Programming Language Concepts

CSCI-344 **Recitation 6**
Term 20161

Monomorphic Type Systems (Typed Impcore)

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. 461) (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))
    ```

- Recall the rules for formation, introduction, and elimination of arrays from the type system of Typed Impcore.

  $\boxed{\tau \text{ is a type}}$

  $$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}} \text{ (ArrayFormation)}$$

  $\boxed{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$

  $$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{INT} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-MAKE}(e_1, e_2) : \text{ARRAY}(\tau)} \text{ (MakeArray)}$$

  $$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{ARRAY}(\tau) \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{INT}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-AT}(e_1, e_2) : \tau} \text{ (ArrayAt)}$$

  $$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{ARRAY}(\tau) \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{INT} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-PUT}(e_1, e_2, e_3) : \tau} \text{ (ArrayPut)}$$

  $$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \text{ARRAY}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-SIZE}(e) : \text{INT}} \text{ (ArraySize)}$$

  Review the implementation of the type checker for Typed Impcore in Appendix B.

  - Groups 1, 5, and 9: Complete the `ty (AMAKE (len, init))` case.
  - Groups 2, 6, and 10: Complete the `ty (AAT (a, i))` case.
  - Groups 3 and 7: Complete the `ty (APUT (a, i, e))` case.
  - Groups 4 and 8: Complete the `ty (ASIZE a)` case.

  Optionally, complete additional cases not assigned to your group.

- (Adapted from Exercise 8 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 452).)

  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:

```
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 `car` or `cdr` to the empty list. Rather, applying `car` or `cdr` to any list (empty or non-empty) should be a well-typed term and applying `car` or `cdr` to the empty list should cause a runtime error.)
  - Explain why the abstract syntax for the empty list is `LNIL of ty` rather than `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:

$\boxed{\langle d, \xi, \phi \rangle \to \langle \xi', \phi' \rangle}$

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \to \langle \xi'\{x \mapsto v\}, \phi \rangle} \text{ (DefineGlobal)} \qquad \frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{EXP}(e), \xi, \phi \rangle \to \langle \xi'\{\texttt{it} \mapsto v\}, \phi \rangle} \text{ (EvalExp)}$$

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \xi, \phi \rangle \to \langle \xi, \phi\{f \mapsto \text{USER}(\langle x_1, \ldots, x_n \rangle, e)\} \rangle} \text{ (DefineFunction)}$$

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:

$\boxed{\langle d, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma'_\phi \rangle}$

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle} \text{ (Val)} \qquad \frac{\langle \text{VAL}(\texttt{it}, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle} \text{ (Exp)}$$

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \qquad \tau_1, \ldots, \tau_n \text{ are types} \\ \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \tau_n \to \tau\} \rangle} \text{ (Define)}$$

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:

$\boxed{\langle d, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma'_\phi \rangle}$

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \qquad x \notin \text{dom } \Gamma_\xi}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle} \text{ (NewVal)} \qquad \frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \qquad \Gamma_\xi(x) = \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi \rangle} \text{ (OldVal)}$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi \rangle} \text{ (Exp)}$$

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \qquad \tau_1, \ldots, \tau_n \text{ are types} \\ f \notin \text{dom } \Gamma_\phi \\ \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \tau_n \to \tau\} \rangle} \text{ (Define)}$$

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \qquad \tau_1, \ldots, \tau_n \text{ are types} \\ \Gamma_\phi(f) = \tau_1 \times \cdots \times \tau_n \to \tau \\ \Gamma_\xi, \Gamma_\phi, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi \rangle} \text{ (Redefine)}$$

B Type Checker for Typed Impcore

```
(*****************************************************************)
(*                                                               *)
(*   TYPES FOR \TIMPCORE                                         *)
(*                                                               *)
(*****************************************************************)

(* types for \timpcore 386g *)
datatype ty    = INTTY | BOOLTY | UNITTY | ARRAYTY of ty
datatype funty = FUNTY of ty list * ty

(*****************************************************************)
(*                                                               *)
(*   TYPE CHECKING FOR \TIMPCORE                                 *)
(*                                                               *)
(*****************************************************************)

(* type checking for \timpcore 394a *)
fun typeof (e, globals, functions, formals) =
  let
 (* function [[ty]], to check the type of an expression, given $\itenvs$ 394b *)
      fun ty (LITERAL v) = INTTY

 (* function [[ty]], to check the type of an expression, given $\itenvs$ 395a *)
        | ty (VAR x) = (find (x, formals) handle NotFound _ => find (x, globals)
                                                                            )

 (* function [[ty]], to check the type of an expression, given $\itenvs$ 395b *)
        | 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 $\itenvs$ 396a *)
        | 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 $\itenvs$ 396b *)
        | ty (WHILEX (e1, e2)) =
            let val tau1 = ty e1
                val tau2 = ty e2
            in  if eqType (tau1, BOOLTY) then
                  UNITTY
                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 $\itenvs$ 396c *)
       | ty (BEGIN es) =
           let val bodytypes = map ty es
           in  List.last bodytypes handle Empty => UNITTY
           end

 (* function [[ty]], to check the type of an expression, given $\itenvs$ 397a *)
       | 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 [[ty]], to check the type of an expression, given $\itenvs$ 397b *)
       | ty (PRINTLN e) = (ty e; UNITTY)
       | ty (PRINT   e) = (ty e; UNITTY)

