MP 9 – MiniOCaml Interpreter — Substitution Model

CS 421 – Spring 2013
Revision 1.2

Assigned Tuesday, March 26, 2012
Due Sunday, March 30, 09:30
Extension (none)

1 Change Log

1.0 Initial Release.

1.1 Fixed second rec rule in figure 3.

1.2 Made the late deadline (Sunday) the on-time deadline. **There is now no late extension.**

1.3 Fixed second let rule in figure 3.

2 Objective

You will write an interpreter in the style of MP6, but for a simplified version of Ocaml. We will provide the lexer, parser, and translation to abstract syntax, so, as in MP 6, your task will be to interpret programs given in abstract syntax. (Actually, you will write two interpreters, this one and another one for MP10. The difference — unlike MP6 and 7 — is not in how many features of the language they implement, but simply in the method of evaluation. This one uses the "substitution model," which we will discuss in class in lecture 18.)

After completing this MP, you will have learned about the following concepts:

- How to implement functional languages using substitution

This MP is not particularly hard, although, as has been the case with most of the MPs, the write-up is complicated. Our solution is about 80 lines of OCaml, and half of it is in functions `applyOp` and `applyUnop`, which are similar to `applyOp` in MP6 and 7. (They are not exactly the same as in MP6 and 7, because we have some different values and operators, but they are identical in concept.)

3 Style Requirements

As in all MPs, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

We will only enforce the rules just listed, but a more comprehensive style guide can be found at `http://caml.inria.fr/resources/doc/guides/guidelines.en.html`.

4 What to submit

You will submit `mp9.ml` using the `handin` program. Rename `mp9-skeleton.ml` to `mp9.ml` and start working from there.

As in previous, once you have finished appropriate sections of your interpreter, you can use the `run` and `run_with_args` functions to test individual programs, or simply add programs to the test suite.

- Download `mp9grader.tar.gz`. This tarball contains all the files you need, including the common AST file (`mp9common.ml`), the MiniOCaml lexer and parser (`miniocamllex.mll` and `miniocamlparse.mly`), and the skeleton solution file (`mp9-skeleton.ml`).
- As always, once you extract the tarball, rename `mp9-skeleton.ml` to `mp9.ml` and start modifying the file. You will modify only the `mp9.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do.
- **Make sure to add several more test cases to the `tests` file; follow the pattern of the existing cases to add a new case.**
- You may use the included `testing.ml` file to run tests interactively. Open the OCaml repl and type `#use "testing.ml";;` to load all of the related modules and enable testing. Further specifics on interactive testing are included in that file.

5 Syntax and Semantics

To simplify the assignment, MiniOCaml lacks some significant features, especially pattern-matching, `type` definitions, and mutually-recursive functions. The concrete syntax of MiniOCaml is the same as OCaml except for those restrictions; you can see the precise syntax in the `mly` file in the tarball. The abstract syntax is given in Figure 1 (and again in `mp9common.ml` in the tarball). The syntactic form "`fun` x `->` e" (which appears in both the concrete and abstract syntax) and the abstract syntax constructor `Rec`, are new. They were discussed at length in class.

In the abstract syntax in `mp9common.ml`, there are several operations that we are not asking you to implement (in your evaluators you should add a default case so that OCaml doesn't complain about not covering all the cases). These are: binary operations `;` (sequencing) and `:=` (assignment), and unary operations `ref` (create a pointer) and `!` (dereferencing, analogous to prefix $\star$ in C). These are the operations that implement side-effects in OCaml, something we have avoided thus far; they are included only so that we don't have to change the concrete and abstract syntax when we discuss them in a couple of weeks. There are also a number of constructors in the abstract syntax that are included with a comment "The following constructors are used only temporarily," which you can ignore because they will never show up in any AST you need to evaluate. (The translation from concrete syntax to abstract syntax — including, for example, reducing all functions to functions of one argument — is somewhat involved; see file `run.ml` if you're curious. These constructors are used to represent intermediate states of the AST during the translation.)

Just as we did at first for MiniJava, you will be implementing a *dynamically-typed* version of MiniOCaml. The rules for interpreting MiniOCaml programs are given in Figure 2 We call this function `reduce`, just to distinguish it from the interpreter you will write in MP10, which we will call `eval`. Note that there is only "evaluation" for MiniOCaml, and not "execution," because there are only expressions and no statements. `reduce` uses a helper function called `subst`, which implements the substitution of a variable with its binding; this function also recursively walks the AST to do its work, so you will find yourself splitting the "work" of the reducer into two separate recursively defined functions. `subst` is defined in Figure 3).

6 Problems

6.1 Concrete and abstract syntax

You will be dealing only with the abstract syntax in this assignment (just as in MPs 6–8); we have provided a lexer and parser for you, which translates concrete syntax to abstract syntax.

However, the translation from concrete to abstract syntax is a bit more involved than was the case for MiniJava, and it may be helpful for you to understand it. As explained in class, the abstract syntax has these properties:

- Let expressions are translated to the form "`Let(`a`,` e`,` e'`)`," where a is a variable name. A let expression that introduces a function — `let f a =` e `in` e' — will become, in the abstract syntax, "`Let(f, Fun(a,` e`),` e'`)`."
- The `Rec` constructor is used to indicate that a function is recursive. A recursive function definition — `let rec f a =` e `in` e' — becomes, in the abstract syntax, "`Let(f, Rec(f, Fun(a,` e`)),` e'`)`."
- Functions in the abstract syntax have only one argument. Functions of multiple arguments are "explicitly curried" — that is, turned into functions whose bodies are functions. Thus, `let f a b c =` e `in` e' becomes, in the abstract syntax, "`Let(f, Fun(a, Fun(b, Fun(c,` e`))),` e'`)`."

