# Appendix G

# Supporting code for the Typed $\mu$Scheme interpreter

## G.1 Printing types and values

This code prints types. It might be desirable to print them using a more ML-like syntax.

683 ⟨*printing types for Typed μScheme* 683⟩≡ (263c)

```
fun typeString (TYCON c) = c
  | typeString (TYVAR a) = a
  | typeString (CONAPP (TYCON "function", [CONAPP (TYCON "tuple", args), result])) =
      "(function (" ^ spaceSep (map typeString args) ^ ") " ^ typeString result ^ ")"
  | typeString (CONAPP (tau, [])) = "(" ^ typeString tau ^ ")"
  | typeString (CONAPP (tau, l)) =
      "(" ^ typeString tau ^ " " ^ spaceSep (map typeString l) ^ ")"
  | typeString (FORALL (l, tau)) =
      "(forall (" ^ spaceSep l ^ ") " ^ typeString tau ^ ")"
```

## G.2 Parsing

684a ⟨*parsing for Typed μScheme* 684a⟩≡ (275a) 684b ▷

```
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 keyword syntax words =
  let fun isKeyword s = List.exists (fn s' => s = s') words
  in  (fn (NAME n) => if isKeyword n then SOME n else NONE | _ => NONE) <$>? token
  end

val expKeyword = keyword "type"       ["if", "while", "set", "begin", "lambda",
                                        "type-lambda", "let", "let*", "@"]
val tyKeyword  = keyword "expression" ["forall", "function"]

val tlformals = nodups ("formal type parameter", "type-lambda") <$>! @@ (many name)

fun nodupsty what (loc, xts) = nodups what (loc, map fst xts) >>=+ (fn _ => xts)
                                                       (* error on duplicate names *)

fun letDups LETSTAR (_, bindings) = OK bindings
  | letDups LET     bindings       = nodupsty ("bound variable", "let") bindings
```

When parsing a type, we reject anything that looks like an expression.

684b ⟨*parsing for Typed μScheme* 684a⟩+≡ (275a) ◁ 684a 685 ▷

```
val tyvar = quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mark)")

fun checkedForall tyvars tau =
  nodups ("quantified type variable", "forall") tyvars >>=+ (fn a's =>
  FORALL (a's, tau))

fun ty tokens = (
     TYCON <$> name
 <|> TYVAR <$> tyvar
 <|> bracket "forall"    "(forall (tyvars) type)"
                         (checkedForall <$> "(" >-- @@ (many tyvar) --< ")" <*>! ty)
 <|> bracket "function" "(function (types) type)"
                         (curry funtype <$> "(" >-- many ty --< ")" <*> ty)
 <|> badExpKeyword <$>! ("(" >-- @@ expKeyword <* scanToCloseParen)
 <|> curry CONAPP <$> "(" >-- ty <*> many ty --< ")"
 <|> "(" >-- literal ")" <!> "empty type ()"
 <|> int <!> "expected type; found integer"
 <|> booltok <!> "expected type; found Boolean literal"
) tokens
and badExpKeyword (loc, bad) =
      errorAt ("looking for type but found '" ^ bad ^ "'") loc
```

tyvar : string parser
ty    : tyex parser

When parsing an expression, we reject anything that looks like a type.

