(Provisional) Lecture 20: ReasonML Fun!

10:00 AM, Oct 21, 2019

Contents

1 Miscellaneous 1
2 Dictionaries 2
3 Well-Ordering Proof Practice 4
4 Summary 6

Objectives

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

- test the equality of sets of different types
- create procedures that take in multiple parameters
- rewrite some Racket procedures in OCaml
- define a lookup procedure for dictionaries in OCaml

1 Miscellaneous

Spike spent the first half of class going over various miscellaneous details:

- Many students are still having trouble installing reason on their laptops. To help with this issue, we will be holding more installation hours specifically for issues related to this. We may also be able to help you on piazza if you tell us what you have done so far and show us the error message(s) you are getting.

- In reason, there are various ways to do the type annotations for functions. Here are two examples:
  - let f : (int => string)= arg => ...
  - let f = (arg : int): string => ...

As shown in the first example, the type annotation can be done before the name of the argument is introduced. In the second, the type of the argument is specified directly after the argument is named. Both are valid ways to show what type your function is, but Spike prefers the type annotation to be completely done before the name of the argument is introduced (first example) so do that one!
• Modules!! Modules are essentially a way of gathering a bunch of functions and other stuff into one entity (this will become much more clear in the final project of CS17, Game). If you have started the homework for this week, you have already implemented some parts of the list module, using empty and non-empty lists and doing operations on lists such as List.hd, List.tl, List.map, and List.rev.

• Remember that we use let in reason, not define (the DrRacket equivalent)

• Type definitions can be very useful in reason. If you find yourself using the same type over and over again (and having to type it over and over again), you might consider defining that type at the top of your file and then using it whenever you need it in your functions. However, keep in mind that type definitions in reason have their limitations: if there is any specific behavior that you want your programs to have that are not inherent to the types that you use in your definitions (lists, integers, etc.), you must specify that in your input output specifications. For example, if you were defining a bignum, you might do something along these lines: type bignum = list(int);. You know that the elements of a bignum should all be one digit long, but all you have specified in the type definition is that all the elements must be integers. In this case, you would specify the fact that each element of a bignum must be between 0 and 9 in your input/output specs.

Check out the slides for more info/details :)

2 Dictionaries

Dictionaries are a special type of data structure used in many different languages. You can think of a dictionary as a function, call it f. This function takes in a key and returns a value associated that key. Thus, you can also think of dictionaries as a set of key-value pairs.

You’ve encountered things resembling dictionaries before! A phone book, for example, can be thought of as a dictionary where names are the keys and phone numbers are the values associated with a certain key. An English-French dictionary has English words as its keys, and French words as the associated values. Even environments can be thought of as dictionaries from identifiers to values bound to those identifiers.

There are some typical operations associated with dictionaries:

• Is this key in the dictionary
• What value is associated with this key
• Add a key-value pair
• Replace a key-value pair
• Delete a key-value pair

Let’s attempt to develop a lookup procedure that takes in a key and a dictionary, and returns the value associated with that key.
(* my_dict is an English to French dictionary *)
my_dict = ["I", "je"], ["you", "tu"], ["dog", "chien"] ;

(* Input: an English word to look up in a dictionary, key
a dictionary from English to French words, dict *)
(* Output: the French words associated with key *)
let rec lookup = (key: string, dict: list((string, string))): string =>
    switch (dict) {
    | [] => ???
    | [(k, v), ...tl] =>
        if (key === k) {
            v;
        } else {
            lookup(key, tl);
        }
    };

But what happens if the key we are looking for is not in our dictionary? We’ll need a way to tell
the user that our lookup procedure did not find a value in our dictionary associated with the key
they entered. Can we do better than outputting an error message in this case?

**Options**  Options are a builtin type for handling success/failure cases!

(* The following defines a type 'a option *)
type 'a option = Some ('a) | None ;

The intended use of an option is to indicate that either what you were looking for was found, or not.
If an answer was found, a procedure returning an option will return Some(x), where x is the answer
you were looking for. If an answer was not found, the procedure will return None.

Having a procedure return an option instead of a value is helpful for dictionaries, where we aren’t
sure if a lookup procedure will have a non-existent key as an input. We can use options to remedy
this issue! Keep in mind, though, that if options are used, a new type signature will be required for
most functions. Notice how the type signature changes for our lookup procedure.

... let rec lookup (key : string) (dict : (string * string) list) : string option = ...

Writing lookup using options is left as a task in Lab 7.

Here is an example of a procedure that makes use of options. Note the use of parentheses to wrap
the value associated with a Some option. This is a good habit to get into.
let rec sum_but_ones =
  fun |
    [] => None
  | [1, ...tl] => sum_but_ones(tl)
  | [hd, ...tl] =>
    switch (sum_but_ones(tl)) {
      | None => Some(hd)
      | Some(s) => Some(hd + s)
    };

3 Well-Ordering Proof Practice

We did a couple of well-ordering proofs in class.

