CPSC-354 Report

Pablo Labbate

Chapman University

December 22, 2021

Abstract

This paper presents a Programming Language Theory through multiple different sections, each building from previous others. It begins by introducing Haskell with the intention of easing the reader into the subject matter. From there, the paper delves into deeper aspects of Programming Language Theory, such as the introduction of Natural Deduction, an overview of Type Systems, and Curry-Howard Isomorphism, with the intention of analyzing the Curry style type checking. Finally, the last part focuses on extending lambda calculus, as we explore other systems.

Contents

1 Introduction

Programming Language Theory seems daunting at first; the subject matter is abstract and rather unfamiliar. Yet, through interest and perseverance, it can prove fruitful in multiple aspects. This paper presents a narrative to the theory, starting with a tutorial and increasing in thematic complexity through latter sections. The first section serves as an introduction to Haskell, a programming language that many might not be as familiar, or comfortable, with as they are with others, such as Python. As such, I introduce similarities and provide coding examples for both with the intention of providing grounds to the reader. From there, other examples are provided with the intention of further showing the benefits and capabilities of Haskell.

With a solid foundation now present, the next section delves deeper into Programming Language Theory. Natural Deduction is introduced and a recap on type systems and type checking is provided. With this information at hand, lambda calculus, both untyped and simply-typed, are presented. Consequently, the link between Natural Deduction and our simply-typed system is revealed and elaborated upon, leading to an analysis of Curry style type checking.

The final section is a philomathic extension of the subjects that were previously covered. Typed lambda calculus is extended with two systems provided as examples, and the beauty of the Curry-Howard Isomorphism is commented on. In essence, it serves both as an extension of the material covered, an invitation for further exploration, and a commentary on the implications and beauty of the isomorphism at hand.

2 Haskell

2.1 A Brief Introduction

The vast majority of readers to this point are familiar, if not proficient, with the concept of object-oriented programming. It is characterized by the representation of things as ”objects”, offering users the option to hide certain processes, pieces of code, from outsiders as well as to store those processes in their respective sections. Another key aspect of object-oriented programming is the manipulation of data through variables.

Functional programming, on the other hand, is based on the evaluation of functions. The manipulation of variables, as seen in object-oriented programming, is not possible—functional programming deals with immutable data. Loops are also not supported here. Instead, iterative data is managed by recursion.

This seemingly radical shift in paradigms sounds daunting at first. And, to an extent, it is. However, all good things are difficult at the beginning and, with some patience and perseverance, we can uncover a lot of interesting material.

2.2 Python, Haskell, and a Calculator

We will first be analyzing the differences between object-oriented and functional programming by building some functions for a simple calculator with a language from each paradigm—Python and Haskell, respectively. This will allow us to understand how functions, variables, and types behave for each language.

Let's begin with addition. For any two numbers, $a,$ b, $a + b = c$, where c, is also a number. How about we code it in Python. We define a function and its parameters, in this case a and b, and implement the operation inside the body. It looks like this:

```
def addition(a, b):
    c = a + b
    return c
```

As trivial as this may seem at first, let's look a bit more in depth at this function. It receives two variables, creates a new one, and returns it. More so, it doesn't specify the required types for addition. It doesn't matter if the input is an integer, double, floating point, or even strings for concatenation, our humble function just does it.

Now, let's do it with Haskell. The way data is treated with Haskell is rather interesting. Behind the scenes, we specify exactly what we wish to compute and with what type of input. In this case, we begin with natural numbers. The first step is to explain to our computer what is a natural number. It is comprised of all positive numbers as well as zero—many mathematicians dispute 0's membership in this number club with much fervour, however we have more important things to do, such as defining addition. Back to it.

Knowledge of discrete mathematics is important here, mainly because we must formally define our number system. Let us define a natural number as any number *NN*. This number is either 0, or a successor of it.

We define it within Haskell as seen below:

```
data NN = 0 | S 0
```

Now that we have defined the data type, let's move on to writing the function. Recall that Haskell is based on the evaluation of functions rather than variables inside functions. Our function adds two natural numbers: in other words, it takes in `NN`, adds it to another `NN`, and finally returns some `NN`, where `NN` could be any natural number. Elegantly put, `NN` $\rightarrow$ `NN` $\rightarrow$ `NN`.

The issue is that we have defined our natural numbers to be either 0 or its successors, therefore our addition will be dependent on this definition. As mentioned before, Haskell is based on recursion, so we will have to incorporate this as well. From Discrete Mathematics, we know that successors are commutative.

That is, adding $1 + 1$ is equivalent to adding $S(0 + 1)$. Furthermore, we know that the addition of zero and any number x is equal to x. Consider this our base case for our oncoming recursive implementation. Basing off from before, adding S(a), b, is equivalent to the sum of S(a + b).

Now that we have defined our data type, how our function treats the data type, as well as how it behaves, we can write it with Haskell. See below.

```
addN :: NN -> NN -> NN
addN O m = m
addN (S n) m = S (addN n m)
```

There we go, our first recursive function in Haskell. Now, let's up the ante a bit. Let's compare a slightly more complex function and analyze at the differences between the languages in which we implement it.

2.3 Fibonacci Sequence, Side Effects, and Benefits

We will first compare the implementation of the Fibonacci code for both languages. In Python, we could write it like this. [GFG]

```
toCalculate = 10

def fibonacci(n):
    a = 0
    b = 1

    # Check is n is less than 0
    if n < 0:
        print("Incorrect input")

    # Check is n is equal to 0
    elif n == 0:
        return 0

    # Check if n is equal to 1
    elif n == 1:
        return b
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b

print(fibonacci(toCalculate))
```

Something interesting is happening here. We are creating a global variable *toCalculate*, giving it an initial value of 10, and passing it to our function. Inside, we see how our local variables a, b, c, are being modified per each iteration. More interestingly, we have the option to reference variables and modify them at will.

Before moving on to Haskell, we must quickly look at pattern matching. Recall piece-wise functions from math. These functions behave differently based on the value of the given input. See the example below.

$$fib(x) = \begin{cases} 1, \text{ if } x = 0 \\ 1, \text{ if } x = 1 \\ fib(x-1) + fib(x-2), \text{ if } x >= 2 \end{cases}$$

