Lecture 28: Even More Rackette  
10:00 AM, Nov 9, 2018

Contents

1 Definitions 1

2 Evaluation 1

3 Palette Cleansing Sherbet 2

3.1 A Contrived Example: "simplify" an Integer 2

4 Back to Evaluation 4

5 Mutually Recursive Functions 4

5.1 An Abstract Example 4

5.2 Concrete Example 5

6 Summary 5

1 Definitions

Definitions in Rackette consist of identifiers and expressions. Meanwhile, bindings consist of identifiers and values. Environments in Rackette are lists of bindings. Thus, in order to add definitions to our environment, we must first evaluate the expression in order to convert it into a value. Then, this new binding can be added to our environment.

Note: In the Rackette handout we have environments represented as something along the lines of $[x \to 5, + \to \text{builtin-addn}]$. Mentally, when adding bindings to the environment, you might think of adding bindings to the end of these lists. However, looking at the structure of lists in OCaml, new bindings should be added to the beginning of the environment lists in order to do this operation in constant time.

2 Evaluation

- The value of an expression depends on whether it has a binding in an environment, for example:

  \[(+ 3 \text{ a})\]

  does not have a value until we have an environment.

- Evaluation has many steps and involves recursion.
- It has many steps: divide and conquer by doing one thing at a time. In other words, write helper functions! Writing helper functions will make your code easier to debug.

- Basic template:

```racket
let eval : environment * environment * expression -> value = function
tle, env, expr) -> match expr with
| NumE n -> ...
| IdentE name -> ...
| AndE (e1, e2) -> ...
| ...
;;
```

3 Palette Cleansing Sherbet

3.1 A Contrived Example: ”simplify” an Integer

Let’s attempt to write a procedure that takes in an integer and does the following:

- Try to divide by 2 (with no remainder).
- Try to divide by 3 (with no remainder).
- Give up.

Making a design recipe out of this:

```racket
(* simplify: int -> int option
   Input: a positive integer, n
   Output:
       Some n/2 if n is a multiple of two
       Some n/3 if n is odd but a multiple of 3
       None, otherwise
   Examples:
       Simplify 12 -> Some 6
       Simplify 15 -> Some 5
       Simplify 19 -> None *)
```

It might be helpful to create a helper procedure, `simplify_2` which does the following:

```racket
(* simplify_2: int -> int option
   Input: a positive integer, n
   Output:
       Some n/2 if n is a multiple of two
       None, otherwise *)

let simplify_2: int -> int option: = function
  n -> if (n mod 2) == 0 then Some (n/2) else None ;;
```
Let’s also create a helper procedure called `simplify_3` which is the same as `simplify_2` except changing all of our 2’s to 3’s:

```racket
(* simplify_3: int -> int option
    Input: a positive integer, n
    Output:
        Some n/3 if n is a multiple of three
        None, otherwise *)

let simplify_3: int -> int option: = function
    n -> if (n mod 3) == 0 then Some (n/3) else None ;;
```

How will we write the main program, `simplify`? We might try writing it in the following way:

```racket
let simplify : int -> int option = function
    n -> if (simplify_2 n) = None
        then
        if (simplify_3 n) = None
        then
            None
        else
            simplify_3 n
        else
            simplify_2 n ;;
```

After testing the code above, we see that it works! However, isn’t there something a bit strange about this code? We are saying if `simplify_3` of `n` is `None`, return `None`, otherwise return `simplify_3` of `n`. In other words, either way we are returning the result of calling `simplify_3` on `n`. Because of this, we could change the code above to the following:

```racket
let simplify : int -> int option = function
    n -> if (simplify_2 n) = None
        then
            simplify_3 n
        else
            simplify_2 n ;;
```

Even in the code above, however, we’re calling `simplify_2` on `n` twice, which is a sign that we could perhaps make our code a little bit cleaner. Now, let’s change our code to the following:

```racket
let simplify : int -> int option = function
    n -> let result = simplify_2 n in
        if result = None
        then
            simplify_3 n
        else
            result ;;
```
However, even in the code above, the if clause tests which of two kinds of “option” we’ve got. In OCaml, looking at the possible cases for different types is a perfect situation to use pattern matching in instead. Because of this, we should change our code to the following final version:

```ocaml
let simplify : int -> int option = function
  n -> let result = (simplify_n) in
  match result with
  | None -> simplify_2 n
  | _ -> result ;;
```

4 Back to Evaluation

- Sometimes the ”rules” say to look up the value in the local environment, and if it’s not found, there, then in the top-level environment.

Suppose we’ve written a procedure `lookup : expression * environment -> value option` which does the following:

- If the expression is not a `SymbolE`, it’s an error.
- If it’s a symbol, and defined in the environment, then return `Some V`, where `v` is the value bound to the symbol.
- If it’s `not` defined in the environment, then return `None`.

**Quiz**: Using `lookup` as a helper procedure, define a procedure `lookup2 : expression * environment * environment -> value option` which takes in a local environment and a top level environment to search.

**Answer**:

```ocaml
let lookup2 : expression * environment * environment -> value option = function (expr, local, tle) ->
  let result = (lookup expr local) in
  match result with
  | None -> (lookup expr tle)
  | _ -> result ;;
```

5 Mutually Recursive Functions

In OCaml, it is not allowed to call a procedure that you haven’t defined yet. In other words, if you try to call a procedure which isn’t defined until further down in your coding file, you’ll get an error saying that the procedure hasn’t been defined yet. Normally, we can fix this by just moving our definition of a procedure to before we ever call it. However, sometimes, you might find it in your interest to create two procedures which call each other. Then, regardless of which one you put first, you’ll run into errors. To fix this problem, we can use a technique called mutually recursive functions.
5.1 An Abstract Example

To show the syntax of this, let’s say we have, this_proc and that_proc which each call each other somewhere in their implementation. Normally, when writing them we’d write something along the lines of:

```racket
let this_proc = function
....... 
    that_proc 
....... ;;

let that_proc = function 
....... 
    this_proc
....... ;;
```

However, once again, trying to write them in this way will result in an error. In order to make these procedures mutually exclusive, we should change the syntax to:

```racket
let this_proc = function
....... 
    that_proc
....... 
and that_proc = function 
....... 
    this_proc
....... ;;
```

In this way, the two procedures will be defined at the same time and we won’t encounter any issues.

5.2 Concrete Example

To give a concrete example of mutually recursive procedures, let’s use a pair of procedures called is_even and is_odd. We will write them in the following way:

```racket
let is_even : int -> bool = function
0 -> true
n -> is_odd n-1

and is_odd : int -> bool = function
0 -> false
n -> is_even n-1 ;;
```

Keep in mind that the example code above, while not the logically best way to write the is_even and is_odd procedures, is a great example to illustrate mutually recursive procedures.
6 Summary

- Evaluation is an important part of the Rackette assignment. It has many steps and involved recursion, but these steps should be divided up into helper functions.
- Once you have working code, it is always important to try and go back and simplify it.
- Mutual recursion will also be very useful in the Rackette assignment.