Contents

Chapter 1

學習狀況報告書

本報告書分為三大部分：其第一部分為我設計（但尚未實作）的程式語言—Ende—的突破性創新。作為上一部分的承接，第二部分是我的程式作品，我實作了一個非常精簡、極為抽象但卻具有代表性意義的程式語言的核心。第三部分則是一系列的筆記，是我在學習程式語言理論數年後對於型別系統所歸結出的學習心得，這樣的心得並非以感想的方式呈現，而是描述仔細的技術細節，希望讀者可以透徹理解並吸收基本的型別系統。整份報告書的內容目前都公佈於網路上各處並持續更新。畢竟目前這方面的中文資源並不多，希望我能貢獻我的知識而達到拋磚引玉的效果。

Chapter 2

Ende

Ende 是我在學習 C, Java, Scala, Haskell, Agda, Idris, Rust …等程式語言後，結合各語言特色，創造出來的心目中最理想的程式語言。當然，並沒有一個客觀的標準決定一個程式語言的好與壞，因此儘管我會根據理論上一般被視為較為細緻，甚至可以說是優勢的型別系統考量加入到系統內，我仍不免將其它部分的設計訴諸主觀偏好。在本報告書接下來的部分中，我會首先以中文表達對於 Ende 的概覽，接著包含較為技術性的內容，為了讓不只華人圈的人能理解我的想法，當初我選擇的是以英文撰寫它們，在此也就不特別把那些部分再重新翻譯回中文。除此之外，語言中包含語法和其它較為不重要抑或是瑣碎的細節被放在我的 Github 專案：Ende-readme上，全部以英文撰寫。

我想先解釋的是，Ende 到底為什麼能被我想出來。我的意思是這樣子：如果 Ende 是那種博士級的成果，例如cubicaltt，我是絕對不可能想到的。以突破性來說，Ende 的充其量只不過是「竊取」我看過語言的一些特色，然後在它們最根本的尖端玩一些小把戲，把一些特殊使用案例做成正確的或者是僅僅延伸它而已。甚至，假如你是所謂「學院派」的學者，我指的是那些推崇最純粹的純函數式語言的人，Ende 對這樣的人並沒有什麼意義。相對於理論界最有代表性的Haskell來說，我最喜歡的程式語言是Rust。幸好就某方面想，的確不是每個人都這麼在乎純函數式的典範。

在我心目中，Ende 的主要特色有 3 個，分別是：

1. 模式
2. 不定參數
3. 基礎實作

接下來的各段落將分別解釋這 3 個特色。

模式 (mode)

模式這個點子是受到了Agda這個語言的啟發，Agda 最讓我喜歡的特色是：在 Agda 中，語言結構的「種類」很少，基本上除了一般常數的宣告之外，就只有資料型態 (`data`) 的定義而已。（註：Agda 中的變數較類似於數學中的變數，並不是真正可變的，為了避免提供錯誤的印象，在此將這類語言中的變數稱為常數。）`record`（在其他語言中又被稱為 `struct`）可以被視為資料型態的另一種定義方式，但本質上和以 `data` 所定義出的資料型態是沒有差別的。

問題是這樣，在 Agda 的世界中，要怎麼模擬一般語言裡面豐富的語言結構呢？Agda 把我所謂「語言的結構」移轉凝聚到了一個特定的結構內，而這樣的結構，也是在函數式編程裡面最核心最重要，甚至賦予了函數式編程其名的那樣結構：函數。

Agda 在函數上……動了些手腳。更精確地來說，是在函數的參數上。參數能以不同的「模式」被傳入函數中，而這樣該被傳入的模式被決定於函數的定義方而非使用方，這些模式分別為：

1. **一般模式**：就一般般的模式，沒有什麼好說的……
2. **隱形模式**：一個隱形模式的參數預設下會盡量被自動推導，也就是說，你不用顯式 (explicitly) 輸入任何參數。這是怎麼辦到的呢？原因是在依值型別 (dependent types)的世界中，參數能被依賴，而被依賴者的資訊就這樣在依賴者上留下了個「印記」，就像母親小時候的教育在孩子身上留下了印記一樣：由這些印記，我們能倒推回其依賴者的身份資訊。
3. **環境模式**：環境模式中，參數同樣可以被隱式 (implicitly) 推導，但卻並不是依照「印記」而這麼做，而是根據「環境」。當有一個這樣的參數該被隱式推導，你 (我是在比喻參數) 便開始問周遭的人：「我該怎麼做？」環境中的人給予了你答覆之後，你再根據自己（其實是編譯器）的判斷做出決定。

以上是 Agda 的部分，接下來要說的是 Ende。Ende 的點子是把模式這樣的概念和「相 (phase)」進行了結合：所謂的「相」指的是程式運行的時期。一般來說（至少就我想討論的概念來說），程式只會有兩個相而已，其一是編譯時期，而另一者為執行時期。在 Agda 裡面，相的概念幾乎被抹去了，因為 Agda 是如此純粹的純函數式語言，以至於兩個相在其上得到了統合。Ende 的看法是，我把這兩個相分開處理，根據這兩個相，我們能引申得到四種模式，而這四種模式分別為……

1. **一般模式**：同樣沒有什麼好說的，參數一般在執行時期被傳入。
2. **常模式**：類似 Agda 的隱形模式，但參數只能在編譯時期被傳入。
3. **環境模式**：和 Agda 一樣，同樣是在編譯時期詢問環境而得到結論。
4. **拍 (Π) 模式**：拍模式無論在各種方面上都和一般模式差距不大，主要的差別在於，拍模式的參數能被依賴，而一般模式的沒有辦法。

總而言之，模式把語言中的結構濃縮成了函數上參數的模式，而之所以要把相和模式掛鉤，是因為這樣能更容易對於程式的效率進行保證。

以下為先前為模式所撰寫，較為技術性的英文介紹：

Modes: An Ubiquitous yet Overlooked Concept in Programming

The first programming language that I learned seriously is Java. I've quickly come out with an interpretation that whatever language features in Java correspond to parts of speech. Variables are nouns; `this` is the subject and the arguments after the name of the method call are objects; methods are verbs and interfaces are adjectives.

But what are the adverbs? Well, it's pretty obvious that programming languages aren't natural languages so not everything has an analogy in natural languages. But, hey, recently I realized there actually *is* something roughly corresponds to adverbs, which I called *argument modes*, **modes**, for short. Modes are ways to call a function or to annotate the data with more information. One may ask, "Is there more than one way to call a method in Java?" Actually, there are 4 modes in Java, but arguments of 2 modes can only be called implicitly. Now let's see an example with all 4 modes in Java.

```
public interface Stream<T> {
    // ...
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
    // ...
}
```

1. the **normal** mode: Here, the argument `mapper` is in the normal mode, which is the only mode in which arguments can always be supplied explicitly at runtime in Java.
2. the **type** mode: They are arguments of generics, which are introduced in Java SE 5. Examples are the `T` in `Stream<T>` and the `R` before `Stream<R>`. They are enclosed in angle brackets (`<>`), and some of them can only be inferred, such as the `<R>` in front of the method declaration and the 2 `?`'s.
3. the **`super`** mode: Appears as `T` in `? super T` in the source code. Always have to be passed implicitly. Defines the lower bound of the type.
4. the **`extends`** mode: Appears as `R` in `? extends R`. Similar to the previous mode.

We can see that modes provide ways to pass some information to methods and classes, either explicitly or implicitly. However, is the concept of modes also

presented in other languages? From now on, let's focus on another language, Haskell, which provides a more advanced type system compared to Java. In Haskell, functions are first-class members, and classes can abstract over higher-kinded types. With language extensions, we can use even more advanced features.

