Appendix F

Supporting code for Typed Impcore

F.1 Printing types and values

This code prints types.

677a ⟨*printing types for Typed Impcore* 677a⟩≡ (235d)

```
fun typeString BOOLTY        = "bool"
  | typeString INTTY         = "int"
  | typeString UNITTY        = "unit"
  | typeString (ARRAYTY tau) = "(array " ^ typeString tau ^ ")"
```

It would be good to figure out how to use `separate` in this code.

677b ⟨*printing values for Typed Impcore* 677b⟩≡ (236d)

```
fun valueString (NUM n) = Int.toString n
  | valueString (ARRAY a) =
      if Array.length a = 0 then
          "[]"
      else
          let val elts = Array.foldr (fn (v, s) => " " :: valueString v :: s) ["]"] a
          in  String.concat ("[" :: tl elts)
          end
```

F.2 Parsing

Typed Impcore can use μScheme's lexical analysis, so all we have here is a parser.

678 ⟨*parsing for Typed Impcore* 678⟩≡ (247d) 679a▷

```
val name    = (fn (NAME  n) => SOME n  | _ => NONE) <$>? token
val booltok = (fn (SHARP b) => SOME b  | _ => NONE) <$>? token
val int     = (fn (INT   n) => SOME n  | _ => NONE) <$>? token
val quote   = (fn (QUOTE)   => SOME () | _ => NONE) <$>? token

fun exp tokens = (
     VAR <$> name
 <|> (LITERAL o NUM) <$> int
 <|> booltok <!> "Typed Impcore has no Boolean literals"
 <|> quote   <!> "Typed Impcore has no quoted literals"
 <|> bracket "if"     "(if e1 e2 e3)"           (curry3 IFX    <$> exp <*> exp <*> exp)
 <|> bracket "while"  "(while e1 e2)"           (curry  WHILEX <$> exp <*> exp)
 <|> bracket "set"    "(set x e)"               (curry  SET    <$> name <*> exp)
 <|> bracket "begin"  ""                        (       BEGIN <$> many exp)
 <|> bracket "print"  "(print e)"               (       PRINT <$> exp)
 <|> bracket "="      "(= e1 e2)"               (curry  EQ    <$> exp <*> exp)
 <|> bracket "array-get"    "(array-get a i)"   (curry  AGET  <$> exp <*> exp)

 <|> bracket "array-set"    "(array-set a i e)" (curry3 ASET  <$> exp <*> exp <*> exp)
 <|> bracket "array-make"   "(array-make n e)"  (curry  AMAKE <$> exp <*> exp)
 <|> bracket "array-length" "(array-length a)"  (       ALEN  <$> exp)

 <|> "(" >-- literal ")" <!> "empty application"
 <|> curry APPLY <$> "(" >-- impcorefun <*> many exp --< ")"
) tokens
and impcorefun tokens =
  (   name
  <|> exp <!> "only named functions can be applied"
  <?> "function name"
  ) tokens
```

679a ⟨*parsing for Typed Impcore* 678⟩+≡ (247d) ◁678 679b▷

```
fun ty tokens = (
     BOOLTY <$ literal "bool"
 <|> UNITTY <$ literal "unit"
 <|> INTTY  <$ literal "int"
 <|> (fn (loc, n) => errorAt ("Cannot recognize name " ^ n ^ " as a type") loc)
     <$>! @@ name
 <|> bracket "array" "(array ty)" (ARRAYTY <$> ty)
 <?> "int, bool, unit, or (array ty)"
 ) tokens

val formal  = "(" >-- ((fn tau => fn x => (x, tau)) <$> ty <*> name --< ")"
                      <?> "(ty argname)")
val formals = "(" >-- many formal --< ")" <?> "((ty1 x1) ... (tyN xN))"

fun define ty f formals body =
      defineDups f formals >>=+ (fn formals =>
      DEFINE (f, { returns = ty, formals = formals, body = body }))
and defineDups f = nodupsty ("formal parameter", "definition of function " ^ f)
and nodupsty what (loc, xts) =
      nodups what (loc, map fst xts) >>=+ (fn _ => xts)
                                          (* error on duplicate names *)
val def =
     bracket "define" "(define ty f (args) body)"
                                    (define <$> ty <*> name <*> @@ formals <*>! exp)
 <|> bracket "val"    "(val x e)"        (curry VAL <$> name <*> exp)
 <|> bracket "use"    "(use filename)"   (USE <$> name)
 <|> literal ")" <!> "unexpected right parenthesis"
 <|> EXP <$> exp
 <?> "definition"
```

679b ⟨*parsing for Typed Impcore* 678⟩+≡ (247d) ◁679a

```
val timpcoreSyntax = (schemeToken, def)
```

--<	664c
<!>	664a
<$>	653c
<$>!	658c
<*>	653b
<*>!	658c
<?>	663c
<|>	654b
>--	664c
>>=+	652a
ARRAYTY	235c
BOOLTY	235c
bracket	665
curry	654a
DEFINE	236b
errorAt	661b
EXP	236b
exp	678
fst	654a
INTTY	235c
literal	664b
many	657d
name	678
nodups	666a
schemeToken	672a
UNITTY	235c
USE	236b
VAL	236b