See how the Python code is somewhat similar? It has different procedures for each specified case. Haskell's pattern matching works in a similar, slightly more elegant way in comparison to Python, and closer to our piece-wise function. We specify different cases for a function so that Haskell knows what to do in each case. When we call the function, Haskell compares the given arguments to the argument patterns we have written. As mentioned in *A Gentle Introduction to Haskell*, "Pattern matching provides a way to 'dispatch control' based on structural properties of a value."[GH] Now that we have a better understanding of pattern matching, we can translate our piece-wise function to Haskell.

```
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

Notice how Haskell's implementation is almost identical to our mathematical representation. However, a valid question would be, "why is this better than Python?" One of the main reasons is because it is more prone to be "mistake free". We can call our Haskell **deterministic**. It will always give the same result given a respective input.

More so, our Haskell code has no external state nor variables that are being mutated. In fact, our Fibonacci sequence is defined upon itself, function-wise rather than in terms of variables. This is crucial because functional programming languages, such as Haskell, don't allow for "side effects". As mentioned by Computerphile, an example of this is a mutable state, in essence a state that you can "mutate, you can update it, [...] so it can change from one value to another as the program progresses." [CPh] In synthesis, this method of implementation allows for clarity and security, as all our "function dependencies are stated explicitly" [MI]. It is important to note, however, that the Haskell code seen above is the "naive version" of implementation. There are multiple methods of implementing the Fibonacci sequence with linear time efficiency. However, those methods are rather advanced with respect to an introductory tutorial to Haskell. The main purpose of this section is to introduce Haskell and its powerful properties, as well as the similarities it shares to certain aspects of mathematics. Regardless, to see interesting examples of linear time implementations, see here.

2.4 Caesar Cipher

A cipher is defined as any method of transforming a message to hide its meaning. It has been used throughout history for efficient and secure communication among parties. One of the most famous and simple ones is the Caesar Cipher—named after Julius Caesar, who used it in his private messages. It is denominated as a substitution cipher, where each letter in the message is replaced by another letter, usually through a specified key. For example, if we wanted to encode "abc" with a key of 2, the result would be "cde". Although it serves no real security methods in today's times, it is a powerful example when it comes to considering the manipulation of strings regarding our comparison between languages.

To code it, we need to have a string to encipher and a key to know how much the shift will be. From there on, we loop through the string and change the individual character by the amount specified by the key. This is done by representing our string through modular arithmetic, where each letter is assigned a number—for example, A=0, B=1, ..., Z=26. Through this method, we can use the following equation for determining how each letter will be updated. [WK]

$$Encipher_n(x) = (x + n) mod 26$$

We use mod26 for cases where our key causes the encrypted letter to shift past 26. In essence, if our letter is Z and the key is 3, $(26 + 3) mod 26 = 3$, which changes Z to C. The code below shows a possible implementation in Python.

```
def Encipher(text,s):
    result = ""
    # loop through text
    for i in range(len(text)):
        char = text[i]
        char = char.lower()

        if char == ' ': # to keep spaces
            result += ' '
        else:
            result += chr((ord(char) + s-65) % 26 + 65).lower()

    return result
```

Notice how, again, our Python code is dependent on manipulation of variables. On the same note, take a look at how we don't have to specify individual character manipulation since the language allows us to do this all from the same function.

For Haskell, the situation is a bit more complicated. We must specify the behavior for each character as well as for the string as a whole. I suggest we begin by looking the shifting procedure for a single character, as seen below.

```
shiftedchar c n = chr((mod ((ord c - ord 'a') + n) 26) + ord 'a')
```

Notice how we are applying the same modular arithmetic equation specified previously. See below:

$$(ordc - ord'a') + n$$
$$x + n$$

All of this is then being modded by 26. Now, we can see the big picture and analyze how our code is being enciphered. [MWS]

```
lowercase m = words (map (toLower) m)

shiftedchar c n = chr((mod ((ord c - ord 'a') + n) 26) + ord 'a')
shiftedword w n = map (`shiftedchar` n) w

encipher m n = unwords(map (`shiftedword` n) (lowercase m))
```

Our enciphering program takes a message and a key, applies the shifted word function to each word as separated by spaces, which itself applies the shiftedchar function to actually shift the letters. Notice how our Haskell function is much more concise than our Python one. Similarly, it acts solely on the functions themselves in a deterministic and side-effect-free manner. No variables are used nor manipulated.

For deciphering, the code is rather similar. By the cyclical nature of the modulo operator on our shifting procedure, we can denote our deciphering equation as follows:

$$Decipher_n(x) = (x - n)mod26$$

The following code is adapted from the enciphering code, changing the necessary manipulations such that an enciphered string can be read in its normal form—given we know the key, of course.

```
lowercase m = words (map (toLower) m)

normalchar c n = chr((mod ((ord c - ord 'a') - n) 26) + ord 'a')
normalword w n = map(`normalchar` n) w
```

```
decipher m n = unwords(map (‘normalword‘ n ) (lowercase m))
```

Finally, the entire program would look as follows, including the necessary imports to run properly. To run it, run the following commands.

1. **ghci *the name you give this file***
2. **encipher *message* *key*** or **decipher *encrypted message* *key***

```
import Data.Char

main :: IO ()
main = return ()

lowercase m = words (map (toLower) m)

shiftedchar c n = chr((mod ((ord c - ord ’a’) + n) 26) + ord ’a’)
shiftedword w n = map (‘shiftedchar‘ n) w

normalchar c n = chr((mod ((ord c - ord ’a’) - n) 26) + ord ’a’)
normalword w n = map(‘normalchar‘ n) w


