1 What is curry-howard correspondence?

.
In programming language theory and proof theory, the Curry-Howard correspondence (also known as the Curry-Howard isomorphism or equivalence, or the proofs-as-programs and propositions- or formulae-as-types interpretation) is the direct relationship between computer programs and mathematical proofs.

1.1 type constructions in FP

But before getting into that let's see type constructions in functional programming *Here I'll use SML as an example*

- Tuple type `(int,string)`
  Creating : `val x : int*string = (4 , ‘‘Hello’’)`
  Using : `val y : int = #1 x`

- Function type `int -> string`
  Creating : `fun f (x:int) :string = ‘‘the value of x is’’ ^ toString(x)`
  Using : `val y: string = f 45`

- Disjunction type `datatype X = Left of int | Right of string`
  Creating : `val x:X = Left 4`
  Creating : `val y:X = Right ‘‘Hello’’`
  Using :
  ```
  fun f (x:X) :bool = case x of
                      Left y = ( y > 0 )
                      Right _ = false
  ```
  the above given is a function that takes an argument of type X and returns a value of type bool

- Unit type `type t = unit`
  Creating : `val x:unit = ()`

1.2 From types to propositions

But how does the types defined just translate themselves into propositions?
consider the sml code `val x:t = .....` now what does this mean? If this code manages to compile without error then there the type t exists in the system.
Let's denote this proposition by $CH(t)$ which means that code has a value of type t or the type t exists in the system.
But that was just for a type t what about all the other things that we have discussed above?

Type	Proposition	Short Notation
t	CH(t)	t
(a,b)	CH(a) and CH(b)	A ∧ B ; A × B
left a | right b	CH(a) or CH(b)	A ∨ B ; A + B
a → b	CH(a) implies CH(b)	A ⟹ B
unit	true	1

- type parameter `'a` can be considered as $\forall a$
- consider the function `fun f (x: 'a) : 'a * 'a = (x,x)` this is logically equivalent to $\forall a \quad a \Longrightarrow a \wedge a$
- The elementary proof task is represented by a sequent. Notation $A, B, C \vdash G$ the premises are A,B and C and the goal is G.
- proofs are achieved through axioms and derivation rules. Axioms : this sequent is true. Derivation: if such sequents are true then other such sequents are true.
- consider the SML expression `val x = 3; val a = Int.toString(x:int) ^ ''abc'';` an expression of type String This is equivalent to the sequent $Int \vdash String$. This sequent uses the fact that it uses the already computed expression x which is an integer. and hence that becomes the premise and what we are computing here is the value of string which becomes the goal.
- Now consider the function `fun f (x:int) :string = Int.toString(x) ^ ''abc''` . this takes an argument of integer type and returns a value of string type.This doesn't assume that x is defined outside. this uses x as an argument . and it is represented by the sequent $\phi \vdash Int \Rightarrow String$. The empty set on the premises implies that this expression doesn't use any variables that are previously computed. and the goal is the function type.
- sequents only describe the types and doesn't say anything about the value that is being computed.

Now lets see how the type constructions stated above translates into axioms and derivation rules.

- the tuple type. $A \times B$
  Creation : $A, B \vdash A \times B$. that is if we have types A and B then we have type $A \times B$.
  Usage: $A \times B \vdash A$ or $A \times B \vdash B$.
- Function type . $A \Rightarrow B$.
  Creation : If $A \vdash B$ then we have $\phi \vdash A \Rightarrow B$.
  Usage : $A \Rightarrow B, A \vdash B$.

- Disjunction type $A + B$:
  Creation : $A \vdash A + B$ and also $B \vdash A + B$
  Usage : $A + B, A \Rightarrow C, B \Rightarrow C \vdash C$. this corresponds to the function (that uses case statements) that we've written in the SML example for disjunction.

- Unit type 1:
  Create : $\phi \vdash 1$

In addition to these constructions we have some trivial constructions also.

- a single unmodified value of type A is a valid expression of type A.
  $A \vdash A$
- if a value can be computed with given data then it can also be computed with additional data.
  If $A_1, A_2, A_3, .., A_n \vdash B$ then $A_1, A_2, A_3, .., A_n, C \vdash B$
  we denote Γ as a sequence of arbitary premises.
- The order in which data is computed doesn't matter.
  ie , $\Gamma, A, B \vdash C$ is the same as $\Gamma, B, A \vdash C$

A few syntax convention.

- The implication operator associates to the right. ie ,$A \Rightarrow B \Rightarrow C$ is $A \Rightarrow (B \Rightarrow C)$.
- precedence order is implication , disjunction , conjunction .
- Implicitly all our type variables are universally quantified.
  which means, $A \Rightarrow B \Rightarrow A$ means $\forall A : \forall B : A \Rightarrow B \Rightarrow A$.

1.2.1 A simple example

consider $A \Rightarrow B \Rightarrow A$ which actually translates to $\forall A : \forall B A \Rightarrow B \Rightarrow A$
now let's see how can we convert this into types and how the process of deriving this logic works.

- Let's start with the trivial axiom $A \vdash A$ this doesn't need any proof.
- And using the fact that we can add extra types and still construct the sequent we have $A, B \vdash A$
  here we are actually adding an unused premise.
- Now consider the function construction rule which states
  $A \vdash B$ then $\phi \vdash A \Rightarrow B$
  but what is the sequent that we have here? it is $A, B \vdash A$. lets apply the create function rule on B and the A on right side
  . so it becomes $A \vdash B \Rightarrow A$.

- lets apply the create function rule once more. so it becomes.
  $\phi \vdash A \Rightarrow B \Rightarrow A$
- Now this is a theorem because its derived from no premises. (here the premise is ϕ)

what code does this describe?

- This is same as the function.

  ```
  fun f (x:A)  (y:B) : A = x;
  f : 'a -> 'b -> 'a
  ```

  we can also write this function as

  ```
  fun f x = fn y => x;
  (* this also has the type *)
  f : 'a -> 'b -> 'a
  ```

- and hence we can see that the unused variable y here corresponds to the type B (or `'b` in SML)

This way we can translate any expression in a code to a sequent. And the proof of that sequent is actually writing a code that defines the type of that function.

1.2.2 some more examples of propositions as codes

.

proposition	**code**
$\forall A : A \Rightarrow A$	`fun f (x:'a) : 'a = x`
$\forall A : A \Rightarrow 1$	`fun f (x:'a) : unit = ()`
$\forall A : A \Rightarrow A + B$	`fun f (x:'a) : Either  = Left x`
$\forall A : A \times B \Rightarrow A$	`fun f (x:'a,y:'b) : 'a = x`
$\forall A : A \Rightarrow B \Rightarrow A$	`fun f (x:'a) (y:'b) : 'a = x`