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:

```java
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:

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

LTXL CFG (1)

```
LTXLPogram → Exp
    | PrimaryExp
```

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

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

LTXL CFG (2)

```
PrimaryExp → LitInt | Ident | \ PrimaryExp | ~ PrimaryExp
            | if Exp then Exp else Exp | ( Exp )
            | let Defs in Exp
```

```
Defs → Def ; Defs
    | Def
Def → Type Ident = Exp
Type → int | bool
```

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?

```plaintext
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?

```plaintext
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.

```haskell
type Id = String

data Type = IntType | BoolType | UnknownType
```

---

**LTXL AST (2)**

```haskell
data UnOp = Not | Neg
data BinOp = Or | And | Less | Equal | Greater | Plus | Minus | Times | Divide
```

---

**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.

```haskell
data Exp a = LitInt Int | Var Id | 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 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:
  ```plaintext
  lookup "x", [("x", a1), ("y", a2), ("z", a3)] => a1;
  ```
- No need for a "close scope" operation. We are in a pure functional setting => persistent data.

---

**LTXL Environment (2)**

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

```haskell
type Attr = (Int, Type)
type Env = [(Id, Attr)]
```

The environment is just an association list:

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

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

```plaintext
let
  int a = 10;
  int b = a + 42
in let
  bool a = b < 20
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))]
```

**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:

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

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

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

**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 the 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
```

**LTXL Environment (5)**

```plaintext
let
  int a = 10;
  int b = a + 42
in let
  bool a = b < 20
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 (6)**

```plaintext
let
  env = [("y", (2, IntType)), ("x", (1, IntType))]
Then:

\[
\begin{align*}
\text{lookupVar } "y" \text{ env} & \Rightarrow \text{Left } (2, \text{IntType}) \\
\text{lookupVar } "x" \text{ env} & \Rightarrow \text{Left } (1, \text{IntType})
\end{align*}
\]
```

**LTXL Environment (7)**

**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
                   -> Either Attr ErrorMsg
```

Example: Before Identification

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

After identification:

```plaintext
Let ["x", IntType, LitInt 7]
(BinOpApp Plus (Var "x" (1, IntType))(LitInt 35))
```
Main identification function:

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

Type signature for auxiliary identification function:

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

Variable case:

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

Block of definitions (let):

```haskell
identAux l env (Let ds e) =
  (Let ds' e', ms1 ++ ms2)
  where
    l' = l + 1
    (ds', env', ms1) = identDefs l' env ds
    (e', ms2) = identAux l' env' e
```

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.

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.
- 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, guarantees near optimal performance.

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