This Lecture (and the next)

Step by step development of a type checker for LTXL:

- LTXL abstract syntax
- LTXL types
- Informal typing rules for LTXL
- Formal typing rules for LTXL
- Additional infrastructure (handout)
- Implementing the type checker (interactively)

LTXL Abstract Syntax

LTXL example program, concrete syntax:

let int x = 7; int y = 5 in x * y + 7

Typing rule/handwriting friendly version of the LTXL abstract syntax:

\[ e \rightarrow n \quad \text{literal integer} \]
\[ x \quad \text{variable} \]
\[ \oplus e \quad \text{unary operator app.} \]
\[ e \odot e \quad \text{binary operator app.} \]
\[ \text{if } e \text{ then } e \text{ else } e \quad \text{conditional expression} \]
\[ \text{let } (T : e) \in e \quad \text{let-expression} \]

LTXL AST Representation (recap)

type Id = String

data Exp
  = LitInt Int
  | Var Id
  | UnOpApp UnOp Exp
  | BinOpApp BinOp Exp Exp
  | If Exp Exp Exp
  | Let [(Id, Type, Exp)] Exp

LTXL Types

LTXL type syntax:

\[ T \rightarrow \text{int} \quad \text{integer type} \]
\[ \text{bool} \quad \text{boolean type} \]
\[ (T', T) \quad \text{product (pair)} \]
\[ T \rightarrow T \quad \text{function} \]

LTXL Operator Types

Unary LTXL operator types:

\[ \ld : \text{bool} \rightarrow \text{bool} \]
\[ \ld : \text{int} \rightarrow \text{int} \quad \text{unary minus} \]

Binary LTXL operator types:

\[ \|, \& \& : (\text{bool, bool}) \rightarrow \text{bool} \]
\[ <,\leq,> : (\text{int, int}) \rightarrow \text{bool} \]
\[ +,-,*,/ : (\text{int, int}) \rightarrow \text{int} \]

LTXL Operator Representation

data UnOp = Not | Neg

data BinOp = Or
  | And
  | Less
  | Equal
  | Greater
  | Plus
  | Minus
  | Times
  | Divide

Example: An LTXL Program

The LTXL example program again:

let int x = 7; int y = 5 in x * y + 7

Representation:

Let [{("x", IntType, LitInt 7)},
    {("y", IntType, LitInt 5)}]
(BinOpApp
  (BinOpApp
    (Var "x")
    (Var "y"))
  (LitInt 7))
The LTXL expression typing relation is a ternary (or trinary) relation:

\[ \Gamma \vdash e : T \]

Read: expression \( e \) has type \( T \) in type environment \( \Gamma \)

1. A literal integer has type \( \text{int} \).
   \[ \Gamma \vdash n : \text{int} \quad (\text{T-LITINT}) \]

2. A variable (or operator) has whatever type it is declared to have.
   \[ \frac{x : T \in \Gamma \quad \Gamma \vdash \_ : T}{\Gamma \vdash \text{let} \ x = e \ \Gamma \vdash x : T} \quad (\text{T-LET}) \]

3. The type of the condition in a conditional expression must be \( \text{bool} \).
   \[ \frac{e_1 : \text{bool} \quad e_2 : T \quad e_3 : T}{\Gamma \vdash \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 : T} \quad (\text{T-IF}) \]

4. The two branches of a conditional expression must have the same type.
   \[ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 : T} \quad (\text{T-IF}) \]

5. The declared type of a variable must match the type of the defining expression.
   \[ \frac{x : T \in \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \text{let} \ x = e \ \Gamma \vdash x : T} \quad (\text{T-LET}) \]

6. The type of the condition in a conditional expression must match the type of the defining expression.
   \[ \frac{x : T \in \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \text{if} \ e \ \text{then} \ e_1 \ \text{else} \ e_2 : T} \quad (\text{T-IF}) \]

7. The type of the argument(s) to a unary or binary operator must match the type(s) of the formal parameters of the operator.
   \[ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \Gamma \vdash e_3 : T_3}{\Gamma \vdash e_1 \cdot e_2 = e_3 : T} \quad (\text{T-BINOPAPP}) \]

A suitable environment implementation is given. These operations enforce scope rules 2 and 3.

```haskell
data Env -- Abstract
type VarAttr = (Int, Type)

initEnv :: [(Id, Type)] -> [(UnOp, Type)] -> Either Env String
enterVar :: Id -> Int -> Type -> Either Env String
lookupVar :: Id -> Env -> Either VarAttr String
lookupUO :: UnOp -> Env -> Type
lookupBO :: BinOp -> Env -> Type
```

Exercise (for home):

The original first LTXL scope rule read:

1. The scope of a variable is all subsequent definitions and the body of the let-expression in which the definition of the variable occurs. A variable is not in scope in the RHS of its definition.

Suggest a version of T-LET that corresponds to this rule, and then change the LTXL implementation correspondingly.
Type-Checking Utilities

compatible :: Type -> Type -> Bool
compatible TpUnknown _ = True
compatible _ TpUnknown = True
compatible t1 t2 = t1 == t2

illTypedOpApp :: Type -> Type -> String
illTypedCond :: Type -> String
incompatibleBranches :: Type -> Type -> String
dclMismatch :: Type -> Type -> String

emitErrD :: SrcPos -> String -> D {}/