NEGATION (DISJUNCTION (VARIABLE “A SIMPLE PROJECT”, VARIABLE “EASILY OPTIMIZED”)

Learning to Optimize and Adequately Test Negational Normalization

Fundamentals of Programming

Mini-Project #2

Brian Scott Bohme

Table of Contents

Introduction

Negational normalization of logical arguments was not a new concept to me. Having done a lot of logic previously in Proofs and philosophy courses, I was expecting this mini-project to be straightforward. I knew that a negation of a disjunction was a conjunction of negations, and vice-versa, alongside the fact that the negation of a negation is just the original variable. Implementing this in OCaml, I initially thought, was just a matter of iterating through an expression, and using match cases to force these negational changes when encountered in the expression. However, I very quickly learned that the obvious transliteration of this was not structurally recursive, as you are adding two more negations with every conjunction or disjunction. This was the beginning of my trial and error in optimizing not only the compilers and interpreters, but also the unit tests. Once I saw the structural methods of negational normalization, it all clicked and makes perfect sense. However, getting to that point of satisfaction was the arduous part of this journey.

The Meaning of Negational Normalization

Before starting on any code, it was important to understand what negational normalization is doing. In formal logic, negation normal form is the "simplified" form in which the negating operator is only applied to variables; therefore, negated conjunctions and/or disjunction must be expressed differently. To solve this, you use De Morgan's laws to such that:

*Not (**P** and **Q**)* becomes *(Not **P** or Not **Q**)*

And:

*Not (**P** or **Q**)* becomes *(Not **P** and Not **Q**)*.

The final rule for negation normal form follows that:

*Not (Not **P**)* becomes ***P***.

Witnessing how an expression becomes negational normal leads intuitively to the fact that a recursive function with match cases will be the most appropriate way to tackle this problem, which is precisely the first steps I took when writing the Boolean expression to negational normal form code.

Boolean Expression to Negational Normal Form

As mentioned previously in this paper, my initial thought was to invoke De Morgan's laws verbatim into an OCaml code. Using matches, I'd iterate through an expression until a negation was reached. At this point, another match case was invoked, where the corresponding De Morgan's laws outlined in the previous section were transliterated in each case. I discovered that the problem with doing this is that the code fails to be structurally recursive. By adding more negations, I was failing to simplify each recursive call as, for example:

*Negation (Disjunction (**e1**, **e2**))* became *Conjunction (Negation **e1**, Negation **e2**)*,

with recursive calls on *Negation **e1*** and *Negation **e2***. I needed to find a way to still iterate through the expression, but only performing the recursive calls on ***e1*** and ***e2***. This requires taking a step back and understanding what is happening during the negational normalization. Imagine we have:

*Not (Not (**P** and Not **Q**) or **P**)*.

There are three negations in this statement. We know that from the recursive structure of the negational normalization that the negations should all be carried forward to the variables. Then we can see that each time, the negation changes the polarity based on whether there is an even number (positive polarity) or odd number (negative polarity) of negations. For example, carrying forward the applicable negations on the last *P* variable results in "not P", which has negative

polarity and therefore stays negated. However, the *P* variable in the conjunctive statement becomes “not not P”, which has positive polarity and therefore is just “P”. The polarity in this case acts as a switch, so if we just change the polarity every time a negation is iterated upon, we have almost successfully implemented a structurally recursive normalizer! However, we still have to consider the conjunctions and disjunctions, which change under negational normalization as mentioned above. With the polarity switch in place however, this is quite straightforward. If there is a positive polarity, then that means there were an even amount of negations affecting the expression, which results in positive polarity. Thus we can keep the conjunction or disjunction the same. If there were an odd amount of negations affecting the expression, then we know there is one negation that is not cancelled out by the others, which then switches a conjunctive statement to a disjunctive or vice versa according to De Morgan’s laws. We do not have to change the polarity when this happens however, because the negation is already being carried through at that point. To summarize: if polarity is positive (true), then there has been an even number of negations affecting the expression being recursed upon at that point, which cancel each other out and maintain the expression as is. If polarity is negative, then there has been an odd number of negations affecting the expression being recursed upon at that point, which means there is still one affective negation that negates a variable or switches the conjunctive/disjunctive statement. Finally, iterating upon a negation just switches the polarity. With all this in mind, we can very simply and structurally implement our Boolean expression to negational normal function:

```
let boolean_to_negational_normal e_init =
  let rec visit e =
    match e with
    | Variable x -> (fun polarity ->
```

```
            if polarity
            then Variable_nnf x
            else Negation_nnf x)
    | Conjunction (e1, e2) -> (fun polarity ->
            if polarity
            then Conjunction_nnf (visit e1 polarity, visit e2 polarity)
            else Disjunction_nnf (visit e1 polarity, visit e2 polarity))
    | Disjunction (e1, e2) -> (fun polarity ->
            if polarity
            then Disjunction_nnf (visit e1 polarity, visit e2 polarity)
            else Conjunction_nnf (visit e1 polarity, visit e2 polarity))
    | Negation e' -> (fun polarity ->
            visit e' (not polarity))
  in visit e_init true;;
```

