The simply-typed lambda calculus

To summarise, the Curry-Howard correspondence between (classical) propositional logic and the lambda calculus does not quite work because

- DNE (or equivalently LEM) cannot be given a general computational interpretation because it contradicts the Halting Problem
- some lambda terms are inconsistent in the sense that they are proof-terms for `A -> Void` (which is logically the same as `A -> Falsity`).

The first problem we deal with by simply dropping DNE as a principle and focussing on the remaining, *constructive* propositional logic.

The second problem we deal with by *typing* the lambda calculus. This is how Agda rejects terms such as `omega` or `Y`. This is called the *simply typed lambda calculus* (STLC), a programming language in which it is impossible to write non-terminating ("diverging") lambda terms.

Before introducing the rules let us define an important syntactic concept, that of *free variable*. These are akin to *global variables* in a programming language, relative to locally-defined variables:

```
FV (x) = { x }
FV (t u) = FV (t) ∪ FV (u)
```

```
FV (\x -> t) = FV (t) \ { x }
```

Obs: You have already seen the concept (briefly) in the Predicate Calculus, e.g. ∀-Elimination.

Example:

```
FV (\x -> (\x -> f x) (g y))
  = FV ((\x -> f x) (g y)) \ { x }
  = FV (\x -> f x) ∪ FV (g y) \ { x }
  = (FV (f x) \ { x }) ∪ FV (g) ∪ FV (y) \ { x }
  = (FV (f) ∪ FV (x)\ { x }) ∪ { g } ∪ { y } \ { x }
  = ({f} ∪ {x} \ {x}) ∪ {g, y} \ {x}
  = ({f, x} \ {x}) ∪ {g, y} \ {x}
  = {f} ∪ {g, y} \ {x}
  = {f, g, y}
```

We are going to only give rules for types made of `Void`, `Unit`, `A -> B`, and type variables. We could easily introduce the rest of the algebraic types (`Sum` and `Prod`) but we will not in order to keep things simpler.

The types are determined by *inference rules*, which are written in the style of *natural deduction sequents*: `Γ ⊢ t : A` where

- `Γ` is a list of elements of the form `x:A` where `x` is a variable and `A` is a type
- `t` is a lambda calculus term

- `A` is a type.

This judgement is interpreted as follows:

> *If the free variables of `t` have types as given in `Γ` then term `t` has type `A`.*

The rules are:

```
Γ ⊢ t : A -> B   Γ ⊢ u : A
---------------------------[App]
Γ ⊢ t u : B
```

```
Γ, x:A ⊢ t : B
--------------[Abs]
Γ ⊢ t : A -> B
```

```
--------------[Axi]
Γ, x:A ⊢ x : A
```

Example:

Show that `⊢ \x y z -> y (x z) : (A -> B) -> (B -> C) -> (A -> C)`:

```
                                                 -----------------
------------[Axi] ----------------------------[Axi]
```

```
                                                x:A->B, y:B->C, z:
A ⊢ x : A->B      x:A->B, y:B->C, z:A ⊢ Z : A
-----------------------------------[Axi] -----------------
-----------------------------------------------[App]
x:A->B, y:B->C, z:A ⊢ y : (B -> C)      x:A->B, y:B->C, z:
A ⊢ x z : B
-----------------------------------------------------------
------------[App]
x:A->B, y:B->C, z:A ⊢ y (x z) : C
-----------------------------------------------------[Abs
x 3]
⊢ \x y z -> y (x z) : (A -> B) -> (B -> C) -> (A -> C)
```

All typed programming languages require the implementation of type-checking as an algorithm. It is usually performed between *parsing* and *optimisations*.

Type inference

Spelling out all the types of all the variables is rather tedious, so programming language designers have asked the question: given a lambda term, can we *infer* (rather than just check) its type? In some sense this is the converse problem to *proof inference* where given a type (proposition) we need to find a term (proof). It's a bit like asking "*I have a proof. What is it a proof of?*"

The type inference algorithm for the STLC uses the concept of

unification, which is related to resolution (and was mentioned earlier in the module). The idea of unification is that two sets of terms which include variables can be made equal by giving the variables certain values. For example if we want to unify types `x, y` and `z, Unit -> x` (i.e. to achieve `x = z` and `y = Unit -> x`) we can substitute `x / z, y / Unit -> z` in both sets of terms. This makes both equal to `z, Unit -> z`. The set of substitutions used is called a *unifier*. Note that in general the unifier is not unique (e.g. `x / Unit -> x', y / Unit -> (Unit -> x'), z / Unit -> x'` which unify both terms to `Unit -> x', Unit -> (Unit -> x')`. However, unifiers can be more or less general. The unifier `x / z, y / Unit -> z` is *the most general unifier* for this unification.