encipher m n = unwords(map (‘shiftedword‘ n) (lowercase m))
decipher m n = unwords(map (‘normalword‘ n ) (lowercase m))
```

For further practice, I recommend incorporating further shifting possibilities to the code above. For example, our code right now only works with alphabetical characters. However, we could extend it such that it can encipher punctuation marks and numbers.

2.5 Haskell, irl

If you are like me from a few weeks ago, you might still be leaning towards Python due to its close relationship semantically to English, facility to write programs, and an understandable comfortable relationship with OOP/imperative programming. However, as programmers, efficiency is one of the most important things we strive for. Haskell, by nature a compiled language—in essence converted directly into machine code for the computer to execute—, is, and will always be, faster than Python—or any other interpreted language.

”Alright, then,” you might say, ”I still see no use of it in real life.” In that case, let’s bring up Facebook and its spam filtering. As of 2021, Facebook boasts over 2.89 billion active users. [FB] For such a big platform, the monitoring of spam is something very difficult to achieve. In comparison, consider the 1.5 billion users Gmail has and the constant spam you can expect to receive [GM]. One of the ways in which Facebook is so successful at monitoring and filtering spam, malware, and other types of abuse is through a system called Sigma. It automatically detects such content and deletes it such that it doesn’t appear on one’s feed.

Sigma serves as a *rule engine*. Every interaction on Facebook is evaluated by sigma against a set of rules that are coded within it. These rules are classified within different groups, which allows for Facebook to determine the types of interactions that occur. The source code for Facebook is running at all times through Sigma. Let’s ponder about this for a second. The source code for a platform consisting of 2.89 billion active users is constantly being run through code for monitoring types of interactions. Clearly, this type of program must be able to run efficiently.

How does Haskell fit in? Recall that Haskell doesn’t allow for side effects and requires strongly typed implementation. This ensures that rules don’t interact with nor modify each other, preventing Sigma from crashing. More so, testing is easier since rules don't interact with each other. Finally, strong types help eliminate bugs from entering Sigma. Other reasons include performance as compared to the language with which Facebook would previously fight spam with: FXL. As mentioned by Facebook, "FXL's slower performance [required] writing anything performance-critical in C++ and putting it in Sigma itself. This had a number of drawbacks, particularly the time required to roll out changes." [SIG]

Regarding performance itself, efficiency is crucial for such a big platform. Facebook states that "Requests to Sigma result from users performing actions on Facebook, such as sending a message on Messenger, and Sigma must respond before the action can take place." For the 25 most common types of requests received by Sigma, Haskell outperforms FXL in all but two cases, often three times faster and for an average of 20-30 percent increased efficiency. [SIG]

I truly recommend reading more on this topic, as it is one of the single most pertaining examples of Haskell's efficiency and industry-wide use. Other companies that also use Haskell include AT&T, Chordify, Deutsche Bank, Eaton, NVIDIA, and many others. [HA]

3 Programming Language Theory

Broadly put, all programming languages are examples and derivations of formal systems: we infer true statements from axioms according to a set of rules. In this section I will introduce Formal Systems. We will look at Natural Deduction and use it to jump towards the Curry-Howard Isomorphism, type checking and similar topics. This section intends to provide a stronger understanding of programming language theory, in essence the spectrum of the subject matter as well as conceptual differences and their significance.

3.1 Introducing Natural Deduction

Natural Deduction is a formal system where inference rules are expressed via logical reasoning. This section assumes basic understanding of logic, truth tables, and proofs.

$$P \vdash Q$$

This is referred to as a conditional assertion—we read it as ”If P, then Q.” In the same manner, we can say the following for some T.

$$\vdash T$$

As a refresher, we read this as T being true without any conditions, persay; similarly, it can be referred to as T being trivial. Note, this is not meant to be a formal and textbook definition of formal systems and their intricacies slash dependencies. I am approaching this through colloquial terms in order to provide solid footing before we approach murkier water. With the basic idea of conditional assertions, we can extend the example to something more interesting. Say we want to prove the following:

$$p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$$

How do we begin? Our conclusion is to the right of our conditional assertion symbol, also referred to as a *turnstile*. My recommendation is to start from the end and work our way back. Let's see what we can derive from our conclusion:

$$\neg\neg p \wedge r$$

Let's keep building upwards. From truth tables, we know that for the statement $p \wedge r$ to be true, both p and r must be true independently. As such, we can extend our tree to the following level.

$$\frac{\neg\neg p \qquad r}{\neg\neg p \wedge r}$$

To make sure we can follow our reasoning steps, we can place our deductions on ”levels”. Formally, this is known as tree proofs, introduced by Gerhard Gentzen in 1934. [GG] For most people, the use of double ”not” logically cancels out in their head—for example, I do not *not* like Programming Language Theory[1]. The same rule applies in our formal system, double negation is the same as an assertion with no negation at all.

$$\neg\neg p == p$$

We can simplify our conclusion from $\neg\neg p \wedge r$ to just $\neg\neg p$ and r independently.[2]

[1]It's an interesting subject, I like it.

[2]Yes, I used unorthodox notation, borrowing ”==” from most programming languages to refer to equality and mixing it with our Natural Deduction syntax.

$$\dfrac{\dfrac{p}{\neg\neg p} \qquad r}{\neg\neg p \wedge r}$$

We are getting close. From our original problem, $p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$, we need to introduce $q \wedge r$ and its double negation. For this to be true, we need to allude to the fact that the left-hand side of our turnstile consists of our assumptions, which we assume to be true. As such, we can say that $q \wedge r$ is true. From truth tables, if our previous statement is true, then q and r, independently, are true. We can present this deduction as another level on our tree.

$$\dfrac{\dfrac{p}{\neg\neg p} \qquad \dfrac{q \wedge r}{r}}{\neg\neg p \wedge r}$$

Given our most recent addition to the tree, it is valid to ask why r is the only deduction from $q \wedge r$ and no q whatsoever. The reason is because it is irrelevant to our proof. Our tree aims to extend the assumption logically and elaborate it, often times using logical manipulation rules to bend it into an equally true form that is closer to the desired conclusion we are trying to prove. With respect to the proof tree, we add our double negation to the respective relation of $q \wedge r$ and we are done!

$$\dfrac{\dfrac{p}{\neg\neg p} \qquad \dfrac{\dfrac{\neg\neg(q \wedge r)}{q \wedge r}}{r}}{\neg\neg p \wedge r}$$

The final version of our tree can be interpreted as follows. The top-most level of any branch is our initial assumption, namely the one given on the left hand side of the turnstile—here, it's $p, \neg\neg(q \wedge r)$. The comma that separates them means they are independent of each other, hence each receiving a separate branch. From thereon, the traversal downwards from level to level follows logical rules, such as elimination of double negation, to manipulate the problem into a form that is closer to what we are trying to prove. It is important to clarify that for academic proof writing, an explanation is included per level of the tree, however I opted to replace that by having a guided narration of the process.

The idea behind this section was to introduce Natural Deduction and get you familiar with it. The exercises below are helpful to strengthen your foundation and understanding of this formal system, but also serve as a fun challenge, equivalent to figuring out a crossword or a Sudoku puzzle.

1. $P, Q \vdash P \wedge Q$
2. $P \wedge (Q \wedge R) \vdash (R \wedge P) \wedge Q$
3. $P \wedge (Q \vee R) \vdash (P \wedge Q) \vee (P \wedge R)$

These three problems above come from Alastair Carr's *The Natural Deduction Pack* [NDP]. I highly recommend going through this resource as it introduces more rules within Natural Deduction and gives countless examples of all the relevant sections. More so, almost all problems have solutions provided with detailed explanations. I highly recommend this pack to almost any student; getting good at Natural Deduction strengthens one's ability to solve complex problems by providing tools for decomposing and rearranging said problems into manageable chunks that lead to solutions. Be it a tool for mathematical proof writing or for computer programming, knowing how to manipulate a problem by using tools at our disposition and tackling it by parts is pivotal. With this being said and Natural Deduction having been introduced, let's move on.

3.2 Type Systems and Type Checking: A Quick Recap

A type system is a logical system consisting of a set of rules which assign types to a computer or formal system's elements, such as variables and functions. The function of these types is to formalize the many present categories that are used within the system in order to differentiate their uses and nature. In most programming languages, for example, you would want to have a distinction between a variable, a function, and a data structure, among other things. More so, you would want to have a way of distinguishing between specific data types such as integers, floats, strings, booleans and so on.

The purpose of a type system serves as a bug reducer, per say. It discovers bugs at "compile-time as opposed to leaving the users discover bugs at run-time. Moreover, a powerful type system enables the compiler to return meaningful error messages to the programmer." [AK] Every value that is generated by the respective program has an associated type. The way type checking works is not universal among all programming languages or formal systems, however. Type checking varies with regard to four main language-typing categories: strongly-typed, weakly-typed, dynamically-typed, and statically-typed languages—although it can be argued that these types mix between themselves, as the following examples might persuade.

Strongly-typed languages have predefined data and all defined constants or variables must be described as part of one of the given data types. This ensures that some operations are reserved strictly for certain data types. An advantage to this is consistency through imposed rigor, minimizing the possibility of errors. [ST] An example is provided below using C++.

```
int age = 0;
string name = "Billy Bob Randall Johnson Owens III"; // a long name, but a valid string

result = age + name; // will cause an error
```

Weakly-typed languages, on the other hand, can make conversions between types implicitly, such as from a string to an integer—they don't require explicit conversion, in essence type casting. [WT] The example below is written in C. [TC]

```
// Code from Geeks for Geeks

int x = 10;   // integer x
char y = 'a'; // character c

// y implicitly converted to int. ASCII
// value of 'a' is 97
x = x + y;

// x is implicitly converted to float
float z = x + 1.0;

printf("x = %d, z = %f", x, z);
return 0;
```

Statically-typed languages are categorized by recognizing variable types at compile-time as opposed to run-time. This means that once a type has been assigned to a variable, it can't be cast to another type or an error will be raised. Although arguably restrictive, it offers the possibility of catching errors in early stages of development. Similarly, it allows for quicker execution of code since the compiler knows what the data types are prior to run-time. [STWT] You can see an example via C++ below.