Haskell has 3 modes. Now I'm going to show all of them in the follow example. The following `fmap` function is similar to the `map` function in Java, but is more flexible by abstracting over the container type.

```
fmap :: forall f a b. Functor f => (a -> b) -> f a -> f b
```

1. the **normal** mode: The actual parameters passed at runtime, such as the values of types `a -> b` and `f a`. The same as the normal mode in Java.
2. the `forall` mode: The arguments after the keyword `forall`, which are `f`, `a`, `b`. Similar to the type mode in Java. Can only be inferred until GHC 8. After GHC 8, type arguments can be explicitly provided using the syntax `@a` Types themselves also have types, which are called kinds. for example, `f` has kind `* -> *`. Ultimately, with extension `TypeInType` turned on, kinds and types are the same, e.g. `* :: *`.
3. the **constraint** mode: The constraints before the fat arrow (`=>`). Constraints are adjectives. In the example, sending an argument (`f`) to the class `Functor` gives you a constraint, but classes can have zero or more arguments and constraints need not be classes with arguments passed in (with language extensions).

We've seen that Java and Haskell both have several modes, and many other languages also do. However, the syntaxes among the modes are "unequal" in either Java or Haskell. Every mode has its own syntax, and they don't look similar at all. In the following section, I will present a hypothetical dependently-typed system programming language without subtyping with more unified syntax and view among the possible modes.

In this hypothetical language, all modes of arguments can be called explicitly, either optionally or not. And I don't use angle brackets, because they could make parsing more difficult or ambiguous. In former C++ versions, you cannot write `A<B<C>>` instead of the uglier `A<B<C> >`, because the trailing `>>` would be parsed as an operator. Rust also faced a similar problem. Thus, the best solution in my opinion is to fully eliminate angle brackets. In fact, I'm going to use only 2 kinds of delimiters, `()` and `[]`, preserving `{}` for other uses.

1. **normal** mode: The same as above. Unlike Haskell, arguments in normal mode cannot be curried because it's closer to the behavior of current machines. If arguments in normal mode could be curried, the compiler often has to generate several uncurried functions or return lambdas. Of course, functions can still be partially applied using explicit lambdas. Examples:

```
fn factorial : (Uint) -> Uint;

fn max : (Int, Int) -> Int;
```

2. **const** mode: Before introducing const mode, I'm going to talk about const functions. Const functions are pure functions that always terminate, e.g. the above `factorial` and `max` function.

   ```
   const fn factorial : (Uint) -> Uint;

   const fn max : (Int, Int) -> Int;
   ```

   The idea is that const functions can simply be evaluated at compile time if all the arguments are known at compile time. Const functions need not to be evaluated at compile time, though. In fact, it must be able to be evaluated at runtime. In most dependently-typed languages, including this hypothetical one, types are themselves first-class values, and user-defined data types are also constants (the type, not the term of the type). All small types have type (or kind) `Type`.

   Now, return to const mode. Arguments in const mode must be evaluated in compile time, so they must be either a literal (including data types) or a const function applied with (recursively) constants. For example,

   ```
   const fn apply : [From : Type, To : Type](From -> To, From) -> To;
   ```

   This function simply applies its first argument to its second argument.

   What's the difference between the previous `max` function and the following `max2` function, though?

   ```
   const fn max2 : [_ : Int, _ : Int]() -> Int;
   ```

   First, `max2` can only be evaluated at compile time, not at runtime. More importantly, arguments in const mode are supposed to be **inferred**, resembling the type mode in java, but much more powerful. Arguments of `max2` can never be inferred, so they should not be in const mode.

   ```
   // Invocations:

   max(0, 1);
   apply(factorial, 42); // Arguments in const mode are inferred.

   // Doesn't work:
   // max2(); // Cannot infer its arguments in const mode.
   ```

   Moreover, arguments in const mode are able to be curried, so `max2[a, b]` and `max2[a][b]` are interchangeable. There are 2 reasons for it:

1. users are not supposed to do heavy calculation at compile time, so performance isn't that important.
2. In Scala, generics cannot be curried, you need to use a hack called *type lambda* if you want to partially apply a generic data type. The reason why it's a hack is not only it's syntactically verbose, but also it makes type inference unpredictable.

Now let's consider another feature of the const mode: The type of the arguments can also be inferred when declaring the function. `apply` can also be written as below:

```
const fn apply : [From, To](From -> To, From) -> To;
```

And the well-known `id` function could be declared as:

```
const fn id : [T](T) -> T;
```

3. **instance** mode: The hypothetical programming language is in fact heavily inspired by Agda. The beauty of Agda in my opinion is that it provides a uniform way to define sorts of information, therefore no `class` versus `interface` in Java or `data` versus `class` in Haskell. The boundary between types and classification of types disappeared, and the differences are represented with different modes.

   The idea of `data` and `class` doesn't necessarily need to be unified in this language, though. But in this language, instances are just another kind of argument, (usually automatically) being passed to functions in a similar syntax. instance arguments are enclosed in `[()]`, for example:

   ```
   const fn map[F, A, B][(Functor[F])](F[A], A -> B) -> F[B];
   ```

   Notice that `[(Foo)]` should not be parsed (either by human beings or compilers) as `[ (Foo) ]` because `[A]` means `[A : _]` and putting a pair of parentheses around the argument doesn't make much sense. Also notice that instance mode is more similar to normal mode than const mode is because `(Bar)` and `[(Bar)]` mean `(_ : Bar)` and `[(_ : Bar)]`, respectively.

4. **pi** mode: In a dependently-typed language, types can depend on terms. But in our language this could not be done … yet. Being enclosed in `([])`, arguments in pi mode are similar to ones in const mode such that `([T])` represents `([T : _])`, but also arguments in pi mode can be provided at runtime and cannot be curried.

Summary

	normal	const	instance	pi
`T` means	`(_ : T)`	`[T : _]`	`[(_ : T)]`	`([T : _])`
type inference	no	yes	no	yes
phase (if not `const fn`)	runtime	compile time	compile time	runtime
curryable	no	yes	yes	no
argument inference	no	yes	proof search	?
can be dependent on	no	yes	yes	yes

不定參數

如果你學的是 Java 或 C# 之類的程式語言的話，你應該會知道如果要呼叫具有不定數量參數的函數，尾端各參數的型別必須是一樣的。但為什麼要有這個限制？如果你聽說過 Haskell 的不定參數實作的話，你會知道 Haskell 中的不定參數不一定要是同一個型別，而可以是兩個型別交替，或者是任意你想要的型別排列。Haskell 的策略是在編譯時期使用介面來達到這樣的效果，在 Ende 裡面，我不使用介面，而是使用上面說的模式，使用常模式在編譯時期把型別的排列傳入，並對於宇宙 (universe)之間是否應該有繼承關係給了一個肯定的建議，同時導入了另一系列的平行宇宙，以最大化對於函數參數數量進行多型的能力。

以下為英文的詳細介紹，在用詞上可能較為精確。

The need for another hierarchy of universes in system programming languages

In the first edition of Martin-Löf's type theory preprinted in 1971, Martin-Löf introduced a universe, normally called $\mathcal{U}$, in his theory. It contains (i.e. is the type of) all types including the universe itself. After the theory was quickly found to be inconsistent by Girard, Martin-Löf then presented a so called "predicative" theory, with a hierarchy of universes, each usually written with a subscript, e.g. $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2$ ..., and the type of each universe is the higher one universe. So, for example, the type of $\mathcal{U}_0$ is $\mathcal{U}_1$, and the type of $\mathcal{U}_1$ is $\mathcal{U}_2$, and going on and on. In this article, I exploit the need for (at least) one other hierarchy of universes, written $..\mathcal{U}_n$, in which n is a natural number in the metatheory, as in the above notation $\mathcal{U}_n$.

To explain why other hierarchies of universes would be useful in a system programming language, one should know what system programming is. System programming is the idea of writing the most efficient program possible; in order to do it, one usually has to have the ability to sacrifice some abstraction and step into the world of direct pointer manipulation, abandon referential transparency, etc. in order to be the closest to the bare metal machine. The abstraction to sacrifice that I will focus on is Currying, a technique implemented

mainly by functional programming languages with the ability to regard functions as first-class objects. In layman's term, Currying is the idea of eliminating multi-argument functions with single-argument ones. To achieve it, one sends the arguments one by one to each function that all of each but the last returns a function abstracting over the rest of the arguments. In the view of category theory, that is possible because of the natural isomorphism between the hom-functor between $\mathsf{A} \times \mathsf{B}$ and C and the hom-functor between A and C^{B} in cartesian closed categories. In a system programming language, we shall minimize the gap between the code the programmers write and the native settings on bare metal. However, computers nowadays have functions that receive several arguments at once. If we apply the well-known simplification from multi- to single-argument functions, either the compiler has to generate several partially applied helper functions, or (sometimes necessarily) function pointers have to be passed around. In either case, Currying induce an overhead to the machine.