685 ⟨*parsing for Typed μScheme* 684a⟩+≡ (275a) ◁684b 686a▷

```
val formal =
  "(" >-- ((fn tau => fn x => (x, tau)) <$> ty <*> name --< ")" <?> "(ty argname)")
val lformals = "(" >-- many formal --< ")"
val tformals = "(" >-- many tyvar  --< ")"

fun lambda xs exp =
      nodupsty ("formal parameter", "lambda") xs >>=+ (fn xs => LAMBDA (xs, exp))
fun tylambda a's exp =
      nodups ("formal type parameter", "type-lambda") a's >>=+ (fn a's =>
      TYLAMBDA (a's, exp))

val br = bracket

fun exp tokens = (
     VAR                <$> name
 <|> (LITERAL o NUM)  <$> int
 <|> (LITERAL o BOOL) <$> booltok
 <|> LITERAL          <$> (quote *> sexp)
 <|> br "if"     "(if e1 e2 e3)"            (curry3 IFX     <$> exp  <*> exp <*> exp
 <|> br "while"  "(while e1 e2)"            (curry  WHILEX  <$> exp  <*> exp)
 <|> br "set"    "(set x e)"                (curry  SET     <$> name <*> exp)
 <|> br "begin"  ""                         (       BEGIN   <$> many exp)
 <|> br "lambda" "(lambda (formals) body)"  (       lambda  <$> @@ lformals <*>! exp
 <|> br "type-lambda" "(type-lambda (tyvars) body)"
                                            (       tylambda <$> @@ tformals <*>! ex
 <|> br "let"    "(let (bindings) body)"    (letx   LET     <$> @@ bindings <*>! exp
 <|> br "letrec" "(letrec (bindings) body)" (letrec <$> bindings <*>! exp)
 <|> br "let*"   "(let* (bindings) body)"   (letx   LETSTAR <$> @@ bindings <*>! exp
 <|> br "@"      "(@ exp types)"            (curry  TYAPPLY <$> exp <*> many1 ty)
 <|> badTyKeyword <$>! ("(" >-- @@ tyKeyword <* scanToCloseParen)
 <|> "(" >-- literal ")" <!> "empty application"
 <|> curry APPLY <$> "(" >-- exp <*> many exp --< ")"
) tokens

and letx kind bs exp = letDups kind bs >>=+ (fn bs => LETX (kind, bs, exp))
and letrec _ _ = ERROR  "letrec is not included in Typed uScheme"
and bindings ts = ("(" >-- (many binding --< ")" <?> "(x e)...")) ts
and binding  ts = ("(" >-- (pair <$> name <*> exp --< ")" <?> "(x e) in bindings"))

and badTyKeyword (loc, bad) =
      errorAt ("looking for expression but found '" ^ bad ^ "'") loc

and sexp tokens = (
     SYM          <$> (notDot <$>! name)
 <|> NUM          <$> int
 <|> BOOL         <$> booltok
 <|> (fn v => embedList [SYM "quote", v]) <$> (quote *> sexp)
 <|> embedList    <$> "(" >-- many sexp --< ")"
) tokens
and notDot "." = ERROR "this interpreter cannot handle . in quoted S-expressions"
  | notDot s   = OK s
```

686a ⟨*parsing for Typed μScheme* 684a⟩+≡ (275a) ◁685 686b▷

```
fun define tau f formals body =
  nodupsty ("formal parameter", "definition of function " ^ f) formals >>=+ (fn xts =>
  DEFINE (f, tau, (xts, body)))

fun valrec tau x e = VALREC (x, tau, e)

val def =
     bracket "define" "(define type f (args) body)"
                                     (define <$> ty <*> name <*> @@ lformals <*>! exp)
 <|> bracket "val"     "(val x e)"              (curry VAL <$> name <*> exp)
 <|> bracket "val-rec" "(val-rec type x e)"     (valrec <$> ty <*> name <*> exp)
 <|> bracket "use"     "(use filename)"         (USE       <$> name)
 <|> literal ")" <!> "unexpected right parenthesis"
 <|> EXP <$> exp
 <?> "definition"
```

686b ⟨*parsing for Typed μScheme* 684a⟩+≡ (275a) ◁686a

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

## G.3 Evaluation

The implementation of the evaluator is almost identical to the implementation in Chapter 5. There are only two significant differences: we have to deal with the mismatch in representations between the abstract syntax LAMBDA and the value CLOSURE, and we have to write cases for the TYAPPLY and TYLAMBDA expressions. 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, not RuntimeError.

686c ⟨*evaluation for Typed μScheme* 686c⟩≡ (273a) 688a▷

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

```
fun eval (e, rho) =
  let fun ev (LITERAL n) = n
        ⟨alternatives for ev for TYAPPLY and TYLAMBDA 272b⟩
        ⟨more alternatives for ev for Typed μScheme 686d⟩
  in  ev e
  end
```

Code for variables is just as in Chapter 5.