```
string myVariable;
myVariable = "Pablo";
myVariable = 10; // will cause compilation error
```

Finally, a dynamically-typed language type checks during run-time. A very common example is Python, where you don't have to declare a variable type while writing code and, instead, the compiler determines it for you. For example:

```
myVariable = "Pablo"
myVariable = 10 // no error
```

In dynamically-typed languages, ”variables are bound to objects at run-time by means of assignment statements, and it is possible to bind the same variables to objects of different types during the execution of the program.” [STWT] Now that we have a better idea of type systems in popular languages, we can look at something different.

3.3 On Lambda Calculus

The previous section aimed to establish a proper distinction between the different type systems. We now will look briefly at λ-calculus and topics related to what we have previously covered. With this, my intention is to bring light to some possible issues that arise with the 'freedom' that (untyped) λ-calculus provides, and offer solutions—and their possible drawbacks—via different type systems.

λ-calculus can be considered one of the simplest or smallest programming languages in the world. The core concept behind it is the evaluation of ”expressions”. We can define expressions recursively: [RR]

```
// author: Raul Rojas
// source in bibliography
<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>
```

Say we have expressions $E_1, E_2, E_3, ..., E_N$. Evaluation of expressions, as we know, follows this order:

$$(...((E_1, E_2)E_3)...E_N)$$

The main challenge regarding λ-calculus comes from the fact that ”every lambda expression can be applied to any other lambda expression. In particular, this allows self-application”. [AK] For example, if e is a lambda expression, so is ee, eee, and so on. There are multiple other examples of self application, such as the following:

$$(\lambda x.x)(\lambda x.x)$$

This reduces to the identity function, its normal form—which we will delve into further ahead. Another example is a bit more interesting:

$$(\lambda x.xx)(\lambda x.xx)$$

This infinite computation proceeds as follows, with subscripts used to differentiate substitutions:

$$(\lambda x.xx)(\lambda x_0.x_0x_0)$$
$$(\lambda x_0.x_0x_0)(\lambda x_0.x_0x_0)$$
$$(\lambda x_0.x_0x_0)(\lambda x_1.x_1x_1)$$

This is infinite self application. We see how in one step, it reduces itself to a copy, which then reduces itself to a copy of that copy, and so on. It is a non-terminating computation. A solution to this would be eliminating self-application. What happens then?

3.4 Simply Typed Lambda Calculus

Adding types to λ-calculus can help us put an end to the paradoxical nature that is presented through the lack of distinction between functions and arguments. Simply-typed λ-calculus, originally presented by Alonzo Church in 1940, is rather similar to its untyped counterpart. The main difference is that simply-typed explicitly states the type of argument. An variable of a certain type is denoted as follows:

$$x : \tau$$

An interesting distinction, which very well illustrates the slight difference between untyped versus simply-typed, comes from the BNF syntax definition of both calculi [WP].

$$e :: x|\lambda x.e|ee$$
$$e :: x|\lambda x : \tau.e|ee|c$$

The first line listed above refers to untyped lambda calculus, whereas the bottom one is that of the simply-typed one. We can see that the equations are almost identical, with the sole difference being the presence of the type specifier that we previously introduced and c, for a constant. It follows the following order, from left to right: variable reference, type specification, application, and a constant [WP].

Expression evaluation is left unchanged despite the presence of type specification. The only way an expression e will get stuck is if it is not a value and there is no expression $e \to e'$ [CS]. The presence of types means that our infinite evaluation of $(\lambda x.xx)(\lambda x.xx)$ will be untypeable, thus removing this issue altogether. An issue, however, is that we end up sacrificing expressiveness for type rigor—a matter of personal taste, I would add.

At this moment I would recommend drawing, not literally, a parallel to the broader spectrum of programming languages. What does the presence of type systems tell us about it as a whole? On one hand, we have untyped lambda calculus, which allows unmatched expressiveness; everything can be run, yet no meaningful errors will be caught by a compiler, if implemented. The other side of the spectrum presents us with typed lambda calculus, among which simply-typed is a member of. All programs here are guaranteed to terminate, yet at the cost of recursion and looping [AK]. Programming languages are somewhat in between. They provide programmers with some freedom, with specific degrees of it varying from language to language, while still offering the ability to catch meaningful errors.

We have now seen Natural Deduction, introduced the idea of type systems, and analyzed two types of lambda calculus. With this knowledge at hand, we will proceed to look at how Natural Deduction comes into play with respect to type checking. But, before that, we need two more quick detours.

3.5 Soundness, Consistency, and Normal Forms

My goal here is to provide necessary background before I explain why Natural Deduction can be used to do type checking on our typed lambda calculus. Consider the following important characterization of Natural Deduction, namely its introduction and elimination rules, which tell us how to infer and use conjunction, disjunction, implication, and so on. The key concept is that each of these connectives should, in a way, act *atomically*; that is, characterized solely by their respective rules with no allusion to other ones. Specifically, introduction and elimination for connectives can only be chosen as long as they follow rules which form coherent systems. This leads to the presence of an important concept, that of *local soundness*. It states that given logical connectives in a formal system, we should not be able to gain information through the introduction of a connective while immediately eliminating it. Said action should be equivalently attainable via other operations without said detour [SD].

This can be demonstrated by local reduction, a local notion of consistency which serves as a pure semantic check on the respective system's inference rules. Mathematically, and specifically within formal systems, a "theory is said to be consistent if falsehood is not provable". Moreover, local reducibility states that "any derivation containing an introduction of a connective followed immediately by its elimination can be turned into an equivalent derivation without its detour". The purpose of this is to show that elimination rules are not too strong, such that they do not "include knowledge not already contained in their premises." The example below illustrates the aforementioned notion [CO].

$$\cfrac{\cfrac{\cfrac{}{\text{A true}} \qquad \cfrac{}{\text{B true}}}{\text{A} \wedge \text{B true}}}{\text{A true}}$$

This reads as follows: we introduce and assume A to be true—from the top left level—and the same for B. From this assumption we can say that A and B are true. From truth tables, we can deduce that A is true. Notice that this derivation is equivalent to the following.

$$\cfrac{}{\text{A true}}$$

The elimination rules must not be that strong such that they include information that is not already within their scope. Overall, main idea behind a type system is that it must be *consistent.* Perhaps of interest and as an aid for understanding, although not a requirement for the overall comprehension of the topic, I want to draw parallel to ω-consistency. An ω-consistent theory avoids the possibility of computations resulting in infinite combinations that are contradictory—in a way, we want our type system to do such a thing. This definition was introduced by Kurt Gödel in his process of proving his famous incompleteness theorem. Although a fascinating subject that deserves an in-depth approach, its content can span a semester's worth of lectures, and so, for now, I will leave simply the previously presented definition as another helpful tool in the analysis of formal system requirements.

If the derivation of a tree follows an order of elimination with subsequent introductions, then it can be considered *normal* [CO]. Most formal systems act this way, such that any derivation has its normal counterpart, which we can refer to as its *normal form.* These allow for an "attractive abstract level of information structure beyond brute details of syntax" [SD]. More abstractly, an object in a system is in a normal form if it can no longer be rewritten, therefore reaching an irreducible state.