These translations all serve to simplify the evaluation process. As noted above, the translation to abstract syntax is somewhat involved, but in the end all that is happening is what we have just described.

6.2 Substitution-based interpreter (`reduce`)

The rules for evaluating expressions using substitution for function application are given in Figure 2. These rely on the definition of substitution (denoted in the evaluation rules by $e[e'/x]$), which is given in Figure 3. You need to define the following functions:

```
reduce (expr:exp) : value

subst (id:id) (v:value) (expr:exp) : exp

applyOp (bop:binary_operation) (v1:value) (v2:value) : value

applyUnop (uop:unary_operation) (v:value) : value
```

The type `value` is just a synonym for `exp`, but is used to indicate that the argument or result in question is in the subset of `exp`s that qualify as values, as explained in class. `applyOp` is very similar to the function of that name in MP 6, except that we have included some more operations than we did there; the arguments are expressions that represent constants — constructors IntConst, StrConst, True, and False.

To simplify the assignment, there are some changes in `applyOp`: no overloading, except for `Equals` (that is, Plus applies only to integers), and no short-circuit evaluation of boolean operations (so `And` and `Or` can be handled in `applyOp`). The types of each of the operators is:

Equals : *value* * *value* → *bool*
NotEquals : *value* * *value* → *bool*
LessThan : *int* * *int* → *bool*
GreaterThan : *int* * *int* → *bool*
And : *bool* * *bool* → *bool*
Or : *bool* * *bool* → *bool*
Plus : *int* * *int* → *int*
Minus : *int* * *int* → *int*
Div : *int* * *int* → *int*
Mult : *int* * *int* → *int*
StringAppend : *string* * *string* → *string*
ListAppend : *value list* * *value list* → *value list*
Cons : *value* * *value list* → *value list*
Not : *bool* → *bool*
Head : *value list* → *value*
Tail : *list* → *value list*
Fst : *tuple* → *value*
Snd : *tuple* → *value*

```
type exp =
     Operation of exp * binary_operation * exp
   | UnaryOperation of unary_operation * exp
   | Var of string | StrConst of string | IntConst of int
   | True | False
   | List of exp list | Tuple of exp list
   | If of exp * exp * exp | App of exp * exp
   | Let of string * exp * exp
   | Fun of string * exp
   | Rec of string * exp

and binary_operation = Semicolon | Comma | Equals | LessThan
   | GreaterThan | NotEquals | Assign | And | Or
   | Plus | Minus | Div | Mult
   | StringAppend | ListAppend | Cons

and unary_operation = Not | Head | Tail | Fst | Snd
```

Figure 1: MiniOCaml abstract syntax

In other words, except for `Equals` and `NotEquals`, for each operation, you need only check for one type of value for each argument, and otherwise raise a type error. (To be clear, since we're using dynamic typing, we could have overloaded operations, like `Plus`, just as in MP6. We're not doing that just to shorten the assignment a little bit.)

(Const) Const x $\Downarrow$ Const x

(Fun) Fun(a,e) $\Downarrow$ Fun(a,e)

(Rec) Rec(f,Fun(a,e)) $\Downarrow$ Fun(a,e[Rec(f,Fun(a,e))/f]

(δ) $e\ op\ e' \Downarrow v\ OP\ v'$
$e \Downarrow v$
$e' \Downarrow v'$

(δ) $op\ e \Downarrow OP\ v$
$e \Downarrow v$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ True
$e_2 \Downarrow v$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ False
$e_3 \Downarrow v$

(List) $[e_1, \ldots, e_n] \Downarrow [v_1, \ldots, v_n]$
$e_1 \Downarrow v_1$
$\vdots$
$e_n \Downarrow v_n$

(App) $e\ e' \Downarrow v$
$e \Downarrow$ Fun(a, e'')
$e' \Downarrow v'$
$e''[v'/a] \Downarrow v$

(Let) Let(a,e,e') $\Downarrow v'$
$e \Downarrow v$
$e'[v/a] \Downarrow v'$

Figure 2: Evaluation rules for substitution model

$x[v/x] = v$
$y[v/x] = y$
(let $x = e$ in e')$[v/x]$ = let $x = e[v/x]$ in e'
(let $y = e$ in e')$[v/x]$ = let $y = e[v/x]$ in $e'[v/x]$
(fun $x \rightarrow e$)$[v/x]$ = (fun $x \rightarrow e$)
(fun $y \rightarrow e$)$[v/x]$ = (fun $y \rightarrow e[v/x]$)
(rec $x\ e$)$[v/x]$ = (rec $x\ e$)
(rec $y\ e$)$[v/x]$ = (rec $y\ e[v/x]$)
The remaining rules either have no variables (literals) or simply apply substitution recursively to all components.

Figure 3: Caption: Definition of substitution, e[v/x], where v is a closed term. y is assumed to be a variable different from x.