0 Introduction

Dear reader,

Most master thesis dive deep into their subject matter with very little to no regard to the uninitiated mind.

While this approach is justified in many cases I want to take this opportunity to experiment with a more gentle introduction to the topic, at the cost of formality and familiarity.

Indeed, I feel like this topic is strange enough yet, simple enough, that it can be taught in the next few pages to an uninitiated student.

The following will only make assumptions about basic understanding of imperative and functional programming.

A note about vocabulary and Jargon

Those technical papers are often hard to approach for the unintiated because of the heavy use of unfamiliar vocabulary and domain-specific jargon. While those practices have their uses, they tend to hinder learning by obscuring basic concepts. Unfortunately I do not have a solution for this problem, but I hope this section will help you mitigate this feeling of helplessness when sentences seem to be composed of randomly generated sequences of letters rather than legitimate words.

Type

A label associated to a collection of values. For example `String` is the type given to strings of characters for text. `Int` is the type given to integer values. `Nat` is the type given to natural numbers.

Linear types

Types that have a usage restriction. Typically a value labelled with a linear type can only be used once, no less, no more.

Linearity / Quantity / Multiplicity

Used interchangably most of the time. They refer the the number of time a variable is expected to be used.

Semiring

A mathematical structure that requires its values to be combined with `+` and `*` in the ways you expect from natural numbers.

Syntax

The structure of some piece of information, usual in the form of *text*. Syntax itself does not convey any meaning.

Semantics

The meaning associated to a piece of data, most often related to syntax.

Pattern matching

Destructuring a value into its constituant parts in order to access them or understand what kind of value we are dealing with.

Generic type / Polymorphic type / Type parameter

A type that has a *type parameter* . For example `Maybe a` takes 1 type as parameter, the `a` has type `Type`.

Indexed type

A *type parameter* that changes with the values that inhabit the type. For example `["a", "b", "c"] : Vect 3 String` has index `3` and a type parameter `String`, because it has 3 elements and the elements are Strings. If the value was `["a", "b"]` then the type would become `Vect 2 String`, the index would change from `3` to `2`, but the type parameter would stay as `String`

Programming recap

If you know about programming, you probably know about types and functions. Types are ways to classify values that the computer manipulates and functions are instructions that describe how those values are changed.

In *imperative programming* functions can perform powerful operations like "malloc" and "free" for memory management or make network requests through the internet. While powerful in a practical sense, those functions are really hard to study, so in order to make life easier we only consider functions in the *mathematical* sense of the word : A function is something that takes an input and returns an output.

```
f : A -> B
```

This notation tells us what type the function is ready to injest as input and what type is expected as the output.

```
input    output
    v    v
f : A -> B
^
name
```

This simplifies our model because it forbids the complexity related to complex operations like arbitrary memory modification or network access. (We can recover those features by using patterns like "monad" but it is not the topic of this brief introduction so we will skip it.)

Functional programming describes a programming practice centered around the use of such functions, and types are used to describe the values those functions manipulate. In addition, traditionally functional programming language have a strong emphasis on their type system which allow the types to describe the structure of the values very precisely.

During the rest of this thesis we are going to talk about Idris2, a purely functional programming language featuring Quantitiative Type Theory, a type theory centered around managing resources.

Idris and dependent types

Before we jump into Idris2, allow me to introduce Idris, its predecessor. Idris is a programming language featuring dependent types. Dependent types in that they allow you to write both programs and theorems within the same language.

Here is an example program featuring dependent types

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = 404
intOrString False = "we got a string"
```

As you can see the return type of this function contains an if-statement that uses the argument of our function as its conditional.

The next snipped disentangles the type signature a little bit.

```
--    name of the argument         return type
--              |                        |
--              |  Type of the argument  |
--              |     |                  |
--              v     v       /----------/ \----------\
intOrString : (b : Bool) -> if b then Int else String
--              ^                ^
--              \----------------/
--                 dependency
```

Since the return type is different depending on the value of `b` it means this is a *dependent type*.

Non-dependent type system cannot represent this behavior and have to resort to patterns like "either" (or, more generally, alternative monads) which encapsulates all the possible meanings our program could have. Even if sometimes we *know* there is only one of them that can occur.

