Midterm Project

Introduction to Computational Science
Yale-NUS College

24 March 2022

Introduction

Midterm is here - this is a report that has all the mini-projects included in the midterm projects. In this midterm project, we revisit, familiarize ourselves with, and deepen our understanding of the topic we have covered since the start of this semester. The topic includes boolean functions, recursive functions, functional abstraction, OCaml expressions and evaluation, polynomial functions, etc. The work is collectively done as a group - SJANN - with my groupmate. I am the S - Swift in the group, but I will be using the pronoun "we" in the following exercises because this report is written for "us" - the readers to understand and follow what is going on. Throughout the exercises, I will note who contributed to solving the exercise, using the first letter of their pen names. Lastly, I would like to state that all the questions in the midterm project are durable, though we don't know the answer from just looking at the question, we are able to figure it out with much consideration. Now, let us dive into the exercises…

"Programming isn't about what you know; it's about what you can figure out."
- Chris Pine

More functional abstraction

Introduction

In this exercise, we implement a maker of boolean functions that is parameterized with the 4 cells of their truth table and make several made-up boolean implementations. We test the made-up implementations with the uni test and check for its coverage. This mini-project is solved with the collaborative effort of S and J.

Solution

We follow the true table shown below:

	b1 = true	b1 = false
b2 = true	tt	tf
b2 = false	ft	ff

```
# let make_boolean_function tt tf ft ff b1 b2 =
    if b1 && b2 then tt
    else if b1 && not b2 then tf
    else if not b1 && b2 then ft
    else ff;;
val make_boolean_function : 'a -> 'a -> 'a -> 'a -> bool -> bool -> 'a =
  <fun>
```

Using this maker, we can parameterize any boolean functions by adding 4 cells of the truth table.

a. Conjunction

```
# let test_conjunction candidate =
    (candidate true true = true)
    &&
    (candidate true false = false)
    &&
    (candidate false true = false)
    &&
    (candidate false false = false);;
val test_conjunction : (bool -> bool -> bool) -> bool = <fun>

# let conjunction_gen =
    make_boolean_function true false false false;;
val conjunction_gen : bool -> bool -> bool = <fun>

# let () = assert (test_conjunction conjunction_gen = true);;
```

b. Disjunction

```
# let test_disjunction candidate =
    (candidate true true = true)
    &&
    (candidate true false = true)
    &&
    (candidate false true = true)
    &&
    (candidate false false = false);;
val test_disjunction : (bool -> bool -> bool) -> bool = <fun>

# let disjunction_gen =
    make_boolean_function true true true false;;
val disjunction_gen : bool -> bool -> bool = <fun>

# let () = assert (test_disjunction disjunction_gen = true);;
```

c. Negated Conjunction

```
# let test_negated_conjunction candidate =
    (candidate true true = false)
    &&
    (candidate true false = true)
    &&
    (candidate false true = true)
    &&
    (candidate false false = true);;
val test_negated_conjunction : (bool -> bool -> bool) -> bool = <fun>

# let negated_conjunction_gen =
    make_boolean_function false true true true;;
val negated_conjunction_gen : bool -> bool -> bool = <fun>

# let () = assert (test_negated_conjunction negated_conjunction_gen =
true);;
```

d. Negated Disjunction

```
# let test_negated_disjunction candidate =
    (candidate true true = false)
    &&
    (candidate true false = false)
    &&
    (candidate false true = false)
    &&
    (candidate false false = true);;
val test_negated_disjunction : (bool -> bool -> bool) -> bool = <fun>

# let negated_disjunction_gen =
    make_boolean_function false false false true;;
val negated_disjunction_gen : bool -> bool -> bool = <fun>

# let () = assert (test_negated_disjunction negated_disjunction_gen =
true);;
```

e. Exclusive Disjunction

```
# let test_exclusive_disjunction candidate =
    (candidate true true = false)
    &&
    (candidate true false = true)
    &&
    (candidate false true = true)
    &&
    (candidate false false = false);;
val test_exclusive_disjunction : (bool -> bool -> bool) -> bool = <fun>

# let exclusive_disjunction_gen =
    make_boolean_function false true true false;;
val exclusive_disjunction_gen : bool -> bool -> bool = <fun>

# let () = assert (test_exclusive_disjunction exclusive_disjunction_gen =
true);;
```

We can say that our implementation is correct, because they passed the unit-test functions. For the functions that take two booleans, since boolean is either true or false, there are only 4 possible cases. In our unit-test functions, we have tested all four cases, therefore the unit test is complete.

Conclusion

In this exercise, we have made a maker function for boolean functions. We went through the process of abstracting a computation with a function in OCaml and instantiating the abstract computation by applying the corresponding function. This functional abstraction and instantiation is something we keep seeing in the next few projects. Considering that every pre-defined functions in OCaml are already abstracted and waiting to be instantiated, we are reminded of the importance of the S-m-n theory.

For testing the functions with the unit-test functions, since the unit-test functions are complete, we can confidently say that our implementations are correct. This would not be the case for the rest of this report where the unit-test coverage will be brought into question.

A miscellany of recursive programs from Week 06

Introduction

In this mini-project, we look at the computational content of mathematical induction and use various implemented functions such as nat_fold_right, and nat_parafold_right, which are generic functions that embody the programming pattern of structural recursion over natural numbers. This project was done by the collective effort of N and A.

Exercise 03

Introduction

In this exercise, along the lines of primitive recursion of natural numbers, we implement nat_of_digits as an instance of nat_fold_right.

Solution

We start off with the unit-test function, which is the same unit test that we implemented in Week 06.

```
# let test_nat_of_digits candidate =
    (candidate "9" = 9)
    && (candidate "89" = 89)
    && (candidate "789" = 789)
    && (candidate "6789" = 6789)
    && (let n = Random.int 10000
        in candidate (string_of_int n) = n)
    (* etc. *);;
val test_nat_of_digits : (string -> int) -> bool = <fun>
```

```
# let nat_of_digits_v2 s =
    let n = String.length s
    in nat_parafold_right (nat_of_digit (String.get s 0)) (fun i' ih ->
ih * 10 + nat_of_digit (String.get s (succ i'))) (n-1);;
val nat_of_digits_v2 : string -> int = <fun>
```

We have used nat_parafold_right function to implement nat_of_digits function. The zero case and succ cases follow the inductive specification. To clarify, let me quote from our group report on Exercise 15 from Week 06, which was my own writing.

Quote:

$$1 = 1$$
$$12 = 1 \times 10 + 2$$
$$123 = (1 \times 10 + 2) \times 10 + 3$$

By the Horner form, we see that the addition of a digit can be expressed as [previous digits] x 10 + [added digit].

The inductive specification of nat_of_digits follows

- base case: a string of length 1 returns a single-digit integer
- induction step: given a number `i'` such that its output is the calculated `ih`, the expected output of `i` is `ih * 10 + i`. (Here, though `i` is an index and not an integer, we use `i` to refer to a character in a string whose index is `i` for readability.)

Quote ended.

nat_parafold_right takes three arguments. Let me break them down one by one.

1. The zero_case, as mentioned in the inductive specification, is the first element (index 0) of the string. The String.get s 0 returns a char, and gets converted to an int by nat_of_digit function.
2. The succ_case, considers ih, inductive hypothesis, which is the recursive call to visit the previous i, and multiplies by 10 following the inductive step. Then, it adds the i-th

element of the string to the ih, which in this case, we have to express in terms of i' because the succ_case is conditioned as succ_case i' ih.
3. Lastly, let n be the length of the string.

```
# test_nat_of_digits nat_of_digits_v2;;
- : bool = true
```

This implementation passes the unit test that we implemented in Week 06. However, we can use a fake function to cheat this unit-test function. This works because, when we implemented the unit-test function, the upper bound for Random.int is 10,000. So if we implement a function that takes an input greater than 10,000, the unit-test function does not detect the error.

```
# let fake_nat_of_digits s =
    if s = "10001"
    then 1
    else nat_of_digits_v2 s
```

```
# test_nat_of_digits fake_nat_of_digits;;
- : bool = true
```

Conclusion

In this exercise, we have implemented nat_of_digits using the nat_parafold_right function that we previously defined. We have familiarized ourselves with the simplification of a complex function using a specialized recursive function. This exercise leads to the later exercise where we will implement more complex functions using the same kind of specialized functions.

Along with the previous mini-project, we also encountered the first limited unit-test function. As we make use of the Random.int function to implement a unit test, any input that is greater than the upper bound cannot be tested. We will see this type of error again later in this report.

Exercise 07

Introduction

In this exercise, we tackle the summatorial function. We write a unit-test function, consider the inductive specification, and explore various ways to implement the sigma function using pre-defined specialized functions.

Solution

A. Compose a unit-test function for sigma, based on its inductive specification.

```
# let test_sigma candidate =
    let b0 = (let f x = x in candidate f 3 = (3*4)/2)
    and b1 = (let f x = x in candidate f 24 = ((24*25)/2))
    and b1r = (let n = random_int () and f x = x in
                candidate f n = (n * (n+1)) / 2)
    and b2 = (let f x = 2 * x in
               candidate f 5 = (5*6))
    and b2r = (let n = random_int () and f x = 2 * x in
                candidate f n = (n * (n+1)))
    and b3 = (let f x = (3 * x) + 5 in
               candidate f 4 = 55)
    and b4 = (let f x = (3 * x) + 5 in
               candidate f 0 = 5)
    and b3r = (let n = random_int () and f x = (3 * x) + 5 in
               candidate f n = (9*n*n + 3*n + 30)/2)
    and is = (let k = random_int () and f x = (3 * x) + 5 in
               candidate f (k+1) = (candidate f k) + ((3 *(k+1)) + 5))
    in b0 && b1 && b1r && b2 && b2r && b3 && b3r && b4 && is;;
val test_sigma : ((int -> int) -> int -> int) -> bool = <fun>
```

In this unit-test function, we made use of the formula of sum of natural numbers, namely **1+2+3+... +n = n(n+1)/2.** We used random_int() function which randomly chooses one integer i within the bound, in our case, 10000.

