Lecture 24: Type Aliases, Variant Types, and Birth Environments

10:00 AM, Oct 31, 2018

Contents

1 Type Aliases and Variant Types
   1.1 Type Aliases ........................................................................................................... 1
   1.2 Variant Types .......................................................................................................... 1

2 Birth environments of procedures .............................................................................. 2

1 Type Aliases and Variant Types

1.1 Type Aliases

There are two uses of the keyword type in OCaml. The simplest is what Professor Klein likes to call a type alias:

type student = string * int

This states that a student is just a pair consisting of a string (the student’s name) and an int (the student’s ID).

This helps me in three ways. First, it helps someone reading the code. Second, I can use student to refer to string * int in type annotations, e.g.

let get_ID: student -> int = function (name, ID) -> ID

I can also use it in other type definitions, e.g.

type gradebook_entry = student * float

This states that a gradebook entry is a pair consisting of a student and a float (the grade).

1.2 Variant Types

But another use of type is in defining completely new types. One new kind of type is called a variant type.

Quiz: Write a variant type definition "transport" that reflects four ways to transport goods: Van, Pickup truck, and Freight truck.
The capacity of each of these types is as follows: Van, 1000 pounds, Pickup truck, 1000 pounds, Freight Truck, 40,000 pounds.

Suppose x is bound to a value of type transport. Write an expression whose value is the capacity of that transport.

type transport =
  Van
| Pickup_truck
| Freight_truck ;;

Define a type "fleet" that can be used to represent a collection of transports.

type fleet = transport list ;;

Define a procedure capacity that, given a fleet, reports the fleet’s total capacity.

let rec capacity : fleet -> int = function
  [] -> 0
| Van :: tail -> 1000 + capacity tail
| Pickup_truck :: tail -> 1000 + capacity tail
| Freight_truck :: tail -> 40000 + capacity tail ;;

What is the type of your procedure?

fleet -> int

Actually, not all freight trucks have capacity 40,000 pounds. Because of this, we should revise our definition of the transport variant type as well as our definition of capacity so that we can have a freight truck of any capacity.

type transport =
  Van
| Pickup_truck
| Freight_truck of int ;;

let rec capacity: fleet -> int = function
  [] -> 0
| Van :: tail -> 1000 + capacity tail
| Pickup_truck :: tail -> 1000 + capacity tail
| Freight_truck c :: tail -> c + capacity tail ;;

2 Birth environments of procedures

Quiz: Write a Racket expression whose value is a procedure with the following spec:
input: increment, an integer
output: a procedure that, given an integer x, returns x+increment

Answer: (lambda (increment) (lambda (x) (+ increment x)))

We think a procedure value has two parts:

- the argument list, in this case (increment)
- the body, in this case (+ increment x)

Quiz: Use the above expression in another expression whose value is a procedure with the following spec:

- input: an integer x
- output: x + 7

Answer: ((lambda (increment) (lambda (x) (+ increment x))) 7)

What are the two parts of the procedure that is the value of this expression?

- the argument list is (x)
- the body is (+ increment x)

When this procedure is applied to something, say, 10, what happens? The symbol x is bound to 10, and the body is evaluated:

> (( (lambda (increment) (lambda (x) (+ increment x))) 7) 10)
17

Is there something a little mysterious about this? The symbol increment in the body—we know it was bound to 7 but how does that binding manage to affect the evaluation of the expression (+ increment x)?

The answer is that a procedure data object has not two but three parts: the arg list, the body, and the birth environment.

The birth environment of a procedure object is the environment consisting of the local variables that are the procedure’s arguments, which are defined locally within the procedure’s scope, but not outside of it. The concept of birth environments will be very important as we begin our next project: Rackette!

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).