Extensions to MiniML

Bennett Parsons

April 27, 2016

There are quite a number of enhancements I made to the MiniML language, including an extension to the `eval_d` functionality, implementation of lexical scoping, the centralization of environment evaluator code, extended semantics and comprehensive, elegant unit tests. They will be discussed in detail here.

While testing, I found that my implementation of the `eval_d` function, which evaluated expressions using a dynamically scoped environment, failed to evaluated expressions with a double application of functions (ie a function applied to a function applied to a Bool and Num literal). I found this to not be representative of a truly dynamically scoped environment model, so I made the following modifications to fix this. I made my recursive helper function, `eval`, return the current environment, in addition the Value of the evaluated expression, so that this environment could be used during function application. Thus, the match case for App looks like this:

```
| App (app_exp, arg_exp) ->
    match eval app_exp env with
    | ...
    | Val (Fun (v, exp1)), env0 ->
```

The variable `env0` represents the environment that is returned from evaluation of the `app_exp`. This is the environment that should be used during evaluation of the application (otherwise the information from the evaluation is lost). The previous implementation used the current environment (specified by the initial argument passed into `eval`), at the time of application, to evaluate then expressions, rather than `env0`. Note that the original implementation of `eval_d` is still available as itself; the extended version has been packaged into a new function, `eval_d_ext`.

Another extension to the language was the implementation of lexical scoping: `eval_s`. This followed generally from the use of closures, binding functions to their lexical environment. Thus, during function application, functions were evaluated based on their lexical, closed environment. This provided an elegant solution to the aforementioned issue with the initial dynamically scoped evaluator (while of course changing the inherent evaluation semantics).

Note also that the functionality called by `eval_d`, `eval_d_ext` and `eval_l`, is all packaged into a single function `eval_all` that takes in the same expression and environment as each evaluator as well as two Boolean flags that let. The first

flag differentiates between dynamic and lexical scoping (true giving dynamic and false giving lexical), while the second indicates the extension to `eval_d` (true gives the extension and false preserves the original). This allows for the `eval_d`, `eval_d_ext` and `eval_l` function definitions to be quite simple and elegant:

```
let eval_d exp env = (eval_all exp env true false) ;;
let eval_d_ext exp env = (eval_all exp env true true) ;;
let eval_l exp env = (eval_all exp env false true) ;;
```

Another enhancement I made to MiniML was the extension of the language semantics to include more operators for its Bool and Num literals. The `"not"` keyword was added to the language as Boolean Unary negation. The Boolean binary operators `">"` and `"<>"` were also added, as well as the Num binary operator `"/"`. These additions consisted of adding these tokens to the declarations provided in `miniml_parse.mly`, as well as the tables and character recognition lists in `miniml_lex.mll`. The Binop and Unop match cases in the evaluators (and the corresponding helper functions) were in turn modified to correctly evaluate expressions containing these operators.

The final, and perhaps most time consuming extension, contributes more to the overall style of the project rather than the functionality of MiniML. Using Professor Shieber's skeleton code for tests, I created a testing environment in `stests.ml` that made use of extension automated testing and further abstraction of the literals, expressions, values and environments used by the functions implemented in `expr.ml` and `evaluation.ml`. For instance, testing for the evaluators consisted of two functions: `test_all` and `test_one`. `test_all`, reproduced below, took in the string one would type into MiniMl and the expected string to be returned, and constructed four tests, one for each of the evaluations, ensuring that these tests cases were supported by each implementation.

```
let test_all str_in str_out =
      dec ();
      let str evaler =
        try
          exp_to_string (exp_of_val (evaler (str_to_exp str_in) empty))
        with
        | EvalError s -> s in

      [``Eval_s'' ^ !ctr, lazy (str eval_s = str_out), str eval_s ;
      ``Eval_d'' ^ !ctr, lazy (str eval_d = str_out), str eval_d ;
      ``Eval_d_ext'' ^ !ctr, lazy (str eval_d_ext = str_out), str eval_d_ext
      ``Eval_l'' ^ !ctr, lazy (str eval_l = str_out), str eval_l]
```

The `str` function evaluates the input string against the evaluator, and eventually returns the string representation of the output, hopefully the same as `str_out`! After a master list of these tests is created, and stored in `tests`, all tests can be run with a single call to: `report (make_test tests)`.