B. Implement this specification as a structurally recursive function expecting a function from int to int and a non-negative integer.

```
# let sigma_rec f n =
    let () = assert (n >= 0) in
    let rec visit i =
      if i = 0
      then f 0
      else (visit (pred i)) + (f i)
    in visit n;;
val sigma_rec : (int -> int) -> int -> int = <fun>
```

This recursive implementation is a rather simple one. The implementation takes two arguments, a function and a non-negatie integer. The visit call recursively calls for the previous number, and using visit(pred i)) as the inductive hypothesis, we keep adding i applied to the function to the inductive hypothesis.

C. Verify that this implementation passes your unit test.

```
# test_sigma sigma_rec;;
- : bool = true
```

D. Express your implementation using either nat_fold_right or nat_parafold_right, your choice. Justify this choice, and verify that your new implementation passes your unit test.

We use nat_parafold_right. The difference between nat_fold_right and nat_parafold_right can be seen in how the succ_case is conditioned in the recursive call. In nat_fold_right, the induction case is succ_case ih. On the other hand, nat_prafold_right has succ_case i' ih in the induction case. For the sigma function, as we are making use of both the inductive hypothesis and the current i, we should be using nat_parafold_right.

The inductive specification is rather simple.

1. Zero case -> apply 0 to the function
2. Induction step -> inductive hypothesis + current i. The inductive hypothesis calls for previous i.

Similar to exercise 03, we can make use of succ(i') as i.

```
# let sigma_gen f n =
    nat_parafold_right (f 0) (fun i' ih -> ih + (f (succ i'))) n;;
val sigma_gen : (int -> int) -> int -> int = <fun>
```

This implementation passes the unit test.

```
# test_sigma sigma_gen;;
- : bool = true
```

E. Re-revisit the summatorial function that sums the first consecutive natural numbers and express it using sigma.

From the .ml file given, our unit-test function is:

```
# let test_summatorial candidate =
    let b0 = (candidate 0 = 0)
    and b1 = (candidate 1 = 0 + 1)
    and b2 = (candidate 2 = 0 + 1 + 2)
    and b3 = (candidate 3 = 0 + 1 + 2 + 3)
    and b4 = (candidate 4 = 0 + 1 + 2 + 3 + 4)
    and b5 = (candidate 5 = 0 + 1 + 2 + 3 + 4 + 5)
    and b6 = (candidate 6 = 0 + 1 + 2 + 3 + 4 + 5 + 6)
    and b7 = (candidate 7 = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7)
    and b8 = (candidate 8 = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8)
    in b0 && b1 && b2 && b3 && b4 && b5 && b6 && b7 && b8;;
val test_summatorial : (int -> int) -> bool = <fun>
```

We implement the summatorial function as folllwing:

```
# let summatorial_gen n =
    sigma_rec (fun x -> x) n;;
val summatorial_gen : int -> int = <fun>
```

As the summatorial function calculates the first consective natural numbers, we use the identity function. We confirm this implementation passes the unit test.

```
# test_summatorial summatorial_gen;;
- : bool = true
```

However, since the unit-test function only considers cadidates up to 8, if the input is bigger than that, the unit-test function will not detect the error. For example,

```
# let fake_summatorial n =
    if n < 9
    then summatorial_gen n
    else -1

# test_summatorial fake_summatorial;;
- : bool = true
```

F. Revisit the function that sums the first consecutive odd natural numbers and express it using sigma.

From the .ml file given, we obtain the unit-test function:

```
# let test_sum_of_the_first_consecutive_odd_natural_numbers candidate =
    let b0 = (candidate 0 = 1)
    and b1 = (candidate 1 = 1 + 3)
    and b2 = (candidate 2 = 1 + 3 + 5)
    and b3 = (candidate 3 = 1 + 3 + 5 + 7)
    and b4 = (candidate 4 = 1 + 3 + 5 + 7 + 9)
    and b5 = (candidate 5 = 1 + 3 + 5 + 7 + 9 + 11)
    and b6 = (candidate 6 = 1 + 3 + 5 + 7 + 9 + 11 + 13)
    and b7 = (candidate 7 = 1 + 3 + 5 + 7 + 9 + 11 + 13 + 15)
    and b8 = (candidate 8 = 1 + 3 + 5 + 7 + 9 + 11 + 13 + 15 + 17)
    in b0 && b1 && b2 && b3 && b4 && b5 && b6 && b7 && b8;;
val test_sum_of_the_first_consecutive_odd_natural_numbers :
  (int -> int) -> bool = <fun>
```

Odd natural numbers can be expressed as 2k + 1 with an integer k. Therefore, using the sigma_rec function we already implemented, we can use fun x -> (2*x) + 1 to express odd numbers.

However, similar to the previous question, the coverage of the unit-test function is not great; it does not detect the error if the input is greater than 8. Hence we can implement a fake function.

```
# let sum_of_the_first_consecutive_odd_natural_numbers_rec n =
    sigma_rec (fun x -> (2*x) + 1) n;;
val sum_of_the_first_consecutive_odd_natural_numbers_rec : int -> int =
<fun>

# test_sum_of_the_first_consecutive_odd_natural_numbers
sum_of_the_first_consecutive_odd_natural_numbers_rec;;
- : bool = true

# let fake_sum_of_the_first_consective_odd_natural_numbers_rec n =
    if n < 9 then sum_of_the_first_consecutive_odd_natural_numbers_rec n
    else -1;;

# test_sum_of_the_first_consecutive_odd_natural_numbers
fake_sum_of_the_first_consecutive_odd_natural_numbers_rec;;
- : bool = true
```

Conclusion

In this exercise, we implemented summatorial function using recursive process. We have learned to be careful with the unit test coverage and made use of the nat_parafold_right function that we have implemented before to implement multiple recursive functions. We have seen that recursive functions can be simplified by breaking the process down to inductive steps. This is a good first step to dive deeper into the recursive functions in the following exercises.

Conclusion

In this mini-project, we have used generic functions that embody the programming pattern of structural recursion over natural numbers. We have reviewed the nat_of_digits function which we have implemented in Week 06 and re-implemented it using the nat_parafold_right function. Then, we implemented the sigma function using a recursive process. We have seen some limitations in the unit-test coverage. If the unit test only covers certain candidates, as we have seen in Ex07 E and F, the test would be weak because it would not detect any error that is not included in the uni-test function. Hence we have to be more careful and use random candidates using functions like Random.int or random_int (self-implemented), though we have to be mindful that these functions are still bounded by their upper bound.

“ Program testing can be used to show the presence of bugs,
but never to show their absence.”
- Dijkstra

The underlying determinism of OCaml

Introduction

In this mini-project, we are going to tackle the underlying determinism of Ocaml (as the title says…). We will look at the order of evaluation in Ocaml expressions and consider their equivalence. We will be touching on the concept of pure and impure functions, and how they affect the equivalence of OCaml expressions. This work has been done by N, J, and A.

Question 1

Introduction

In this question, we examine which way the operand (+) in addition is evaluated from in Ocaml.

Solution

```
# an_int 12 + an_int 20;;
processing 20...
processing 12...
- : int = 32
```

We can see that the operand evaluated addition from right to left.

Conclusion

Now that we know the operand of addition evaluates the expression from right to left, we can think about other evaluations of expressions.

Question 2

Introduction

In this question, we consider if the two sub-expressions (namely the one in position of a function, on the left, and the one that is the actual parameter of the function, i.e., the argument, on the right) are evaluated from left to right or from right to left.

Solution

```
let foo x = x + 3
let y = 5
```

First, we prepare the two sub-expressions, one that is a function and the other that would be the argument of the function. We observe, when applying this y to the function foo, how they are evaluated. We make use of the an_int function and a_function that are defined at the start of this mini-project.

```
# a_function foo (an_int y);;
processing 5...
processing a function...
- : int = 8
```

In fact, we can see that first the right parameter y is evaluated, and then the function foo is evaluated. This suggests that two sub-expressions are evaluated from right to left. This result is compatible with the earlier result, as the infix operand '+' is an alternative form of a function Int.add, and we can think about the earlier case as three sub-expressions – one that is a function, and the others that are parameters – being evaluated from right to left.

Conclusion

We have seen that two sub-expressions are evaluated from right to left. This will be important to unpack later questions. However, before that, we will look at one different type of expression.

Question 3

Introduction

In this question, we look at the order of evaluation of components for the construction of a tuple in OCaml.

Solution

We make use of the an_int function and construct a tuple. We can see that the components of the tuple are evaluated from right to left.

```
# (an_int 1, an_int 12, an_int 17);;
processing 17...
processing 12...
processing 1...
- : int * int * int = (1, 12, 17)
```

Conclusion

In this question, we constructed a tuple in OCaml and checked its order of evaluation. We have seen that OCaml evaluates three components of a tuple from right to left.