Sometimes zero-cost abstractions is achieved by fallbacking to a lower-level outset, retaining the ability to build higher abstraction. In Currying, the case we focus on, I propose to provide not only uncurried functions by default, but also special mechanisms to be generic over arity. Below is an example of such a function, in which we sum up 2 natural numbers inductively:

$$\begin{aligned}&\mathtt{sum}(\mathtt{0} &&: \mathbb{N}, \mathtt{r} : \mathbb{N}) : \mathbb{N} = \mathtt{r}\\ &\mathtt{sum}(\mathtt{Succ}(\mathtt{l}) &&: \mathbb{N}, \mathtt{r} : \mathbb{N}) : \mathbb{N} = \mathtt{Succ}(\mathtt{sum}(\mathtt{l}, \mathtt{r}))\end{aligned}$$

Note that $\mathtt{sum}(\mathtt{a})$ shouldn't be allowed without an explicit lambda because of what is explained. The next one is a function summing up its arbitrarily many arguments, being generic over arity:

$$\begin{aligned}&\mathtt{sumAll}() &&: \mathbb{N} = \mathtt{0}\\ &\mathtt{sumAll}(\mathtt{head} : \mathbb{N}, \ulcorner \mathtt{tail} \lrcorner : ?_1) &&: \mathbb{N} = \mathtt{sum}(\mathtt{head}, \mathtt{sumAll}(\ulcorner \mathtt{tail} \lrcorner))\end{aligned}$$

$\ulcorner \mathtt{foo} \lrcorner$ can be seen as flattening a list of arguments. As usual, the pattern matching syntax $\ulcorner \lrcorner$ is for deconstructing value, and the term syntax $\ulcorner \lrcorner$ is for constructing one. Specifically, if someone writes $\mathtt{sumAll}(1, 2, 3)$, pattern matching matches $\mathtt{head}$ to 1 and $\mathtt{tail}$ to $2, 3$; the $\ulcorner \lrcorner$ syntax is meant to collect several arguments. On the other hand, the recursive call on the right hand side is applied with the arguments $\ulcorner \mathtt{tail} \lrcorner$. Here, the same syntax expands the arguments we collected.

What should be the type of $\ulcorner \mathtt{tail} \lrcorner$? The most obvious choice is that it should be a homogeneous list generic over some type T. In lots of programming languages without Currying, variable arguments is implemented that way. However, it could sometimes be too restrictive. For example, it cannot express the arguments in the argument list is alternating between some type T and another type U. What is the way to build abstraction in programming languages? An obvious and primitive notion are functions. Therefore, directly or indirectly, what should be filled into $?_1$ should call a compile-time function that generates the argument list. Why "compile-time"? We can then monomorphize the function upfront before running it, making the overhead minimal. The whole idea of

compile-time functions and arguments is described previously with the concept of modes. I'm just going to make a very brief review of the notion we need to proceed.

I introduce another way to pass arguments, called const mode, in which the arguments must be supplied at compile time. Arguments in const mode are written inside brackets ([]) and can be inferred. Actually, the type of the argument list should be an argument in const mode. We can call purely functional const function in const mode. Now the sumAll example can be fully defined using const mode and a const function called replicate, which is a function generating the list of arguments of the same type replicating $\mathfrak{n}$ times:

$$\begin{aligned}
&\mathsf{replicate}[0:\mathbb{N}]\quad(T:\mathcal{U}):?_2 = \ulcorner\lrcorner\\
&\mathsf{replicate}[\mathsf{Succ}(n):\mathbb{N}](T:\mathcal{U}):?_2 = \ulcorner T, \mathsf{replicate}[n](T)\lrcorner\\
&\mathsf{sumAll}\quad()\quad:\mathbb{N}=0\\
&\mathsf{sumAll}[\mathsf{Args}:\mathsf{replicate}(\mathbb{N})](\mathsf{head}:\mathbb{N},\ulcorner\mathsf{tail}\lrcorner:\mathsf{Args}):\mathbb{N}=\mathsf{sum}(\mathsf{head},\mathsf{sumAll}(\ulcorner\mathsf{tail}\lrcorner))
\end{aligned}$$

How would $?_2$ be, then? If we say the type of a heterogeneous list is the list of the types of each member of the list, we might want the type of the list to also be heterogeneous. Fortunately, we don't have to recurse forever to build higher-order argument lists because of a feature of normal type systems: The type of the type of any value is a universe. With a cumulative hierarchy of universes $\mathcal{U}_0 <: \mathcal{U}_1 <: \mathcal{U}_2...$, we can assign each $\mathcal{U}_n$ a universe $..\mathcal{U}_n$ which is a list the types of which is $\mathcal{U}_n$ and have arbitrary length; because of cumulativity, the types of all arguments belong to a sufficiently large $\mathcal{U}_n$.

Therefore, $?_2$ would be $..\mathcal{U}_n$, for a sufficiently large $\mathfrak{n}$. The new hierarchy of types should also be cumulative, and it seems straightforward to say that $..\mathcal{U}_0 <: ..\mathcal{U}_1 <: ..\mathcal{U}_2...$, i.e. $..\mathcal{U}_n$ is covariant over $\mathcal{U}_n$.

基礎實作

在 Agda 裡面，環境模式的作用和 Haskell 裡面的介面類似：環境模式中你會向環境進行詢問，而環境中的其他人事實上比喻的是所謂的「實作」。但為何 Haskell 要創造新的語言結構，亦即介面，而不用原本的 record 呢？乍看之下，它們似乎是很不一樣的概念，但卻在 Agda 中得到了統合。Haskell 不像 Agda 這麼做的原因是介面之間有繼承關係，任何實作子介面的資料型態都必定也實作母介面。在 Ende 中，類似於 Agda，並沒有介面這樣的獨立概念。和 Agda 的不同之處在於，Ende 確實提供了一個能定義介面間繼承關係的方法，它被稱為「基礎實作」。基礎實作事實上不是只能用來實作介面間的繼承，但..... 我也想不到它的其他用處了..... 不過就語言的角度來看，基本實作只不過是語言中的一個後門而已，和傳統的實作差異並不大，因此我認為它嚴格上能算是某些概念的統合，而不是硬生生的把兩個不同的概念試圖牽扯在一起。我在探索 Ende 的時候還意外發現了一件事，事實上我不僅能模擬介面，連傳統物件導向中的類別也能被模擬，但問題.....又是出在繼承，我目前還不確定那樣的問題是否能被解決。

有關基礎實作的資訊，我並不像上面的兩個創新曾在部落格上發布獨立的英文文章，詳細的細節被寫在 Ende-readme，但可能需要前後文較容易讀懂，以下仍然節錄 Ende-readme 中談論基礎實作的小部分

fundamental impl

What we don't have yet is the ability to define one `record` to be a supertrait of another `record`. In other words, asserting if you implement a trait, another trait must be implemented. You may ask, isn't the above `impl` functions enough? Not really. Imagine you want to define an `Abelian` trait, the code you need to add would be:

```
record Abelian[T] = abelian {
    unit : T,
    fn append(self : T, T) -> T,
    fn inverse(self : T) -> T,
};

impl abelianToGroup[T][(Abelian[T])] -> Group[T] = {
    group {
       unit => unit,
       append => append,
       inverse => inverse,
    }
};
```

That's a lot of boilerplate! The code is very similar to implementing `Monoid`s for `Group`s; we have to write this kind of code again and again while creating an inheritance tree of `record`s. That is not tolerable. Imagine if we can write code like this:

```
-- The type `Abelian` carries no additional data,
-- but it extends the trait `Group`.
data Abelian[T] = abelian;
-- The extension happens here.
impl abelianExtendsGroup[T] ;
    -> Extends[Abelian[T], Group[T]] = extension
```

It's really concise code! But how do we get an instance of type `Group[T]` given the instances of types `Abelian[T]` and `Extends[Abelian[T], Group[T]]`? It's actually a hack: we need more `impl`s. I'll try to explain it.