To illustrate this overarching notion in somewhat simpler terms—as I can admit that the past couple of paragraphs might have been a lot of information packed in a punch—consider these simple rewriting rules:

$$aa \to a$$
$$bb \to a$$
$$ab \to b$$
$$ba \to b$$

Given our previous definition of normal forms, we can deduce that the only two irreducible states are a and b. For example, say we have some string of a's, call it a^n, it would follow this sequence of reduction:

$$a^n \to ... \to aaaa \to aaa \to aa \to a$$

Interestingly enough, the following computation is also equally valid:

$$a \leftarrow bb \leftarrow bbb \to ba \to b$$

You might have noticed that at *bbb* has two arrows stemming from it, reduction towards normal form *a* and another towards *b*. More so, regarding the rightwards computation stemming from *bbb*, rather than simplifying to *ba*, we could have done the opposite, namely *ab*. The rules allow for this, and both derivations lead to *b*. In my opinion, this is the clearest example of normal forms.

The previous examples regarding untyped lambda calculus' infinite computations are key samples as to the system's lack of normalizing properties. The lack of types allows for infinite evaluations through the fact that anything can be considered an expression. Simply-typed lambda calculus, on the other hand, offers normalization via its type checking. It is because of this normalization property that it can be considered a programming language; namely because it terminates [NF]. Because of Natural Deduction's consistency [3] [4]—through which reduction to normal forms follows—, we can use it as the backbone for λ-calculus' type checking system. The reason stemming from an underlying isomorphism between logic and type systems.

3.6 Curry-Howard Correspondence With Respect to Natural Deduction

Readers that have taken some logic or abstract math class in conjunction with their computer science curriculum might have noticed a certain level of similarity between these subjects. Discrete math, for example, introduces the idea of lists and the operations that can be done with or to them. It defines a list by its head and tail, $(x : K)$ for some list K. Well, as we have previously seen, this is rather identical to Haskell's methods of operating on lists. This example is of countless.

From the early 20th century onwards, mathematics, logic, and computer science—rather new back then—began to blend together in terms of foundations. Gödel's Completeness theorem was followed by λ-calculus, the Incompleteness theorem paralleled by Turing Machines, and, almost at the same time, the Undefinability theorem emerged alongside the Halting problem [PMP]. There was an emerging sense of a bijective relationship between these topics. A zigzag of sorts. Near the early mid 20th century, Haskell Curry and William Alvin Howard discovered[5] the direct relationship mathematical proofs and computer programs. A main concept regarding the discovery being that proofs compute! To further illustrate this bijection[6], refer to the table below. Note that one column is called *Logic*, loosely referring to the umbrella which formal systems and mathematical proofs, for example, fall under. Type theory and programming languages are very much similar, but one serves to illustrate a more formal, generalized concept from which the other one, programming languages, borrows.

[3]I want to stress that this section is not, by any means, meant to serve as a proof of Natural Deduction's consistency. For further reading, however, I recommend *Direct consistency proof of Gentzen's system of natural deduction* by Andres R. Raggio, which indirectly inspired many interesting points of conversation above.

[4]Although I haven't been able to find other proofs regarding Natural Deduction's consistency, Raggio's is the most comprehensive and complete example I was able to come across. I recommend giving it a try, as it will solidify the understanding of what we have covered, as well as what is to come.

[5]Independently, Curry in 1958, Howard in 1969.

[6]Referring to the one-to-one relationship between formal systems and computer science.

Logic	**Type Theory**	**Programming Languages**
Proposition: a claim something is true. • There is an equivalence between proposition and contract.	*Contract*: picks out set of values that pass said contract. • In essence a claim that there are values that satisfy a property • If input provided passes a contract, it is proof such input exists	*Types*: • `char x = 'a';` • strings, chars, ints, doubles, so on.
Proof: • From an assumption, we can deduce via constructive means that another proposition holds.	*Guarded Function*: • If a given input passes the input contract, the function outputs something that passes its output contract. • Inputs to a function are parallel to a proof's premises and the return parallel the proof's statement.	*Program*: • Detailed, unambiguous procedure for solving a problem with a computer. • We can ask a program to run certain code and pass in different values to see responses.
Implication ($\implies$): "If P, then Q." • $F \implies F$ • $F \implies T$ • $T \implies T$ • $T \not\Longrightarrow F$	*Right Arrow* ($\rightarrow$): A guarded function between contracts P and Q. • $\{\} \rightarrow \{\}$ • $\{\} \rightarrow \{\cdot\}$ • $\{\cdot\} \rightarrow \{\cdot\}$ (*identity*) • No function from $\{\cdot\} \rightarrow \{\}$	*Function From A to B*: Behaves similarly to Type Theory's right arrow. Implementation would be via If statements. Falsity would fall under the "else" case, for example. The impossibility of True implying False would be equivalent to entering a portion of an If statement through an incorrect conditional.

The similarities continue, this table merely serves to illustrate some of the direct relationship between these fields. The following table is a continuation of this, but strictly within Logic and Computer Science [PMP].

Logic	**Computer Science**
More on Logic:	*Its CS equivalent*:
Axiom	System primitive
Soundness theorem	Compiler
Completeness theorem	Debugger
Incompleteness theorem	Infinite loops
$A \wedge B$	Pair of A and B
$A \vee B$	Tagged union of A and B
Falsity	Void type, 0
Truth	Singleton type
Axiom	Variable
Introduction rule	Constructor
Elimination rule	Destructor

"Cool, cool, cool... and?" One might ask. *And?!* This isomorphism is the bridge that allows logic, mathematics, and computer science to be interconnected. The Curry-Howard Isomorphism is the single reason why we can use Natural Deduction and program a type checker for our typed λ-calculus. Recall our Natural Deduction derivation.

$$\dfrac{\dfrac{\dfrac{}{\text{A true}} \qquad \dfrac{}{\text{B true}}}{\text{A} \wedge \text{B true}}}{\text{A true}}$$

Which was the same as:

$$\dfrac{}{\text{A true}}$$

This is because of a direct correspondence to β-reduction via the Curry-Howard Isomorphism. It has a direct relation to Natural Deduction's concept of *local reducibility*, which we covered in a previous section. Since this topic is deeply covered in class, I will simply paraphrase it in the following sentences. β-reduction states that an application of the form $(\lambda x.\tau)$ reduces to $\tau[x := s]$. Recall our $(\lambda x.xx)(\lambda x.xx)$ evaluation. Following β-reduction, the following occurred[7]: [WP]

$$(\lambda x.xx)(\lambda x.xx) \to (xx)[x := \lambda x.xx] = (x[x := \lambda x.xx])(x[x := \lambda x.xx]) = (\lambda x.xx)(\lambda x.xx)$$

The term would reduce itself to a single β-reduction, continuously evaluation itself in an endless reduction process. This is conceptually identical to the computations we carried out in the respective section which

[7] Within untyped lambda calculus

showed us the same endless evaluation. However, when we begin to introduce types—that is, we work with a typed version of lambda calculus—, these issues, as we have seen above, go away. As explained by John D. Cook: [JDC]

> In simply typed lambda calculus, we assign types to every variable, and functions have to take the right type of argument. This additional structure prevents examples such as those above that fail to normalize. If x is a function that takes an argument of type A and returns an argument of type B then you can't apply x to itself. This is because x takes something of type A, not something of type function from A to B. You can prove that not only does this rule out specific examples like those above, it rules out all possible examples that would prevent β-reduction from terminating.