Question 4

Introduction

In this question, we consider the equivalence of expressions. We examine the order of evaluation of the expression's components and see if expressions are evaluated in the same order. Here are the two expressions we tackle in this question:

```
(fun x1 -> fun x2 -> e0) e1 e2

(fun (x1, x2) -> e0) (e1, e2)
```

Solution

To answer this question, we follow the format of week07 solution for Exercise 07.

1) Evaluating (fun x1 -> fun x2 -> e0) e1 e2 first involves evaluating e2 which yields v2 of type t2 if the evaluation completes. Then, having e2 evaluated, (fun x1 -> fun x2 -> e0) e1 is evaluated. This process first involves evaluating e1 that yields v1 of type t1, if the evaluation completes. Next, the function (fun x1 -> fun x2 -> e0) is evaluated and yields a value, if the evaluation completes. Then evaluating (fun x1 -> fun x2 -> e0) e1 reduces to applying the function (fun x1 -> fun x2 -> e0) to v1, which yields a function t2 -> e0, where t0 is the type of e0. Evaluating (fun x1 -> fun x2 -> e0) e1 e2 reduces to applying this function to v2 in e0.
2) Evaluating (fun (x1, x2) -> e0) (e1, e2) first involves evaluating (e1, e2). Since we know from question 3 that a pair (a tuple) is evaluated from right to left, firstly e2 is evaluated to yield v2 if the evaluation completes, and then e1 is evaluated to yield v1 if the evaluation completes. Then, we evaluate the function (fun (x1, x2) -> e0) and the function yield a value, if the evaluation completes. Next, evaluating (fun (x1, x2) -> e0) (e1, e2) reduces applying the function to v1 and v2 in e0.

In both expressions, e2 is evaluated first, then e1 is evaluated, and then e0 is evaluated. Therefore, we conclude that these two expressions are observationally equivalent.

This answer is compatible with the answers from question 02 and question 03, well, in fact, we have used the answer to question 2 and question 3 to solve the mystery. If we were to assume that e0, x1, x2 all have type int, the expression fun x1 -> fun x2 -> e0 has type int -> int -> int and the expression fun (x1, x2) -> e0 has type int * int -> int. The former type is what we are familiar with from some functions such as Int.add. The latter is also easy to construct one; one of the options could be a function that takes in two integer arguments and returns an integer value:

```
# let f (x,y) = x + y;;
val f : int * int -> int = <fun>
```

Conclusion

This question is the first step of determining the equivalence of the two expressions. We have learned that to show the equivalence we need complete proof rather than a mere example. We have considered the process of evaluation of an expression in the same manner as we construct a proof tree of it. In the coming questions, we will see more of this.

Question 5

Introduction

In this question, we will look at two expressions (again) and consider if evaluating them carry out the same computation, i.e., if the expressions are equivalent.

```
(fun x1 -> e0)  e1
let x1 = e1 in e0
```

Solution

1) Evaluating (fun x1 -> e0) e1 requires first evaluating e1. If evaluating e1 completes, it yields a value v1 of type t1. The evaluating (fun x1 -> e0) e1 requires evaluating (fun x1 -> e0) and it yields a value. Then evaluating (fun x1 -> e0) e1 reduces to applying this function to v1, which yields a function t1 -> t0 where t0 is the type of e0.
2) Evaluating let x1 = e1 in e0 requires first evaluating e1. If evaluating e1 completes, it yields a value v1 and type t1 and this will be assigned to x1. Then evaluating let x1 = e1 in e0 reduces to evaluating e0, which yields v0 of type t0.

   In both cases, e1 is first evaluated. If the evaluation completes, then e0 is evaluated. Therefore, we conclude that these expressions are observationally equivalent.

Conclusion

In this question, along the line of checking the equivalence of two expressions, we have looked at two observationally equivalent expressions; one that takes a form of a function, and the other uses the let-expression. We confirm that, though many expressions are evaluated from right to left in OCaml, by using a let-expression, we can control the direction of evaluation. Thinking about how the evaluation is operated in let-expression will become particularly important later when we tackle the recursive function.

Question 6

Introduction

In this question, we answer the following question: in a (local or global) let-expression declaring several bindings at once, i.e., with and, are the deficiencies evaluated from left to right or from right to left?

Solution

```
# let x = (an_int 10) and y = an_int(100) in x + y;;
processing 10...
processing 100...
- : int = 110
```

```
#
```

They are evaluated from left to right.

Conclusion

Along with the previous question, we confirmed that let-expression evaluates the parameters from left to right. This may come in handy when dealing with functions that take several bindings and when we want to control the order they are evaluated.

Question 7

Introduction

Given three expressions, e0, e1, and e2, we look at if the two following expressions are equivalent.

```
let x1 = e1 and x2 = e2 in e0

(fun (x1, x2) -> e0) (e1, e2)
```

Solution

1) From question 5, we know that let expression evaluates from left to right. And from question 6, we know that when let-expression has several bindings it evaluates from left to right. Combining these two, we observe that in the first expression, e1 is evaluated first; if the evaluation completes, then e2 is evaluated; if this evaluation completes, it finally evaluates e0.
2) This expression is the same expression we have seen in question 4. We know that the order of evaluation is the following: it first evaluates e2, and if it completes, then evaluates e1. If the evaluation completes, finally evaluates e0.

Therefore, the order of evaluation differs between the two expressions. We conclude that these two expressions are not observationally equivalent.

Conclusion

In this question, we have used answers to the previous questions to examine the order of evaluation in expressions. We find that two expressions do not carry out the same computation when evaluated. In the following exercise, we move on from let-expression and visit boolean evaluations.

Question 8

Introduction

The question we will be thinking about is the following: in boolean conjunction, are the conjuncts evaluated from left to right or right to left?

Solution

```
# a_bool (a_bool true && a_bool false);;
processing true...
processing false...
processing false...
- : bool = false
# a_bool (a_bool false && a_bool true);;
processing false...
processing false...
- : bool = false
# a_bool (a_bool true && a_bool true);;
processing true...
processing true...
processing true...
- : bool = true
# a_bool (a_bool false && a_bool false);;
processing false...
processing false...
- : bool = false
#
```

We see that the conjuncts are evaluated from left to right. This design may make sense due to the short-circuit evaluation. This evaluation examines the two arguments, and if the first argument for the boolean conjunction is false, then the latter argument does not need to be evaluated because if one of the arguments is false, boolean conjunction emits false.

However, as the truth table is symmetric (as in, whether OCaml evaluates the first argument or the second argument first should not really change the efficiency of the evaluation), there may be a better reason to evaluate from left to right.

Question 9

Introduction

In this question we look at pure expressions and potentially impure expressions and examine some statements about their equivalence.

Solution

1. For any pure expression v of type int, would it be valid to simplify v*0 into 0? -> Yes.

   For any expressions, multiplying 0 to an expression yields a 0. Since the expression is pure, there will be no observable side effects. 0 is also a pure expression. So v*0 and 0 can be equated to one another.

2. For any potentially impure expression e of type int, would it be valid to simplify e*0 into 0? -> No.

Unlike the previous statement, we consider each side carefully. On the one hand, 0 is a pure expression and it will not yield any observable side effects. On the other hand, a potentially impure expression e of type int could emit a trace, and thus they cannot be observationally equivalent.

3. For any pure expression v of type int, would it be valid to simplify *1 into v? -> Yes.

   1 is a multiplicative identity and a pure expression. Therefore, v*1 reduces to v. Since both v*1 and v are pure expressions, they are observationally equivalent.

4. For any potentially impure expression e of type int, would it be valid to simplify e*1 into e? -> yes.

   1 is a multiplicative identity and a pure expression, as stated above. For e*1, 1 will be evaluated first as Ocaml evaluates expressions from right to left. Then the expression e will be evaluated and may emit some observable side effects. Still, e*1 will return e as 1 is a multiplicative identity. This is identical to e, and therefore it is valid to simplify the expression e*1 to e.

Conclusion

In this question, we have looked at four statements and examined if they are true. If both expressions are pure and yield the same value for any given input, they are equivalent and can be simplified. If they are potentially impure, we also need to look at how the expression is evaluated. If the observable side effects are the same in two impure expressions, they are equivalent.

Question 10

Introduction

In this question, we let e1 and e2 be both potentially impure expressions and see two expressions are equivalent.

Solution

a) Are the two following expressions equivalent:

```
let x1 = e1 and x2 = e2 in (x1, x2);;

let x2 = e2 and x1 = e1 in (x1, x2);;
```

They are not equivalent. Since they are potentially impure expressions, we have to consider how each parameter is evaluated. As we have seen in question 6, when let-expression has several bindings, they are evaluated from left to right. Therefore, in the former expression, e1 is evaluated first, and if this evaluation completes, e2 is evaluated. On the other hand, in the latter expression, e2 is evaluated first, and if this evaluation completes, then e1 is evaluated. We see that the order of evaluation is different in the two expressions. Hence, they are not equivalent.

