `x` and `y`. You solution should be of the form

```
let
    add be ...
in
    ...
end
```

and you can test the function by writing a test case in the body of the let.

Task 2 (5%). Define a function `mult : (nat -> (nat -> nat))` such that `mult @ x @ y` computes the product of `x` and `y`. You solution should be of the form

```
let
    add be ...
in

  let

      mult be ...
  in

      ...
  end

end
```

because you will need to use your solution from the previous task.

Task 2 (5%). Define a function `sub : (nat -> (nat -> nat))` such that `sub @ x @ y` is equal to $x - y$ *assuming that* x *is* $\geq y$ (otherwise, the behavior is unspecified).

3 Progress and Preservation for Lists

Suppose we extend Gödel's T with a type of lists of natural numbers:

New expressions:

$$\begin{array}{l} \mathsf{nil} \\ \mathsf{cons}(e_1, e_2) \\ \mathsf{listrec}(e_0, x.xs.r.e_1, e) \end{array}$$

With the following operational semantics:

$$\frac{}{\mathsf{nil}\ \mathsf{done}} \qquad \frac{e_1\ \mathsf{done} \quad e_2\ \mathsf{done}}{\mathsf{cons}(e_1, e_2)\ \mathsf{done}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1', e_2)} \qquad \frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1, e_2')}$$

$$\frac{e \mapsto e'}{\mathsf{listrec}(e_0, x.xs.r.e_1, e) \mapsto \mathsf{listrec}(e_0, x.xs.r.e_1, e')} \qquad \frac{}{\mathsf{listrec}(e_0, x.xs.r.e_1, \mathsf{nil}) \mapsto e_0}$$

$$\frac{\mathsf{cons}(e_h, e_t)\ \mathsf{done}}{\mathsf{listrec}(e_0, x.xs.r.e_1, \mathsf{cons}(e_h, e_t)) \mapsto e_1[e_h/x][e_t/xs][\mathsf{listrec}(e_0, x.xs.r.e_1, e_t)/r]}$$

Task 1 (10%). Give typing rules for nil and cons and listrec.

Task 2 (15%). Recall the progress theorem:

For all e, τ, if $\cdot \vdash e : \tau$ then either e done or there exists an e' such that $e \mapsto e'$.

Progress is proved by rule induction on the derivation of $\cdot \vdash e : \tau$. Prove the case of progress for your typing rule for listrec (you don't need to do the cases for nil and cons).

Task 3 (15%). Recall the preservation theorem:

For all e, e', τ, if $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$.

Preservation is proved by rule induction on $e \mapsto e'$. Prove the cases of preservation for listrec (you don't need to do the cases for nil and cons).

You may use the following lemmas:

- Inversion of typing: Note where you are using inversion.
- Weakening: For all $\Gamma, \Gamma', e, \tau, \tau_1, x$, if $\Gamma \vdash e : \tau$ and $x \notin \Gamma, \Gamma'$ then $\Gamma, x : \tau_1, \Gamma' \vdash e : \tau$.
- Substitution: For all $\Gamma, \Gamma' e, x, \tau, \tau'$, if $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma, \Gamma' \vdash e'[e/x] : \tau'$