We can see that these match cases do exactly what is described above. Each match case is a function of polarity, which checks what the current polarity is and implements the proper structural change based on the current expression being recursed upon. I built my unit tests such that covered all the permutations of the simplest expressions, as any other statement will just be a composition of these, and all passed!

The reason polarity is actually outside the parameters of the visit function is so it can be made fold right, which will be covered in a later section. If polarity was also a parameter, then our fold right function would also need polarity as a parameter, which would in turn mean our interpreter function would need polarity as a parameter. In the next section, we will see this is not the case, and that the interpreter function is actually more straightforward to implement.

Implementing an Interpreter

The interpreter's function is to express a Boolean answer to the different Boolean inputs within the variables of Boolean expressions. By literally following the structure of how you'd evaluate such statements on paper logically, it is easy to create a structurally recursive function:

```
let rec boolean_expression_interpreter e_init =
  let rec visit e =
    match e with
    | Variable x ->
          x
    | Conjunction (e1, e2) ->
          ((visit e1) && (visit e2))
    | Disjunction (e1, e2) ->
          ((visit e1) || (visit e2))
    | Negation e' ->
          match e' with
          | Variable x ->
                not (x)
          | Conjunction (e1, e2) ->
                not ((visit e1) && (visit e2))
          | Disjunction (e1, e2) ->
                not ((visit e1) || (visit e2))
          | Negation e'' ->
                visit e''
  in visit e_init;;
```

However, this implementation isn't as functional as we'd like it to be. We want to know the result of interpreting the expression for all possible variable inputs: thus if an expression had variables "x", "y", and "z", we'd want the function to interpret the result of (true, true, true), (true, true, false), (true false false), and et cetera. At the moment, the above function isn't powerful enough to do this. That is why we wrap this function in an environment. The environment model is superior to a substitution model (in which you evaluate sub-expressions to values and substitute the argument value for the formal parameter and evaluate) because is combines the process of a substitution with the process of evaluation within a single pass over the code. We will talk about actually constructing the environment in the next section. However, we can complete our implementation of the interpreter by parameterizing the environment in the code, and looking up the associated value (this will also be explained in the next section).

```
let rec boolean_expression_interpreter e_init env_init =
  let rec visit e env =
    match e with
    | Variable x ->
          lookup x env
    | Conjunction (e1, e2) ->
          ((visit e1 env) && (visit e2 env))
    | Disjunction (e1, e2) ->
          ((visit e1 env) || (visit e2 env))
    | Negation e' ->
          match e' with
          | Variable x ->
                not (lookup x env)
```

```
        | Conjunction (e1, e2) ->
              not ((visit e1 env) && (visit e2 env))
        | Disjunction (e1, e2) ->
              not ((visit e1 env) || (visit e2 env))
        | Negation e'' ->
              visit e'' env
  in visit e_init env_init;;
```

The interpreter function for negation normalized expressions is almost identical. However, we know there will only be negations on variables, so we don't need a separate match case for negated statements:

```
let boolean_expression_nnf_interpreter (e_init :
boolean_expression_nnf) env_init : bool =
  let rec visit e env =
    match e with
    | Variable_nnf x ->
        lookup x env
    | Conjunction_nnf (e1, e2) ->
        ((visit e1 env) && (visit e2 env))
    | Disjunction_nnf (e1, e2) ->
        ((visit e1 env) || (visit e2 env))
    | Negation_nnf x ->
        not (lookup x env)
  in visit e_init env_init;;
```

Before we verify that passing an expression through both interpreters produces the same result, we need to understand and construct our environment.

