COMP3048: Lecture 7

Contextual Analysis: Scope II

Matthew Pike

University of Nottingham, Ningbo

This Lecture

An Illustrative Identification Algorithm in Haskell

- LTXL Syntax and Semantics, particularly scope rules.
- Abstract syntax representation
- Environment/Symbol Table representation and operations.
- The Identification Algorithm

Recap: Identification

Identification is the task of relating each applied identifier occurrence to its declaration or definition:

```
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of `set`, the one applied occurrence of

- `x` refers to the ***instance variable*** `x`
- `n` refers to the ***argument*** `n`.

Identification for LTXL

We are now going to study a concrete Haskell implementation of identification for ***LTXL***:

Less Trival eXpression Language

- LTXL $\approx$ TXL $+$ typed definitions $+$ `if`-expression $+$ new operators
- Slides only show highlights: complete code available on-line.

LTXL CFG (1)

$$
\begin{array}{lcl}
\mathit{LTXLProgram} & \rightarrow & \mathit{Exp} \\
\\
\mathit{Exp} & \rightarrow & \mathit{Exp} \; \texttt{||} \; \mathit{Exp} \quad | \quad \mathit{Exp} \; \texttt{\&\&} \; \mathit{Exp} \\
& | & \mathit{Exp} \; \texttt{<} \; \mathit{Exp} \quad | \quad \mathit{Exp} \; \texttt{==} \; \mathit{Exp} \quad | \quad \mathit{Exp} \; \texttt{>} \; \mathit{Exp} \\
& | & \mathit{Exp} \; \texttt{+} \; \mathit{Exp} \quad | \quad \mathit{Exp} \; \texttt{-} \; \mathit{Exp} \\
& | & \mathit{Exp} \; \texttt{*} \; \mathit{Exp} \quad | \quad \mathit{Exp} \; \texttt{/} \; \mathit{Exp} \\
& | & \mathit{PrimaryExp}
\end{array}
$$

LTXL CFG (2)

Operator precedence is used to disambiguate.
In increasing order of precedence:

1. `||`
2. `&&`
3. `<`, `==`, `>`
4. `+`, `-`
5. `*`, `/`

LTXL CFG (3)

$$
\begin{array}{lcl}
\mathit{PrimaryExp} & \rightarrow & \underline{\mathit{LitInt}} \\
& | & \underline{\mathit{Ident}} \\
& | & \texttt{\textbackslash}\ \mathit{PrimaryExp} \\
& | & \texttt{-}\ \mathit{PrimaryExp} \\
& | & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
& | & \texttt{(}\ \mathit{Exp}\ \texttt{)} \\
& | & \texttt{let}\ \mathit{Defs}\ \texttt{in}\ \mathit{Exp}
\end{array}
$$

LTXL CFG (4)

$$
\begin{array}{lcl}
\mathit{Defs} & \rightarrow & \mathit{Def}\ \texttt{;}\ \mathit{Defs} \\
& | & \mathit{Def} \\
\\
\mathit{Def} & \rightarrow & \mathit{Type}\ \underline{\mathit{Ident}}\ \texttt{=}\ \mathit{Exp} \\
\\
\mathit{Type} & \rightarrow & \texttt{int} \\
& | & \texttt{bool}
\end{array}
$$

LTXL Example 1

```
let
    int a = 10;
    bool b = a < 2
in let
    int c = a * 10;
    bool a = a == 42;
    int d = if a then 1 else 2
in
    if a && b then c else 42
```

LTXL Scope Rules

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 own definition.
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 Example 1 (again)

Which scope rules are used where?

```
let
    int a = 10;
    bool b = a < 2
in let
    int c = a * 10;
    bool a = a == 42;
    int d = if a then 1 else 2
in
    if a && b then c else 42
```

LTXL Example 2

What about this LTXL example?

```
let
    int a = 1;
    int b = c * 2;
    bool a = a < 1
in
    a + b
```

LTXL AST (1)

The following Haskell data types are used to represent LTXL programs.

```
type Id = String

data Type = IntType
          | BoolType
          | UnknownType
```

LTXL AST (2)

```
data UnOp = Not | Neg

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

LTXL AST (3)

`Exp` is a ***parameterized*** type. The ***type parameter*** `a` allows variables to be ***annotated*** with an attribute of type `a`. This facility is used by the identification function.

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

LTXL AST (4)

Example: The LTXL program

```
let int x = 7 in x + 35
```

would be represented like this ***before*** identification (type `Exp ()`):

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" ())
              (LitInt 35))
```

(***After*** identification, type will be `Exp Attr`.)

LTXL Environment (1)

- An ***association list*** is used to represent the environment/symbol table to keep things simple.
- By ***prepending*** new declarations to the list, and searching from the beginning, we will always find an identifier in the closest containing scope. For example:

  $\texttt{lookup "x" [("x",}a_1\texttt{),("y",}a_2\texttt{),("x",}a_3\texttt{)]}$

  $\Rightarrow\ a_1$

- No need for a "close scope" operation. We are in a pure functional setting $\Rightarrow$ persistent data.

LTXL Environment (2)

The environment associates identifiers with ***variable attributes***. Our attributes are the ***scope level*** and the ***declared type***.