To define the unification algorithm (for types) first we define:

- *Type reduction*: Any equation of the form `t1 -> t2 = t1' -> t2'` is broken up into equations `t1 = t1', t2 = t2'`.
- *Variable elimination*: If `x = u` is an equation where `x` is a variable and `t` any type (could be just a variable) the substitution `x / u` is applied to all other equations.
- A set of equations is *solved* if
    - all equations have form `x = t` with `x` a variable
    - every variable on the left occurs only in that place.

Basic unification algorithm

1. Change all equations `t = x` where `t` is not a variable and `x` is a variable to `x = t`.

2. Remove all equations `x = x` where `x` is a variable.
3. Select an equation `u = v` where neither is a variable and apply *type reduction*.

   If type reduction is impossible, *fail*.
4. Select an equation `x = t` where `x`, where `t ≠ x`. If `x ∈ t` then *fail* else apply *variable elimination*.
5. Repeat until *solved*.

Basic type inference

S1. Construct the type derivation tree for the term using variables instead of types.

S2. Add constraints as follows:

```
Γ ⊢ t : A   Γ ⊢ u : B
----------------------------[A = B -> C]
Γ ⊢ t u : C
```

```
Γ, x:A ⊢ t : B
--------------[C = A -> B]
Γ ⊢ t : C
```

```
--------------[B = A]
Γ, x:A ⊢ x : B
```

S3. Collect and solve the system of constraints.

> *Theorem (Soundness): If type inference for a term* `u` *computes a type* `A` *such then* `⊢ u : A`.
> *Theorem (Completeness): If there is a derivation of* `⊢ u : A` *then type inference applied to* `u` *will recover a type* `A'` *which generalises* `A`.
> *Theorem (Termination): For any term* `u` *the algorithm will terminate.*

The proofs of the theorems are outside the scope of this module.

Example: Let us infer the type of `\x y z -> y (x z)`. The derivation tree is:

```
                                    ------------------------------
-[J = B] ----------------------[K = F]
                                    x:B, y:D, z:F ⊢ x : J
         x:B, y:D, z:F ⊢ Z : K
---------------------[H = D] ------------------------------
--------------------------------[J = K -> I]
x:B, y:D, z:F ⊢ y : H          x:B, y:D, z:F ⊢ x z : I
-----------------------------------------------------[H =
I -> G]
x:B, y:D, z:F ⊢ y (x z) : G
----------------------------[E = F -> G]
x:B, y:D ⊢ \z -> y (x z) : E
-----------------------------[C = D -> E]
x:B ⊢ \y z -> y (x z) : C
```

```
--------------------------[A = B -> C]
⊢ \x y z -> y (x z) : A
```

We collect the following constraints:

```
A = B -> C
C = D -> E
E = F -> G
H = I -> G
H = D
J = K -> I
B = J
F = K
```

Applying the unification algorithm we get:

```
A = (K -> I) -> (I -> G) -> K -> G
C = D -> F -> G
E = F -> G
H = I -> G
D = I -> G
J = K -> I
B = K -> I
F = K
```

which is, up to variable renaming, the desired type.

Note: This is a *simple* algorithm. A more efficient algorithm is the *Hindley-Milner W* [PDF] algorithm. It can be easily generalised to `Prod`, `Sum`, and arbitrary type definitions.

Typing non-termination

Just like DNE violates the Curry-Howard correspondence from the logical side, so do terms like `omega` and `Y` violate it from the functional side. The programming language Agda, which is based on the CH correspondence, bans such terms, which lead to errors.

Terms such as `Y` and `omega` are non-terminating because somewhere in them you will find sub-terms `\x -> x x`. Such terms cannot type check:

```
-----------[D = B] ------------[E = B]
x:B ⊢ x : D         x:B ⊢ x : E
--------------------------------[D = C -> E]
x:B ⊢ x x : C
-----------------[A = B -> C]
⊢ (\x -> x x) : A
```

The constraints

```
D = B
E = B
D = C -> E
```

```
A = B -> C
```

after the eliminination of `D` and `E` it becomes

```
B = C -> B
A = B -> C
```

Which *fails* as variable `B` occurs on both sides of an equation.

The same failure of type inference occurs in attempting to type the `Y` combinator.

Observation

Recursive functions can be defined without lambdas in languages that allow such definitions (as opposed to allowing explicitly-defined or built in recursion combinators).

```
unsafe a = unsafe a
```

noting that for any argument we have that

```
unsafe x = unsafe x = unsafe x = ...
```

Such functions are disallowed. In general any function that Agda cannot determine that it is terminating is disallowed. The reason they are

disallowed is that non-terminating functions can have *any type*.

We can temporarily suspend Agda's termination checker and, indeed:

```
{-# TERMINATING #-}
unsafe : {A : Set} -> A -> Void
unsafe a = unsafe a
```

Through the CH correspondence `unsafe` is a proof that `A -> Falsity`, making the system inconsistent.