The Hack

First, I'll provide the hacky part of the source code:

```
record Extends[A, B] = extension;

-- Ignore the `const` keyword before `fn` for now.
const fn implicitly[T][(inst : T)] -> T = inst;

fundamental impl superTrait[A, B][(A, Extends[A, B])]
    -> B = implicitly[B];
```

The basic idea is that no matter what `A` and `B` are, if `impl`s of `A` and `Extends[A, B]` are in scope, `impl` of `B` is made in scope. In the revised version of example of `Abelian`, because both `impl`s of `Abelian[T]` and `Extends[Abelian[T], Group[T]]` are in scope, `impl` of `Group[T]` is also made in scope. Why is the keyword `fundamental` before the `impl superTrait` required then? To know why it's needed, we need to go deeper to know how an `impl` is found.

Searching for `impl`s

First, we search for `impl` objects. We add an `impl` object in scope to the current **`impl` context** if the trait it implements is also in scope.

Second, we add the `impl` functions to the `impl` context. There is a necessary limitation of normal `impl` functions: a normal `impl` function can only have a return type that is not a variable, so the example below does't work:

```
-- Doesn't compile.
-- impl abuseOfImplicitly[T] -> T = implicitly[T];
```

The reason why some limitation is needed is because we want to make searching `impl`s more predictable, so that we can filter out the `impl` functions that doesn't retern an `impl` of a type in scope. Without the limitation, the `impl` searching process could stuck at some weird recursive `impl`.

And `fundamental impl`s surpass that limitation. It has to be used more carefully, but I don't think there's a lot of uses of it. In fact the only one I can think of is trait inheritance. The `impl`s of the return types of the `fundamental impl`s are recursively added to the `impl` context no matter whether the type it implements is in scope or not.

Chapter 3

PTS (Pure Type Systems)

I'll only present the type checking part and omit other things because the report is limited to have below 30 pages; The full implementation including tests is put on a Github repository, pts. Currently, this project can only be used as a library, not an application, because I haven't dealt with parsing stuff.

This is an implementation of *pure type systems* written in the Rust programming language. It's basically a rewrite from the Haskell version, Simpler, Easier!

Installation:

1. First make sure Rust (and obviously also `git`) has already been installed on your machine.
2. Clone this repository: `git clone https://github.com/AndyShiue/pts.git`
3. Navigate to the root of the project: `cd pts`
4. Run `cargo test` to run all the tests in the project. It might takes some time.

Originally, lambda calculus is invented to be a Turing-complete model of computation. Subsequent works add type systems on top of the lambda calculus, usually making it **not** Turing-complete, but stronger type systems lead to the ability to write mathematical proofs in it. Pure type systems are a generalization of the lambda cube, which consists of the simply typed lambda calculus, system F, calculus of constructions, etc.. They can be served as fundamental core

languages for real-world functional programming languages. Dependent types, the idea that types can be dependent on terms, is also a very powerful feature of a type system. The users of this library can implement their own pure type systems either having or not having dependent types. More generally speaking, with this library, you can define your own pure type systems, consisting one or more, or even infinite sorts, and the relationship between them.

Source code below.

```rust
use std::fmt::{self, Debug, Display};
use std::hash::Hash;
use std::collections::{HashSet, HashMap};

// A newtype wrapper representing a symbol.
#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub struct Symbol(pub String);

impl Display for Symbol {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}", self.0)
    }
}

// The type of terms generic over pure type systems.
#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub enum Term<System: PureTypeSystem> {
    Var(Symbol),
    App(Box<Term<System>>, Box<Term<System>>),
    Lam(Symbol, Box<Term<System>>, Box<Term<System>>),
    Pi(Symbol, Box<Term<System>>, Box<Term<System>>),
    Sort(System::Sort),
}

impl<System: PureTypeSystem> Display for Term<System> {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        use Term::*;
        match *self {
            Var(Symbol(ref str)) => write!(f, "{}", str),
            App(ref left, ref right) =>
                write!(f, "({} {})", left, right),
            Lam(ref bound, ref ty, ref inner) =>
                write!(f, "(\\{}: {}. {})", bound, ty, inner),
            Pi(ref bound, ref left, ref right) =>
                write!(f, "({}: {}) -> {}", bound, left, right),
```

```
            Sort(ref sort) => write!(f, "{}", sort),
        }
    }
}

// The trait classifying a pure type system.
// It consists of 3 things, respectively:
// 1. Its sort, this is represented as an associated type.
// 2. `axiom`, which is a function from any sort to its super-sort.
//    It returns an `Option` because some sort may not belong to any other sorts,
//    i.e. it's already the largest sort in the system.
// 3. `rule`, the purpose of this function is to specify the type of a function
//    from the type of its argument and its return type.
//    If a function has type T1 -> T2, the type of T1 is s1 and the type of T2
//    is s2, then the type of the whole function is rule(s1, s2).
//    Again, `rule` returns an `Option` because the function type isn't always
//    well-formed.
pub trait PureTypeSystem: Clone + Debug {
    type Sort: Copy + Clone + Debug + Display + Eq + Hash;
    fn axiom(sort: Self::Sort) -> Option<Self::Sort>;
    fn rule(s1: Self::Sort, s2: Self::Sort) -> Option<Self::Sort>;
}

impl<System: PureTypeSystem> Term<System> {

    // The starting point of type checking.
    pub fn type_check(self) -> Result<Term<System>, String> {
        self.type_check_with_context(HashMap::new())
    }

    // And the real implementation of the type checking.
    // We need to store typing information in a map called `context`.
    pub fn type_check_with_context(self, context: HashMap<Symbol, Term<System>>)
        -> Result<Term<System>, String> {
        use self::Term::*;
        match self {
            // Simply lookup the context if I hit a variable.
            Var(v) => {
                match context.get(&v) {
                    Some(ty) => Ok(ty.clone()),
                    None => Err(format!("Cannot find variable {}.", &v.0))
                }
            }
            // If I hit an application ...
            App(left, right) => {
                // First see if the left hand side type checks.
```

```
        let left_ty = try!(
            left.type_check_with_context(context.clone())
        );
        // If `left_ty` isn't a function in its `whnf` form, output an
        // error.
        match left_ty.whnf() {
            Pi(bound, ty_in, ty_out) => {
                // Let's then type check the right hand side.
                let right_ty = try!(right.clone()
                                        .type_check_with_context(
                                            context.clone())
                                        );
                // If the type of the right hand side matches the type
                // of the argument of the `Pi` type, substitute the
                // return type with the right hand side.
                // The return type can have free occurences of the bound
                // variable because now we are working with dependent
                // types.
                if right_ty.beta_eq(&ty_in) {
                    Ok(ty_out.substitute(&bound, &right))
                } else {
                    // If the types doesn't match, return an error.
                    Err(
                        format!(
                            "Expected something of type {}, \
                             found that of type {}.",
                            ty_in, right_ty
                        )
                    )
                }
            }
            left_ty =>
                Err(format!("Expected lambda, found value of type {}.",
                            left_ty))
        }
    }
    // If I hit a lambda ...
    Lam(bound, ty, inner) => {
        // Check if the type of the argument is well-formed,
        // if it is, proceed ...
        try!(ty.clone().type_check_with_context(context.clone()));
        let mut new_context = context;
        // Insert the bound variable into the new context.
        new_context.insert(bound.clone(), *ty.clone());
        // And type check the right hand side of the lambda with the new
        // context.
```

```
        let inner_type =
            try!(inner.type_check_with_context(new_context));
        Ok(Pi(bound, ty, Box::new(inner_type)))
    }
    // If I hit a `Pi` ...
    Pi(bound, left, right) => {
        // First, type check the type of the bound variable.
        // It must be a `Sort`, otherwise output an error.
        if let Sort(left_sort) = try!(left.clone()
                                          .type_check_with_context(
                                              context.clone()
                                           )
                                          .map(Term::whnf)) {
            // Create a new context, the same as what we did in the case
            // of `Lam`.
            let mut new_context = context;
            // Insert the bound variable.
            new_context.insert(bound, *left);
            // type check the right hand side of the `Pi` with the new
            // context.
            let right_kind = try!(right.clone()
                                       .type_check_with_context(
                                           new_context
                                        )
                                       .map(Term::whnf));
            // Again, check if the type of the return type is a `Sort`.
            if let Sort(right_sort) = right_kind {
                // Call `rule` to get the type of the whole function
                // type.
                let new_sort =
                    System::rule(left_sort.clone(), right_sort.clone());
                match new_sort {
                    Some(sort) => return Ok(Sort(sort.clone())),
                    // If such rule doesn't exist, output an error.
                    None => {
                        let error_message = format!(
                            "Rule ({}, {}, _) doesn't exist.",
                            left_sort, right_sort
                        );
                        Err(error_message)
                    }
                }
            } else {
                Err(format!("Type {} isn't inhabited.", right))
            }
        } else {
```

```
                    Err(format!("Type {} isn't inhabited.", left))
                }
            }
            // Finally, type check the sorts. It's an easy case. We just need to
            // call `axiom`.
            Sort(sort) => {
                match System::axiom(sort) {
                    Some(new_sort) => Ok(Sort(new_sort.clone())),
                    None =>
                        Err(format!("Sort {} doesn't have a super-sort.", sort)),
                }
            }
        }
    }

    // This function substitutes all occurences of the variable `from` into the
    // term `to`.
    pub fn substitute(self, from: &Symbol, to: &Term<System>) -> Term<System> {
        panic!("Source code omitted for brevity.");
    }

    // The purpose of this function is to get the *Weak Head Normal Form* of a
    // term.
    pub fn whnf(self) -> Term<System> {
        panic!("Source code omitted for brevity.");
    }

}
```