F.3 Evaluation

680a ⟨*definition of* eval *for Typed Impcore* 680a⟩≡ (681a)

```
fun eval (e, globals, functions, formals) =
  let fun toBool (NUM 0) = false
        | toBool _       = true
      fun ofBool true    = NUM 1
        | ofBool false   = NUM 0
      val unitVal = NUM 1983 (* all values of unit type must test equal with = *)
      fun eq (NUM n1,   NUM n2)   = (n1 = n2)
        | eq (ARRAY a1, ARRAY a2) = (a1 = a2)
        | eq _                    = false
      fun findVar v = find (v, formals) handle NotFound _ => find (v, globals)
      fun ev (LITERAL n)          = n
        | ev (VAR x)              = !(findVar x)
        | ev (SET (x, e))         = let val v = ev e in v before findVar x := v end
        | ev (IFX (cond, t, f))   = if toBool (ev cond) then ev t else ev f
        | ev (WHILEX (cond, exp)) =
            if toBool (ev cond) then
                (ev exp; ev (WHILEX (cond, exp)))
            else
                unitVal
        | ev (BEGIN es) =
            let fun b (e::es, lastval) = b (es, ev e)
                  | b (   [], lastval) = lastval
            in  b (es, unitVal)
            end
        | ev (EQ (e1, e2)) = ofBool (eq (ev e1, ev e2))
        | ev (PRINT e)     = (print (valueString (ev e)^"\n"); unitVal)
        | ev (APPLY (f, args)) =
            (case find (f, functions)
               of PRIMITIVE p  => p (map ev args)
                | USERDEF func => ⟨apply user-defined function func to args 680b⟩)
      ⟨more alternatives for ev for Typed Impcore 250b⟩
  in  ev e
  end
```

ev : exp -> value

To apply a function, we build an evaluation environment. We strip the types off the formals and we put the actuals in mutable ref cells.

⟨*apply user-defined function* func *to* args 680b⟩≡ (680a)

```
let val (formals, body) = func
    val actuals         = map (ref o ev) args
in  eval (body, globals, functions, bindList (formals, actuals, emptyEnv))
    handle BindListLength =>
        raise BugInTypeChecking "Wrong number of arguments to function"
end
```

formals : name list
actuals : value ref list

The implementation of the evaluator uses the same techniques we use to implement μScheme in Chapter 5. Because of Typed Impcore's many environments, the evaluator does more bookkeeping. Another difference is that many potential run-time errors should be impossible because the relevant code would be rejected by the type checker. If one of those errors occurs anyway, we raise the exception `BugInTypeChecking`.

681a ⟨*evaluation for Typed Impcore* 681a⟩≡ (246a) 681b▷

```
eval : exp * value ref env * func env * value ref env -> value
```

⟨*definition of* `eval` *for Typed Impcore* 680a⟩

681b ⟨*evaluation for Typed Impcore* 681a⟩+≡ (246a) ◁681a 681d▷

```
evaldef : def * value ref env * func env * (string->unit) -> value ref env * func env
fun evaldef (d, globals, functions, echo) =
  case d
    of VAL (name, e) => ⟨evaluate e and bind to name 681c⟩
     | EXP e => evaldef (VAL ("it", e), globals, functions, echo)
     | DEFINE (f, {body=e, formals=xs, returns=rt}) =>
         (globals, bind (f, USERDEF (map #1 xs, e), functions))
         before echo f
     | USE _ => raise RuntimeError "Internal error - 'use' was evaluated"
```

681c ⟨*evaluate* e *and bind to* name 681c⟩≡ (681b)

```
let val v = eval (e, globals, functions, emptyEnv)
    val _ = echo (valueString v)
in  (bind (name, ref v, globals), functions)
end
```

Here are the primitives. As in Chapter 5, all are either binary or unary operators. Type checking should guarantee that operators are used with the correct arity.

681d ⟨*evaluation for Typed Impcore* 681a⟩+≡ (246a) ◁681b 681e▷

```
unaryOp  : (value         -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)
fun binaryOp f = (fn [a, b] => f (a, b) | _ => raise BugInTypeChecking "arity 2")
fun unaryOp  f = (fn [a]    => f a      | _ => raise BugInTypeChecking "arity 1")
```

Arithmetic primitives expect and return integers.

681e ⟨*evaluation for Typed Impcore* 681a⟩+≡ (246a) ◁681d 682a▷

```
arithOp   : (int * int -> int) -> (value list -> value)
arithtype : funty
fun arithOp f =
      binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                 | _ => raise BugInTypeChecking "arithmetic on non-numbers")
val arithtype = FUNTY ([INTTY, INTTY], INTTY)
```

As in Chapter 5, we use the chunk ⟨*primops for Typed Impcore* :: 681f⟩ to cons up all the primitives into one giant list, and we use that list to build the initial environment for the read-eval-print loop. The big difference is that in Typed Impcore, each primitive has a type as well as a value.

681f ⟨*primops for Typed Impcore* :: 681f⟩≡ (247b) 682b▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```

Comparisons take two arguments. Most comparisons (except for equality) apply only to integers.

682a ⟨*evaluation for Typed Impcore* 681a⟩+≡ (246a) ◁681e

```
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
comptype   : funty
```

```
fun embedPredicate f args = NUM (if f args then 1 else 0)
fun comparison f = binaryOp (embedPredicate f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeChecking "comparing non-numbers")
val comptype = FUNTY ([INTTY, INTTY], BOOLTY)
```

682b ⟨*primops for Typed Impcore* :: 681f⟩+≡ (247b) ◁681f

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
```