⟨*more alternatives for* ev *for Typed μScheme* 686d⟩≡ (686c) 687a▷

```
| ev (VAR v) = !(find (v, rho))
| ev (SET (n, e)) =
    let val v = ev e
    in  find (n, rho) := v;
        v
    end
```

Code for control flow is just as in Chapter 5.

687a ⟨*more alternatives for* ev *for Typed μScheme* 686d⟩+≡ (686c) ◁686d 687b▷

```
| ev (IFX (e1, e2, e3)) = ev (if bool (ev e1) then e2 else e3)
| ev (WHILEX (guard, body)) =
    if bool (ev guard) then
      (ev body; ev (WHILEX (guard, body)))
    else
      unitVal
| ev (BEGIN es) =
    let fun b (e::es, lastval) = b (es, ev e)
          | b (   [], lastval) = lastval
    in  b (es, unitVal)
    end
```

Code for a `lambda` has to remove the types from the abstract syntax.

687b ⟨*more alternatives for* ev *for Typed μScheme* 686d⟩+≡ (686c) ◁687a 687c▷

```
| ev (LAMBDA (args, body)) = CLOSURE ((map (fn (n, ty) => n) args, body), rho)
```

Code for application is almost as in Chapter 5, except if the program tries to apply a non-function, we raise `BugInTypeChecking`, not `RuntimeError`, because the type checker should reject any program that could apply a non-function.

687c ⟨*more alternatives for* ev *for Typed μScheme* 686d⟩+≡ (686c) ◁687b 687d▷

```
| ev (APPLY (f, args))  =
       (case ev f
          of PRIMITIVE prim => prim (map ev args)
           | CLOSURE clo => ⟨apply closure clo to args 218c⟩
           | v => raise BugInTypeChecking "applied non-function"
       )
```

Code for the `LETX` family is as in Chapter 5.

687d ⟨*more alternatives for* ev *for Typed μScheme* 686d⟩+≡ (686c) ◁687c

```
| ev (LETX (LET, bs, body)) =
    let val (names, values) = ListPair.unzip bs
    in  eval (body, bindList (names, map (ref o ev) values, rho))
    end
| ev (LETX (LETSTAR, bs, body)) =
    let fun step ((n, e), rho) = bind (n, ref (eval (e, rho)), rho)
    in  eval (body, foldl step rho bs)
    end
```

Evaluating a definition can produce a new environment. The function `evaldef` also returns a string which, if nonempty, should be printed to show the value of the item. Type soundness requires a change in the evaluation rule for `VAL`; as described in Exercise 37 in Chapter 3, `VAL` must always create a new binding.

688a ⟨*evaluation for Typed μScheme* 686c⟩+≡ (273a) ◁686c 688b▷

```
fun evaldef (d, rho) =      evaldef : def * value ref env -> value ref env * string
  case d
    of VAL    (name, e)      =>
         let val v   = eval (e, rho)
             val rho = bind (name, ref v, rho)
         in  (rho, showVal name v)
         end
     | VALREC (name, tau, e) =>
         let val rho = bind (name, ref NIL, rho)
             val v   = eval (e, rho)
         in  find (name, rho) := v;
             (rho, showVal name v)
         end
     | EXP e => (* differs from VAL ("it", e) only in what it prints *)
         let val v   = eval (e, rho)
             val rho = bind ("it", ref v, rho)
         in  (rho, valueString v)
         end
     | DEFINE (name, tau, lambda) => evaldef (VALREC (name, tau, LAMBDA lambda), rho)
     | USE filename => raise RuntimeError "internal error -- 'use' reached evaldef"
```

In the `VALREC` case, the interpreter evaluates `e` while `name` is still bound to `NIL`—that is, before the assignment to `find (name, rho)`. Therefore, as described on page 271, evaluating `e` must not evaluate `name`—because the mutable cell for `name` does not yet contain its correct value.