We can confirm this by using an_int function we previously defined:

```
# let x1 = an_int 3 and x2 = an_int 4 in (x1, x2);;
processing 3...
processing 4...
- : int * int = (3, 4)
# let x2 = an_int 4 and x1 = an_int 3 in (x1,x2);;
processing 4...
processing 3...
- : int * int = (3, 4)
```

b) Are the two following expressions equivalent:

```
let x1 = e1 in let x2 = e2 in (x1, x2);;

let x2 = e2 in let x1 = e1 in (x1, x2);;
```

Since e1 and e2 are both potentially impure expressions, we have to look at how the expressions are evaluated. From what we know from question 5, let-expression evaluates from left to right. So for the former expression, e1 is evaluated first, and if the evaluation completes, e2 is evaluated. For the latter expression, e2 is evaluated first, if the evaluation completes, e1 is evaluated. As the order of evaluation is different, we conclude that they are not equivalent.

We can confirm this by using an_int function we previously defined:

```
# let x1 = an_int 1 in let x2 = an_int 2 in (x1,x2);;
processing 1...
processing 2...
- : int * int = (1, 2)
# let x2 = an_int 2 in let x1 = an_int 1 in (x1,x2);;
processing 2...
processing 1...
- : int * int = (1, 2)
```

Conclusion

In this question, we have looked at two potentially impure expressions and examined the equivalence of two expressions. As they were impure, we have to look at the order of evaluation. We made use of what we have learned from previous questions about let-expressions. Next, we will look at pure expressions.

Question 11

Introduction

In this question, we let v1 and v2 be pure expressions and examine if two expressions are equivalent. Since they are pure expressions, if they are evaluated to the same result, we can conclude that they are equivalent.

Solution

a) Are the two following expressions equivalent:

```
let x1 = v1 and x2 = v2 in (x1, x2);;

let x2 = v2 and x1 = v1 in (x1, x2);;
```

Since both e1 and e2 are pure expressions, the main expressions are pure. As both expressions evaluate to (x1, x2), they are equivalent.

b) Are the two following expressions equivalent:

```
let x1 = v1 in let x2 = v2 in (x1, x2);;

let x2 = v2 in let x1 = v1 in (x1, x2);;
```

Since both e1 and e2 are pure expressions, the main expressions are pure. As both expressions evaluate to (x1, x2), they are equivalent.

Conclusion

In this question, we looked at expressions that contain two pure sub-expressions. Since they do not emit any observable side effects, if they emit the same result for any given input, we can conclude that they are equivalent. Both pairs of expressions we have seen are equivalent, as the outputs are all (x1, x2).

Question 12

Introduction

In mathematics, a function is said to be strict if it uses its argument and non-strict if it does not. For example, in the syntax of OCaml, fun x -> x is such a strict function, and fun _ -> 42 is such a non-strict function. So in mathematics, applying this non-strict function to any argument yields 42, always. Is that the case in OCaml too?

Solution

In OCaml evaluates expressions from right to left. This means that the parameters are evaluated before evaluating the function itself. So if the parameters included an expression that does not terminate, the function would not be evaluated. The function cannot return, 42 in the example above. Therefore, in OCaml, the non-strict function does not necessarily yield the same value every time it runs.

Conclusion

In this question, we have considered a non-strict function. We have only dealt with terminating expressions in the previous questions, but we have shown what happens if one of the parameters is a non-terminating expression. In the question's context, a non-strict function in mathematics sense cannot yield the same value for any given input, as the input can not terminate.

Conclusion

In this mini-project, we have looked at the order of evaluation of both pure and potentially impure expressions. We have seen that for many of OCaml's expressions, the parameters are evaluated from right to left, as long as the order is not specifically instructed by let-expression. We have also seen how to determine the equivalence of expressions when the expressions are pure/impure. When the expressions are pure, if the expressions return the identical result for any given input, we can conclude that they are equivalent. If they are potentially impure, meaning that they may emit observable side effects or traces, we have to look into the order of evaluation to make sure that they carry out the same computation. This mini-project is very important because it elaborates on the definition of observational equivalence which will appear a few more times in the later mini-project.

Palindromes, string concatenation, and string reversal

Introduction

In this mini-project, we will play around with the strings. We try to concatenate and reverse the string and make palindromes, using recursive processing. We will learn how to access each element of the given string, how to use various String built-in functions, and most importantly, how to follow the flow of evaluation in a function. This mini-project has been done by S and A.

Question 1

Introduction

In this question we implement an OCaml function of type string -> string -> string that concatenates two strings, using String.init.

Solution

First, let us implement a unit-test function. Since this is string concatenation, we use some examples, a corner case (empty string), and random strings.

```
let test_string_concatenation candidate =
let b0 = (candidate  "abc" "bcd" = "abcbcd")
and b1 = (candidate "123" "456" = "123456")
and b2 = (candidate "123" "abc" = "123abc")
and b3 = (candidate "12a" "Univer4" = "12aUniver4")
and br = (let s1 = random_string (Random.int 10)
          and s2 = random_string (Random.int 10)
          in candidate s1 s2 = s1^s2)
and bb = (candidate "" "" = "")
in b0 && b1 && b2 && b3 && br && bb
```

Now that we are ready, let us implement the concatenating function. The function we use is String.init. It takes two arguments; the first argument is the number of times it operates on, and the second argument is the function that is operated. Since we want to make a concatenated string, the first argument shall be the length of the desired string, which is the combined length of two original strings. The second argument shall return the elements of the first string if the index is still within the length of the first string, and return the elements of the second string if the index is beyond the length of the first string. The actual implementation follows:

```
let string_append s1 s2 =
  String.init
    (String.length (s1^s2))
    (fun i -> if i < String.length s1
              then String.get s1 i
              else String.get s2 (i - String.length s1));;
val string_append : string -> string -> string = <fun>
```

This implementation has type string -> string -> string. And it passes the unit test.

```
# test_string_concatenation string_append;;
- : bool = true
```

Conclusion

This is our first exposure to String.init function. This function resembles the String,mapi function, but differs as init creates a new string whereas mapi operates on the pre-existing string. So if we want to get a string that has the same length as the original string but is altered by a function, mapi is enough. But since we are trying to make a new function whose length is longer than a pre-existing string, we should use init. In the following questions, we deepen our understanding of string concatenation.

Question 3

Introduction
In this question, we look at the OCaml's built-in function String.map and see how the function is operated on each character in the given string.

Solution

a. In which order are the characters accessed in the given string?

```
#  String.map (fun c -> char_of_int(an_int(int_of_char c))) "abcde";;
processing 97...
processing 98...
processing 99...
processing 100...
processing 101...
- : string = "abcde"
```

We can check the order of operation by using an_int function that is previously defined. We see that String.map accesses the string from the smallest index to the largest index, i.e., left to right.

Before moving on to the next part of this question, let us implement a unit-test function for String.map.

```
let test_string_map candidate =
  let b0 = (candidate (fun c -> char_of_int (int_of_char c + 1)) "abc" =
"bcd")
  and b1 = (candidate (fun c -> 'a') "12345" = "aaaaa")
  and b2 = (candidate (fun c -> char_of_int (int_of_char c - 1)) "bcd" =
String.map (fun c -> char_of_int (int_of_char c - 1)) "bcd")
  and b3 = (candidate (fun c -> char_of_int (int_of_char c - 1)) "" = "")
  in b0 && b1 && b2 && b3;;
```

b. Using a recursive function that operates over a natural number, implement a left-to-right version of String.map.

Let us see the implementation first.

```
# let string_map_up f s =
    let n = String.length s
    in let () = assert (n >= 0) in
       if n = 0
       then ""
       else let rec visit i =
           if i = 0
           then String.make 1 (f(String.get s 0))
           else let i' = pred i
                in let ih = visit i'
                   in ih ^ (String.make 1 (f(String.get s i)))
  in visit (n-1)
```

Though the recursive function will operate on the natural number in our case, for dealing with the empty string, we start with when String.length returns a zero. In that case, no further operation is necessary, and the function should return an empty string.
Having a non-empty string, we move on to the recursive step. In the last call, the function visit n-1, which is the last element of the string. When evaluating the last element, it also considers the ih, which calls the predecessor of the last element, whose ih contains a call for the predecessor of the element, which recursively calls till to the first element of the string. Once the recursive process reaches the first element of the string, it starts evaluating the ih - going back to the last element of the string. Therefore, the evaluation is from the smallest index to the largest index, i.e., left to right. And this implementation passes the unit test.

```
# test_string_map string_map_up;;
- : bool = true
```

c. Using a recursive function that operates over a natural number, implement a right-to-left version of String.map.

```
# let string_map_down f s =
   let n = String.length s
   in let () = assert (n >= 0)
      in let rec visit i =
         if i = 0
         then  ""
         else let i' = pred i
              in let ih = visit i'
                 in (String.make 1 (f (String.get s (n - i))) ^ ih)
   in visit n;;
val string_map_down : (char -> char) -> string -> string = <fun>
```

Let us see how this function operates. In the function, first, n is assigned to the length of the string. We go with the assumption that n is non-negative and move on to the recursive steps. The last call tells us to visit n. So let's visit n. In the second last line, we see n-i, in this case, n-n = 0. So the first part of the second last line is accessing the first element of the string. Now, what about the ih part? If we look at how ih is defined in the previous line. ih , is a visit call to the predecessor of i, which in this case, n-1. When we substitute i = n-1, we realize we are accessing the element of index 1 of the given string, as n - (n-1) = 1. We keep doing this process until i = 0, and for this, we return an empty string because index n is out of bound. Now, having accessed to the very end of the recursive call, we can finally evaluate them all, starting from i = 0, going back to i = n. This implementation accesses from largest to smallest index, i.e., from right to left. And it passes the unit test.

```
# test_string_map string_map_down;;
- : bool = true
```

Just like how we did the first part of this question, we can confirm our implementations are correct using the same arguments:

```
# string_map_up (fun c -> char_of_int(an_int(int_of_char c))) "abcde";;
processing 97...
processing 98...
processing 99...
processing 100...
processing 101...
- : string = "abcde"
# string_map_down (fun c -> char_of_int(an_int(int_of_char c))) "abcde";;
processing 101...
processing 100...
processing 99...
processing 98...
processing 97...
- : string = "abcde"
```

Conclusion

In this question, we considered the OCaml built-in String.map function. We have shown that String.map operates from the smallest to the largest index, i.e., left to right. We have used a recursive function that operates on natural numbers to implement our own version of the String.map. When doing so, it was crucial to think about which element of the given string we are accessing using the visit call, how the call starts, and where it ends. We have learned that by using different ways of accessing the indices of the given string, we can implement both the left-to-right and right-to-left versions of the String.map function.