```
eitherIntOrString :: Bool -> Either Int String
eitherIntOrString True = Left 404
eitherIntOrString False = Right "we got a string"
```

This small example is one reason why dependent types are desirable for general purpose programming, they allow the programmer to state the behavior of the program without ambiguity and without superfluous checks that, in addition to hinder readability, can have a negative impact on the runtime performance of the program.

Idris and type holes

A very useful feature of Idris is *type holes*, one can replace any term by a variable name prefixed by a question mark, like this : `?hole` . This tells the compiler to infer the type a this position and report it to the user in order to better understand what value could possibly fit the expected type. When asked about a hole, the compiler will also report what it knows about the surrounding context in order to help us figure out which values could suit the expected type.

If we take our example of `intOrString` and replace the implementation by a hole we have the following:

```
intOrString : (b : Bool) -> if b then Int else String
intOrString b = ?hole
```

Asking the Idris compiler what the hole is supposed to contain we get

```
b : Bool
--------------------------------
hole : if b then Int else String
```

This information does not tell us what value we can use. However it informs us that the type of the value *depends* on the value of `b` therefore, pattern matching on `b` might give us more insight.

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = ?hole1
intOrString False = ?hole2
```

Asking again what is in hole1 get us

```
------------
hole1 : Int
```

and hole2 gets us

```
-------------
```

```
hole2 : String
```

Which we can fill with literal values like `123` or `"good afternoon"`. The complete program would look like this:

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = 123
intOrString False = "good afternoon"
```

A note about Either

Interestingly enough using the same strategy in `eitherIntOrString` does not yield such precise results, indeed

```
eitherIntOrString : Bool -> Either Int String
eitherIntOrString b = ?hole
```

Asking for `hole` gets us

```
b : Bool
------------------------
hole : Either Int String
```

While this type might be easier to read than `if b then Int else String` it does not tell us how to proceed in order to find a more precise type to fill. Indeed, pattern matching further on `b`

```
intOrString' : Bool -> Either Int String
intOrString' True = ?hole1
intOrString' False = ?hole2
```

does not provide any additional information about the return types to use.

```
-----------
hole1 : Either Int String


-----------
```

```
hole2 : Either Int String
```

A note about usage

In itself using `Either` isn't a problem however this lack of information manifests itself in other ways during programming, take the following program

```
checkType : Int
checkType = let intValue = intOrString True in ?hole
```

The hole gives us the following:

```
intValue : Int
---------------
hole : Int
```

If we want to manipulate this value we can treat it as any other integer, there is nothing special about it, except for the fact that it comes from a dependent function.

```
checkType : Int
checkType = let intValue = IntOrString True
                doubled = intValue * 2 in ?hole
```

```
intValue : Int
doubled : Int
---------------
hole : Int
```

We can then return the modified value without any fuss:

```
checkType : Int
checkType = let intValue = intOrString True
                doubled = intValue * 2 in doubled
```

Constrast that with the non-dependent implementation `intOrString'`

```
checkType : Int
```

```
checkType = let intValue = eitherIntOrString True in ?hole
```

```
intValue : Either Int String
---------------
hole : Int
```

The compiler is unable to tell us if this value is an `Int` or a string. despite us *knowing* that `IntOrString` returns an `Int` we cannot use this fact to convince the compiler to simplify the signature for us. We have to go through a runtime check to ensure that the value we are inspecting is indeed an `Int`. This is one aspect where using dependent types results in more efficient program too.