 (* function [[ty]], to check the type of an expression, given $\itenvs$ 397c *)
       | ty (APPLY (f, actuals)) =
           let val actualtypes                     = map ty actuals
               val FUNTY (formaltypes, resulttype) = find (f, functions)
               (* definition of [[parameterError]] 398a *)
               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 [[ty]], to check the type of an expression, given $\itenvs$ ((prototype)) 405 *)
      | ty (AMAKE (len, init)) = raise LeftAsExercise "AMAKE"
      | ty (AAT (a, i))        = raise LeftAsExercise "AAT"
      | ty (APUT (a, i, e))    = raise LeftAsExercise "APUT"
      | ty (ASIZE a)           = raise LeftAsExercise "ASIZE"
(* 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
(* 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
```

Solutions

- (un)soundness of Typed Impcore
  - `f - g`
    - does not go wrong; at the time that the expression `(f 5)` is evaluated, the function `g` has been added to the function-definition environment (ϕ) and the expression `(g x)` can be evaluated.
    - is not well-typed according to the old rules ("`variable g not found`"); at the time that `f` is defined, the function `g` is not present in the function type environment (Γ_ϕ) and the expression `(g x)` is not well-typed.
    - is not well-typed according to the new rules ("`variable g not found`"); at the time that `f` is defined, the function `g` is not present in the function type environment (Γ_ϕ) and the expression `(g x)` is not well-typed.
  - `g - f - g`
    - goes wrong ("`Wrong number of arguments to function`"); at the time that the expression `(f 5)` is evaluated, the "second" function `g` has been added to the function-definition environment (ϕ) and the expression `(g x)` cannot be evaluated, because the "second" function `g` requires two arguments.
    - is well-typed according to the old rules; at the time that `f` is defined, the "first" function `g` is present in the function type environment (Γ_ϕ) with type $\text{INT} \to \text{INT}$ (a one-argument function type) and the expression `(g x)` is well-typed, and the "second" function `g` can be accepted according to the (old) DEFINE rule.
    - is not well-typed according to the new rules ("`Function g of type (int -> int) may not be redefined with type (int int -> int)`"); at the time that `f` is defined, the "first" function `g` is present in the function type environment (Γ_ϕ) with type $\text{INT} \to \text{INT}$ (a one-argument function type) and the expression `(g x)` is well-typed, but the "second" function `g` cannot be accepted according to the (new) DEFINE rule (because $\mathtt{g} \in \text{dom}\ \Gamma_\phi$) and cannot be accepted according to the (new) REDEFINE rule because $\Gamma_\phi(\mathtt{g}) = \text{INT} \to \text{INT} \neq \text{INT} \times \text{INT} \to \text{INT}$.
  - `f - a`
    - does not go wrong; at the time that the expression `(f 5)` is evaluated, the variable `a` has been added to the global-variable environment (ξ) with the value `1` and the expression `(+ x a)` can be evaluated.
    - is not well-typed according to the old rules ("`variable a not found`"); at the time that `f` is defined, the variable `a` is not present in the global-variable type environment (Γ_ξ) and the expression `(+ x a)` is not well-typed.
    - is not well-typed according to the old rules ("`variable a not found`"); at the time that `f` is defined, the variable `a` is not present in the global-variable type environment (Γ_ξ) and the expression `(+ x a)` is not well-typed.
  - `a - f - a`
    - goes wrong ("`arithmetic on non-numbers`"); at the time that the expression `(f 5)` is evaluated, the "second" variable `a` has been added to the global-variable environment (ξ) with the value `[5 5 5 5 5]` and the expression `(+ x a)` cannot be evaluated, because it attempts to add an integer and an array.
    - is well-typed according to the old rules; at the time that `f` is defined, the "first" global variable `a` is present in the global-variable type environment (Γ_ξ) with type INT and the expression `(+ x a)` is well-typed, and the "second" global variable `a` can be accepted according to the (old) VAL rule.
    - is not well-typed according to the new rules; at the time that `f` is defined, the "first" global variable `a` is present in the global-variable type environment (Γ_ξ) with type INT and the expression `(+ x a)` is well-typed, but the "second" global variable `a` cannot be accepted according to the (new) VALNEW rule (because $\mathtt{a} \in \text{dom}\ \Gamma_\xi$) and cannot be accepted according to the (new) VALOLD rule (because $\Gamma_\xi(\mathtt{a}) = \text{INT} \neq \text{ARRAY(INT)}$).

- Extending Typed Impcore with lists
  - Rules for formation, introduction, and elimination of lists

$\boxed{\tau \text{ is a type}}$

$$\frac{\tau \text{ is a type}}{\text{LIST}(\tau) \text{ is a type}} \text{ (ListFormation)}$$

$\boxed{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$

$$\frac{\tau \text{ is a type}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LNIL}(\tau) : \text{LIST}(\tau)} \text{ (ListNil)}$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{LIST}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LCONS}(e_1, e_2) : \text{LIST}(\tau)} \text{ (ListCons)}$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \text{LIST}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LNULLP}(e_1, e_2) : \tau} \text{ (ListNull?)}$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \text{LIST}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LCAR}(e) : \tau} \text{ (ListCar)}$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \text{LIST}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LCDR}(e) : \text{LIST}(\tau)} \text{ (ListCdr)}$$

  - LNIL(τ)/`(nil t)`

    We require the concrete/abstract syntax for the empty list to include the type of the list elements in order to make the type system deterministic: every well-typed expression has exactly one type.

    An alternative concrete/abstract syntax for the empty list that omits the type of the list elements would make the type system non-deterministic:

$\boxed{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$

$$\frac{\tau \text{ is a type}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LNIL} : \text{LIST}(\tau)} \text{ (ListNil)}$$

    When type checking a `LNIL` expression, the implementation would need to "guess" the type of the list elements that makes the rest of the program type check.