Appendix H

Supporting code for μML

H.1 Printing types and constraints

This code prints types without too many redundant parentheses. It uses infix notation for the `function` and `tuple` constructors.

693 ⟨*printing types* 693⟩≡ (295c)

```
⟨definition of separate 218d⟩
local
  (* precedences *)
  val CONp   = 3
  val STARp  = 2
  val ARROWp = 1
  val NONEp  = 0

  fun parens s = "(" ^ s ^ ")"
  fun bracket (s, context, prec) = if prec <= context then parens s else s
  fun p (context, CONAPP (TYCON "tuple", l)) = bracket (ptuple l, context, STARp)
    | p (context, CONAPP (TYCON "function", [arg, ret])) =
        bracket (p (ARROWp, arg) ^ " -> " ^ p (ARROWp, ret), context, ARROWp)
    | p (context, CONAPP (n, []))  = p (context, n)
    | p (context, CONAPP (n, [t])) = p (CONp, t) ^ " " ^ p (CONp, n)
    | p (context, CONAPP (n, l)) =
        "(" ^ separate ("", ", ") (map typeString l) ^ ") " ^ p (CONp, n)
    | p (context, TYCON n) = n
    | p (context, TYVAR v) = "'" ^ v
  and ptuple l = separate ("unit", " * ") (map (fn t => p (STARp, t)) l)
  and typeString ty = p (NONEp, ty)
in
  val typeString = typeString
  val ptuple = ptuple
end
```

A constraint can be printed in full, but it's easier to read if its first passed to `untriviate`, which removes as many `TRIVIAL` sub-constraints as possible.

694a ⟨*printing constraints* 694a⟩≡ (320c)

```
fun untriviate (c /\ c') = (case (untriviate c, untriviate c')
                              of (TRIVIAL, c) => c
                               | (c, TRIVIAL) => c
                               | (c, c') => c /\ c')
  | untriviate atomic = atomic

fun constraintString (c /\ c')  = constraintString c ^ " /\\ " ^ constraintString c'
  | constraintString (t =*= t') = typeString t ^ " =*= " ^ typeString t'
  | constraintString TRIVIAL = "TRIVIAL"
```

H.2 Parsing

μML can use μScheme's lexical analysis, so all we have here is a parser. As with μScheme, we begin with error-detection functions.

694b ⟨*parsing* 694b⟩≡ (331c) 695a▷

```
fun letDups LETSTAR (loc, bindings) = OK bindings
  | letDups kind    (loc, bindings) =
      let val names    = map (fn (n, _) => n) bindings
          val kindName = case kind of LET => "let" | LETREC => "letrec" | _ => "??"
      in  nodups ("bound name", kindName) (loc, names) >>=+ (fn _ => bindings)
      end
fun noExp kind _ _ =
  ERROR ("uML does not include '" ^ kind ^ "' expressions")
```

=*=	320b
>>=+	652a
ERROR	651a
LET	287a
LETREC	287a
LETSTAR	287a
nodups	666a
OK	651a
TRIVIAL	320b
typeString	693

695a ⟨*parsing* 694b⟩+≡ (331c) ◁694b 695b▷

```
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
 <|> (LITERAL o BOOL) <$> booltok
 <|> LITERAL          <$> (quote *> sexp)
 <|> bracket "if"     "(if e1 e2 e3)"           (curry3 IFX <$> exp <*> exp <*> exp)
 <|> bracket "while"  "(while e1 e2)"           (noExp "while" <$> exp <*>! exp)
 <|> bracket "set"    "(set x e)"               (noExp "set"   <$> name <*>! exp)
 <|> bracket "begin"  ""                        (       BEGIN <$> many exp)
 <|> bracket "lambda" "(lambda (names) body)"   (     lambda  <$> @@ formals  <*>!
 <|> bracket "let"    "(let (bindings) body)"   (letx LET     <$> @@ bindings <*>!
 <|> bracket "letrec" "(letrec (bindings) body)" (letx LETREC  <$> @@ bindings <*>!
 <|> bracket "let*"   "(let* (bindings) body)"  (letx LETSTAR <$> @@ bindings <*>!
 <|> "(" >-- literal ")" <!> "empty application"
 <|> curry APPLY <$> "(" >-- exp <*> many exp --< ")"
) tokens
and lambda xs exp =
  nodups ("formal parameter", "lambda") xs >>=+ (fn xs => LAMBDA (xs, exp))
and letx kind bs exp = letDups kind bs >>=+ (fn bs => LETX (kind, bs, exp))
and formals  ts = ("(" >-- many name --< ")") ts
and bindings ts = ("(" >-- (many binding --< ")" <?> "(x e)...")) ts
and binding  ts = ("(" >-- (pair <$> name <*> exp --< ")" <?> "(x e) in bindings")) ts

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
```

695b ⟨*parsing* 694b⟩+≡ (331c) ◁695a 696a▷

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

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

696a ⟨*parsing* 694b⟩+≡ (331c) ◁695b
```
val mlSyntax = (schemeToken, def)
```

H.3 Initial basis

These parts of the initial basis are identical to what we find in μScheme.