Chapter 4

Lambda Calculus 系列文章

以下為我寫的 lambda calculus 系列教學文章，所有文章集結於我的部落格上，因篇幅關係，這邊僅節錄其中幾篇。

Lambda Calculus (2 - STLC)

講完 untyped lambda，接下來就要講 typed lambda 啦。顧名思義，typed lambda 就是有 type 的 lambda，因此必須要被 type check；一個合理的 type checker 不會讓所有程式都 type check 成功，而是只接受特定的一些「正確的」程式，因此讓程式的錯誤能在編譯時期就被找出來。

定義

我們首先從 simply typed lambda calculus 開始，以下將會取簡稱 STLC。STLC 的 term 中，唯一的差別就在於 lambda 的綁定變數增加了型別標記，原本的 lambda

$\lambda x.t$

會變成

$\lambda x : T.t$

其中的 T 是一個 type，代表了參數應要有的型態，假如傳入的參數有著其它型態，type check 便不會通過。

現在問題來了，lambda 的型態又是什麼？一個 lambda 的資訊主要由兩個部分構成，一個是參數的 type，另外一個則是值的 type，我們把一個 lambda 的 type 寫成 $T \to U$，其中 T 是參數的 type，U 則是回傳值的 type。一個 type 為 $T \to U$ 的 lambda，在接收一個 type 為 T 的參數後，會傳回 type 為 U 的運算式。

全部組合起來吧，我們的新系統裡面會有 term 和 type 兩個階層，接下來我要分別定義 term 和 type 由什麼組成，其中 type 包含了：

1. Base，這是最基本的 type，它不能被當成一個函數。假設 $t_1 : \mathsf{Base}$(這代表著 t_1 的型別是 Base)，則 $t_1 t_2$ 絕對無法通過 type check。
2. $T_1 \to T_2$，對所有 type T_1 和 T_2，函數型別 $T_1 \to T_2$ 是一個 type。也就是說，我們能夠從 Base 開始，建構一系列更複雜的 type，例如像 $\mathsf{Base} \to \mathsf{Base}$、$\mathsf{Base} \to (\mathsf{Base} \to \mathsf{Base})$、$(\mathsf{Base} \to \mathsf{Base}) \to \mathsf{Base}$……

term 則包含了：

1. 變數
2. 對所有變數名稱 x，所有 term t，和所有 type T，$\lambda x : T.t$ 是一個 term。
3. 對所有運算式 t_1，和所有運算式 t_2，$t_1 t_2$ 是一個 term。

在以上對 term 和 type 的定義中，我省略了這句話：

> 除上述之外，沒有任何其它東西是運算式。

因為這是顯而易見的，從今以後我都將省略之，並且我要導入一個更簡潔的表示法：

$$
\begin{aligned}
\mathsf{term}(t) ::= \quad & \mathbf{var}(x) \\
\mid \quad & \lambda x : T.t \\
\mid \quad & t\,t \\
\mathsf{type}(T) ::= \quad & \mathsf{Base} \\
\mid \quad & T \to T
\end{aligned}
$$

這樣的表示法相當於上面用文字對 term 和 type 的定義，兩相對照之下，應該不難理解它的意思。用 Haskell 可以寫成這樣：

```haskell
newtype Symbol = Symbol String
data Term = Var Symbol
          | Lam Symbol Type Term
          | App Term Term
data Type = Base
          | Arrow Type Type
```

值得一提的是，$\to$ 是**右結合**的，換句話說，$T \to U \to V$ 代表的是 $T \to (U \to V)$。為什麼呢？還記得前面提到的柯里化 (Currying) 吧，f a b 中的 f 可以被理解成接收兩個參數的函數，假設 a 的型別為 A、b 的型別為 B，則 f 的型別會是 $A \to (B \to C)$。在大多的情況中，我們都希望柯里化能運作，因此我們把 $\to$ 定義成右結合的。

歸約

STLC 的歸約和 untyped lambda 中的歸約差不多，畢竟兩者唯一的差別就只在綁定變數上多了個型別，而目前型別在執行時期是不需要的，不過現在我們大可不必管歸約策略了，為什麼呢？因為在 STLC 中，所有 term 都有 NF，並且任何 term 都能在有限步 β-歸約後達到它的 NF，所以你大可把所有 term 歸約成 NF，這是型別系統帶來的好處。當然，這代表著我們沒有辦法在 STLC 裡面跑一個無窮迴圈，例如像是前面提到的 Ω 在 STLC 中便不存在，也就是說 STLC 並不是圖靈完全的。

為什麼 Ω 不存在？複習一下，$\Omega \overset{\mathrm{def}}{\equiv} (\lambda x.xx)(\lambda x.xx)$，現在假設我們要為它的綁定變數加上型別，變成 $\Omega \overset{\mathrm{def}}{\equiv} (\lambda x : ?.xx)(\lambda x : ?.xx)$，問號中要填入什麼？這個問題是無解的。

type checking

接下來要講 type checking 了，先來讓大家嘗鮮一下，以下這堆莫名其妙的符號解釋了 STLC 的 type checking 是怎麼進行的：

$$(\mathrm{var}) \quad \Gamma \vdash x : T \quad \text{if } x : T \in \Gamma$$

$$(\mathrm{appl}) \quad \frac{\Gamma \vdash t_1 : T \to U \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 t_2 : U}$$

$$(\mathrm{abst}) \quad \frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash (\lambda x : T.t) : T \to U}$$

接下來我會用直覺解釋要怎麼進行 type check，然後再引導讀者理解上面這些符號代表的意義。

概述

首先我先來定義何謂 term 的內外層，以 untyped lambda 來說，$(\lambda x.xx)$ 在 $(\lambda x.xx)(\lambda x.xx)$ 的內層，而 xx 又在 $(\lambda x.xx)$ 的更內層。注意綁定變數 x 不在 $(\lambda x.xx)$ 的內層，那只是一個變數而已，不是一個 term。

