Types & Programming Languages
Notes on the BFL (Better Functional Language) Typechecker

The BFL typechecker is derived from the SFL typechecker. The new features are function types and function values, and `let` declarations.

Location

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

Grammar Specification

The file `bfl.grm` specifies the grammar of BFL in a form suitable for SableCC. A program is now just an expression, as we can use `let` to define functions. Remember that brackets must be used within expressions, for example `(1+2)+3`. The same is true for structured types: we must write `int->(int->int)` and cannot write `int->int->int`. The syntax `fn x:T => e` is used for $\lambda x : T.e$. The syntax for function application does not include brackets: if `f` is a function of type (say) `int->int` then we can simply write `f 2`. However, brackets are needed around a function expression before it can be applied: for example, `(fn x:int => x+1)2`.

An example program:

```
let val a = 1
    fun f(x:int):int->int =
           fn y:int => (x+y)+a
in  (f 2) 3
end
```

The `fun` syntax for function definition only allows one argument to be defined. Curried functions of two or more arguments can be defined in a similar way to the above example.

Syntax Tree Classes

These are generated by SableCC in the usual way.

Environments

The implementation of environments is similar to environments for SFL. The difference is that we now have just one kind of environment entry: now that we have function types, there is no need to distinguish between function entries and variable entries. The environment just stores identifiers with their types.

Note that it was wrong to say that a Java `Hashtable` cannot be used: I have since realised that using `String` keys in a `Hashtable` *does* result in lookup by string value. So the definition of `Env` in both SFL and BFL has been changed and now uses `Hashtable`s. It is still necessary to use a stack of `Hashtable`s in order to deal with nested scopes.

Typechecker

The typechecker is slightly simpler than the SFL typechecker because we no longer allow direct definition of multi-argument functions; this makes typechecking function applications simpler too. For further simplification, mutually recursive function definitions are not allowed, but could be supported by processing all of the function definitions in each `let` in two passes.

There are three cases in which it is necessary to override the `case` method of `DepthFirstAdapter`: `fn` expressions, `fun` definitions, and `let` expressions. These are the cases in which a new scope must be opened in the environment *before* adding new environment entries and checking an expression. It might be possible instead to use the `in` method to open the new scope, use `outATypedArg` etc. to add the environment entries, and use `out` to close the scope; this would avoid overriding `case`, but I find it clearer to keep the code together in one method.

Error Reporting

The error messages are slightly different: some have disappeared and some have changed. For example: it is no longer an error to use a function name without applying it to parameters; typechecking a function application involves finding a function type and matching its argument type with the type of the actual parameter.

Top-level Driver

This is more like the top-level driver from SEL.