The report below is a summary of the proof assisatnt project.

What was implemented in the project

In the project, I managed to implement all the all the requirerd portions except for the associativity and commutativity of multiplication. The code is primarly divided into 2 sections, one for simple types and another for dependent types. Both proof assistants the support for logical connectives as well as natural numbers. Along side the main implementaiton, there are also error messages explaining the errors when writing the proof, which can hint the user on what would be an appropriate blank to fill in, this is further elaborated in the later sections.

Simple types

Simple types are defined as follows

```
(** Type variables. *)
type tvar = string
type ty =
  | Var of tvar
  | Imp of ty * ty
  | Conj of ty * ty
  | Truth
  | Disj of ty * ty
  | False
  | Unit
  | Nat
```

Here we define a type (in ocaml) tvar which is just a string, this is done so we are able to distinguish between the various strings which may appear later on. For the remaining simple types, they are defined as a type (in ocaml) as ty, the types and their defintions are as follows :

1. Var - which takes a tvar(string) as input, this is simple a name to call the various types we encounter
2. Imp - whch takes two types (A, B), this represents an implication between simple types (A -> B)
3. Conj - which takes two types (A, B), this represents a conjunction (A \and B)
4. Truth - a type for truth which is represented as "T" when printed.
5. Disj - which takes two types (A, B), this represents a disjunction (A \or B)
6. False - a type for false represetned by "⊥"
7. Unit - a type for unit represented by "()"
8. Nat - a type for natural numbers

In addition to the types there are also the terms defined as follows

```
type var = string
type tm =
  | Varm of var
  | Appm of tm * tm
```

```
    | Absm of var * ty * tm
    | Pairm of tm * tm
    | Fstm of tm
    | Sndm of tm
    | Casem of tm * var * tm * var *tm
    | Rcasem of tm * ty
    | Lcasem of tm * ty
    | Trum
    | Falm of tm * ty
    | Unitm
    | Zero
    | Suc of tm
    | Rec of tm * tm * var * var * tm
```

Here we introduce a new type var which is just a string, but it serves to distinguish between strings and variables for terms, as tvar does for types. For the remaining terms, they are defined as follows :

1. Varm - which takes a var(string), this representation for the terms which are introduced or encountered
2. Appm - taking 2 terms (a, b), this is a representation of a being applied to b "a(b)"
3. Absm - taking a var, a type and a term (x, ty, tm), this represents an abstraction with x being the name of the arbitrary variable in the term, ty being the type of x, and tm being the term which depends on x. "x (type ty) -> tm(x)"
4. Pairm - taking two terms (a, b), this is a representation of a pair (b \times b)
5. Fstm - taking a Pairm(tm) (a, b), which represents the first term of the pair (a)
6. Sndm - taking a Pairm(tm) (a, b), which represents the second term of the pari (b)
7. Casem - taking a term (t) and variables with terms bounded to them (x -> u) (y -> v), which is a representation of case analysis
8. Rcasem - taking a term and type, which represents the right case of a Casem term
9. Lcasem - taking a term and type, which represents the left case of a Casem term
10. Trum - representing the truth term
11. Falm - which takes a term (a) and type (A), and represents a proof of A (which is anything) given that a is false.
12. Unitm - representing the unit term
13. Zero - the term representing the natural number 0
14. Suc - taking a term (n) (in particular a natural number), representing the successor of n
15. Rec - taking 2 terms (t, u), 2 variables (x, y) and another term (v), representing the recursor function for natural numbers

Using the proof assistant however would not require the knowledge of the terms and types above, but simply their existences. In the context of using the proof assistant, you can run

```
#Interactive mode
make manual
#Proof from a file
make proof
```

where the manual mode will allow the user to manually type in the commands while proof mode allows the user to input a file, and running the commands in the file.

Firstly we need a statement to prove. The full list of syntaxes available are in the lexer.mll file The various commands available are :

1. exact, which takes an input variable, if the input variable is in the context, and the type of the variable corresponds to the type we are trying to prove, it completes the proof
2. intro, if the type we are trying to prove has type

- Imp, then it also demands an input variable, it then associates the variable to the input type of the implication, and then asks for a proof of the output type of the implication
- Conj, then it requests a proof for the first and second variable
- Truth, then it introduces a proof of true
- Nat, then in also demands an input variable (which has to be a natural number in the context). If the variable is 0, then in introduces a proof of 0, if the variable is a successor of something (call it n), then it asks for a proof of n

3. fst, which takes an input variable, if the variable is in the context and is of a type conj, it checks if the first term of the conjunction has the same type as the type we are trying to proof, and completes the proof if the type matches.
4. snd, which does the same things as fst, but just on the second type of the conjunction.
5. left, if the type we are proving is a disjunction, it asks for a proof of the left term
6. right, the same as left, but for the right term of disjunction.
7. elim, which requires at least an input, and if the first input has type

- Imp, it checks the input type of the implication with the type we are trying to prove, if it matches, it only requires a proof of the output type of the Implication.
- Disj, it gives a new variable in the context with type of the right term of the disjunction, after it has been used, it gives another variable with the type of the left term of the disjunction.
- False, it returns a proof of false
- Nat, it then requires 4 other arguments (in total 5), the base case, 2 variable x and y, and the recursion function which depends on x and y. It then asks for a proof of the base case, as well as the recursion function, given the variables x and y provided. If all the proves are given, it completes the proof of the first input (of type Nat)

8. cut, which takes a variable, and introduces a the variable as a part of the functon.