還記得目前有哪些 term 吧：變數、lambda、函數調用。當我們在 term 的最外層時，可能遇到的肯定是這三者的其中一個，現在我打算對這三個可能性逐個擊破，首先先講遇到函數調用時怎麼辦：當遇到函數調用時，有兩個內層的 term，左邊那個必須是函數，右邊則是丟給左邊那個函數的參數，而且右邊那個 term 的型別必須要和左邊函數綁定變數後面標示的型別互相吻合。我們只要分別對左右邊進行 type check，再確定左邊是不是個函數，而且它標示的型別和右邊 term 相吻合就好。

如果遇到了一個 lambda $\lambda x : T.t$，則我們要把 x 的型別是 T 這個事實記錄下來，然後在這個事實的大前提下，檢查 t 的型別是什麼。就把 t 的型別叫做 U 好了，那麼整個 lambda 的型別就是 $T \to U$，因為有可能 $x \in \mathrm{FV}(t)$，所以 x 的型別資訊在檢查 t 的型別時是必要的。

用更精確的詞來說，當遇到一個 lambda $\lambda x : T.t$ 時，我們把 x 的型別是 T 這樣的資訊存放在**語境**中，然後用這樣的語境檢查 t 的型別，如果在 lambda 的內層遇到另一個 lambda 要怎麼辦？只要把它的綁定變數也加入語境就好了，然後用這個更改過的語境檢查更內層 term 的 type。

最後，如果遇到變數怎麼辦？要是單獨一個 x 在檢查型別時肯定是會失敗的，我們根本沒辦法決定 x 的型別啊！但 $\lambda x : Base.x$ 卻應該被成功編譯，為什麼呢？因為在檢查內層的 x 時，我們的語境已經含有了 $x : Base$ 的資料，現在能用這樣的資料檢查內層 x 的型態。換句話說，遇到變數時，我們只要在語境中進行搜索就好了。

回到那堆符號上。

符號

$$(var) \quad \Gamma \vdash x : T \quad \text{if } x : T \in \Gamma$$

我們把 $\Gamma \vdash x : T$ 叫做一個**裁決**，裁決什麼？裁決了 x 的 type 是 T。由什麼裁決？由語境 Γ，Γ 代表著語境，要如何進行這個裁決呢？看看後面的 $\text{if } x : T \in \Gamma$，這告訴我們如果在語境 Γ 中找到了 x 的 type 是 T，我們便能進行裁決：$\in$ 讀成屬於。整個組合起來是這樣：如果 $x : T$ 在語境 Γ 中，則由 Γ 可以裁決 x 的型別是 T。

不難對吧？而且還很像是廢話。

$$(appl) \quad \frac{\Gamma \vdash t_1 : T \to U \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 t_2 : U}$$

你應該立刻會注意到那條橫線，橫線上頭的裁決被稱為**前提**，下面則是**結論**。要是前提都成立的話，結論便也成立，前提有兩個，第一個是 $\Gamma \vdash t_1 : T \to U$，這是一個 Γ 對 $t_1 : T \to U$ 的裁決；第二個是 $\Gamma \vdash t_2 : T$，這是一個 Γ 對 $t_2 : T$ 的裁決；結論是 $\Gamma \vdash t_1 t_2 : U$，這是一個 Γ 對 $t_1 t_2 : U$ 的裁決。全部組合起來，如果在 Γ 裁決 $t_1 : T \to U$ 和 $t_2 : T$ 的前提下，Γ 對 $t_1 t_2 : U$ 的裁決成立。以目前來說也能反過來讀，要檢查 $t_1 t_2$ 的型別，只要檢查 Γ 能否裁決 $t_1 : T \to U$ 和 $t_2 : T$ 就好，然後回傳結果 U。這也就是我們 type check 函數調用的方法，接下來要講如何 type check lambda：

$$(abst) \quad \frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash (\lambda x : T.t) : T \to U}$$

這邊出現了新的符號：逗號，只要解釋這個逗號的意義，整個應該就能被讀懂了。$\Gamma, x : T$ 代表的是在 Γ 上加上 $x : T$ 的新語境，一整個讀起來會是這樣：若 Γ 加上 $x : T$ 的新語境能裁決 $t : U$，Γ 便能裁決 $x : T.t$ 的型別是 $T \to U$，type check 時只要反過來讀就好。

實例

來舉個實際例子好了，我們試著找找看 $(\lambda x : Base \to Base.x)(\lambda y : Base.y)$ 的型別。首先看看最外層的 term，它是一個函數調用，因此我們應該要先試圖檢

查左右兩邊的型別，然後根據這樣的資料檢查一整個 term 的型別，用符號表示是這樣：

$$(\text{appl})\quad \frac{\Gamma \vdash (\lambda x : \text{Base} \to \text{Base}.x) : ?_1 \qquad \Gamma \vdash (\lambda y : \text{Base}.y) : ?_2}{\Gamma \vdash (\lambda x : \text{Base} \to \text{Base}.x)(\lambda y : \text{Base}.y) : ?_3}$$

只要解決了 $?_1$ 和 $?_2$，你就知道 $?_3$ 要填入什麼。先解決 $?_1$，注意到了這個 term 是個 lambda，因此我們要在這裡套用 (abst)：

$$(\text{abst})\quad \frac{\Gamma, x : \text{Base} \to \text{Base} \vdash x : ?_4}{\Gamma \vdash (\lambda x : \text{Base} \to \text{Base}.x) : ?_1}$$

現在我們想知道 $?_4$ 是什麼。我們已經遇到了一個變數，因此接下來應該要用 (var) 求得 $?_4$。在這邊，我們根本不需要 Γ，因為 $\vdash$ 左邊的語境中已經有 x 的型別資訊了，而這代表著 Γ 可以是個**空語境**，寫作 $\emptyset$，我來提出一個解釋語境究竟是什麼的想法：語境是自由變數的型別資料，為什麼檢查 lambda 的型別時要把約束變數的資料加入到語境裡面呢？因為原本在 $\lambda x.t$ 中不自由的 x 變數，到了 t 裡面變成自由的了！因此檢查 t 的型別時要在語境中加入 x 的資料。回到原本的主題上，Γ 可以是個空語境代表著什麼？這代表著我們不需要任何其它自由變數的資料就能進行判決，由這個判決，可以得到 $x : \text{Base} \to \text{Base}$，也就是說 $?_4 = \text{Base} \to \text{Base}$。回到 (abst) 上，要來解 $?_1$，我們可以把 $?_4 = \text{Base} \to \text{Base}$ 填入了：

$$(\text{abst})\quad \frac{\emptyset, x : \text{Base} \to \text{Base} \vdash x : \text{Base} \to \text{Base}}{\Gamma \vdash (\lambda x : \text{Base} \to \text{Base}.x) : ?_1}$$

根據 (abst)，我們可以得到 $?_1 = (\text{Base} \to \text{Base}) \to (\text{Base} \to \text{Base})$。用和上述類似的方法，我們得到 $?_2 = \text{Base} \to \text{Base}$，因此上面提到過的 (appl) 變成這樣：

$$(\text{appl})\quad \frac{\emptyset \vdash (\lambda x : \text{Base} \to \text{Base}.x) : (\text{Base} \to \text{Base}) \to (\text{Base} \to \text{Base}) \qquad \emptyset \vdash (\lambda y : \text{Base}.y) : \text{Base} \to \text{Base}}{\Gamma \vdash (\lambda x : \text{Base} \to \text{Base}.x)(\lambda y : \text{Base}.y) : ?_3}$$

得到結論，$\emptyset \vdash (\lambda x : \text{Base} \to \text{Base}.x)(\lambda y : \text{Base}.y) : \text{Base} \to \text{Base}$。也就是說我們不需要其他自由變數的資料，就能得到 $(\lambda x : \text{Base} \to \text{Base}.x)(\lambda y : \text{Base}.y)$ 的型別是 $\text{Base} \to \text{Base}$。

補充

啊對了，差點忘記說一件事情，你可能會想問，假設有個函數 $\lambda x : \text{Base}.x$，要怎麼丟一個 Base 型態的參數給它呢？我根本就沒提到怎麼建構 type 為 Base 的 term 啊！答案是……沒有辦法…… 至少在我介紹的這個最簡化的系統裡面沒辦法，但就算沒辦法建構型別為 Base 的 term，這個系統還是有某些用處的。這個系統內存在 type 為 $\text{Base} \to \text{Base}$，或者 $(\text{Base} \to \text{Base}) \to (\text{Base} \to \text{Base})$ 的 term，即使沒有 Base 型態的 term，我們還是能由系統內的 term 進行一些有意義的運算。。

