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:

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

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

```plaintext
e  →  n     \text{literal integer}
    \mid x     \text{variable}
    \mid e \circ e  \text{unary operator app.}
    \mid e \otimes e  \text{binary operator app.}
    \mid \text{if } e \text{ then } e \text{ else } e  \text{ conditional expression}
    \mid \text{let } (T x = e) \ast \text{ in } e  \text{ let-expression}
```

LTXL AST Representation (recap)

```plaintext
type Id = String

data Exp
  = LitInt Int
    \mid Var Id
    \mid UnOpApp UnOp Exp
    \mid BinOpApp BinOp Exp Exp
    \mid If Exp Exp Exp
    \mid Let [(Id, Type, Exp)] Exp
```
**LTXL Types**

LTXL type syntax:

\[
T \rightarrow \text{int} \quad \text{integer type} \\
| \quad \text{bool} \quad \text{boolean type} \\
| \quad (T, T) \quad \text{product (pair)} \\
| \quad T \rightarrow T \quad \text{function}
\]

**LTXL Type Representation**

The following Haskell data type is used to represent LTXL types:

```haskell
data Type = TpUnknown \\
| TpBool \\
| TpInt \\
| TpProd Type Type -- pair \\
| TpArr Type Type -- function \\
deriving Eq
```

**LTXL Operator Types**

Unary LTXL operator types:

- \( \neg : \text{bool} \rightarrow \text{bool} \)
- \(- : \text{int} \rightarrow \text{int} \quad \text{unary minus}\)

Binary LTXL operator types:

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

**LTXL Operator Representation**

```haskell
data UnOp = Not | Neg
data BinOp = Or \\
| And \\
| Less \\
| Equal \\
| Greater \\
| Plus \\
| Minus \\
| Times \\
| Divide
```
**Example: An LTXl Program**

The LTXl example program again:

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

**Representation:**

Let 

- `x`, IntType, LitInt 7
- `y`, IntType, LitInt 5

(`BinOpApp` Plus)

- `(Var "x")`
- `(Var "y")`

(LitInt 7)

**LTXl Typing Rules (1)**

The LTXl expression typing relation is a *ternary* (or *trinary*) *relation*:

Γ ⊢ e : T

Read: expression e has type T in type environment Γ

1. A literal integer has type `int`.

   Γ ⊢ n : int  (T-LITINT)

2. A variable (or operator) has whatever type it is declared to have.

   \[ x : T \in \Gamma \]  (T-VAR)

---

**Exercise: LTXl Typing Rules**

Let us use the rules we have seen thus far to type check the program

\[ x + 3 \]

in the environment:

\[ \Gamma_1 = +: (\text{int, int}) \rightarrow \text{int}, \]

\[ *: (\text{int, int}) \rightarrow \text{int}, \]

\[ x: \text{int} \]

(On whiteboard)
LTXL Typing Rules (3)

5. The type of the condition in a conditional expression must be `bool`.

6. The two branches of a conditional expression must have the same type.

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T
\]

(T-IF)

LTXL Typing Rules (4)

7. The declared type of a variable must match the type of the defining expression.

\[
\begin{align*}
\Gamma &\vdash x : T \\
\Gamma &\vdash e_1 : T
\end{align*}
\]

(T-VAR)

\[
\begin{align*}
\Gamma &\vdash e_1 : T_1 \\
\Gamma, x : T_1 &\vdash e : T
\end{align*}
\]

(T-LET)

All LTXL Typing Rules

\[
\begin{align*}
\Gamma &\vdash n : \text{int} \quad \text{(T-LITINT)} \\
\Gamma &\vdash x : T \quad \text{(T-VAR)} \\
\Gamma &\vdash e_1 : T_1 \\
\Gamma &\vdash e_2 : T_2 \\
\Gamma &\vdash e_3 : T_3
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{bool} \\
\Gamma &\vdash e_2 : T \\
\Gamma &\vdash e_3 : T
\end{align*}
\]

(T-LET)

\[
\begin{align*}
\Gamma &\vdash e_1 : T_1 \\
\Gamma, x : T_1 &\vdash e : T
\end{align*}
\]

Modified LTXL Scope Rules

1. The scope of a variable is **only** the body of the `let`-expression in which the definition of the variable occurs. (Implied by T-LET.)

2. A definition of a variable hides, for the extent of its scope, any definition of a variable with the same name from an outer `let`-expression.

3. At most one definition may be given for a variable in the list of definitions of a `let`-expression.
LTXL Type Environment

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

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

initEnv :: [(Id, Type)] -> [(UnOp, Type)]
        -> [(BinOp, Type)] -> Env
enterVar :: Id -> Int -> Type -> Env
        -> 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

```haskell
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
declMismatch :: Type -> Type -> String

emitErrD :: SrcPos -> String -> D ()
```