Lets say we'd like to prove the commutativity of the and operator, then the following statements is a proof of the statement.

```
A /\ B => B /\ A
intro x
intro
snd x
fst x
```

The various proof for the different parts of the assignment are in the folders Part2/Part3_proofs

Dependent types

In the case of dependent types, the inital types are all already provided in the assignment document, as such I will not dwell on them too much. Instead the focus of this section will be to explain how to use the dependent type prover. As with the case of the simple prover, there syntaxes available are in the lexer.mll file, the commands are the same as well, you just have to ensure that you are in the correct directory. The various commands that are available are also explained in the assignment document.

There is a complete proof of the associativity and commutativity of addition in the dnat.proof file, as well as a few auxilary function. The proof of multiplicaiton however is incomplete.

A brief explaination of how the prover works :

```
define pp = fun (n : Nat) -> Nat
define sp = fun (m : Nat) -> (fun (l : Nat) -> m)
define pred = fun (k : Nat) -> Ind pp Z sp k
eval pred Z
eval pred (S (S (S Z)))
```

the following commands is a construction of the predecessor function. Here we want to define the predecessor function using induction, as such we begin by defining:

1. pp, the predicate, which is of type Nat -> Type.
2. sp, the step function, which given m and pp(m) gives a proof of pp(S(m)), where S(m) is the successor of m.
3. pred, the induction on the natural number k. Here Ind is a function which takes the predicate pp, base case (which we set to Zero), step function sp, and k the variable do induction on.

There is a similar function for equality, which is the J function, which is also explained in the assignment docuement.

Difficulties encountered

One of the biggest difficulties encountered was in the normalization of terms. There were many points in this project where the proofs I provided were not being accepted by the assistant. However when inspecting the outputs, it was usually the result of certain terms remaining in their unnormalized form. While it may seem like a simple fix, to just simple normalize the output where needed, it did not seem obvious when I was coding the project. There were certain outputs which already seemed normalized, as the outputs were composed of normalized terms. However it was necessary to then normalize the term as a whole as there might be further normalization that could occur in the composition.

In addition to the bugs that appeared in the proof assistant, at times the proves for the functions were also difficult to check. Some proves which seemed obvious to me were not evident to the prover, and left me asking if it was an issue with the prover, or was the proof provided not sufficient. This however could be remedied by back checking with another proof assistant, such as agda, where if it worked with agda, I could deduce that there was a bug in my implementation.

Implementation choices

Most implementation choices in the simple type prove assistant are detailed in the simple types section.

Some additional implementation choices for dependent types made here were to add error comments when ever there was clear problems with a proof. For example, if we tried to apply a natural number to a function which takes in boolean types. This helps explicitly tell the user where in the proof is problem and what should be the correct input type. This is especially useful, if we don't know what to proof. We can simply put in an arbitrary function, (fun (x : Nat) -> x) for example, and see what the prover returns. Here the prover will explicitly tell us what we need to prove, letting us know what we should be proving.

Possible extensions

Some quality of life improvements:

1. Most outputs has did not have too many variables, as most intermediate variables are not present in the final representation. A simple quality of life:w improvement would be to relable the final variables into something more readable.
2. A graphical user interface would also be a great quality of life, listing out the various definitions available as blocks, which can then be connected using the various functions.

More technical improvements:

1. Automatic prover (For both the simple and dependent type). It became clear at a certain point that the prover actually tells us what we need to prove. If the necessary definitions are available in the context it shouldn't be difficult to implement a brute force search on the available defintions to find a possible solution. Being a brute force approach here would inevitably result in a larger computation time. Which would the require implementations of better and smarter search algorithms.
2. Mathematical equation converter, as well as support for more symbols and notations. It is much easier to write some of the results out on paper than its coded counterpart. Having a mathematical equation interpreter would save a lot of time in the conversion of notation.

Conclusion

In conclusion, this was a pretty fun project. Despite being stuck for many hours on a single question, the pleasure from resolving the problem was very enlightining. There are many points for improvements in this project, however with the limited time as well as other concurrent project make it difficult to finish. Having done this project, I have really learnt a lot more about the inner workings of OCaml as well as Agda, giving me a better apprectiation and understanding of how these functional programming languages work.