1 Introduction

You will undertake a few simple problems in order to develop basic skills with a monomorphic type system.

2 Description

- (Adapted from Exercise 30 of Chapter 6 from Programming Languages: Build, Prove, and Compare (p. 471) (itself adapted from an earlier version of this recitation problem!).)

The rules for elaboration (i.e., type checking) of definitions in Typed Impcore may seem complicated, especially when compared to the rules for evaluation of definitions in Impcore. The purpose of this problem is to demonstrate that simpler rules for elaboration would be unsound, which is to say that they would not actually guarantee the safety property that they are meant to enforce. (In fact, an earlier draft of Programming Languages: Build, Prove, and Compare used these simpler, but unsound, rules.)

Review the inference rules that define the evaluation and elaboration of definitions for Impcore/Typed Impcore in Appendix A.

Answer the following questions:

- Does the program “go wrong” when evaluated? Why or why not?
- Is the program well-typed according to the old rules? Why or why not?
- Is the program well-typed according to the new rules? Why or why not?

for each of these Impcore programs (sequences of definitions):

- (define int f ([x : int]) (g x))
  (define int g ([y : int]) (+ y 1))
  (val ans (f 5))

- (define int g ([y : int]) (+ y 1))
  (define int f ([x : int]) (g x))
  (define int g ([y : int] [z : int]) (+ y z))
  (val ans (f 5))

- (define int f ([x : int]) (+ x a))
  (val a 1)
  (val ans (f 5))

- (val a 1)
  (define int f ([x : int]) (+ x a))
  (val a (array-make 5 5))
  (val ans (f 5))
Imagine we want to add lists to Typed Impcore using the same techniques we use for arrays. We introduce the following abstract syntax (as expressed in SML, with potential concrete syntax in comments) to support lists:

```sml
datatype ty = ...
  | LISTTY of ty (* (list t) *)

datatype exp = ...
  | LNIL of ty (* (nil t) *)
  | LCONS of exp * exp (* (cons e1 e2) *)
  | LNULLP of exp (* (null? e) *)
  | LCAR of exp (* (car e) *)
  | LCDR of exp (* (cdr e) *)
```

– Give appropriate rules for formation, introduction, and elimination of lists. 
(Note: Do not attempt to create a type system that prevents programmers from applying \texttt{car} or \texttt{cdr} to the empty list. Rather, applying \texttt{car} or \texttt{cdr} to any list (empty or non-empty) should be a well-typed term and applying \texttt{car} or \texttt{cdr} to the empty list should cause a runtime error.)

– Review the implementation of the type checker for Typed Impcore in Appendix B. Complete the \texttt{ty (LNIL ty)}, \texttt{ty (LCONS (h, t))}, \texttt{ty (LNULLP l)}, \texttt{ty (LCAR l)}, \texttt{ty (LCDR l)} cases.

– Explain why the abstract syntax for the empty list is \texttt{LNIL of ty} rather than \texttt{LNIL}.

### 3 Requirements and Submission

At the end of class, submit the group’s solutions as hard-copy; be sure to include the names of all group members in the submission.
A Evaluation and Elaboration for Typed Impcore

A.1 Evaluation

The operational semantics of Impcore uses the following rules for the evaluation of definitions:

\[ (d, \xi, \phi) \rightarrow (\xi', \phi') \]

\[ \langle e, \xi, \phi, \{\} \rangle \downarrow \langle v, \xi', \phi', \rho' \rangle \] (DefineGlobal)

\[ \langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi', x \mapsto v, \phi \rangle \] (Eval)

\[ \langle \text{EXP}(e), \xi, \phi \rangle \rightarrow \langle \xi', \text{it} \mapsto v, \phi \rangle \] (EvalExp)

\[ x_1, \ldots, x_n \text{ all distinct} \]

\[ \langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n, e : \tau \rangle, \xi, \phi) \rangle \rightarrow \langle \xi, \phi \{ f \mapsto \text{USER}(\langle x_1, \ldots, x_n, e \rangle) \} \rangle \] (DefineFunction)

A.2 Elaboration (Old Rules)