>
> [In essence], β-reduction is not normalizing, not even weakly, in the context of untyped lambda calculus, but it is strongly normalizing in the context of simply typed lambda calculus.

We can finally now look at how simply-typed λ-calculus' type checker works.

3.7 Curry Style and Type Checking

The introduction of types and strong normalization create a stricter environment, providing rigidity and eliminating cases of endless self-evaluation. We say goodbye to $(\lambda x.xx)$. Our simply-typed λ-calculus consists of types and terms—which have only one unique type. For example, if we have $x : \tau$, we mean to say that x is a term of type τ.

The following are all types in simply-typed λ-calculus [BA].

- **Atomic types**: the type 0.
- **Product types**: if A and B are types, their product $A \times B$ is also a type.
- **Coproduct types**: if A and B are types, their product $A + B$ is also a type.
- **Function types**: if A and B are types, the function $A \to B$ is also a type.

The composition of types is also a type. For example:

- $((A \times B) + C) \to D$ is a type.
- $0 \to ((A + B) \to C)$ is a type.

Term construction falls under two categories:

- **Introduction construction**: allows to construct terms a type specified above.
- **Elimination construction**: allows the use of terms of a certain type to construct terms of another type.

It also is comprised of a **computation rule**: it explains the behavior that occurs if introduction is followed by elimination construction[8] [BA2].

Consider the Curry-Howard Isomorphism between logic and our simply-typed λ-calculus. Types are analogous to propositions and terms are analogous to proofs. More so, there is a direct relation between propositions and types, see table below [BA3].

[8]Note, an in-depth explanation behind term construction is not included here. The reasoning behind this comes in two parts. The first part being that a lot of this is covered as class material. The second reason comes from the fact that the explanation behind term construction is not directly relevant to this section. Knowing what terms and types are is enough, but for further information I recommend checking out Benedikt Ahren's video, "Simply-Typed Lambda Calculus, part 2: Terms".

Propositions	**Types**
$\perp$	0
$A \vee B$	$A + B$
$A \to B$	$A \to B$
$A \wedge B$	$A \times B$

Free and bound variables are still a thing in simply-typed λ-calculus. To deal with them, we apply typing rules with respect to a context of the free variables that appear in a given term. We can refer to the context that contains our variable declaration of the form $x : \tau$ as $\Gamma, x : \tau$, where Γ is the set of said declarations [AK]. Let's analyze this a bit more. Terms, or expressions, are defined as follows:

$$e :: x|(\lambda x : \tau.e)|(e\ e)|c$$

Recall the similarities regarding the definitions for expressions between untyped and simply-typed λ-calculus. The main difference here is that the argument variable has to have some type annotated to it. Types are defined in the following manner:

$$\tau, s ::= Z|\tau \to \tau$$

Z itself is not part of the formal definition, but I placed it there to denote a variable for types such as ints, chars, strings, so on. The other possible type is a function type, where the first τ represents the input type, and the latter τ the output. Finally, our context, Γ, is as follows:

$$\Gamma ::= \{x_1 :: \tau_1, ..., x_n :: \tau_n\}$$

Our context is thus nothing more than a mapping from a term to its respective type. We use typing rules to ensure our simply-typed λ-calculus follow our desired behavior. The "typing judgement $\Gamma \vdash e :: \tau$ defines a valid relation among the type [context] Γ, the [term] e, and the type τ[9]." [KM] We can define these rules via deduction systems—again, notice the similarity between our Natural Deduction section and our simply-typed λ-calculus.

See the next page for typing rules and their explanations.

[9]Notice how our definitions are defined via logical relations.

Rule	**In Vernacular**
(1) $\dfrac{(x :: \tau) \in \Gamma}{\Gamma \vdash x :: \tau}$	Term x is of type τ under our context Γ if and only if a pattern can be matched from x to τ in the given context Γ
(2) $\dfrac{\Gamma \cup \{x :: \tau\} \vdash e :: s}{\Gamma \vdash (\lambda(x :: \tau) \to e) :: \tau \to s}$	*Abstraction:* the λ abstractions for a term e of type s, in a context with term x of type τ, then in the same context but without x, $(\lambda x :: \tau) \to e$ has the type $\tau \to s$. [WP]
(3) $\dfrac{c \text{ is a constant of type T}}{\Gamma \vdash c :: T}$	*Trivial case.* A term c is under our context Γ if and only if it has an appropriate base type.
(4) $\dfrac{\Gamma \vdash e_1 :: \tau \to s \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash (e_1 e_2) :: s}$	*Function application*: e_1 applied to e_2 has a type s in our context Γ if and only if we can prove: (1). Under the same Γ, e_1 has a type τ that goes to s and (2). Again under the same Γ, e_2 is of type τ.
(5) $\dfrac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash (e_1 + e_2) :: \tau}$	The sum of terms e_1 and e_2 are of type τ under our context Γ if and only if e_1 and e_2, independently, are of type τ under the same context.

With respect to our typing rules, there are two main facets to typing: static and dynamic. Static typing does type checking and type inference "before the execution of the program. This allows the compiler to statically find programming mistakes at compile time and prevents run-time errors (which are much more costly as they may only appear after code has been shipped)" [AK]. On the other hand, dynamic typing only occurs during run-time, meaning that mistakes are to be found during run-time as well, and those errors can't be prevented beforehand.

As a nice exercise before we wrap the section up, I suggest we put together our knowledge of Natural Deduction and type checking to analyze that an expression, in this case $(\lambda x :: \mathbf{int}.x + 40)2$, is well-typed[10]. Our goal is to construct a derivation of the form $\vdash e :: \tau$ to type check our term e of type τ. Note that the reasoning on each level of the tree refers to the rule in the table above—e.g: a reasoning of **(3)** on a level refers to the third rule in our table, in this case the trivial one.

$$
\textbf{(4)}\ \dfrac{\textbf{(2)}\ \dfrac{\textbf{(5)}\ \dfrac{\textbf{(1)}\ \dfrac{}{x :: \mathbf{int} \vdash x :: \mathbf{int}} \qquad \textbf{(3)}\ \dfrac{}{x :: \mathbf{int} \vdash 40 :: \mathbf{int}}}{x :: \mathbf{int} \vdash x + 40 :: \mathbf{int}}}{\vdash \lambda x :: \mathbf{int}.x + 40 :: \mathbf{int} \to \mathbf{int}} \qquad \textbf{(3)}\ \dfrac{}{\vdash 2 :: \mathbf{int}}}{\vdash (\lambda x :: \mathbf{int}.x + 40)2 :: \mathbf{int}}
$$

[HPL] We can read the deduction tree top-down. The top-left branch maps our term x to a type **int** in the current context—in essence, since we state in our expression that $x :: \mathbf{int}$, then our typing assumption

[10]This exercise comes from Harvard's CS 152: Programming Languages. I found it to be both a challenging yet approachable exercise, as the problem itself is not that complicated, but the sudden jump from theory to exercise might be daunting.

is the same as our typing judgement. Likewise, the top-right branch states that if we assume x to be of type **int**, then we can judge 40 in the same manner. The merging of these two branches comes via the use of **(5)**, the sum of terms: if our typing assumption is that $x :: \textbf{int}$, then $x + 40$, based on the information deduced from levels above, is also of type **int**. The deduction level below refers to **(2)**, abstraction—from type **int** to **int**. On the same level yet deduced on the rightmost branch, we see another trivial case where our typing judgement recognizes 2 as **int**. Because we were able to prove both conditions for **(4)** via the two last deductions, we can conclude that the application of these terms is of type **int**.

With this in hand, we have successfully gone over Natural Deduction, the requirements needed from it to be used as a type system, and the correspondence between logic and computer science, all with the purpose of leading towards an analysis of simply-typed λ-calculus' type-checking system. With this section now fully wrapped up, and with a short way left ahead, I invite you now to enjoy some interesting finds and important consequences regarding overarching subjects in this paper.