```
checkType : Int
checkType = let intValue = eitherIntOrString True in
                case intValue of
                     (Left i) => ?hole1
                     (Right str) => ?hole2
```

This introduces the additional problem that we now need to provide a value for an impossible case (`Right`). What do we even return? we do not have an Int to double. Our alternatives are:

- Panic and crash the program
- Make up a default value, silencing the error but hiding a potential bug
- Change the return type to `Either Int String` and letting the caller deal with it.

None of which are ideal nor replicate the functionality of the dependent version we saw before.

This conclude our short introduction to dependent types and I hope you've been convinced of their usefulnesss. In the next section we are going to talk about linear types.

Idris2 and linear types

Idris2 takes things further and introduces *linear types* in its type system, allowing us to define how many times a variable will be used. Three different quantities exist in Idris2 : `0`, `1` and `ω`. `0` means the value cannot be used in the body of a function, `1` means it has to be used exactly once, no less, no more. `ω` means the variable isn't subject to any usage restrictions, just like other non-linear programming languages.

We are going to revisit this concept later as there are more subtleties, specially about the `0` usage. For now we are going to explore some examples of linear function and linear types. Take the following function:

```
increment : Nat -> Nat
increment n = S n
```

As we've seen before with our `intOrString` function we can name our arguments in order to refer to them later in the type signature. We can do the same here even if we do not use the argument in a dependent type. Here we are going to name our first argument `n`.

```
increment : (n : Nat) -> Nat
increment n = S n
```

In this case, the name `n` doesn't serve any other purpose than documentation/ but our implementation of linear types has one particularity: quantities have to be assigned to a *name*. Since the argument of `increment` is used exactly once in the body of the function we can update our type signature to assign the quantity `1` to the argument `n`

```
increment : (1 n : Nat) -> Nat
increment n = S n
                ^
                |
              We use n once here
```

Additionally, idris2 feature pattern matching and the rules of linearity also apply to each variable that is bond when matching on it. That is, if the value we are matching is linear then we need to use the pattern variables linearly.

```
sum : (1 n : Nat) -> (1 m : Nat) -> Nat
sum Z m = m
    ^
    We match on the argument here

sum (S n) m = S (sum n m)
     ^
     we match on the argument and bind the values used by the constructor to `n`
```

Obviously this programming discipline does not allow us to express every program the same way as before. Here are two typical examples that cannot be expressed

```
drop : (1 v : a) -> ()

copy : (1 v : a) -> (a, a)
```

We can explore what is wrong with those functions by trying to implement them and making use of holes.

```
drop : (1 v : a) -> ()
drop v = ?drop_rhs
```

```
0 a : Type
1 v : a
------------
drop_rhs : ()
```

As you can see, each variable is annotated with an additional number on its left, `0` or `1`, they inform us of how many times each variable has to be used (If there is no restriction, the usage number is simply removed, just like our previous examples didn't show any usage numbers).

As you can see we need to use `v` (since it maked with `1`) but we are only allowed to return `()`. This would be solved if we had a function of type `(1 v : a) -> ()` to consume the value and return `()`, but this is exactly the signature of the function we are trying to implement!

If we try to implement the function by returning `()` directly we get the following:

```
drop : (1 v : a) -> ()
```

```
drop v = ()
```

```
There are 0 uses of linear variable v
```

Which indicates that `v` is supposed to be used but no uses have been found.

Similarly for `copy` we have

```
copy : (1 v : a) -> (a, a)
copy v = ?hole
```

```
0 a : Type
1 v : a
-----------
hole : (a, a)
```

In which we need to use `v` twice but we're only allow to use it once. Using it twice result in this program with this error

```
copy : (1 v : a) -> (a, a)
copy v = (v, v)
```

```
There are 2 uses of linear variable v
```

Interestingly enough, partially implementing our program with a hole give use an amazing insight

```
copy : (1 v : a) -> (a, a)
copy v = (v, ?hole)
```

```
0 a : Type
0 v : a
-----------
hole : a
```

The hole has been updated to reflect the fact that though `v` is in scope, no uses of it are available. Despite that we still need to make up a value of type `a` out of thin air, which is impossible.

While there are experimental ideas that allow us to recover those capabilities they are not currently present in Idris2. We will talk about those limitations and how to overcome them our "limitations and future work" section.

Something about 0

In the following snipped you will notice that we use an implicit argument that hold the type of a list

```
length : {a : Type} -> List a -> Nat
length [] = Z
length (x :: xs) = S (length xs)
```

If an argument is superfluous it can be annotated with 0 indicating that it will not and cannot appear in the body of our function. While this might be strange at first it makes a lot of sense in a dependently typed setting.