Both `VAL` and `VALREC` show names as follows:

688b ⟨*evaluation for Typed μScheme* 686c⟩+≡ (273a) ◁688a 688c▷

```
and showVal name v =
      case v
        of CLOSURE   _ => name
         | PRIMITIVE _ => name
         | _ => valueString v
```

| | |
|---|---|
| bind | 214 |
| BugInTypeChecking | 273a |
| CLOSURE | 268a |
| DEFINE | 268b |
| eval | 686c |
| EXP | 268b |
| find | 214 |
| LAMBDA | 268a |
| NIL | 268a |
| PRIMITIVE | 268a |
| RuntimeError | 268a |
| USE | 268b |
| VAL | 268b |
| VALREC | 268b |
| valueString | 217a |

## G.4 Primitives of Typed μScheme

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.

688c ⟨*evaluation for Typed μScheme* 686c⟩+≡ (273a) ◁688b 689a▷

```
                        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.

689a ⟨*evaluation for Typed μScheme* 686c⟩+≡ (273a) ◁688c 689c▷

```
arithOp   : (int * int -> int) -> (value list -> value)
arithtype : tyex
```

```
fun arithOp f =
      binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                 | _ => raise BugInTypeChecking "arithmetic on non-numbers")
val arithtype = funtype ([inttype, inttype], inttype)
```

As in Chapter 5, we use the chunk ⟨*primitive functions for Typed μScheme* :: 689b⟩ 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 μScheme, each primitive has a type as well as a value.

689b ⟨*primitive functions for Typed μScheme* :: 689b⟩≡ (274b) 689d▷

```
("+", 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.

689c ⟨*evaluation for Typed μScheme* 686c⟩+≡ (273a) ◁689a

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

```
fun embedPredicate f args = BOOL (f args)
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 = funtype ([inttype, inttype], booltype)
```

689d ⟨*primitive functions for Typed μScheme* :: 689b⟩+≡ (274b) ◁689b 689e▷

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
("=", comparison (fn (NIL,     NIL    ) => true
                   | (NUM  n1, NUM  n2) => n1 = n2
                   | (SYM  v1, SYM  v2) => v1 = v2
                   | (BOOL b1, BOOL b2) => b1 = b2
                   | _                  => false)
    , FORALL (["'a"], funtype ([tyvarA, tyvarA], booltype))) ::
```

The list primitives have polymorphic types.

689e ⟨*primitive functions for Typed μScheme* :: 689b⟩+≡ (274b) ◁689d 690a▷