```
type Attr = (Int, Type)
```

The environment is just an association list:

```
type Env = [(Id, Attr)]
```

Note: our environment does ***not*** store variable ***definitions***.

LTXL Environment (3)

Example:

```
let
    int a = 10;              (1)
    int b = a + 42
in let
    bool a = b < 20          (2)
in
    if a then b else 13
```

Env. after (1): `[("a",(1,IntType))]`
Env. after (2): `[("a",(2,BoolType)), ("b",(1,IntType)), ("a",(1,IntType))]`

LTXL Environment (4)

`enterVar` inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the ***resulting environment*** is returned.
- Otherwise an ***error message*** is returned.

```
enterVar :: Id -> Int -> Type -> Env
            -> Either Env ErrorMsg
```

Aside: The Haskell Type `Either`

The standard Haskell type `Either` comes in handy when one needs to represent a value that has one of two possible types:

```
data Either a b = Left a | Right b
```

A typical example is when a function needs to return one of two kinds of results:

```
foo :: Int -> Either Bool String
foo x | x < 100    = Left (x < 0)
      | otherwise = Right "Too big"
```

LTXL Environment (5)

```
enterVar i l t env
  | not (isDefined i l env)
    = Left ((i,(l,t)) : env)
  | otherwise
    = Right (i ++ " already defined.")
  where
    isDefined i l [] = False
    isDefined i l ((i',(l',_)) : env)
      | l < l' = error "Should not happen!"
      | l > l' = False
      | i == i' = True
      | otherwise = isDefined i l env
```

LTXL Environment (6)

Let

```
env = [("y", (2,IntType)),
       ("x", (1,IntType))]
```

Then:

```
enterVar "x" 2 BoolType env
⇒ Left [("x", (2,BoolType)),
        ("y", (2,IntType)),
        ("x", (1,IntType))]

enterVar "y" 2 BoolType env
⇒ Right "y already defined."
```

LTXL Environment (7)

`lookupVar` looks up a variable in an environment.

- Returns ***variable attributes*** if found.
- Returns an ***error message*** otherwise.

```
lookupVar :: Id -> Env
             -> Either  Attr  ErrorMsg
lookupVar i [] = Right (i ++ " not defined.")
lookupVar i ((i',a) : env)
      | i == i'    = Left a
      | otherwise = lookupVar i env
```

LTXL Environment (8)

Let

```
env = [("x", (2,BoolType)),
       ("y", (2,IntType)),
       ("x", (1,IntType))]
```

Then:

```
lookupVar "y" env
⇒ Left (2,IntType))
lookupVar "x" env
⇒ Left (2,BoolType))
lookupVar "z" env
⇒ Right "z not defined."
```

LTXL Identification (1)

Goals of LTXL identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.
  I.e., map unannotated AST **`Exp ()`** to annotated AST **`Exp Attr`**.
- Report conflicting variable definitions and undefined variables.

```
identification ::
    Exp () -> (Exp Attr,  [ErrorMsg])
```

LTXL Identification (2)

Example: Before Identification

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" ())
              (LitInt 35))
```

After identification:

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" (1, IntType))
              (LitInt 35))
```

LTXL Identification (3)

Main identification function:

```
identification :: Exp ()
                    -> (Exp Attr, [ErrorMsg])
identification e = identAux 0 emptyEnv e
```

Type signature for auxiliary identification function:

```
identAux :: Int -> Env -> Exp ()
            -> (Exp Attr, [ErrorMsg])
```

LTXL Identification (4)

Variable case:

```
identAux l env (Var i _) =
  case lookupVar i env of
    Left a  -> (Var i a, [])
    Right m -> (Var i (0, UnknownType), [m])
```

LTXL Identification (5)

Binary operator application (typical recursive case):

```
identAux l env (BinOpApp op e1 e2) =
  (BinOpApp op e1' e2', ms1 ++ ms2)
  where
    (e1', ms1) = identAux l env e1
    (e2', ms2) = identAux l env e2
```

LTXL Identification (6)

Reminder: LTXL scope rules

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.
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 Identification (7)

Block of definitions (`let`):

```
identAux l env (Let ds e) =
  (Let ds' e', ms1 ++ ms2)
  where


      (e', ms2)          = identAux l' env' e
```

LTXL Identification (8)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
      case enterVar i l t env of
        Left env' -> (env', [])
        Right m   -> (env, [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```

`i` ***not*** in scope (rule 1)

impl./checks rules 2 & 3

`i` in scope (rule 1)

Efficient Symbol Table Implementation

Lists don't make for very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where n is the number of symbols.

Some more efficient options:

- Balanced trees:
  - Insertion and lookup are both $O(\log n)$.
  - One way of handling nested scopes would be a stack of trees.

Efficient Symbol Table Implementation

- Hash tables:
  - Insertion and lookup are both $O(1)$ as long as the ratio between the number of symbols and the hash table size is kept below a small constant factor.
  - Algorithms such as ***linear hashing*** allows the table to grow and shrink gracefully, guaranteeing near optimal performance.

See e.g. Aho, Sethi, Ullman (1986) for further details.