696b ⟨*additions to the μML initial basis* 696b⟩≡ 696c▷
```
(define caar (l) (car (car l)))
(define cadr (l) (car (cdr l)))
(define cdar (l) (cdr (car l)))
(define length (l)
  (if (null? l) 0
    (+ 1 (length (cdr l)))))
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))
```

696c ⟨*additions to the μML initial basis* 696b⟩+≡ ◁696b 696d▷
```
(define append (xs ys)
  (if (null? xs)
     ys
     (cons (car xs) (append (cdr xs) ys))))
(define revapp (xs ys)
  (if (null? xs)
     ys
     (revapp (cdr xs) (cons (car xs) ys))))
```

696d ⟨*additions to the μML initial basis* 696b⟩+≡ ◁696c 696e▷
```
(define o (f g) (lambda (x) (f (g x))))
(define curry   (f) (lambda (x) (lambda (y) (f x y))))
(define uncurry (f) (lambda (x y) ((f x) y)))
```

696e ⟨*additions to the μML initial basis* 696b⟩+≡ ◁696d 696f▷
```
(define filter (p? l)
  (if (null? l)
    '()
    (if (p? (car l))
      (cons (car l) (filter p? (cdr l)))
      (filter p? (cdr l)))))
```

car P 329b
cdr P 329b
cons P 329b
def 695b
null? P 329a
schemeToken 672a

⟨*additions to the μML initial basis* 696b⟩+≡ ◁696e 697a▷
```
(define map (f l)
  (if (null? l)
    '()
    (cons (f (car l)) (map f (cdr l)))))
```

697a ⟨*additions to the μML initial basis* 696b⟩+≡ ◁696f 697b▷

```
(define exists? (p? l)
  (if (null? l)
    #f
    (if (p? (car l))
      #t
      (exists? p? (cdr l)))))
(define all? (p? l)
  (if (null? l)
    #t
    (if (p? (car l))
      (all? p? (cdr l))
      #f)))
```

697b ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697a 697c▷

```
(define foldr (op zero l)
  (if (null? l)
    zero
    (op (car l) (foldr op zero (cdr l)))))
(define foldl (op zero l)
  (if (null? l)
    zero
    (foldl op (op (car l) zero) (cdr l))))
```

697c ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697b 697d▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

697d ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697c 697e▷

```
(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))
(define mod (m n) (- m (* n (/ m n))))
(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
(define lcm (m n) (* m (/ n (gcd m n))))
```

697e ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697d 697f▷

```
(define min* (l) (foldr min (car l) (cdr l)))
(define max* (l) (foldr max (car l) (cdr l)))
(define gcd* (l) (foldr gcd (car l) (cdr l)))
(define lcm* (l) (foldr lcm (car l) (cdr l)))
```

car	P 329b
cdr	P 329b
cons	P 329b
not	696b
null?	P 329a

697f ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697e 698▷

```
(define list1 (x)             (cons x '()))
(define list2 (x y)           (cons x (list1 y)))
(define list3 (x y z)         (cons x (list2 y z)))
(define list4 (x y z a)       (cons x (list3 y z a)))
(define list5 (x y z a b)     (cons x (list4 y z a b)))
(define list6 (x y z a b c)   (cons x (list5 y z a b c)))
(define list7 (x y z a b c d) (cons x (list6 y z a b c d)))
(define list8 (x y z a b c d e) (cons x (list7 y z a b c d e)))
```

698 ⟨*additions to the μML initial basis* 696b⟩+≡ ◁697f

```
(define takewhile (p? l)
  (if (null? l)
      '()
      (if (p? (car l))
          (cons (car l) (takewhile p? (cdr l)))
          '())))
(define dropwhile (p? l)
  (if (null? l)
      '()
      (if (p? (car l))
          (dropwhile p? (cdr l))
          l)))
```

car 𝒫 329b
cdr 𝒫 329b
cons 𝒫 329b
null? 𝒫 329a