```
("null?", unaryOp (embedPredicate (fn (NIL   ) => true | _ => false))
    , FORALL (["'a"], funtype ([listtype tyvarA], booltype))) ::
("cons", binaryOp (fn (a, b) => PAIR (a, b))
    , FORALL (["'a"], funtype ([tyvarA, listtype tyvarA], listtype tyvarA))) ::
("car",  unaryOp  (fn (PAIR (car, _)) => car
                    | v => raise RuntimeError
                                  ("car applied to non-list " ^ valueString v))
    , FORALL (["'a"], funtype ([listtype tyvarA], tyvarA))) ::
("cdr",  unaryOp  (fn (PAIR (_, cdr)) => cdr
                    | v => raise RuntimeError
                                  ("cdr applied to non-list " ^ valueString v))
    , FORALL (["'a"], funtype ([listtype tyvarA], listtype tyvarA))) ::
```

The print primitive also has a polymorphic type.

690a ⟨*primitive functions for Typed μScheme* :: 689b⟩+≡ (274b) ◁689e
```
("print", unaryOp (fn x => (print (valueString x^"\n"); unitVal)),
   FORALL (["'a"], funtype ([tyvarA], unittype))) ::
```

In plain Typed μScheme, all the primitives are functions, so this chunk is empty. But you might add to it in the Exercises.

690b ⟨*primitives that aren't functions, for Typed μScheme* :: 690b⟩≡ (274b)
```
(* if this space is completely empty, something goes wrong with the software OMIT *)
```

## G.5 Initial basis

Because programming in Typed μScheme is an awful lot of trouble, Typed μScheme has a smaller initial basis than μScheme. Some of the basis functions are defined in Chapter 6. The rest are here.

Becauses lists in Typed μScheme must be homogeneous, the funny list functions built from car and cdr are much less useful than in μScheme.

690c ⟨*additions to the Typed μScheme initial basis* 690c⟩≡ 690d▷
```
(val caar
   (type-lambda ('a)
      (lambda (((list (list 'a)) l))
          ((@ car 'a) ((@ car (list 'a)) l)))))
(val cadr
   (type-lambda ('a)
      (lambda (((list (list 'a)) l))
          ((@ car (list 'a)) ((@ cdr (list 'a)) l)))))
```

The Boolean functions are almost exactly as in Typed Impcore.

690d ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁690c 690e▷
```
(define bool and ((bool b) (bool c)) (if b  c  b))
(define bool or  ((bool b) (bool c)) (if b  b  c))
(define bool not ((bool b))          (if b #f #t))
```

Here is list append.

690e ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁690d 691a▷
```
(val-rec (forall ('a) (function ((list 'a) (list 'a)) (list 'a))) append
  (type-lambda ('a)
    (lambda (((list 'a) xs)  ((list 'a) ys))
      (if ((@ null? 'a) xs)
        ys
        ((@ cons 'a) ((@ car 'a) xs) ((@ append 'a) ((@ cdr 'a) xs) ys))))))
```

| | |
|---|---|
| FORALL | 260 |
| funtype | 267b |
| tyvarA | 267b |
| unaryOp | 688c |
| unittype | 267b |
| unitVal | 267c |
| valueString | 217a |

In Typed μScheme, an association list must be represented as a list of pairs. The only sensible way to write a lookup function for an association list is to use continuation-passing style. These problems are given as exercises.

We provide just some of the list functions found in μScheme. Here is `filter`.

691a ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁690e 691b▷

```
(val-rec (forall ('a) (function ((function ('a) bool) (list 'a)) (list 'a))) filter
   (type-lambda ('a)
      (lambda (((function ('a) bool) p?)  ((list 'a) l))
         (if ((@ null? 'a) l)
             (@ '() 'a)
             (if (p? ((@ car 'a) l))
                 ((@ cons 'a) ((@ car 'a) l) ((@ filter 'a) p? ((@ cdr 'a) l)))
                 ((@ filter 'a) p? ((@ cdr 'a) l)))))))
; missing exists?
; missing all?
```

Here is map.

691b ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁691a 691c▷

```
(val-rec (forall ('a 'b) (function ((function ('a) 'b) (list 'a)) (list 'b))) map
   (type-lambda ('a 'b)
      (lambda (((function ('a) 'b) f)  ((list 'a) l))
         (if ((@ null? 'a) l)
             (@ '() 'b)
             ((@ cons 'b) (f ((@ car 'a) l)) ((@ map 'a 'b) f ((@ cdr 'a) l)))))))
```

Function `foldr` is also given as an exercise.

Integer comparisons are easy, but to define `!=` we need a type abstraction.

691c ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁691b 691d▷

```
(define bool <= ((int x) (int y)) (not (> x y)))
(define bool >= ((int x) (int y)) (not (< x y)))
(val != (type-lambda ('a) (lambda (('a x) ('a y)) (not ((@ = 'a) x y)))))
```

Integer functions are also easy, but we must be careful to instantiate polymorphic equality.

691d ⟨*additions to the Typed μScheme initial basis* 690c⟩+≡ ◁691c

```
(define int max ((int x) (int y)) (if (> x y) x y))
(define int min ((int x) (int y)) (if (< x y) x y))

(define int mod ((int m) (int n)) (- m (* n (/ m n))))
(define int gcd ((int m) (int n)) (if ((@ = int) n 0) m (gcd n (mod m n))))
(define int lcm ((int m) (int n)) (* m (/ n (gcd m n))))
```