Building the Environments

The environment model is perhaps the most powerful part of this code. As mentioned previously, we want an environment that passes both true and false into every variable. To build our environment, we want to use an association list, which is a list of pairs in which the first item is the variable and the second item is the associated value of the item. Thus, we need to do two things if we want an environment model consisting of all variables of an expression and all associated values of true/false: first we need to collect all variables in an expression, then we need to generate all possible Boolean environments with these variables. The first part of this task is for efficiency; by first collecting all variables in the expression, you are only going to evaluate the expression with the necessary variables. This is much more optimized then, say, generating an arbitrary list of variables. If you have an expression with only "x" and "y" as variables but create an environment with all variables "a" through "z", there is an exponential amount of wasted space that the garbage collector must sift through since "x" and "y" are the only active variables. Remember that we have to evaluate the expression for every true/false combination of all variables, so it is imperative that our environment is only using the active variables in our expression. By iterating through the expression, we can collect the variables:

```
let boolean_expression_variables e_init =
    let rec visit e lst =
      match e with
      | Variable x ->
          if List.mem x lst
```

```
            then lst
            else x :: lst
       | Conjunction (e1, e2) ->
            visit e1 (visit e2 lst)
       | Disjunction (e1, e2) ->
            visit e1 (visit e2 lst)
       | Negation x ->
            visit x lst
in visit e_init [];;
```

Then we need to generate all possible Boolean environments for the variables. We want to pass our list of variables through, and somehow couple each one with a true and a false as described above. Thus we can iterate through the list, and for each head, extend the environment with a true and a false using:

```
let extend n d e
      = (n, d) :: e
```

Therefore, by mapping the extend over all variables, all possible environments will be generated as an abstraction, ready to use in our code:

```
let rec all_possible_Boolean_environments ns =
  match ns with
  | [] ->
     [Environment.empty]
  | n :: ns' ->
     let es = all_possible_Boolean_environments ns'
     in List.append
```

```
(List.map (fun e -> Environment.extend n true e) es)
(List.map (fun e -> Environment.extend n false e) es);;
```

It is easier to understand what lookup in our interpreter is doing now: it is just search through the environment for our variable, and returning the associated true/false! With all the interpreters built and the environment optimized for efficiency, the next thing we need to do is test, test, and test.

Generating Tests

The core of this assignment is making sure that evaluating the Boolean expression produces the same results as evaluating the normalized expression, or else we are in trouble. Structurally, we want to make sure that mapping the environment over the interpreted expression is the same as mapping the environment over the interpreted normalized expression:

```
let verify_all_possible_Boolean_environments e =
  List.map (boolean_expression_interpreter e) environment =
  List.map (boolean_expression_nnf_interpreter
(boolean_to_negational_normal e) environment;;
```

We know how to construct our environment, we just want to make sure we collect the variables first then pass them through the environment generator:

```
let verify_all_possible_Boolean_environments e =
  let variables = all_possible_Boolean_environments
(boolean_expression_variables e)
  and
normalized = boolean_to_negational_normal e
  in
```

```
  List.map (boolean_expression_interpreter e) variables
  =
  List.map (boolean_expression_nnf_interpreter normalized) variables;;
```

If the function evaluates to true, then we know that for any input for all variables, the outputs will be the same for both the original expression and the normalized form. I used a view unit tests to verify this, but I wanted a hardcore test to *really* test it. My idea was to create a function that generates a random boolean function each time, and recursively tests the verify_all_possible_Boolean_environments function for *n* different and random expressions. It was a fun function to build; essentially, I used a random number generator from 0-5, then matched the numbers to a different constructor. For example, 0 would lead to a negation, 1 a conjunction, and 2 a disjunction. Any other number will be a variable with a random variable labelled 1 through 10. The highest chance goes to getting a Variable because this the only way the function will terminate; when my chance of getting a variable was even with the other constructors, I blew the stack multiple times. This function looked like:

```
let rec random_boolean_expression_generator () =
    match Random.int 6 with
    | 0 -> Negation (random_boolean_expression_generator ())
    | 1 -> Conjunction (random_boolean_expression_generator (),
random_boolean_expression_generator ())
    | 2 -> Disjunction (random_boolean_expression_generator (),
random_boolean_expression_generator ())
    | _ -> Variable (string_of_int (Random.int 10));;
```