In an earlier draft of *Programming Languages: Build, Prove, and Compare*, the type system of Typed Impcore used the following rules for the elaboration of definitions:

\[ (d, \Gamma_\xi, \Gamma_\phi) \rightarrow (\Gamma_\xi', \Gamma_\phi') \]

\[ \Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \]

\[ \langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \{ x \mapsto \tau \}, \Gamma_\phi \rangle \] (Val)

\[ \langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi' \Gamma_\phi \rangle \] (Exp)

\[ x_1, \ldots, x_n \text{ all distinct} \]

\[ \tau_1, \ldots, \tau_n, \tau \text{ are types} \]

\[ \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_1 \times \cdots \times \tau_n \mapsto \tau \}, \{ x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \} \vdash e : \tau \]

\[ \langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n, e : \tau \rangle, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \Gamma_\phi \{ f \mapsto \tau_1 \times \cdots \times \tau_n \mapsto \tau \} \rangle \] (Define)

A.3 Elaboration (New Rules)

In the current draft of *Programming Languages: Build, Prove, and Compare*, the type system of Typed Impcore uses the following rules for the elaboration of definitions:

\[ (d, \Gamma_\xi, \Gamma_\phi) \rightarrow (\Gamma_\xi', \Gamma_\phi') \]

\[ \Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \quad x \notin \text{dom } \Gamma_\xi \]

\[ \langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \{ x \mapsto \tau \}, \Gamma_\phi \rangle \] (NewVal)

\[ \langle \text{VAL}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \Gamma_\phi \rangle \] (Val)

\[ \Gamma_\xi, \Gamma_\phi \vdash e : \tau \quad \Gamma_\xi(x) = \tau \]

\[ \langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle \] (Exp)

\[ x_1, \ldots, x_n \text{ all distinct} \]

\[ \tau_1, \ldots, \tau_n, \tau \text{ are types} \]

\[ f \notin \text{dom } \Gamma_\phi \]

\[ \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_1 \times \cdots \times \tau_n \mapsto \tau \}, \{ x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \} \vdash e : \tau \]

\[ \langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n, e : \tau \rangle, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \Gamma_\phi \{ f \mapsto \tau_1 \times \cdots \times \tau_n \mapsto \tau \} \rangle \] (Define)

\[ \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_1 \times \cdots \times \tau_n \mapsto \tau \}, \{ x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \} \vdash e : \tau \]

\[ \langle \text{REDEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n, e : \tau \rangle, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle \] (Redefine)
B Type Checker for Typed Impcore

(* exceptions used in languages with type checking *)
exception TypeError of string
exception BugInTypeChecking of string

(* types for \timpcore *)

datatype ty = INTTY | BOOLTY | UNITTY | LISTTY of ty
datatype funty = FUNTY of ty list * ty

(* definitions of [[typeString]] and [[funtyString]] for \timpcore 1110c *)
fun typeString BOOLTY = " bool "
| typeString INTTY = " int "
| typeString UNITTY = " unit "
| typeString ( LISTTY tau ) = "( list " ^ typeString tau ^ " )"

(* definitions of [[typeString]] and [[funtyString]] for \timpcore 1110d *)
fun funtyString ( FUNTY (args , result )) = "( ^ spaceSep ( map typeString args ) ^ " -> " ^ typeString result ^ " )"

(* type declarations for consistency checking *)
val _ = op typeString : ty -> string
val _ = op funtyString : funty -> string

(* types for \timpcore 395b *)
fun eqType ( INTTY , INTTY ) = true
| eqType ( BOOLTY , BOOLTY ) = true
| eqType ( UNITTY , UNITTY ) = true
| eqType ( LISTTY t1 , LISTTY t2) = eqType (t1 , t2)
| eqType (_ , _) = false

and eqTypes ([], []) = true
| eqTypes (t1::ts1, t2::ts2) = eqType (t1, t2) andalso eqTypes (ts1, ts2)
| eqTypes (_ , _) = false

(* types for \timpcore 395c *)
fun eqFunty ( FUNTY (args , result ), FUNTY (args , result )) = ListPair.allEq eqType (args, args) andalso eqType (result, result)