```
length : {0 a : Type} -> List a -> Nat
```

Indeed, variables with linearity `0` do not appear in the execution of the program but are allowed to be used during the compilation of the program and therefore are allowed unrestricted use as along as its constrained to functions which takes argument with linearity `0`.

This feature is available for every value, here the same example with vector showcases an erased Nat:

```
--               n isn't consumed by Vect
--                           v
length : {0 n : Nat} -> Vect n a -> Nat
length [] = Z -- n doesn't appear in the body
length (x :: xs) = S (length xs)
```

Note: This function uses curly brackets instead of parenthesis, this indicates that the argument between curly brackets is *implicit*, that is, the used doesn't have to give the argument to the function, the compiler can figure out the argument to use by itself.

The subtleties about `0` do not end here, take this example from the idris compiler.

```
getLocName : (idx : Nat) -> Names vars -> (0 p : IsVar name idx vars) -> Name
getLocName Z (x :: xs) First = x
getLocName (S k) (x :: xs) (Later p) = getLocName k xs p
                                       ^                 ^
            we match on `p` but its type is              |
            (0 : p : IsVar name idx vars)                |
                                                         |
                          We use `p` in the body of the function
```

This function matches on an erased argument, and then binds the arguments of its constructor to a new value `p` and uses it in a recursive call. Why does that work?

If we replace the implementation of the second clause by a hole we get:

```
getLocName (S k) (x :: xs) (Later p) = ?erased
```

```
 0 name : Name
   k : Nat
   xs : Names ns
   x : Name
 0 p : IsVar name k ns
 0 vars : List Name
------------------------------------
erased : Name
```

Which indicates that `p` is inaccessible, updating the program to make the recursive call

```
getLocName (S k) (x :: xs) (Later p) = getLocName k xs ?erased
```

```
 0 name : Name
   k : Nat
   xs : Names ns
   x : Name
 0 p : IsVar name k ns
 0 vars : List Name
------------------------------------
erased : IsVar ?name k ns
```

Tell us that we need to pass a proof of type `IsVar`, the context would suggest that we cannot use `p` because of its `0` linearity, but thankfully the recursive call does not consume the proof either. Therefore it can safely be passed along:

```
getLocName (S k) (x :: xs) (Later p) = getLocName k xs p
```

Exercises

Drop and copy cannot be written for arbitrary types *but* if you know how to construct your type you can *work really hard* to implement them. Here is an example with Nat

```
dropNat : (1 _ : Nat) -> ()
dropNat Z = ()
dropNat (S n) = dropNat n

copyNat : (1 _ : Nat) -> (Nat, Nat)
copyNat Z = (Z, Z)
copyNat (S n) = let (a, b) = copyNat n in
                    (S a, S b)
```

It is worthy to notice that while dropNat effectively spends O(n) doing nothing, copy *simulates* allocation by constructing a new value that is identical and takes the same space as the original one (albeit very inefficiently, one would expect a memcpy to be O(1), not O(n) in the size of the input)

Here is an interface for this behavior

```
interface Drop a where
    drop : (1 _ : a) -> ()

interface Copy a where
    copy : (1 _ : a) -> (a, a)
```

Can you complete the following file?

```
module Main

import Data.List

data Tree a = Leaf a | Branch a (Tree a) (Tree a)

interface Drop a where
    drop : (1 _ : a) -> ()

interface Copy a where
    copy : (1 _ : a) -> (a, a)

Drop a => Drop (List a) where
    copy ls = ?drop_list_impl

Copy a => Copy (List a) where
    Copy ls = ?copy_list_impl
```

```
Drop a => Drop (Tree a) where
    copy tree = ?drop_Tree_impl

Copy a => Copy (Tree a) where
    Copy tree = ?copy_tree_impl
```

What about String and Int? What's wrong with them?