I think this is a super cool function, and probably my favorite of the project. Using this to extensively test our verify function is fun too. I implemented a counter, that just calls the verify function with the randomly generated expression *n* times, and ensures all come out to true:

```
let rec extensive_test n =
    if n = 0
    then verify_all_possible_Boolean_environments (random_boolean_expression_generator ())
    else
    (verify_all_possible_Boolean_environments (random_boolean_expression_generator ()))
&& (extensive_test (n-1));;
```

This is an amazingly fast function too, considering the amount of computations being performed. As the master of all tests, I wrote the ultimate test code:

```
let _ = assert(extensive_test 1 && extensive_test 10 && extensive_test
100 && extensive_test 1000);;
```

Which is testing the verify function with 1111 random expressions. Nothing beats the feeling of seeing this pass.

Executing Proper Fold-Right Functions

As the final exercise to this project, we had to implement our interpreters and our Boolean expression to normalized form functions as fold right functions. Luckily, as this was considered when building our original functions, this was quite straightforward. We know the fold right function should be parameterized by 5 inputs. Four of these are our functions (what we want to do) with each constructor, and the last being the expression.

```
let fold_right_boolean_expression var con dis neg exp =
  let rec visit e =
    match e with
    | Variable x ->
```

```
        var x
    | Conjunction (e1, e2) ->
        let e1' = visit e1
        and e2' = visit e2
        in con (e1', e2')
    | Disjunction (e1, e2) ->
        let e1' = visit e1
        and e2' = visit e2
        in dis (e1', e2')
    | Negation e' ->
        neg (visit e')
in visit exp;;
```

The first function to express in fold right was the boolean to negational normal. Luckily, we can copy and paste from our original function what we are doing with each constructor because the structural recursion is the same! And since polarity is not a parameter in the original function but rather a function of the function, we can parameterize polarity at the end of the function:

```
let fold_right_boolean_to_negational_normal e_init =
  fold_right_boolean_expression
  (fun x -> (fun polarity ->
     if polarity
     then Variable_nnf x
     else Negation_nnf x))
  (fun (c1, c2) -> (fun polarity ->
     if polarity
```

```
    then Conjunction_nnf (c1 polarity, c2 polarity)
    else Disjunction_nnf (c1 polarity, c2 polarity)))
  (fun (c1, c2) -> (fun polarity ->
    if polarity
    then Disjunction_nnf (c1 polarity, c2 polarity)
    else Conjunction_nnf (c1 polarity, c2 polarity)))
  (fun c -> (fun polarity ->
    c (not polarity)))
  e_init
  true;;
```

The interpreter for Boolean expressions is even more straightforward. We can transcribe each action as a function for the respective constructors:

```
let fold_right_boolean_expression_interpreter e env =
  fold_right_boolean_expression
  (fun x -> lookup x env)
  (fun (c1, c2) -> c1 && c2)
  (fun (d1, d2) -> d1 || d2)
  (fun x -> not x)
  e;;
```

Finally, the fold right and fold right interpreter for the negational normal form is almost the same. However, we know that there are only negations on variables, so we can adjust accordingly.

Parting Thoughts

This project was probably one of the most helpful for me in terms of bettering my computer science skills because it really made me think deeply about recursion (especially structural recursion), environments, and optimization. There were a lot of considerations that I honestly skipped over when first completing this project. However, I began to saw a lot of missed opportunities to optimize functions or optimize tests and even go all out and build mega-tests. There is still work that could be done however; is there a way to build tail recursive normalizers and/or interpreters so the stack is never blown? That would be my next topic of exploration. I believe that this topic was a very nice compliment to the first mini-project because it focused more on understanding recursion, optimization and test models rather than structure, data types and patterns. Together however there is a breadth of topics between the two that I now understand much more deeply. I cannot say either were easy (despite my initial reservations), but both were immensely rewarding and eye opening.