Question 4

Introduction

In this question, along the line of the previous question, we are going to implement our own version of String.mapi function. What is different this time is that String.mapi takes indices as an argument, meaning that it can access a specific element of the given string, whereas String.map applied the function to all elements of the given string. This task is optional, and has been done by A and S, separately, combined later together.

Solution

First, let us start with the unit-test function, as usual.

```
let test_stringmapi candidate =
  let b1 = (candidate (fun i _ -> 'a') "abcde" =
            String.mapi (fun i _ -> 'a') "abcde")
  and b2 = (candidate (fun i _ -> char_of_int (i+48)) "gondor" =
            String.mapi (fun i _ -> char_of_int (i+48)) "gondor")
  and b3 = (candidate (fun i _ -> '%') "" =
            String.mapi (fun i _ -> '%') "")
  and b4 = (candidate (fun i c -> if i = 3 then '-' else c) "test" =
            String.mapi (fun i c -> if i = 3 then '-' else c) "test")
  and b4r = (let s = random_string 100
             in candidate (fun i c -> if i = 57 then '$' else c) s =
             String.mapi (fun i c -> if i = 57 then '$' else c) s)
  in b1 && b2 && b3 && b4 && b4r;;
```

a. In which order ar ethe character accessed in the given string?

```
# String.mapi (fun i _ -> char_of_int (an_int i)) "abcde";;
processing 0...
processing 1...
processing 2...
processing 3...
processing 4...
- : string = "\000\001\002\003\004"
```

We can see that String.mapi accesses from the smallest index to the largest index, i.e., from left to right.

b. Using a recursive function that operates over a natural number, implement a left-to-right version of String.mapi.

```
# let string_mapi_up f s =
    let n = String.length s
    in if n = 0
       then ""
       else let rec visit i  =
              if i = 0
              then String.make 1 (f 0 (String.get s 0))
              else let i' = pred i
                   in let ih = visit i'
                      in ih ^ String.make 1 (f i (String.get s i))
            in visit (n-1);;
val string_mapi_up : (int -> char -> char) -> string -> string = <fun>
```

This is quite straightforward, as it is quite similar to what we did for String.map. The only change made was that the index was added to the argument. We will show an example later to show that this implementation operates from left to right.

c. Using a recursive function that operates over a natural number, implement a right-to-left version of String.mapi.

```
# let string_mapi_down f s =
    (* Right to left *)
    let n = String.length s in
    if n = 0 then ""
    else let rec visit i =
           if i = 0
           then String.make 1 (f (n-1) (String.get s (n-1)))
           else  let i' = pred i
                 in let ih = visit i'
                    in String.make 1 (f (n-1-i) (String.get s (n-1-i))) ^ ih
         in visit (n-1);;
val string_mapi_down : (int -> char -> char) -> string -> string = <fun>
```

Well, there is no explanation needed for the first half of the function, as it is the same step as the previous questions. The last call tells us to visit n-1. Let us follow that. Since the argument is n-1-i, when i = n-1, this becomes 0. Here, we realize again that we are accessing to the first element of the given string. In the ih, it calls to the predecessor of i, which in this case is n-2. For i = n-2, n-1-i yields 1, and indeed we are accessing the next element of the string in the recursive step. When i reaches 0, we are accessing the last element of the string of index n-1. Since we reached the end of the recursive process, evaluation can start. The evaluation starts from the last element of the string, going back to the very first element of the string.

Two implementations above pass the unit-test function.

```
# test_string_mapi string_mapi_up;;
- : bool = true
# test_string_mapi string_mapi_down;;
- : bool = true
```

And we can show that our implementations are correct using the same arguments we used for part a.

```
# string_mapi_up (fun i _ -> char_of_int (an_int i)) "abcde";;
processing 0...
processing 1...
processing 2...
processing 3...
processing 4...
- : string = "\000\001\002\003\004"
# string_mapi_down (fun i _ -> char_of_int (an_int i)) "abcde";;
processing 4...
processing 3...
processing 2...
processing 1...
processing 0...
- : string = "\000\001\002\003\004"
```

Conclusion

In this question, similar to the previous question, we have implemented our own version of String.mapi using recursive function. Though String.mapi differs from String.map as it takes another argument, namely index, the implementation is actually quite similar. Again we have learned to be careful with how we define the ih and how the recursive calls will be carried out by visit.

Question 5

Introduction

In this question, we implement two Ocaml functions of type string -> string that reverse a string. The unit test function that our implementation should pass follows:

```
let test_string_reverse candidate =
  let b1 = (candidate "abcde" = "edcba")
  and b2 = (candidate "" = "")
  and b3 = (candidate "abba" = "abba")
  and b4 = (candidate "sauron" = "noruas")
  and br = (let c0 = random_char ()
            and c1 = random_char ()
            and c2 = random_char ()
            in candidate (warmup c0 c1 c2) = (warmup c2 c1 c0))
in b1 && b2 && b3 && b4 && br;;
```

Solution

a. A function that uses String.mapi

```
let string_reverse_mapi s =
    String.mapi (fun i _ -> String.get s ((String.length s)-i-1) ) s;;
val string_reverse_mapi : string -> string = <fun>
```

```
# test_string_reverse string_reverse;;
- : bool = true
```

We make String.mapi to access from the last element to the first element of the given string. For example, when i = 0, ((String.length s) - i - 1) would yield n - 1, where n is the length of the string. That is the last element of the string, and as i increases by 1, the index decreases by 1. Therefore, we can access from the last element to the first, making a reversed version of the same string. We make sure this implementation passes the unit test.

b. A function that uses recursion over a non-negative integer

```
# let string_reverse_rec s =
      let n = String.length s
      in let () = assert (n >= 0)
         in if n = 0
            then ""
            else let rec visit i =
                 if i = 0
                 then String.make 1 (String.get s (n-1))
                 else let i' = pred i
                      in let ih = visit i'
                         in ih ^ (String.make 1 (String.get s
((String.length s)-i-1)))
    in visit (n-1) ;;
val string_reverse_rec : string -> string = <fun>
```

```
# test_string_reverse string_reverse_rec;;
- : bool = true
```

Let us look at this recursive implementation of the string_reverse. First, since the recursinon operates over a non-negative integer, we have to be mindful about an empty string. We put an if-else statement for this corner case, making the function return the same empty string. For the next step, as showin in the earlier part of this exercise, ((String.length s)-i-1)) helps us to access to the last element of the string when i = 0, and the first element when i = n-1. This time, the visit call starts from n-1. As we can see in the ih definition, visit calls to the predecessor of i, enabling it to access the next element of the given string. In the end, we will be accessing to the last element of the string. Now the recursion reached the end, we can evaluate the ih. Ih evaluates from the last element to the first element, constructing a reversed version of the given string. This implementation passes the unit-test function.

Can we express our recursive string_reevrsing function using nat_fold_right?

- No, we cannot use nat_fold_right because that function only takes in one argument, namely ih, in the function succ_case. However, in our own implementation, we need both the ih and the string to which the ih will be added to. We could implement this using nat_parafold_right, however.

Conclusion

In this question, we have looked at how to reverse a string. We implemented this reverse function in two ways, one that uses String.mapi and the other that uses a recursive function. For the recursive function, it is important to be mindful of how the corner case is handled - such as an empty string.

Question 6

Introduction
For any sting denoted by s1 and s2, what is the relation between string_reverse s1 and string_reverse s2, and s1^s2?

Solution
s1^s2 is a concatenated string. If we reverse this concatenated string, this string will be equivalent to the concatenated (string_reverse s2) ^ (string_reverse s1).

```
let unit_test_string_reverse_for_question_5 candidate =
let b0 = (candidate "" "" = (string_reverse_mapi "")^(string_reverse_mapi
""))
and b1 = (candidate "abc" "123" = (string_reverse_mapi
"123")^(string_reverse_mapi "abc"))
and b2 = (candidate "kingfisher" "yalenus" = (string_reverse_mapi
"yalenus")^(string_reverse_mapi "kingfisher"))
and b3 = (candidate "1234" "4321" = (string_reverse_mapi
"4321")^(string_reverse_mapi "1234"))
and br = (let s1 = random_string (Random.int 10)
          and s2 = random_string (Random.int 10)
          in candidate s1 s2 = (string_reverse_mapi
s2)^(string_reverse_mapi s1))
in b0 && b1 && b2 && b3 && br;;
```

Along with some normal examples, we have implemented empty strings and random strings to strengthen the unit-test coverage.

