G53CMP: Lecture 11

Contextual Analysis: Implementing a Type Checker

Henrik Nilsson

University of Nottingham, UK
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:

\[
e \rightarrow n \quad \text{ literal integer}
\]

\[
| \quad x \quad \text{ variable}
\]

\[
| \quad \ominus e \quad \text{ unary operator app.}
\]

\[
| \quad e \otimes e \quad \text{ binary operator app.}
\]

\[
| \quad \text{if } e \text{ then } e \text{ else } e \quad \text{ conditional expression}
\]

\[
| \quad \text{let } (T x = e)^* \text{ in } e \quad \text{ let-expression}
\]
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:

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

\ : bool → bool
- : int → int \hspace{1cm} \textit{unary minus}

Binary LTXL operator types:

||, && : (bool, bool) → bool
<, ==, > : (int, int) → bool
+, -, *, / : (int, int) → int
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:

```plaintext
Let [("x", IntType, LitInt 7),
    ("y", IntType, LitInt 5)]
(BinOpApp Plus
  (BinOpApp Times
    (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 \( \Gamma \)
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 \texttt{int}. 
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 \texttt{int}.

\[ \Gamma \vdash n : \texttt{int} \quad \text{(T-LITINT)} \]
The LTXL expression typing relation is a \textit{ternary} (or \textit{trinary}) relation:

\[
\Gamma \vdash e : T
\]

Read: expression $e$ has type $T$ in type environment $\Gamma$

1. A literal integer has type \texttt{int}.

\[
\Gamma \vdash n : \texttt{int} \quad (\text{T-LITINT})
\]

2. A variable (or operator) has whatever type it is declared to have.
The LTXL expression typing relation is a \textit{ternary} (or \textit{trinary}) \textit{relation}:

\[ \Gamma \vdash e : T \]

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

1. A literal integer has type \texttt{int}.

\[ \Gamma \vdash n : \texttt{int} \quad (T\text{-}LIT\text{INT}) \]

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

\[ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (T\text{-}VAR) \]
3. The types of the argument(s) to a unary or binary operator must match the type(s) of the formal parameters of the operator.
3. The types of the argument(s) to a unary or binary operator must match the type(s) of the formal parameters of the operator.

4. The result type of a unary or binary operator application is the result type of the operator.
3. The types of the argument(s) to a unary or binary operator must match the type(s) of the formal parameters of the operator.

4. The result type of a unary or binary operator application is the result type of the operator.

\[
\frac{\Gamma \vdash \ominus : T_1 \rightarrow T_2 \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \ominus e_1 : T_2} \quad \text{(T-UNOPAPP)}
\]

\[
\frac{\Gamma \vdash \otimes : (T_1, T_2) \rightarrow T_3 \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \otimes e_2 : T_3} \quad \text{(T-BINOPAPP)}
\]
Exercise: LTXL Typing Rules

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

\[ x + 3 \]

in the environment:

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

(On whiteboard)
5. The type of the condition in a conditional expression must be \texttt{bool}.
5. The type of the condition in a conditional expression must be \texttt{bool}.

6. The two branches of a conditional expression must have the same type.
5. The type of the condition in a conditional expression must be \texttt{bool}.

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

\[
\begin{align*}
\Gamma \vdash e_1 : \texttt{bool} & \quad \Gamma \vdash e_2 : T & \quad \Gamma \vdash e_3 : T \\
\Gamma \vdash \texttt{if} \, e_1 \, \texttt{then} \, e_2 \, \texttt{else} \, e_3 : T
\end{align*}
\]  
\text{(T-IF)}
7. The declared type of a variable must match the type of the defining expression.
7. The declared type of a variable must match the type of the defining expression.

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\]

\[
\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e : T}{\Gamma \vdash \text{let } T_1 \ x = e_1 \ \text{in} \ e : T} \quad \text{(T-LET)}
\]
All LTXL Typing Rules

\[ \Gamma \vdash n : \text{int} \quad (T\text{-LITINT}) \]

\[ x : T \in \Gamma \]
\[ \Gamma \vdash x : T \quad (T\text{-VAR}) \]

\[ \Gamma \vdash \ominus : T_1 \rightarrow T_2 \]
\[ \Gamma \vdash e_1 : T_1 \]
\[ \Gamma \vdash \ominus e_1 : T_2 \quad (T\text{-UNOPAPP}) \]

\[ \Gamma \vdash \otimes : (T_1 , T_2) \rightarrow T_3 \]
\[ \Gamma \vdash e_1 : T_1 \]
\[ \Gamma \vdash e_2 : T_2 \]
\[ \Gamma \vdash e_1 \otimes e_2 : T_3 \quad (T\text{-BINOPAPP}) \]

\[ \Gamma \vdash e_1 : \text{bool} \quad (T\text{-IF}) \]
\[ \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T \]
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T \]

\[ \Gamma \vdash e_1 : \overline{T_1} \quad \Gamma, x : \overline{T_1} \vdash e : T \]
\[ \Gamma \vdash \text{let } \overline{T_1} x = e_1 \text{ in } e : T \quad (T\text{-LET}) \]
Modified LTXL Scope Rules

1. The scope of a variable is *only* the body of the \texttt{let}-expression in which the definition of the variable occurs. (Implied by T-LET.)
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.
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.
A suitable environment implementation is given. These operations enforce scope rules 2 and 3.

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 LTXML 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 LTXML 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
declMismatch :: Type -> Type -> String

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