Lambda Calculus (6 - Curry-Howard correspondence)

這篇文章會討論有點不同的主題，從這個問題開始：你要怎麼寫一個 term，使得他的 type 為 ∀X.X？

這是一個有點有趣的型別，屬於它的 term 必須要能被當作任何型別的值，顯然這有點無理。假如你用的是 Haskell，或者是任何一個普通的程式語言，事實上可行的辦法是讓這個函數永遠不要回傳值，舉例來說讓它進入無窮迴圈，或者是直接終止程式。不過這些基本上都是作弊，不是嗎？除了作弊之外，的確你是不應該有辦法寫出這樣的 term 的。

有沒有可能設計一個沒辦法作弊的程式語言呢？有的，沒辦法作弊這樣的特性事實上有個聽起來比較專業的名詞：**一致的**。前面我們介紹的所有 lambda calculus 都是一致的，也就是說不存在型別為 ∀X.X 的 term，一般的程式語言是在 lambda calculus 的基礎上加了太多其它結構，讓你有辦法作弊。為什麼那些程式語言要被這麼設計呢？因為作弊有時的確是好的，當你寫真正的程式時，你確實可能希望進入一個無窮迴圈，永遠不要結束函數。在先前介紹的各種最精簡的 typed lambda 裡面，所有函數都有 NF，因此我們無法辦到這點。

我要來直接切入一致性這樣的特性有什麼特別的意義了，接下來給出的資訊可能有點多，如果還沒辦法理解的話就一次背下來再繼續讀吧。

二十世紀中的數學家們發現了一件事情，型別系統和邏輯是有直接的對應的，更精確地說是如此：不同的 type 可以被視為不同的**命題**，而每個 type 的 term 則是它的**證明**，這樣的關係被稱為**柯里—霍華德對應**。

先來解釋一下命題是什麼：一個命題可以被看為一個邏輯中可或不可證明的合法語句。假如一個命題無論在任何前提下都成立的話，則它被稱為**套套句**，既然套套句是命題，那麼便存在（至少）一個 term 的 type 是該命題對應到的 type。

舉個最早被發現的對應吧，由柯里在 1934 年發現。這樣的對應是：型別系統裡面的 →(函數型別) 對應到了邏輯裡面的 →(蘊含)，如果讀者不熟悉邏輯裡面的蘊含的話，這邊稍微介紹一下：A → B 在邏輯裡面的意思是A **蘊含** B，換句話說就是由 A 的真實性可以推導至 B 的真實性。單單只用 →，我們就能建立某些套套句，舉例來說好了，像是 A → A(由任何事實能推導至它本身)，或者是 A → (B → A)，（如果一個命題 A 是事實，那麼任何其它命題 B 是事實都能推導出 A 是事實。）咦，有沒有覺得有種熟悉的感覺？

事實上，這不就是 **I** 和 **K** 的 type 嗎？這印證了套套句能被證明這樣的說法。但 **I** 和 **K** 的 type 前面的 ∀ 又要怎麼解釋呢？根據柯里—霍華德對應，∀ 對應到了「對於所有 type……」，type 被對應到了命題，所以事實上也能解釋成「對於所有命題……」。這麼一來，**I** 的 type 能被解釋成對於所有命題 A，A 蘊含自身；**K** 的 type 也能被類似的方式解讀，只要在前面加上對於所有命題

A 和 B 就可以了。不存在 type 為 $\forall X.X$ 的 term 也因此會是型別系統裡很重要的特色，一旦系統不一致，任何命題都能被證明，而顯然對於一個邏輯來說這並不是我們想要的。在 CoC 裡面 $\forall$ 和 $\rightarrow$ 已經被統一，$\forall A....$ 指的事實上是 $(A : \star) \rightarrow ...$，其實我們也能有 $(A : \mathrm{Bool}) \rightarrow ...$ 或 $(A : \mathrm{Nat}) \rightarrow ...$，分別指對於所有布林值和對於所有自然數，依此類推。

現在我們不知不覺進入了一階邏輯的領域，這是型別理論和傳統邏輯不一樣的地方；在傳統邏輯中，一階邏輯的變數和命題是兩種完全不同的東西，而在型別論中，命題只不過是 $\star$ 的元素罷了，而述詞（接收值傳回命題的函數）或高階述詞也能被輕鬆定義。

命題邏輯除了 $\rightarrow$ 之外還有很多運算子，現在我將一一定義。

首先，我把 $\forall X.X$ 當作一個最無法被證明的命題，一旦它能被證明，任何命題都能被推導出來，$\forall X.X$ 在符號上因此被寫作 $\bot$，顯然 $\bot$ 蘊含任何命題。這邊來練習寫出第一個證明吧，我想要證明的是 $(A : \star) \rightarrow \bot \rightarrow A$。證明應該很顯而易見吧，只要寫出一個 type 為 $(A : \star) \rightarrow \bot \rightarrow A$ 的 term 就好了，而這樣的 term 是 $\lambda A : \star.\lambda X : (A : \star) \rightarrow A.XA$，有趣的是我們調用了一個永遠不可能存在的函數（型別為 $(A : \star) \rightarrow A$），以符合我們想要證明的命題。

$\bot$ 有時又被稱為**矛盾**。現在我們可以定義否定了，一個命題的否定能被證明代表它本身能推得矛盾，寫成符號的話：

$$\neg A \stackrel{\mathrm{def}}{\equiv} A \rightarrow \bot$$

如果你有一點邏輯的基礎的話，你可能會猜想，現在我們能用 $\rightarrow$ 和 $\neg$ 用你所想像的那套方式來定義其它邏輯運算子。很可惜這邊並不適用，接下來我要來解釋為什麼。

在一般邏輯學的教科書上，你學到的那種邏輯被稱為**古典邏輯**，而我們在這邊考慮的是一套稍微不同的系統，被稱為**直構邏輯**。問題在於，在我們的詮釋下，邏輯系統和型別系統已經成為了一體兩面，而證明應該要能被當作程式調用：例如像是前面提到的 $\mathbf{I}$ 和 $\mathbf{K}$，除了作為命題的證明，還能被用於真正的程式中。可惜，古典邏輯裡面有些定理（套套句）完全無法被當作程式運行，例如像雙重否定律。

$$\neg\neg A \rightarrow A$$

$\neg\neg A$ 依定義相等於 $(A \rightarrow \bot) \rightarrow \bot$，雙重否定律是一個函數，要有適當的參數才能調用它，但事實是一個都沒有，所以直覺上是不可能從這個函數得到 A 的。換言之，不應該存在符合這個型別的程式。

因此這邊要定義一個稍微比古典邏輯弱的系統，你可能會覺得這樣有點可惜，沒辦法證明一些原本能證明的定理。好消息是，只要假定雙重否定律，直構邏輯就會變成古典邏輯，這意味著你只要在需要的時候假定雙重否定律就好了，只不過它們沒辦法被當作程式運行而已，而且直構邏輯還提供了比古典邏輯更強大的表達能力，就某種程度上，直構邏輯裡面的一切命題都是能被建構的。

我接下來要說邏輯中的**或**和**且**怎麼在 CoC 中被定義，首先解釋且好了：A 且 B 在邏輯上寫作 $A \wedge B$，但一般型別系統中寫作 $A \times B$。我們知道在 CoC 裡面幾乎只有函數這種東西而已，因此 $A \times B$ 也應該被定義成某個函數，首先，我們