Conclusion
In this question, we considered the relation between concatenated strings and reversed strings. We found that if we reverse a concatenated string, it will be equivalent to the concatenated two reversed strings, but the order is reversed. It is because when the string is reversed, the first element becomes last, so when considering two strings, the first strings will be placed in the later half of the latter string.

Question 7

Introduction

In this question, we implement a generator of palindromes of type int -> string, i.e., an OCaml function that maps a non-negative integer n to a palindrome of length n.

Solution

```
let make_palindrome n =
  let s = String.init (n / 2) (fun i -> random_char ())
  in if n mod 2 = 0
     then s ^ (string_reverse_mapi s)
     else s ^ "x" ^ (string_reverse_mapi s);;
```

Our strategy is following:

1) Check the length of the desired string, i.e., n. If n is an even number, the desired palindrome can be split into two, and the second half should be a reversed version of the first half.
2) If n is an odd number, we still split the palindrome, and add one character in between a string, and a reversed version of itself.

We can show this works with couple of examples. Shown below is three scenario, one when n is even, one when n is odd, and one when n is 0.

```
# make_palindrome 10;;
- : string = "(,1\"11\"1,("
# make_palindrome 0;;
- : string = ""
# make_palindrome 5;;
- : string = "9>x>9"
```

Conclusion

In this question, we implemente a generator of a palindrome. Along the line of previous questions, we made use of the string_reverse functions to make palindromes. Now that we can make palindromes, we will be implementing functions that can detect palindromes.

Question 8

Introduction

In this question, we implement a palindrome detector of type string -> bool, such that applying it to a palindrome yields true whereas applying it to a string that is not a palindrome yields false.

Solution

First, let us prepare a unit-test function:

```
let test_palindrome_detector candidate =
let b0 = (candidate "abcdefedcba" = true)
and b1 = (candidate "a" = true)
and b2 = (candidate "aaaaaaaaa" = true)
and b3 = (candidate "123454321" = true)
and b4 = (candidate "ab" = false)
and b5 = (candidate "yale" = false)
and b6 = (candidate "nus" = false)
and b7 = (candidate "" = true)
and br = (let p = make_palindrome 5
          in candidate p = true)
and bp = (let q = make_palindrome 6
          in candidate q = true)
in b0 && b1 && b2 && b3 && b4 && b5 && b6 && b7 && br && bp
```

a. Implement a palindrome detector using String.mapi

This one is simple. We reverse the given string using string_reverse_mapi that we have previously implemented, and check if the reversed string is the same as the given string. If true, then it is a palindrome, if not, then it is not.

```
# let palindromep_mapi s = string_reverse_mapi s = s;;
val palindromep_mapi : string -> bool = <fun>

# test_palindrome_detector palindromep_mapi;;
- : bool = true
```

b. Implement a palindrome detector using recursion over a non-negative integer.

This implementation makes use of the made-up conjunction function from the previous mini-project. We look at elements of the string from both ends, and if they match, it returns a true. Since it is conjunction, if any one pair of the elements are not same, the overall result will be false. We only have to start from (n-1)/2 because once we check the half of the string, we also have checked the other half of the string already.

```
# let palindromep_rec s =
    let is_palindrome = true
    in let n = String.length s
       in if n = 0
          then true
          else let () = assert (n > 0)
               in let rec visit i =
                    if i = 0
                    then is_palindrome = (String.get s 0 = String.get s
(n-1))
```

```
                   else let i' = pred i
                        in let ih = visit i'
                           in is_palindrome = conjunction_gen ih
(String.get s i = String.get s (n-1-i))
    in visit (n/2) ;;
val palindromep_rec : string -> bool = <fun>

# test_palindrome_detector palindromep_rec;;
- : bool = true
```

Conclusion

In this question, we have implemented two function that detect palindromes. The first implementation is rather simple, we made use of the string_reverse function we implemented and checked if the reversed string is still the same as the original string. The second one, a rather complex one, uses a recursive function and compares elements of the given string from both ends. The function uses the made-up conjunction function from the More functional abstraction.

Question 9

Introduction

Implement an Ocaml function that reverses a palindrome.

Solution

Let us think…a reversed palindrome is…still the same palindrome.

```
let reverse_palindrome s = s
```

Conclusion

This question says it is NOT OPTIONAL whereas other questions that are mandatory do not say they are mandatory…why is it not optional? The answer is right above us.

"First, solve the problem. Then, write the code"
- *John Johnson*

Question 12

Introduction

In this question, we implement a function that, given a predicate and a string, applies the function to each character of the string and returns the concatenation of the result.

Solution

a. Implement a function string_andmap that has type (char -> bool) -> string -> bool and that, given a predicate and a string, applies this predicate to each character of the string and returns the conjunction of the results.

The implementation should pass the following unit test.

```
let digitp c =
  '0' <= c && c <= '9';;

let test_string_andmap candidate =
  (candidate digitp "" = true) &&
  (candidate digitp "123" = true) &&
  (candidate digitp "abc" = false) &&
  (candidate digitp "123abc" = false);;
```

First, we can make use of the made-up conjunction function from the first mini-project. We can prepare a ‘flag’ - like a place holder for the boolean, and update it eerie time the function returns a boolean. In this case, since we want to have conjunction, we set the first flag to be true. If any of the following recursive calls return a false, the flag will also be false.

```
# let string_andmap predicate s =
  let is_true = true
  in let n = String.length s
      in if n = 0
         then is_true
         else let () = assert (n > 0)
              in let rec visit i =
                 if i = 0
                 then is_true = predicate (String.get s 0)
                 else let i' = pred i
                      in let ih = visit i'
                         in is_true= conjunction_gen ih (predicate
(String.get s i))
    in visit (n-1);;

# test_string_andmap string_andmap;;
- : bool = true
```

And my groupmate pointed out that this does not necessarily need a flag. Also, we do not have to use our own version of conjugation, since there is one bult-in function for this. So below is the more simplified version of string_andmap. Both implemetatin passes the unit test.

```
let string_andmap predicate s =
 let n = String.length s
     in if n = 0 then true
        else let rec visit i =
               if i = 0
               then predicate (String.get s 0)
               else let i' = pred i
                    in let ih = visit i'
                       in ih && predicate (String.get s i)
             in visit (n-1);;

# test_string_andmap string_andmap;;
- : bool = true
```

However, as the unit-test function only has digitp as a candidate function and test certain strings, the coverage is not great - and can be cheated :

```
let fake_string_andmap predicate s =
  if s = "onlydigits"
  then true
  else string_andmap predicate s;;

# test_string_andmap fake_string_andmap;;
- : bool = true
```

b. Implement a function string_ormap that has type (char -> bool) -> string -> bool and that, given a predicate and a string, applies this predicate to each character of the string and returns the disjunction of the results.

   The implementation should pass the following unit test.

```
let test_string_ormap candidate =
  (candidate digitp "" = false) &&
  (candidate digitp "123" = true) &&
  (candidate digitp "abc" = false) &&
  (candidate digitp "123abc" = true);;
```

This time, we no longer use a flag. Now that we want the disjunction of the result, we use the built-in expression of the disjunction, ||. Only if both booleans (ih and i-th element's predicate) are false, the function returns a false.

```
let string_ormap predicate s =
   let n = String.length s
     in if n = 0 then false
```

```
        else let rec visit i =
               if i = 0
               then predicate (String.get s 0)
               else let i' = pred i
                    in let ih = visit i'
                       in ih ||  predicate (String.get s i)
             in visit (n-1);;
```

c. What is the logic of making string_andmap return true when applied to the empty string, and of making string_ormap return false when applied to the empty string?

   This is for the case when we have to concatenate the strings. For disjunction, having an empty string equal to false does not affect otherwise true strings. For conjunction, having an empty string equal to true does not affect otherwise true strings.

Conclusion

In this question, we have implemented a function that applies a function to each element of a string and returns conjunction/disjunction of the result. We have defined what to do with an empty string so that it does not affect otherwise true strings.

Conclusion

In this mini-project, we have implemented functions that concatenate strings, reverse strings, make/detect palindromes, and apply a function to each element of strings. Through these questions, we have familiarized ourselves with the strings and accessing elements of the string. Also, the frequent use of recursive functions helped us gain confidence in the recursive process and deepened our understanding of how recursion is operated in a function. We have found that there are multiple ways of solving the same question, but we should prioritize simpler answers, as it will be more reader-friendly (including the future myself).

"Simplicity is the soul of efficiency."
- *Austin Freeman*

Cracking polynomial functions

Introduction

In this mini-project, we implement a function that can compute the coefficients of a given polynomial function. We implement a polynomial function cracker till the fourth degree and consider forms of polynomials other than the standard form (e.g., Horner form). This project has been done by S, a.k.a, myself.

Second-degree

Introduction

Since the answer to the first-degree cracker is given in the lecture note, we start with the second-degree polynomials. We define make_polynomial_2 function that takes three coefficients a2, a1, a0 of type int and returns a polynomial function of degree 2.

```
let make_polynomial_2 a2 a1 a0 =
  fun x -> a2 * x * x + a1 * x + a0;;
```

Our goal for this task is to implement a function crack_2 that, given a polynomial function of degree 2, can crack the polynomial and returns the three coefficients.

