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*:

*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)

\[
\begin{align*}
\text{LTXLProgram} & \rightarrow \text{Exp} \\
\text{Exp} & \rightarrow \text{Exp} \; \| \; \text{Exp} \; \&\& \; \text{Exp} \\
& \quad | \; \text{Exp} < \text{Exp} \; | \; \text{Exp} =\,\,= \; | \; \text{Exp} > \; | \; \text{Exp} \\
& \quad | \; \text{Exp} \; + \; \text{Exp} \; | \; \text{Exp} \; - \; \text{Exp} \\
& \quad | \; \text{Exp} \; \times \; \text{Exp} \; | \; \text{Exp} \; / \; \text{Exp} \\
& \quad | \; \text{PrimaryExp} 
\end{align*}
\]

LTXL CFG (2)

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

1. \(\|\)
2. \(\&\&\)
3. \(<,\; =\,\,=,\; >\)
4. \(+,\; -\)
5. \(*,\; /\)

LTXL CFG (3)

\[
\begin{align*}
\text{PrimaryExp} & \rightarrow \text{LitInt} \\
& \quad | \; \text{Ident} \\
& \quad | \; \backslash \; \text{PrimaryExp} \\
& \quad | \; - \; \text{PrimaryExp} \\
& \quad | \; \text{if} \; \text{Exp} \; \text{then} \; \text{Exp} \; \text{else} \; \text{Exp} \\
& \quad | \; ( \; \text{Exp} \; ) \\
& \quad | \; \text{let} \; \text{Defs} \; \text{in} \; \text{Exp} 
\end{align*}
\]

LTXL CFG (4)

\[
\begin{align*}
\text{Defs} & \rightarrow \text{Def} \; ; \; \text{Defs} \\
& \quad | \; \text{Def} \\
\text{Def} & \rightarrow \text{Type} \; \text{Ident} \; = \; \text{Exp} \\
\text{Type} & \rightarrow \text{int} \\
& \quad | \; \text{bool}
\end{align*}
\]
**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 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 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 2**

What about this LTXL example?

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

- Not defined
- Defined twice at same scope level
The following Haskell data types are used to represent L TXL programs.

```haskell
type Id = String

data Type = IntType
  | BoolType
  | UnknownType
```

```
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 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)
```

Example: The L TXL program

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

would be represented like this before identification (type \( \text{Exp} () \)):

```haskell
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:

  ```
  lookup "x" ["x", a₁], ["y", a₂], ["x", 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**.

```
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:

```ocaml
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.

```haskell
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:

```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 (5)

```haskell
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
```

decl. prepended

LTXL Environment (6)

Let

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

Then:

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

```haskell
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.

```haskell
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
```

variable attributes
error message
LTXL Environment (8)

Let

\[ \text{env} = \{ ("x", (2, \text{BoolType})), ("y", (2, \text{IntType})), ("x", (1, \text{IntType})) \} \]

Then:

- \( \text{lookupVar "y" env} \Rightarrow \text{Left (2, \text{IntType})} \)
- \( \text{lookupVar "x" env} \Rightarrow \text{Left (2, \text{BoolType})} \)
- \( \text{lookupVar "z" env} \Rightarrow \text{Right "z not defined."} \)

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 \( \text{Exp ()} \) to annotated AST \( \text{Exp Attr} \).
- Report conflicting variable definitions and undefined variables.

\[
\text{identification :: Exp () -> (Exp Attr, [ErrorMsg])}
\]

LTXL Identification (2)

Example: Before Identification

Let \( \{ ("x", \text{IntType, LitInt 7}) \} \)
(\( \text{BinOpApp Plus} \))
(Var "x" ())(LitInt 35))

After identification:

Let \( \{ ("x", \text{IntType, LitInt 7}) \} \)
(\( \text{BinOpApp Plus} \))
(Var "x" (1, \text{IntType}))
(LitInt 35))

LTXL Identification (3)

Main identification function:

\[
\text{identification :: Exp () -> (Exp Attr, [ErrorMsg])}
\]

Type signature for auxiliary identification function:

\[
\text{identAux :: Int -> Env -> Exp () -> (Exp Attr, [ErrorMsg])}
\]
LTXL Identification (4)

Variable case:

```plaintext
defaultAux 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):

```plaintext
defaultAux l env (BinOpApp op e1 e2) =
  (BinOpApp op e1’ e2’, ms1 ++ ms2)
  where
    (e1’, ms1) = defaultAux l env e1
    (e2’, ms2) = defaultAux 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`):

```plaintext
defaultAux l env (Let ds e) =
  (Let ds’ e’, ms1 ++ ms2)
  where
    l’ = l + 1
    (ds’, env’, ms1) = defaultDefs l’ env ds
    (e’, ms2) = defaultAux l’ env’ e
```

Note that `defaultDefs` returns an
**updated environment** to be used when
checking the body of the `let` (rule 1).
**LTXL Identification (8)**

\[
\begin{align*}
\text{identDefs } l \ env \ [\] &= ([], \ env, []) \\
\text{identDefs } l \ env \ ((i, t, e): ds) &= \\
((i, t, e'): ds', \ env'', \ ms1++ms2++ms3) \\
\text{where } i \not\in \text{scope (rule 1)} \\
(e', ms1) &= \text{identAux } l \ env \ e \\
(\text{env'}, ms2) &= \impl/\text{checks rules 2 & 3} \\
\text{case enterVar } i \ l \ t \ env\ of \\
\quad \text{Left } \text{env}' \to (\text{env}', []) \\
\quad \text{Right } m \to (\text{env}, [m]) \\
(\text{ds'}, \text{env}'', \text{ms3}) &= i \in \text{scope (rule 1)} \\
\text{identDefs } l \ \text{env}' \ ds
\end{align*}
\]

**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, guaranteeing near optimal performance.

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