The first proof was to prove that if \( H : \mathbb{N} \rightarrow \mathbb{N} \) satisfies the recurrence

\[
\begin{align*}
H(0) &= A \\
H(n) &\leq B + H(n - 1) \quad \text{for } n > 0
\end{align*}
\]

then for every \( n \in \mathbb{N} \),

\[
H(n) \leq Bn + A.
\]

(Eq 1)

The statements we filled out together in class are in red

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Let ( S = { n \in \mathbb{N} \mid (\text{Eq 1}) \text{ is false for } n } )</td>
<td>Definition of ( S )</td>
</tr>
<tr>
<td>2 Suppose that ( S ) is nonempty</td>
<td>Contradiction hypothesis</td>
</tr>
<tr>
<td>3 Let ( k ) be the least element of ( S )</td>
<td>Well-ordering</td>
</tr>
<tr>
<td>4 (Eq 1), for ( n = 0 ), says that ( H(0) \leq B \cdot 0 + A = A )</td>
<td>Restatement of (Eq 1).</td>
</tr>
<tr>
<td>5 (Eq 1), for ( n = 0 ), is true</td>
<td>Line 1 of recurrence</td>
</tr>
<tr>
<td>6 ( k \neq 0 )</td>
<td>If it were, then (Eq 1) would not be true for ( n = 0 ); S5.</td>
</tr>
<tr>
<td>7 ( k &gt; 0 )</td>
<td>( k ) is a natural number, and ( k \neq 0 ) by S6.</td>
</tr>
<tr>
<td>8 (Eq 1) is true for ( n = k - 1 )</td>
<td>S1, S3, S7</td>
</tr>
<tr>
<td>9. ( H(k - 1) \leq B \cdot (k - 1) + A )</td>
<td>Restatement of S8 with ( k - 1 ) plugged in.</td>
</tr>
<tr>
<td>10 ( H(k) \leq B + H(k - 1) )</td>
<td>Because ( k &gt; 0 ) (see S7) and line 2 of recurrence.</td>
</tr>
<tr>
<td>11 ( H(k) \leq B + [B \cdot (k - 1) + A] )</td>
<td>S10, S9.</td>
</tr>
<tr>
<td>12 ( H(k) \leq B + Bk - B + A )</td>
<td>S11, algebra.</td>
</tr>
<tr>
<td>13 ( H(k) \leq Bk + A )</td>
<td>S12, algebra.</td>
</tr>
<tr>
<td>14 ( k \notin S )</td>
<td>S13, definition of ( S ) in S1.</td>
</tr>
<tr>
<td>15 Contradiction</td>
<td>S14, S3. Hence S2 is false!</td>
</tr>
<tr>
<td>16 S2 is false, so (Eq 1) holds for all natural numbers</td>
<td>S2, S15.</td>
</tr>
</tbody>
</table>
Then, we worked on finding the upper bound for this recurrence.

Suppose $H : \mathbb{N} \mapsto \mathbb{N}$ satisfies the recurrence

$$H(0) = A$$
$$H(n) \leq B + 2H(n-1) \quad \text{for } n > 0$$

where $A > 0, B > 0$. Use plug-n-chug to find an upper-bound for $H(k)$ for $k \in \mathbb{N}$.

In class we worked to find this upper bound. A useful theorem we used in class is that for $k \in \mathbb{N}$,

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1.$$

Then we worked to fill out a second well-ordering proof to prove this upper bound.

Claim: if $H : \mathbb{N} \mapsto \mathbb{N}$ satisfies the recurrence

$$H(0) = A$$
$$H(n) \leq B + 2H(n-1) \quad \text{for } n > 0$$

where $A > 0, B > 0$, then for all $n \in \mathbb{N}$,

$$H(n) \leq (2^n)A + (2^n - 1)B$$

The statements we filled out in class are in red.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Let $S = {n \in \mathbb{N} \mid (\text{Eq 1}) \text{ is false for } n}$</td>
<td>Definition of $S$</td>
</tr>
<tr>
<td>2 Suppose that $S$ is nonempty</td>
<td>Contradiction hypothesis</td>
</tr>
<tr>
<td>3 Let $k$ be the least element of $S$</td>
<td>Well-ordering says a least element must exist, by S2</td>
</tr>
<tr>
<td>4 (Eq 1), for $n = 0$, says that $H(0) \leq 2^0A + (2^0 - 1)B = A$</td>
<td>Restatement of (Eq 1).</td>
</tr>
<tr>
<td>5 $H(0) = A$</td>
<td>Line 1 of recurrence</td>
</tr>
<tr>
<td>6 (Eq 1), for $n = 0$, is true</td>
<td>S4, S5</td>
</tr>
<tr>
<td>7 $k \neq 0$</td>
<td>If it were, then (Eq 1) would not be true for $n = 0$; S5.</td>
</tr>
<tr>
<td>8 $k &gt; 0$</td>
<td>S7, A nonzero natural number must be a positive integer.</td>
</tr>
<tr>
<td>9 (Eq 1) is true for $n = k - 1$</td>
<td>S1, S3, S8</td>
</tr>
<tr>
<td>10. $H(k - 1) \leq (2^{k-1})A + (2^{k-1} - 1)B$</td>
<td>S9, restated as a formula</td>
</tr>
<tr>
<td>11 $H(k) \leq B + 2H(n-1)$</td>
<td>Because $k &gt; 0$ (see S7) and line 2 of recurrence.</td>
</tr>
</tbody>
</table>
\[ H(k) \leq B + 2[(2^{k-1})A + (2^{k-1} - 1)B] \]
\[ H(k) \leq B + (2^k)A + (2^k)B - 2B \]
\[ H(k) \leq (2^k)A + (2^n - 1)B \]

15. \( k \not\in S \)

16. Contradiction

17. S2 is false, so (Eq 1) holds for all natural numbers

S11, S10.
S12, algebra.
S13, algebra.
S14, definition of \( S \) in S1.
S15, S3. Hence S2 is false!
S2, S16.

4 Summary

Ideas

- Dictionaries are a very useful data structure that allow us to store pairs of data with a key, and an associated value.

- Options are a builtin type for handling success/failure cases. If an answer was found, a procedure returning an option will return \texttt{Some}(x), where \( x \) is the answer you were looking for. If an answer was not found, the procedure will return \texttt{None}.

Skills

- Understand the importance and usage of dictionaries.
- Understand how to use options
- Practice writing a well-ordering proof

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).