(* type declarations for consistency checking *)
val _ = op eqType : ty * ty -> bool
val _ = op eqTypes : ty list * ty list -> bool
(* type declarations for consistency checking *)
val _ = op eqFunty : funty * funty -> bool
fun typeof (e, globals, functions, formals) =
  let
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 402b *)
    fun ty (LITERAL v) = INTTY
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 403a *)
    | ty (VAR x) = (find (x, formals) handle NotFound _ => find (x, globals))
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 403b *)
    | ty (SET (x, e)) =
      let val tau_x = ty (VAR x)
      val tau_e = ty e
      in
        if eqType (tau_x, tau_e) then
          tau_x
        else
          raise TypeError ($"Set variable " ^ x ^ " of type " ^ typeString tau_x ^ " to value of type " ^ typeString tau_e)
      end
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 404a *)
    | ty (IFX (e1, e2, e3)) =
      let val tau1 = ty e1
      val tau2 = ty e2
      val tau3 = ty e3
      in
        if eqType (tau1, BOOLTY) then
          if eqType (tau2, tau3) then
            tau2
          else
            raise TypeError ($"In if expression, true branch has type " ^ typeString tau2 ^ " but false branch has type " ^ typeString tau3)
        else
          raise TypeError ($"Condition in if expression has type " ^ typeString tau1 ^ ", which should be " ^ typeString BOOLTY)
      end
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 404b *)
    | ty (WHILEX (e1, e2)) =
      let val tau1 = ty e1
      val tau2 = ty e2
      in
        if eqType (tau1, BOOLTY) then
          UNITY
        else
          raise TypeError ($"Condition in while expression has type " ^ typeString tau1 ^ ", which should be " ^ typeString BOOLTY)
      end
    (* function [[ty]], to check the type of an expression, given $\text{itenvs}$ 404c *)
    | ty (BEGIN es) =
      let val bodytypes = map ty es
      in
        List.last bodytypes handle Empty => UNITY
      end
(* function \([\text{ty}]\), to check the type of an expression, given \(\text{itenvs}\) 405a *)
| ty (EQ (e1, e2)) =
  let val (tau1, tau2) = (ty e1, ty e2)
  in if eqType (tau1, tau2) then
    BOOLTY
  else raise TypeError ("Equality compares values of different types " ^
                         typeString tau1 ^ " and " ^ typeString tau2)
  end

(* function \([\text{ty}]\), to check the type of an expression, given \(\text{itenvs}\) 405b *)
| ty (PRINTLN e) = (ty e; UNITTY)
| ty (PRINT e) = (ty e; UNITTY)

(* function \([\text{ty}]\), to check the type of an expression, given \(\text{itenvs}\) 405c *)
| ty (APPLY (f, actuals)) =
  let val actualtypes = map ty actuals
  val FUNTY (formaltypes, resulttype) = find (f, functions)
  (* definition of [[parameterError]] 406a *)
  fun parameterError (n, atau::actuals, ftau::formals) =
    if eqType (atau, ftau) then
      parameterError (n+1, actuals, formals)
    else raise TypeError ("In call to " ^ f ^ ", parameter " ^
                           intString n ^ " has type " ^
                           typeString atau ^
                           " where type " ^ typeString ftau ^
                           " is expected")
  |
  | parameterError _ =
    raise TypeError ("Function " ^ f ^ " expects " ^
                     intString (length formaltypes) ^
                     " parameters " ^
                     " but got " ^ intString (length actualtypes))

  (* type declarations for consistency checking *)
  val _ = op parameterError : int * ty list * ty list -> 'a
  in if eqTypes (actualtypes, formaltypes) then
    resulttype
  else parameterError (1, actualtypes, formaltypes)
  end

(* function \([\text{ty}]\), to check the type of an expression, given \(\text{itenvs}\) ((prototype)) 413 *)
| ty (LNIL t) = raise LeftAsExercise "LNIL"
| ty (LCONS (h, t)) = raise LeftAsExercise "LCONS"
| ty (LNULLP l) = raise LeftAsExercise "LNULLP"
| ty (LCAR l) = raise LeftAsExercise "LCAR"
| ty (LCDR l) = raise LeftAsExercise "LCDR"

(* type declarations for consistency checking *)
val _ = op typeof : exp * ty env * funty env * ty env -> ty
val _ = op ty : exp -> ty
in ty e
end