4 Project

I want to treat this section somewhat differently. Until now, we have followed a path that traversed some of the more important, yet mostly surface-level parts of programming language theory. This is not to say that the subjects covered were trivial; the introduction to Haskell served both as a micro-tutorial for the language as well as a bridge towards the more abstract subject of programming language theory; said theory section introduced types, type checking via deduction systems, and the general idea—alongside some intricacies—of λ-calculus, both typed and untyped. An overarching theme was seen throughout these sections, namely the bridge, or correspondence, via logic, mathematics, and computer science—you guessed it, the Curry-Howard Isomorphism.

Yet we have not even scratched the surface of what is the iceberg of programming language theory. As I was researching material—to both enrich the narrative I was presenting as well as for pure self-interest—I kept finding myself in situations where I would get engrossed in a certain subject and fall through its little rabbit hole. I would end up trying to fit it into my paper but, despite its fascinating subject matter, it simply wasn't right for that specific section.

Regardless, because of my insatiable interest[11] in sharing these finds, I want to treat this section as a momentary exploration of some of the things that lie underneath the tip of our iceberg. With this in mind, let's jump to what comes *after* simply-typed λ-calculus.

4.1 HM System

Recall simply-typed λ-calculus' type system, namely its sole constructor[12], $\rightarrow$, and its primitive assortment of types. The Hindley-Milner Type System, abbreviated as the HM System, extends the capabilities of typed λ-calculus by introducing parametric polymorphism—that is, it allows for the power of polymorphism while still offering the safety mechanisms of a statically-typed language. Typing rules are almost identical to simply-typed, except one major difference:[13]:

Rule	**In Vernacular**
$\dfrac{\Gamma \cup \{x :: \tau\} \vdash e :: s}{\Gamma \vdash (\lambda x \rightarrow e) :: \tau \rightarrow s}$	*Abstraction:* Notice how, compared to simply-typed λ-calculus, x no longer has a type annotation.

The result of the HM System's lack of type annotation for abstraction is rather large: functions ca have multiple types! More so, HM consists of monomorphic and polymorphic types. A monomorphic type τ is defined as either a variable α or an abstraction $\tau \rightarrow \tau$. On the other hand, a polymorphic type σ is either a monomorphic type τ or a quantifier—that is, $\forall \alpha.\ \sigma$ [NL]. The identity function, $(\lambda x\ .\ x)$ is of polymorphic type and can be applied to any variable $\alpha \rightarrow \alpha$.

To be able to apply polymorphism, we require a generalization rule (see below):

[11] Or, arguably, because of my infinite stubbornness?
[12] See the table for the Curry-Howard Isomorphism between type systems and computer science
[13] I would recommend going back to section 3.7 and comparing that section's rule table to this one.

Rule	In Vernacular
$\dfrac{\Gamma \vdash e :: \sigma \qquad \alpha \notin free(\Gamma)}{\Gamma \vdash e :: \ \forall \alpha.\ \sigma}$	*Generalization:* If our typing judgment is that e is of type σ and our variable α is not free within our context Γ, then under this context we can generalize e to be σ for all α.

Instantiation is required to be able to use polymorphic types:

Rule	In Vernacular
$\dfrac{\Gamma \vdash e :: \sigma\prime \qquad \sigma\prime \sqsubseteq \sigma}{\Gamma \vdash e :: \sigma}$	*Instantiation:* If e is of type $\sigma\prime$, and we know that σ is more specific than $\sigma\prime$, then we can type-judge e to be σ [NL].

Regarding the table above, we prefer σ over $\sigma\prime$ because its higher specificity, in essence its set of constraints, is what we aim for in our strongly-typed system. Instantiation thus allows us to go from a polymorphic type to a specific type. One of the last things to cover, considering we have introduced the differences in type rules, is expression definition. We can define expressions almost like we did for simply-typed λ-calculus, except we no longer have type annotations and, instead, add the following to the end of the definition:

$$e :: x|(\lambda x.e)|(e\ e)|\ \mathbf{let}\ x = e\ \mathbf{in}\ e$$

The let rule is in a way a mixture of the abstraction and application rules:

Rule	In Vernacular
$\dfrac{\Gamma \vdash e_0 :: \sigma \qquad \Gamma \cup x :: \sigma \vdash e_1 :: \tau}{\Gamma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 :: \tau}$	*Let:* If e_0 is of type σ, and if in the same context, under the assumption that x is of type σ, the typing judgement is that e_1 is of type τ, then **let** $x = e_0$ **in** e_1 is of type τ.

This powerful, polymorphic version of typed λ-calculus was used for multiple functional programming languages, such as Haskell 98.[14][15][SF] It's inference capacities allow it to deduce "the most general type of a given program without programmer supplied type annotations or other hints." [HM]

4.2 System F

System F shares many similarities with the HM System[16], namely its parametric polymorphism. System F, however, formalizes its polymorphism, thus "forming a theoretical basis for languages such as Haskell." [SF] It follows closely to simply-typed λ-calculus' typing rules, with the addition of its own application and abstraction rules. Although the previous section went over the differences in rules more in depth, the

[14]Recall section 2.

[15]For further reading and information, I recommend checking out Nicolas Laurent, whose amazing explanations and examples ended up influencing and even appearing in this discussion. The proper citation, as always, can be found at **References**.

[16]Given that the HM System is a restrictive version of System F, as we will briefly discuss in the next page.

interesting parts of System F are elsewhere, and since it shares a lot of similarities with the HM System, I leave the matter here. However, for further reading, as well as the explicit typing rules, see this.

Recall that at the beginning of section 3, we dealt a lot with logic operators. System F, interestingly enough, manages to incorporate them explicitly. Consider booleans: [AM]

Formally	**Name**
$\forall\gamma.\gamma \rightarrow \gamma \rightarrow \gamma$	Bool
$\Lambda\gamma \,.\, \lambda x,\ y\ :\ \gamma\ .x ::$ Bool	True
$\Lambda\gamma \,.\, \lambda x,\ y\ :\ \gamma\ .y ::$ Bool	False

The inclusion of booleans allows for logical operators such as *or*, *and*, and *negation*—$\vee$, $\wedge$, and $\neg$, respectively—of type $Boolean \rightarrow Boolean \rightarrow Boolean$. [AK2] [SF]

Formally	**Name**
$\lambda x^{\text{Boolean}}\lambda y^{\text{Boolean}}.x$ Boolean y **False**	And
$\lambda x^{\text{Boolean}}\lambda y^{\text{Boolean}}.x$ Boolean **True** y	Or
$\lambda x^{\text{Boolean}}.x$ Boolean **False True**	Not

The incorporation of logical operators and booleans allows for the presence of recursive functions, as we can now evaluate conditions and assign truth values to them. As we can see, System F is powerful: it allows for strong type-checking as well as increased versatility with respect to our simply-typed λ-calculus. A striking characteristic of System F, however, is that type inference is impossible—that is, a "type checking is undecidable for a Curry-style variant of System F, that is, one that lacks explicit typing annotations." [SF].

The result was proved by J.B Wells in 1996, in his paper titled *Typability and type checking in System F are equivalent and undecidable.* He states, "typability asks for a term whether there exists some type it can be given. Type checking asks, for a particular term and type, whether the term can be given that type." His proof uses reduction from semi-unification[17] and states that "because there is an easy reduction from typability to type checking, the two problems are equivalent." He proves that semi-unification can be reduced to type checking "using simple encoding". Wells then shows that since semi-unification is undecidable[18], so is type checking. From there, he further reduces type checking to typability "using a novel method of building λ-terms which simulate arbitrarily chosen *type environments.*" [JW]