知道如果 A 和 B 都能被證明，則 $A \times B$ 也應該能被證明，用符號可以寫成這樣：

$$A \to B \to A \times B$$

上面的式子應被讀為 $A \to B \to (A \times B)$，而非 $(A \to B \to A) \times B$。這邊要進入有點困難的部分了，來探討 A、B、$A \times B$ 和一任意 C 的關係，要探討另一命題的原因是因為在 CoC 裡面唯一能使用一個值的方式就是把他丟進一個函數，定義 $A \times B$ 時又勢必要用到 A 和 B，而定義 $A \times B$ 所用到的函數應當能回傳任何型別的值 (下寫作 C)，所以 $A \times B$ 可以被定義成：

$$(C : \star) \to A \to B \to C...$$

這樣的定義還沒辦法滿足 $A \to B \to A \times B$，因為一個隨意的 C 被導入了：假如把 A 和 B 丟進 $(C : \star) \to A \to B \to C$ 裡面，它會丟回一個 type 為 C 的 term。怎麼辦呢？因為 C 其實本質上是不重要的，不如我們就直接丟回 C 吧，把 $A \times B$ 的定義改成 $(C : \star) \to (A \to B \to C) \to C$，而這就是 $A \times B$ 的定義了：

$$A \times B \overset{\text{def}}{\equiv} (C : \star) \to (A \to B \to C) \to C$$

接下來講 $A \times B$ 在寫程式上有什麼意義，在程式的領域，$A \times B$ 被稱為一個**A 和 B 的二元組**，它同時記錄了 type 為 A 的 term 和 type 為 B 的 term 兩者的資訊。$A \times B$ 本身是個 type，要建構 $A \times B$ 的 term 的方式如下：

$$(a, b) : A \times B$$
$$(a, b) \overset{\text{def}}{\equiv} \lambda C : \star.\lambda f : A \to B \to C.fab$$

這樣的定義其實有點理所當然，它告訴我們可以把任何依賴 a 和 b 的函數丟進去以產生一個新的 C 的 term。接著來介紹如何從 (a, b) 中取回 a 和 b 兩者的資訊，稍微思考一下之後可以知道，只要把適當的 f 丟進去 (a, b) 的定義就好了，要取得 a 的話用 `fst`，要取得 b 的話用 `snd`。

$$\texttt{fst} : A \times B \to A$$
$$\texttt{fst} \overset{\text{def}}{\equiv} \lambda t : A \times B.tA(\lambda a : A.\lambda b : B.a)$$
$$\texttt{snd} : A \times B \to B$$
$$\texttt{snd} \overset{\text{def}}{\equiv} \lambda t : A \times B.tB(\lambda a : A.\lambda b : B.b)$$

其實有點像是循環論證啦，畢竟我們定義出二元組就是用來能讓 `fst` 和 `snd` 能被寫出來啊..... 一個雞生蛋蛋生雞的概念。

接著來介紹邏輯中的或，A 或 B 在邏輯上寫作 $A \vee B$，但通常型別系統中寫作 $A + B$，不囉唆直接給定義了：

$$A + B \overset{\text{def}}{\equiv} (C : \star) \to (A \to C) \to (B \to C) \to C$$

$A + B$ 的定義同樣會傳回一個任意的命題 C，但不同的是它不需要同時接收 A 和 B 兩個參數，而是接收 $A \to C$ 和 $B \to C$。為什麼呢？因為現在我們不要求 A 和 B 同時成立，而是其中一個成立就好了，假如是 A 成立好了，我們能把它丟給 $A \to C$ 得到 C；假如是 B 成立也類似。用這樣的寫法，我們只需要求兩者其中之一成立就好了。

現在來講 $A + B$ 在寫程式上頭的用途，它儲存了 A 或 B 其中一者的 term。在 $A \times B$ 中，要建構一個 term 只需要一個函數，要獲得它的內容則有 `fst` 和 `snd` 兩種方式，$A + B$ 則相反，有兩種方式能建構一個 type 為 $A + B$ 的 term，但要使用它只有一種方式。

要獲得 $A + B$ 的 term 的第一個方式是 `inl`，以 $A + B$ 來說，`inl` 代表的是它的內容事實上是左邊的 A；第二個方式則是 `inr`，代表的是它的內容事實上是右邊的 B。

$$\texttt{inl} : A \to A + B$$
$$\texttt{inl} \stackrel{def}{\equiv} \lambda a : A.\lambda C : \star.\lambda f : A \to C.\lambda g : B \to C.fa$$
$$\texttt{inr} : B \to A + B$$
$$\texttt{inr} \stackrel{def}{\equiv} \lambda a : A.\lambda C : \star.\lambda f : A \to C.\lambda g : B \to C.ga$$

至於要如何使用 $A + B$，$(C : \star) \to (A \to C) \to (B \to C) \to C$ 這個 type 說得很清楚了：我們要決定一個目標 type，就叫他 T 好了，然後提供一個 $A \to T$ 的函數和一個 $B \to T$ 的函數，把它丟給 $A + B$ 的 term 後就會拿回一個 T。

至於表示兩個命題相等的雙箭頭 $\leftrightarrow$，則可以被定義成 $A \leftrightarrow B \stackrel{def}{\equiv} (A \to B) \times (B \to A)$，命題邏輯的部分就到此告一段落了。

在一階邏輯中，還有另外一個符號用來表示存在某個值使得一述詞成立，相對於對於所有值一述詞成立；邏輯符號上寫作 $\exists x.P$。在型別論裡面，對於所有值一述詞成立的對應是 Π-型別，也就是回傳 type 裡面可使用輸入值的函數，是原本 $\to$ 的推廣；存在某個值使得一述詞成立的對應則是 $A \times B$ 的推廣，其中 B 可以依賴 type 為 A 的 term，這樣的型別被稱為 Σ-型別，符號上則寫作

$$\sum_{a:A} B$$

其中 a 被稱為 Σ 的綁定變數。和 Π-型別一樣，Σ-型別中綁定變數的型別也不一定要是 $\star$，而可以是一個一般的 type。舉例來說，如果你想表示對於某個自然數一述詞成立，你可以寫 $\sum_{n:Nat} B$。

要怎麼在 CoC 裡定義它呢？因為 Σ-型別是 $\times$ 的推廣，Σ-型別的定義方法和 $\times$ 的也類似，只是二元組裡面的第二個物體的 type 必須要能提到第一個物體的值，事實上我們只要把原本定義裡面的 A 改成可以被依賴的形式就好了：

$$\sum_{a:A} B \stackrel{def}{\equiv} (C : \star) \to ((a : A) \to B \to C) \to C$$

一旦 a 能被依賴，B 這個 type 中便可以使用到 a。如果要建構一個 Σ-型別的 term，我在此使用和普通 $\times$ 型別一樣的語法：

$$(a, b) : \sum_{a:A} B$$
$$(a, b) \stackrel{def}{\equiv} \lambda C : \star.\lambda f : (a : A) \to B \to C.fab$$

`fst` 和 `snd` 的定義則如下：

$$\texttt{fst} : (\sum_{a:A} B) \to A$$

$\mathtt{fst} \overset{def}{\equiv} \lambda t : \sum_{a:A} B.tA(\lambda a : A.\lambda b : B.a)$

$\mathtt{snd} : (t : \sum_{a:A} B) \to B[a := \mathtt{fst}\ t]$

$\mathtt{snd} \overset{def}{\equiv} \lambda t : \sum_{a:A} B.t(B[a := \mathtt{fst}\ t])(\lambda a : A.\lambda b : B.b)$

以程式的角度來探討，Σ-型別可以被想成是抹去 type 中一部份的資料。舉例來說好了，假設我們的系統裡有個叫 Array 的 type，Array 的型別是 $\star \to Nat \to \star$，意思是它接收一個型別和一個自然數，建構一個元素為那個型別且長度等同於那個自然數的陣列。假如我想要寫一個函數，把原本陣列裡重複的值刪除，你不能把它的型別寫作 $(T : \star) \to (n : Nat) \to Array\ T\ n \to Array\ T\ n$，這樣寫的話，你傳入和傳回陣列的 n 必須是同一個值，問題是一旦你刪除了陣列裡的某些元素，陣列的長度就會縮短！事實上應該被表達的是回傳的 Array 的長度是某個特定的值，但我們不確定那個值是什麼，換句話說，我想抹去 type 中長度的資料，所以我應該用 Σ-型別這麼寫：

$$(T : \star) \to (n : Nat) \to Array\ T\ n \to \sum_{m:Nat} Array\ T\ m$$