Lab 8: Environments
12:00 PM, Oct 28, 2018

Contents

1 Rackette Environment 1
2 Debugging 2

Objectives

By the end of this lab, you will know:

- how to use environments for the Rackette project

By the end of this lab, you will be able to:

- make a new environment
- add a binding to an environment
- extend an environment by a collection of new bindings
- look up identifiers in an environment

1 Rackette Environment

Note: The code you write in this lab may end up in your Rackette project. Even though your lab partner might not be your project partner, this is fine!

For Rackette, in order to keep track of bindings from identifiers to values, we’ll need environments, which are sequences of bindings. For simplicity, we can represent these as a list. And if you think back to last week’s lab, you may recall the lookup function, which searches through a list for a pair that contains the key you’re looking for and returns the value associated with the key. We can use our environments as dictionaries, and use something like lookup to search for the values bound to different identifiers.

We can say a binding is something like this, where the string component is the identifier:

```
type binding = string * value
```

It’s helpful, though, to define types that let Ocaml’s type-system help us. Strings will be used for multiple things in the Rackette project, and by defining several different types, each type a wrapper for a string that has a different purpose, we can avoid many tricky type bugs.

So for Rackette, rather than just treating an identifier as a string, we’re going to recommend this as the type-definition for the representation of an identifier:
And for values, we recommend naming a type as well. As was mentioned in class, for this assignment, we can choose a very simple version of “value” – something that allows for multiple possibilities, but isn’t yet complete:

```
type value = Int_val of int | String_val of string
```

Then a binding now becomes

```
type binding = identifier * value
```

Now that we’ve specified what a binding is, here are the things we need to do with an environment:

- Make a new empty environment (for creating the top-level environment, but also for creating the environment that’ll be part of every closure that we create)
- Add a binding to an environment
- Extend one environment by another. (When we extend $E$ by $E'$, the bindings in $E'$ should override those in $E$, so that if a name $s$ is bound in $E'$ and $E$, the value of $s$ in the extended environment is found from $E'$ rather than $E$.)
- Look up an identifier in the environment to find out the value it is bound to (or fail with in the event that the identifier is unbound). (An alternative design is to return a value option, returning None when the identifier doesn’t appear in the environment.)

Task: Write type definitions for bindings and environments (for which you’ll need the type-definition for values, but you can use a placeholder for that, like `type value = Int | String`, for instance). First, define `empty_env` as what an empty environment would be.

Then write three procedures to: (1) add a binding to an environment, (2) extend one environment by another and return the resulting larger environment, and (3) look up the value associated with an identifier (represented as ”ID of string”) in an environment (or fail as described).

Note: Don’t overthink the definition or procedures.

Your code (aside from the check-expects) should be just a dozen lines or so. Since you’ll be using this code in the Rackette project, you have a strong incentive to get it right!

| You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

2 Debugging

Now, we’ll cover a topic near and dear to every computer scientist’s heart; debugging. First, copy the source code from the course directory into your `lab08` directory.

One of the files we’ve provided, seen below, is a solution to the `lab07` problem `lookup`. You can see the code below.
let rec lookup: (int * string) list * int -> string option = function
    ([], _) -> None
  | ((k, v)::tl, key) -> if k == key then (Some v) else lookup(tl, key) ;;

#trace lookup ;;

lookup([((1, "one"), (2, "two"), (3, "three")), 3) ;;

The first part of the file makes sense; it’s just the solution itself. And the last line is also pretty clear; it’s just a call to lookup. But what’s this #trace business? Try running lookup.ml in the command line and see what it prints.

Pretty nifty, right? What #trace does is prints out information every time a procedure is called. When you see the procedure name with a left arrow next to it, the expression after the arrow is the argument the procedure was called with. When you see the procedure name with a right arrow next to it, the expression after the arrow is what the procedure returned. As you can see here, lookup is called three times, each time with a shorter list, and then returns Some "three" once for each time it was called.

Now, as you can see, we gave a type annotation to our code above. What if we removed this type annotation? Logically, since == can take any data type, the code should still work, but should now take dictionaries and keys of any type! We’ve provided this code as well, in generic_lookup.ml.

let rec lookup = function
    ([], _) -> None
  | ((k, v)::tl, key) -> if k == key then (Some v) else lookup(tl, key) ;;

#trace lookup ;;

lookup([(1, "one"), (2, "two"), (3, "three"), 3) ;;

Try running this code in the command line and see what it produces.

As you can see, while the code still worked (the end result was what we wanted), the trace was not nearly as useful. Unfortunately, OCaml needs to know the specific types of the inputs and outputs to use #trace properly. This can be a pain when testing procedures that take in generic types. To avoid these issues, if you want to use #trace, you can temporarily give your code a very specific type annotation, test it using #trace, and then remove the type annotation once you think you’ve found your bug.

Now, we’ve seen #trace on a procedure with one recursive call. But what about #trace on a procedure with two recursive calls? Well, let’s try it! We’ve provided a nice, neat procedure to calculate fibonacci numbers, provided in fibonacci.ml.

let rec fibonacci: int -> int = function
  0 -> 1
  | 1 -> 1
  | x -> (fibonacci (x - 1)) + (fibonacci (x - 2)) ;;

#trace fibonacci ;;

fibonacci 3 ;;
Run this in the command line and see what it prints.

While the trace is much harder to follow, we can still use this tool to tell what’s going on. You might also notice that the recursive calls aren’t printing or executing in the order you expect them to. However, we can still gain some useful information from this trace.

Now, it’s time to try some things on your own. We’ve provided buggy solutions to the problems from lab05 in subsequence.ml, lcs.ml, and gap.ml respectively. (Since OCaml doesn’t use symbols, all three of these procedures take in string lists.) You could open the files, read the code, and try to find the bug that way, but we’re going to try a different way. Instead, use the OCaml interpreter to find the bugs. We’ve traced the relevant procedures for you, so to use the OCaml interpreter for this, all you have to do is open it and type `#use "<filename>"` to get access to our implementation of the relevant problem.

**Task:** Figure out the bugs in subsequence, lcs, and gap using `#trace` and the interpreter.

---

| You’ve reached a checkpoint! Please sign up to get a lab TA to review your work. |
| Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written. |

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)