Types & Programming Languages
Notes on the SFL (Simple Functional Language) Typechecker

The SFL typechecker is derived from the SEL typechecker. The new features are the use of environments and the code for typechecking function definitions and function applications.

Location

The SFL typechecker can be obtained from `http://www.dcs.gla.ac.uk/~simon/teaching/tpl`

Grammar Specification

The file `sfl.grm` specifies the grammar of SFL in a form suitable for SableCC. Notice that a function definition must be followed by a semicolon, to avoid problems of ambiguity.

Syntax Tree Classes

These are the syntax tree classes from SEL, with new classes for function definitions and function applications.

Environments

Environments are implemented by the files in `env`. An environment is a lookup table indexed by strings and containing two kinds of entry: `VarEntry` and `FunEntry`, which are extensions of `EnvEntry`. The obvious idea for implementing an environment is to use a `Hashtable`. However, we have to allow for nested scopes and the possibility that an identifier may exist in both an outer and an inner scope. In SFL there is only one level of nesting: function definitions are global, and the body of each function has its own nested scope. SFL does not have nested function definitions or nesting of scopes within a single function body, but later we will be interested in languages with scope nesting of arbitrary depth. There are various ways of implementing environments with nested scopes, but one of the simplest (which we use here) is for an environment to be a stack of hash tables, and for lookup to search down the stack if necessary. If an identifier is not found in the innermost scope (top of the stack) then the next outer scope is searched, and so on. You can see this in `env/Env.java`, which uses a `Vector` to implement a stack (not for any very good reason; it could have been done with a `Stack` instead).

Typechecker

In `checker/Checker.java` the method `outAAppTerm` checks function applications. To check function definitions, we redefine the method `caseAFunDef`, because we only want to visit a function body if there are no errors in the header.

Unlike SEL, SFL has syntax for types, so there are now two representations of types: the classes extending `ATy` in the syntax tree representation, and the classes extending `Type` defined in `sfl/types`. We have to convert one to the other, and this is done by the methods `outAIntTy` and `outABoolTy`. These methods work with the same hash table as the rest of the typechecking methods: as well as storing the type of each expression, the `out` table now stores the `Type` representation of each `ATy`.

Dealing with mutual recursion introduces a complication. The depth first traversal of the syntax tree corresponds to reading once through the program from beginning to end, so each function application must be preceded by a definition of the function. This rules out mutual recursion because if two functions call each other, one of the calls must precede the corresponding definition. There are several possible ways of dealing with this problem.

1. Allow a function to be declared separately from its definition. Pascal adopted this approach, using the mechanism of *forward* function declarations, for example:

   ```
   function f(x:int):int; forward;

   function g(x:int):int
   begin
     g := f(x+1);
   end;

   function f(x:int):int
   begin
     f := g(x);
   end;
   ```

   With this approach, typechecking can still be done in a single pass over the syntax tree.

2. Introduce another traversal of the syntax tree, which gathers all the function headers into an environment which is used by the second traversal to check all the function bodies. This means that all the functions in the program are allowed to be mutually recursive.

3. An intermediate approach requires groups of mutually recursive functions to be explicitly declared. Standard ML adopts this approach, using the keyword `and`, for example:

   ```
   fun f(x:int) = g(x)

   and g(x:int) = f(x+1)
   ```

   This can be implemented with a single traversal of the syntax tree, but requires two passes over each collection of mutually recursive functions.

The SFL typechecker uses approach (2). The first traversal of the syntax tree is defined by `checker/Mutual` and the second traversal is defined by `checker/Checker`. In `Mutual`, only `caseAFunDef` is defined, and the only purpose of this traversal is to build an environment of function declarations and detect any errors in the declarations. `Checker` uses this environment, and checks the function bodies and the rest of the code.

Error Reporting

There are some new error messages relating to function definitions and function applications.

Top-level Driver

Mutual recursion is optional: calling the checker as `check -mutual file.sfl` invokes both passes over the syntax tree and allows mutually recursive functions, whereas `check file.sfl` does not do the initial pass and therefore does not allow mutually recursive functions.