Solution

Below is the unit-test function that will be used.

```
let test_crack_2_once candidate =
  let a2 = random_int ()
  and a1 = random_int ()
  and a0 = random_int ()
  in let p2 = make_polynomial_2 a2 a1 a0
     in let expected_result = (a2, a1, a0)
        and ((a2', a1', a0') as actual_result) = candidate p2
        in if actual_result = expected_result
           then true
           else let () = Printf.printf
                           "%s\ntest_crack_2_once failed for %i * x^2 +
%i * x^1 + %i\nwith %i instead of %i, %i instead of %i, and %i instead of
%i\n"
                           (Array.get commiseration (Random.int
(Array.length commiseration)))
                           a2 a1 a0 a2' a2 a1' a1 a0' a0
                in false;;
```

First, let us express each coefficients using the given polynomial function of degree 2. For visual clarity, I used a, b, and c for a2, a1, and a0.

$$f(x) = ax^2 + bx + c$$

where $a_0 = c$
$a_1 = b$
$a_2 = a$

$$f(0) = c$$
$$f(1) = a + b + c$$
$$f(-1) = a - b + c$$
$$\Rightarrow a+b = f(1) - f(0), \quad a-b = f(-1) - f(0)$$

let $\text{a\_plus\_b} = f(1) - f(0)$

and $\text{a\_minus\_b} = f(-1) - f(0)$

then $\dfrac{\text{a\_plus\_b} + \text{a\_minus\_b}}{2} = \dfrac{(a+b)+(a-b)}{2} = a$

and $b = \text{a\_plus\_b} - a$

Therefore $a_2 = a = \dfrac{\text{a\_plus\_b} + \text{a\_minus\_b}}{2}$
$a_1 = b = \text{a\_plus\_b} - a$
$a_0 = c = f(0)$

Hence, we can use this to implement the crack_2, like below:

```
let crack_2 p =
  let p_0 = p 0
  and p_1 = p 1
  and p_m1 = p (-1)
  in let a0 = p_0
     and a2_plus_a1 = p_1 - p_0
     and a2_minus_a1 = p_m1 - p_0
     in let a2 = (a2_plus_a1 + a2_minus_a1) / 2
        in let a1 = a2_plus_a1 - a2
           in (a2,a1,a0)
```

As point out in the Solution for the first-degree task, applying the polynomial function to 0 yields a0. And as we have two more unknown parameters, namely a1 and a2, we need two more applied functions. Thus we prepare p_1 and p_m1 and use them to compute the rest of the coefficients.

And this implementation passes the unit test.

```
let () = assert (test_crack_2 crack_2)
```

Now we think about the subsidiary question - can crack_2 be used to implement crack_1 - which is the cracker of a polynomial function of degree 1. If we think about it, a polynomial function of degree of 1 has the same structure as the polynomial function of degree 2, just that the first coefficient (a2) is 0. Therefore, we can use crack_2 to compute three coefficients, the first of which is 0, and return only the last two coefficients. The implementation follows:

```
let crack_1_alt p =
  let (a2, a1, a0) = crack_2 p
  in (a1, a0)
```

This implementation passes the unit test.

```
let () = assert (test_crack_1 crack_1_alt);;
```

Conclusion

In this task, we implemented a cracker for polynomial function of degree 2. We have built the basis for what to come - cracker for polynomial function of degree 3 and 4. The way we implemented crack_1 using crack_2 will also be repeatedly used in the later tasks. The perspective that polynomial functions with lesser degree are just polynomial functions with many coefficients of 0 will become handy.

Zeroth degree

Introduction

But before we get into the tiring ones - let;s take a break and implement three distinct OCmal functions crack_0 that, given a polynomial function of degree 0, returns its coefficient.

Here is the unit-test function our implementation shold pass.

```
let test_crack_0_once candidate =
  let a0 = random_int ()
  in let p0 = make_polynomial_0 a0
     in let expected_result = a0
        and (a0' as actual_result) = candidate p0
        in if actual_result = expected_result
           then true
           else let () = Printf.printf
```

```
                            "%s\ntest_crack_0_once failed for %i\nwith %i
instead of %i\n"
                            (Array.get commiseration (Random.int
(Array.length commiseration)))
                            a0 a0' a0'
                in false;;

let test_crack_0 candidate =
  test_crack_0_once candidate && test_crack_0_once candidate &&
test_crack_0_once candidate;;
```

Solution

1) Since polynomial functions with degree 0 is a constant, we can substitute any x as an input. The function should return a constant. We could make use of Random.int function, but to keep it simple, let us substitute 0 and get the output. And it passes the unit test.

```
let crack_0a p = p 0;;

let () = assert (test_crack_0 crack_0a);;
```

2) We can make use of the crak_2 function we already implemented. The logic is the same as the previous question. The first two coefficients will be 0, so just return the last coefficient. And it passes the unit test.

```
let crack_0b p =
  let (a2, a1, a0) = crack_2 p
  in a0;;

let () = assert (test_crack_0 crack_0b);;
```

3) We can also use crack_1 function, which is copied from question 1 of this mini-project.

```
let crack_0c p =
  let (a1, a0) = crack_1 p
  in a0;;

let () = assert (test_crack_0 crack_0c);;
```

Conclusion

In thie task, we have implemented polynomial functions with degree zero in three different ways. Since these functions are merely a constant, we can make use of polynomial crackers that we already implemented and make them return only the last coefficient.

Third-degree

Introduction

Along the line of implementing the cracker for polynomial functions of degree 2, we implement a function crack_3 that, given a polynomial function of degree 3, return the four coefficients, a3,a2, a1, and a0.

The unit-test function that our implementation should pass follows:

```
let test_crack_3_once candidate =
  let a3 = random_int ()
  and a2 = random_int ()
  and a1 = random_int ()
  and a0 = random_int ()
  in let p3 = make_polynomial_3 a3 a2 a1 a0
     in let expected_result = (a3, a2, a1, a0)
        and ((a3', a2', a1', a0') as actual_result) = candidate p3
        in if actual_result = expected_result
           then true
           else let () = Printf.printf
                           "%s\ntest_crack_3_once failed for %i * x^3 +
%i * x^2 + %i * x^1 + %i\nwith %i instead of %i, %i instead of %i, %i
instead of %i, and %i instead of %i\n"
                           (Array.get commiseration (Random.int
(Array.length commiseration)))
                           a3 a2 a1 a0 a3' a3 a2' a2 a1' a1 a0' a0
                in false;;

let test_crack_3 candidate =
  test_crack_3_once candidate && test_crack_3_once candidate &&
test_crack_3_once candidate;;
```

Solution

First, let us use pen and paper to express each coefficients using a given polynomial function of degree 3. For clarity, a3, a2,a1, and a0 are replaced with a,b,c, and d.

$P(x) = ax^3 + bx^2 + cx + d$

where $a = a_3$, $b = a_2$, $c = a_1$, and $d = a_0$

$P(0) = d$ ... ①

$P(1) = a + b + c + d$ ... ②

$P(-1) = -a + b - c + d$ ... ③

$P(2) = 8a + 4b + 2c + d$ ... ④

② + ③ − ① × 2

$\Rightarrow (a+b+c+d) + (-a+b-c+d) - 2d$

$= 2b + 2d - 2d = 2b$

Therefore, $b = \frac{1}{2}(P(1) + P(-1) - 2P(0))$

Now, $a + c = P(1) - b - d$ ... ⑤

and $8a + 2c = P(2) - 4b - d$

$\Leftrightarrow 4a + c = \frac{1}{2}(P(2) - 4b - d)$ ... ⑥

⑥ − ⑤

$3a = \frac{1}{2}(P(2) - 4b - d) - \{P(1) - b - d\}$

$= \frac{P(2) - 4b - d - 2P(1) + 2b + 2d}{2}$

$= \frac{P(2) - 2P(1) - 2b + d}{2}$

$\Leftrightarrow a = \frac{P(2) - 2P(1) - 2b + d}{6}$

$\Rightarrow c = P(1) - a - b - d$

Highlighted in yellow are the coefficients. We can use these definitions to implement crack_3 and check if the implementation passes the unit test.

```
let crack_3 p =
  let p_0 = p 0
  and p_1 = p 1
  and p_2 = p 2
  and p_m1 = p (-1)
  in let a0 = p_0
     and a2 = (p_1 + p_m1 - (p_0 * 2)) / 2
     in let a3 = (p_2 - (2 * a2) + a0 - (2 * p_1)) / 6
        in let a1 = p_1 - a0 - a2 - a3
           in (a3, a2, a1, a0);;


let () = assert (test_crack_3 crack_3);;
```

Then, we tackle the subsidiary questions:

1) Can crack_3 be used to implement crack_2?

   Yes. The process is the same as when we implemented crack_1 using crack_2. We calculate the coefficients using crack_3, and make the function return only the needed coefficients. We make sure the implementation passes the appropriate unit test.

```
(* Can crack_3 be used to implement crack_2? *)
let crack_2_alt_v1 p =
  let (a3, a2, a1, a0) =  crack_3 p
  in (a2, a1, a0)

let () = assert (test_crack_2 crack_2_alt_v1)
```

2) Can crack_3 be used to implement crack_1?

   Yes. We just need the last two coefficients, as the first two are zeros. And it passes the unit test.

```
(* Can crack_3 be used to implement crack_1?*)
let crack_1_alt_v2 p  =
  let (a3, a2, a1, a0) = crack_3 p
  in (a1, a0)

let () = assert (test_crack_1 crack_1_alt_v2)
```

3) Can crack_3 be used to implement crack_0?

