Additional Material for Lecture 6 Type checking

```
data Exp
  =  While Exp Exp
  ...
  | Bool Bool
  | If Exp Exp Exp

  | Int  Int
  | Add  Exp Exp
  | Sub  Exp Exp
  | Mul  Exp Exp
  | Div  Exp Exp
  | Leq  Exp Exp

  | Char Char
  | Ceq Exp Exp

  | Pair Exp Exp
  | Fst Exp
  | Snd Exp

  | Cons Exp Exp
  | Nil
  | Head Exp
  | Tail Exp
  | Null Exp
```

```
data Value
     = IntV Int
     | PairV Addr
     | CharV Char
     | BoolV Bool
     | ConsV Addr
     | NilV
```

```
data Typ
   = TyVar String        -- a, b , c
   | TyPair Typ Typ      -- (Int . Bool)
   | TyFun [Typ] Typ     -- Int -> Bool -> Int
   | TyList Typ          -- [ Int]
   | TyCon String        -- Bool, Char, etc
```

We divide the universe of values into types.
Many expression forms are used to construct values of some type, or to consume values of some type.

Composite types

- Built from type constructors.
- In E6
  - (Int . Bool)
  - [Int]
  - Int -> a -> (Bool . Char)

```
data Typ
   = TyVar String        -- a, b , c
   | TyPair Typ Typ      -- (Int . Bool)
   | TyFun [Typ] Typ     -- Int -> Bool -> Int
   | TyList Typ          -- [ Int]
   | TyCon String        -- Bool, Char, etc
```

Static Type Checking

- Based on declarations of types

```
(fun append [a] (l [a]  m [a] )  { return (l ++ m) }
     (if (@isnil l) m
         (cons (head l) (@append (tail l) m))))

{ generate the list [1,2,...,n] }
(fun gen [Int] (n Int)
     (local (temp nil)
     (block
       (:= temp nil)
       (while (@not (@eq n 0)) (block
                  (:= temp (cons n temp))
                  (:= n (- n 1))))
        temp)))
```

Dynamic Type Checking

- Based on runtime predicates
- Recall language E5

```
exp := var
…
| '(' 'ispair' exp ')'
| '(' 'ischar' exp ')'
| '(' 'ispair' exp ')'
| '(' 'isint' exp ')'
```

Typing judgments 1

- In the context of an environment that maps names to types.

$$\frac{TE(x) = t}{TE \vdash x : t} \ \text{(Var)}$$

```
infer fs vs (term@(Var s pos)) =
  case lookup s vs of
    Just sch -> instantiate sch
    Nothing ->
      error ("\nNear "++show pos++
             "\nUnknown var: "++ s)
```

More about this later

Typing judgments 2

$$\frac{TE \vdash e_1 : \texttt{Int} \quad TE \vdash e_2 : \texttt{Int}}{TE \vdash (+\ \ e_1\ \ e_2) : \texttt{Int}}\ (\text{Add})$$

```
infer fs vs (term@(Add x y)) =
  do { check fs vs x intT "(+)"
     ; check fs vs y intT "(+)"
     ; return intT}
```

$$\frac{TE \vdash e_1 : \texttt{Bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash (\texttt{if}\ e_1\ e_2\ e_3) : t} \text{(If)}$$

```
infer fs vs (term@(If x y z)) =
  do { check fs vs x boolT "if statement test"
     ; t1 <- infer fs vs y
     ; t2 <- infer fs vs z
     ; unify t2 t1 (loc term)
         ["\nWhile infering the term\n   "
         ,show term
         ,"\nThe branches don’t match" ]
     ; return t1}
```

Polymorphism

We add to the types, the notion of a type variable

This can take on any type.

```
(fun append [a] (l [a]  m [a] )
      (if (@isnil l) m
          (cons (head l)
                (@append (tail l)
                         m))))
```

Implementation

- The notion of a type variable

```
data Typ
   = TyVar String         -- a, b , c
   | TyPair Typ Typ       -- (Int . Bool)
   | TyFun [Typ] Typ      -- Int -> Bool -> Int
   | TyList Typ           -- [ Int]
   | TyCon String         -- Bool, Char, etc
   | TyFresh (Uniq,Pointer Typ)
```

Fresh instances

```
(fun hd a (xs [a]) (head x))
```

[a23] -> a23

```
(pair (@ hd (@list3 1 2 4))
      (@ hd (cons True
              (cons False nil))))
```

[a25] -> a25

Each instance of hd gets a fresh instance of the type [a] -> a

We see
a23 = Int
a25 = Bool

Making a fresh instance

```
infer fs vs (term@(Var s pos)) =
  case lookup s vs of
    Just sch -> instantiate sch
    Nothing ->
      error ("\nNear "++show pos++
             "\nUnknown var: "++ s)
```

More about this later

User defined types

```
data Temp = F Float | C Float

boiling (F x) = x >= 212.0
boiling (C x) = x >= 100.0
```

Recursive types

```
data Inttree
   = Branch Inttree Inttree
   | Leaf Int

sumleaves (Leaf i) = i
sumleaves (Branch l r)
   = sumleaves l + sumleaves r
```

Parameterized types

```
data Bintree a
   = Branch (Bintree a) (Bintree a)
   | Leaf a

depth (Leaf i) = 0
depth (Branch l r)
   = 1 + max (depth l) (depth r)
```

Enumerations

```
data Day = Mon | Tue |Wed |Thu
         |Fri |Sat | Sun

weekday Sat = False
weekday Sun = False
weekday  x = True

data Bool = True | False
```