Free and Bound Variable Identifiers, Desugaring, in Oz

A Grammar for Oz with Sugars

Consider the grammar in Figure 1, which describe data structures (records) called abstract syntax trees. These are to be thought of as representing in Oz data the statements and expressions of Oz itself. Thus the grammar in the figure represents the abstract syntax of a sugared version of Oz. For example the following Oz statement:

```
A = B
```

is represented by the record structure:

```
assignStmt("A" varIdExp("B"))
```

Note that the grammar uses Oz strings to represent variable identifiers. A more complex example that illustrates more features of the representation is given in Figure 2 on the next page.

⟨Statement⟩ ::= `skipStmt`
| `seqStmt(`⟨List ⟨Statement⟩⟩`)`
| `localStmt(`⟨String⟩ ⟨Statement⟩`)`
| `assignStmt(`⟨String⟩ ⟨Expression⟩`)`
| `ifStmt(`⟨Expression⟩ ⟨Statement⟩ ⟨Statement⟩`)`
| `caseStmt(`⟨Expression⟩ ⟨Pattern⟩ ⟨Statement⟩ ⟨Statement⟩`)`
| `applyStmt(`⟨Expression⟩ ⟨List ⟨Expression⟩⟩`)`
| `namedFunStmt(`⟨String⟩ ⟨List ⟨Pattern⟩⟩ ⟨Expression⟩`)`
| `inStmt(`⟨Pattern⟩ ⟨Expression⟩ ⟨Statement⟩`)`
| `threadStmt(`⟨Statement⟩`)`

⟨Expression⟩ ::= `varIdExp(`⟨String⟩`)`
| `atomExp(`⟨Atom⟩`)` | `boolExp(`⟨Bool⟩`)`
| `numExp(`⟨Number⟩`)`
| `recordExp(`⟨Expression⟩ ⟨List ⟨Field⟩⟩`)`
| `procExp(`⟨List ⟨Pattern⟩⟩ ⟨Statement⟩`)`
| `ifExp(`⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩`)`
| `caseExp(`⟨Expression⟩ ⟨Pattern⟩ ⟨Expression⟩ ⟨Expression⟩`)`
| `applyExp(`⟨Expression⟩ ⟨List ⟨Expression⟩⟩`)`
| `threadExp(`⟨Expression⟩`)`

⟨Pattern⟩ ::= `varIdPat(`⟨String⟩`)`
| `atomPat(`⟨Atom⟩`)` | `boolPat(`⟨Bool⟩`)`
| `recordPat(`⟨Atom⟩ ⟨List ⟨Field⟩⟩`)`

⟨Field⟩ ::= `colonFld(`⟨Atom⟩ ⟨Expression⟩`)`
| `posFld(`⟨Exp⟩`)`

Figure 1: Grammar for a simplified subset of the practical version of the declarative language of chapters 3 and 4, based on the textbook's section 2.6 [VH04]. The type ⟨String⟩ is the type of character strings in Oz, ⟨Atom⟩ is the type of atoms in Oz, etc.

The following Oz statement:

```
fun {AddToEach A#B Ls}
   case Ls of
     (X#Y)|T then ({Plus A X}#{Plus B Y})|{AddToEach A#B T}
   else nil
   end
end
```

is represented by the following record structure:

```
namedFunStmt("AddToEach"
             [recordPat('#'
                        [posFld(varIdExp("A")) posFld(varIdExp("B"))])
              varIdPat("Ls")]
   caseExp(varIdExp("Ls")
           recordPat('|'
                     [posFld(recordExp(atomExp('#')
                                       [posFld(varIdExp("X"))
                                        posFld(varIdExp("Y"))]))
                      posFld(varIdExp("T"))])
           recordExp(atomExp('|')
                     [posFld(recordExp(atomExp('#')
                                       [posFld(applyExp(varIdExp("Plus")
                                                        [varIdExp("A")
                                                         varIdExp("X")]))
                                        posFld(applyExp(varIdExp("Plus")
                                                        [varIdExp("B")
                                                         varIdExp("Y")]))]))
                      posFld(applyExp(varIdExp("AddToEach")
                                      [recordExp(atomExp('#')
                                                 [posFld(varIdExp("A"))
                                                  posFld(varIdExp("B"))])
                                       varIdExp("T")]))])
           atomExp(nil)))
```

Figure 2: Example showing how Oz is parsed into the record structures (abstract syntax trees) from Figure 1 on the previous page.

The Main Functions

The interpreter provides the following main functions to users.

1. The function

   ```
   FreeVarIds: <fun {$ <Statement>}: <Set <String>>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 1 on page 1 and returns a set of all the variable identifiers that occur free in its argument. Its implementation is in `FreeVarIds.oz`. There are tests in the file `FreeVarIdsTest.oz`.

   See the file `TestOutput.txt` for results of testing.

2. The function

   ```
   BoundVarIds: <fun {$ <Statement>}: <Set <String>>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 1 on page 1 and returns a set of all the variable identifiers that occur bound in its argument. Its implementation is in `BoundVarIds.oz`. There are tests in the file `BoundVarIdsTest.oz`.

   See the file `TestOutput.txt` for results of testing.

3. The function

   ```
   Desugar: <fun {$ <Statement>}: <Statement>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 1 on page 1 and returns a ⟨Statement⟩ in the subset of that grammar that represents Oz statements in the kernel language of the textbook [VH04, Section 2.3].

   An implementation is in `Desugar.oz`. There are tests in the file `DesugarTest.oz`.

   See the file `TestOutput.txt` for results of testing.

4. The function

   ```
   Reduce1: <fun {$ <Config>}: <Pair String Config>>
   ```

   takes a non-terminal configuration and returns a new rule name and a configuration for the next step in the operational semantics of Oz, starting at the given configuration. See `Configuration.oz` for details on configurations.

References

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.