   Yes. Let the function return only the last coefficient. We make sure it passes the unit test.

```
(* Can crack_3 be used to implement crack_0? *)
let crack_0d p =
  let (a3, a2, a1, a0) = crack_3 p
  in a0

let () = assert (test_crack_0 crack_0d)
```

Conclusion

In this task, we dealt with polynomial function of degree 3. What we do is very similar to the previous question about polynomial functions of degree 2, but a little more complexed. In the next task, we will further complicate the implementation, by trying to crack the polynomial functions with degree 4.

Fourth degree

Introduction

In this task, we implement an OCaml function crak_4 of type (int -> int) -> int * int * int * int * int that, given a polynomial function of degree 4, return its 5 coefficients - a4, a3, a2, a1, and a0. In the previous questions, maker of polynomial functions and the unit-test functions were given. However, in this task, we are going to start from scratch.

Solution

First, let us implement a maker of polynomial function of degree 4. We use the power function that is defined in the .ml file. This function takes 6 arguments; 5 of which are the coefficients and the last is the variable x.

```
let make_polynomial_4 a4 a3 a2 a1 a0 x =
  a4 * power x 4 + a3 * power x 3 + a2 * power x 2 + a1 * power x 1 + a0
* power x 0;;
```

Now we can prepare a unit-test function. We follow the format of previous questions.

```
let test_crack_4_once candidate =
  let a4 = random_int ()
  and a3 = random_int ()
  and a2 = random_int ()
  and a1 = random_int ()
  and a0 = random_int ()
  in let p4 = make_polynomial_4 a4 a3 a2 a1 a0
     in let expected_result = (a4, a3, a2, a1, a0)
        and ((a4', a3', a2', a1', a0') as actual_result) = candidate p4
        in if actual_result = expected_result
           then true
           else let () = Printf.printf
                            "test_crack_4_once failed for %i * x^4 + %i *
x^3 + %i * x^2 + %i * x^1 + %i with (%i, %i, %i, %i, %i)\n"
                            a4 a3 a2 a1 a0 a4' a3' a2' a1' a0'
                in false;;

let test_crack_4  candidate =
  test_crack_4_once candidate && test_crack_4_once candidate &&
test_crack_4_once candidate;;
```

Next, let us crack the coefficients:

let $f(x) = ax^4 + bx^3 + cx^2 + dx + e$.

then, $f(0) = e$

$f(1) = a + b + c + d + e \cdots ①$

$f(-1) = a - b + c - d + e \cdots ②$

$f(2) = 16a + 8b + 4c + 2d + e \cdots ③$

$f(-2) = 16a - 8b + 4c - 2d + e \cdots ④$

①−②: $f(1) - f(-1) = 2b + 2d \Leftrightarrow b + d = \frac{1}{2}(f(1) - f(-1)) \cdots ⑤$

③−④: $f(2) - f(-2) = 16b + 4d \Leftrightarrow 4b + d = \frac{1}{4}(f(2) - f(-2)) \cdots ⑥$

⑥−⑤: $3b = \frac{1}{4}(f(2) - f(-2)) - \frac{1}{2}(f(1) - f(-1))$

$\Leftrightarrow b = \left\{ \frac{f(2) - f(-2) - 2f(1) + 2f(-1)}{4} \right\} \times \frac{1}{3}$

$\Leftrightarrow b = \frac{1}{12}\{f(2) - f(-2) - 2f(1) + 2f(-1)\}$

Using ⑤, $d = \frac{1}{2}(f(1) - f(-1)) - b$

$\Leftrightarrow d = \frac{1}{12}\{6f(1) - 6f(-1) - f(2) + f(-2) + 2f(1) - 2f(-1)\}$

$\Leftrightarrow d = \frac{1}{12}\{8f(1) - 8f(-1) - f(2) + f(-2)\}$

①: $a + c = f(1) - b - d - e \Leftrightarrow 4a + 4c = 4f(1) - 4b - 4d - 4e \cdots ⑦$

③: $16a + 4c = f(2) - 8b - 2d - e \cdots ⑧$

⑧−⑦: $12a = f(2) - 8b - 2d - e - \{4f(1) - 4b - 4d - 4e\}$

$\Leftrightarrow a = \frac{1}{12}\{f(2) - 4f(1) - 4b + 2d + 3e\}$

$\Rightarrow c = f(1) - a - b - d - e$

We follow this step and implement the crack_4 function as follows:

```
let crack_4 p =
  let p_0 = p 0
  and p_1 = p 1
  and p_m1 = p (-1)
  and p_2 = p 2
  and p_m2 = p (-2)
  in let a0 = p_0
     and a3 = (p_2 - p_m2 - (2 * p_1) + (2 * p_m1)) / 12
     and a1 = ((8 * p_1) - (8 * p_m1) - p_2 + p_m2) / 12
     in let a4 = (p_2 - (4 * p_1) - (4 * a3) + (2 * a1) + (3 * a0)) / 12
        in let a2 = p_1 - a4 - a3 - a1 - a0
           in (a4,a3,a2,a1,a0);;

let () = assert (test_crack_4 crack_4);;
```

This implementation passes the unit-test function.

Conclusion

In this task, we implemented a function that can crack polynomial functions of degree 4. The implementation got more comlex than the last task, but essentially what we do is the same. We avoid calculating the same value again and again by assigning them a name (e.g., p_0 = p 0) and go step by step to crack the polynomial functions.

A generalization

Introduction

In this task, we consider polynomial functions that are not defined in the standard way, such as:

```
fun x -> ((2 * x) + 3) * x + 4

fun x -> (x + 1) * (x + 1)

fun x -> (x + 1) * (x + 1) * (x + 1)
```

Can such a polynomial function be cracked, and if so, how?

Solution

```
# crack_2 (fun x -> ((2 * x) + 3) * x + 4);;
- : int * int * int = (2, 3, 4)

# crack_3 (fun x -> (x+1) * (x+1) *(x+1));;
- : int * int * int * int = (1, 3, 3, 1)

# crack_2 (fun x -> (x+1) * (x+1));;
- : int * int * int = (1, 2, 1)
```

It seems that there is no problem cracking these polynomial functions. But why? To think about this, let us consider one particular example, crack_2 (fun x -> (x+1) * (x+1)). An expression (x+1) * (x+1) can be expressed as x*x + 2*x + 1 in the standard form. Now let us compare these two expressions.

Firstly, both expressions have type int:

```
# let x = 10;;
val x : int = 10
# (x+1) * (x+1);;
- : int = 121
# x*x + 2*x + 1;;
- : int = 121
#
```

We also see that x has type int.

Secondly, in an environment where x denotes a given value of type int, both expressions would yield the same value of type int, without any error message or observable side effects (pure expressions). Therefore, we can conclude that they are observationally equivalent.

Thirdly, we think about applying this expression to the function. Given two functions:

```
# fun x -> (x+1) * (x+1);;
- : int -> int = <fun>
# fun x -> x*x + 2*x + 1;;
- : int -> int = <fun>
```

applying these functions to a given integer x reduces to evaluating their body - the expressions. As the two expressions are observationally equivalent, the two expressions are functionally equivalent - given the same input, they would yield the same output.

We have solved the mystery of why this cracking function can be applied to other forms of polynomial functions. It's applicable because these expressions are functionally equivalent.

Conclusion

In this task, we have seen how to show the functional equivalence of the expressions. We have considered the type of the input, the type of the expressions, the observational equivalence of the expressions, and the type of the function. Since the expressions are pure and emit the same output for any given input, we have concluded that they are observationally equivalent, and thus the functions are also equivalent.

Conclusion

In this mini-project, through cracking the polynomial functions, we first re-familiarized ourselves with the let-expression. It is a useful technique to assign a name to a value to avoid calculating the same value multiple times. And in the last task, we see how to connect the concept of functional equivalence being equal to the observational equivalence of their body. This reduction of the evaluation links back to the partial evaluation where we parameterized a general function to a specialized function.

"A function is a parameterized expression!"
- *Prof O Danvy*

Conclusion

This marks the end of the midterm project. In this midterm project, through writing a number of OCaml expressions and functions (parameterized expressions), we have come to understand more of what the OCaml's determinism is, how recursive functions operate on the given input, how functions can be parameterized to make them more generic, and how to make most of the let-expressions and built-in String functions.

As a part of our routine when implementing a function, we prepare a unit-test function to test the function later. We have learned to be mindful of the coverage and limitation of these unit-test functions. We should include some corner cases such as empty string/character, and also randomly chosen candidates.

From five mini-projects collectively, we have learned many things. The first mini-project reminded us of how to write a parameterized function, which links back to partial evaluation and specialized functions from Week 02. The next mini-project, the nat_fold_right exercise is a more elaborate version of the specialized function, and we have seen how we can break down the recursive process into inductive steps and implement it using those specialized functions. In the next project, we have looked at how expressions are evaluated in OCaml and how to determine the equivalence of expressions. The former led us to the following mini-project about strings, where we used built-in String functions to implement mostly recursive functions. The latter led us to the very last mini-project where we cracked polynomial functions and considered functional equivalence of expressions.

Overall, by writing hundreds of lines of codes and encountering errors, we have learned about OCaml and implementing desired functions a lot more than we thought we would. After all, the best way to familiarize ourselves with programming language is to code in it.

"The only way to learn a new programming language is by writing programs in it."

- Dennies Ritchie