Although the proof is beyond the scope of this class, the implications, as previously mentioned, are very interesting: type inference is impossible in System F. This is one of the contrasting aspects between System F and the HM System. The HM System allows for type inference, as we showed above via its typing rules and weaker restrictions on its parametric polymorphism. However, Wells goes on to state the following on pages 114 and 115 of his proof: [JW]

> System F exhibits a kind of polymorphism called parametric polymorphism, meaning that a λ-term operates in the same way over a range of types, the particular type used in any instance being determined by a type parameter. The Hindley/Milner type system supports a weak restriction of parametric polymorphism which allows a polymorphic function to be passed as a parameter,

[17]For Wells' definition, see pages 122-123 of his proof.
[18]Proven by Kfoury, Tiuryn, and Urzyczyn in the 1990's.

but only to a single predetermined non-polymorphic function. Also, a polymorphic function can have only monomorphic arguments. In the full System F, a polymorphic function can be passed as a parameter to another non-predetermined polymorphic function, which itself can be passed as a parameter to other such functions, and so on. This flexibility is reflected in the fact that System F types more λ-terms than does the Hindley/Milner type system.

If you have used Haskell, either to run the examples provided on the earlier parts of this paper or for class assignments—or perhaps for your own enjoyment, which I commend—you most likely have used GHC[19], a Haskell compiler. GHC "goes beyond HM (as of 2008) and uses System F extended with non-syntactic type equality." [SF]. It is rather visible the influence and deep ties that λ-calculus has with programming language theory, be it via untyped, simply-typed, HM, or System F "styles". The impressive thing, however, are the different variations, each with their own impacts on other fields, of λ-calculus, such as System U, System F_ω, System LF, λ-calculi with dependent types, and the lambda cube by Henk Barendregt. I invite you, dear reader, to explore some of these subjects, as they all have their own abstract beauty that can only be fully appreciated after some time has been dedicated to understanding them, as I hope these past sections have shown true.

4.3 Final Notes on the Curry-Howard Isomorphism

Although the Curry-Howard Isomorphism was already covered in section 3, I want to end this section by stressing both its importance and the beauty behind it. The isomorphism is the basis for a wide body of work regarding programming language theory as well as the leap in computational advancement over the past four decades. It provides not only a model for derivations and normalization, but mainly a type checker for programs—a system which has been extended, modified, and strengthened throughout the past decades, each time evolving. The isomorphism "establishes that well-formed terms can always be type checked by a derivation in normal form. Since derivations in normal form have the subformula property, searching for a derivation that type checks a given program can (often) be done effectively." [RZ] The Curry-Howard Isomorphism guarantees type safety. As seen in our comparison tables in section 3.6, having the capacity of type checking programs are programming languages' versions of soundness and completeness.

On a more general basis, however, the isomorphism is truly something beautiful. It is difficult to grasp at first and, even when finally understood, it seems like nothing other than something that was once proven true. Yet it is more than that. It acts as a unifying force between different subjects within Logic, Computer Science, and Mathematics[20]. It is one thing to understand the correspondence amongst the different topics and another to fully appreciate it. The appreciative part, however, is rather difficult to do. One can not be taught to appreciate it, only to understand it. The isomorphism can be seen in almost all of the topics and sections of this report, either hiding behind certain themes or being explicitly out there. The entirety of computer science and Programming Language Theory can be stated to depend on the Curry-Howard Isomorphism. It is one of the deepest and most beautiful ideas discovered; it's beauty can't be taught, only found through exploration and analysis of multiple subjects all while having an inquisitive, eager, persistent, and patient mind at hand. And the day the isomorphism's beauty starts to seem apparent is the day that all the subjects that it relates to suddenly seem to morph into one big, interconnected, amazing field of study.

[19]Or GHCI, which is GHC's interactive environment.

[20]Given the right conditions hold, as specified in previous sections.

5 Conclusions

The development of this report was characterized, personally, by a growing sense of thematic-connection, almost like its own small isomorphism. The overarching structure of the paper changed multiple times throughout the semester. As the Haskell section was being written, I found myself suddenly discovering interesting topics related to the subject matter—or, if not directly related, capable of leading towards in later sections. The process of researching articles, proofs, tutorials and other sources of information slowly uncovered a multitude of doors through which exploration was not only possible, but desired.

The process, I would say, followed that of opening a door with a certain path in mind, and with each door suddenly finding a range of other interesting possible trails, often exploring, enjoying, and then backtracking to resume the main exploration. This was done with the intention of finding the optimal path for exploring and presenting programming language theory. It is a daunting subject, not easily approachable, especially given that the rigor and subjects are different from what most students expect. But this apparent exploration from "door" to "door" led me down a wonderful road—from Haskell to λ-calculus, and from there to the many intricacies of the system and its multiple variations. I came to find an incomparable beauty in the study of Programming Language theory, one that sky-folded my appreciation for these abstract and at first seemingly-unapproachable topics.

This paper serves as an invitation to learn about one of the most beautiful ideas in computer science, the Curry-Howard Isomorphism, through an indirect approach, guided by different subjects that slowly build on each other. It serves as an invitation to draw parallels across sections and subjects which lead to a better understanding of the bigger picture. Yet, most importantly, it serves as an invitation to learn something new. The most fascinating part, perhaps, is that through each page turned, each section and subsection completed, and each door gone through, countless more topics and doors pop up. I invite you to continue this exploration.

References

[PL] Programming Languages 2021, Chapman University, 2021.

[AK] Type Checking, Type Inference Typed Lambda Calculus.

[AK2] Lambda Calculus: Extensions Alexander Kurz.

[AM] An Introduction to System F Alexandre Miquel.

[BA] Simply-Typed lambda Calculus, part 1: Types Benedikt Ahrens.

[BA2] Simply-Typed Lambda Calculus, part 2: Terms Benedikt Ahrens.

[BA3] Simply-Typed Lambda Calculus, part 3: Curry-Howard Benedikt Ahrens.

[CO] Consistency, completeness, and normal forms.

[CPh] Programming Paradigms - Computerphile ComputerPhile.

[CS] Simply-typed lambda calculus Programming Languages and Logics.

[FB] How Effective is Facebook at Detecting Bad Content? Statista.

[GFG] Python Program for Fibonacci Numbers Geeks for Geeks.

[GG] Proof Format Internet Encyclopedia of Philosophy.

[GH] Case Expressions and Pattern Matching *A Gentle Introduction to Haskell.*

[GM] Gmail Wikipedia.

[HA] Haskell in Industry Haskell.

[HM] Hindley–Milner type system.

[HPL] Simply-typed lambda calculus CS 152.

[JDC] Beta reduction: The difference typing makes John D. Cook.

[JW] Typability and type checking in System F are equivalent and undecidable J.B. Wells.

[KM] 3 type checking and simply typed lambda calculus Kenny Zhuo Ming LU.

[MI] Pure Functions and Side Effects Maksim Ivanov.

[MWS] Introduction to Haskell, Caesar Shift Cipher MultiWingSpan.

[NDP] The Natural Deduction Pack Alastair Carr.

[NF] Normal form (abstract rewriting).

[NL] The Hindley-Milner Type System Nicolas Laurent.

[PMP] The Curry-Howard Isomorphism Pierre-Marie Pédrot.

[RR] A Tutorial Introduction to the Lambda Calculus Raul Rojas.

[RZ] The Significance of the Curry-Howard Isomorphism Richard Zach.

[SD] Handbook of Proof Theory.

[SF] System F.

[SIG] Fighting spam with Haskell Facebook engineering.

[ST] Strongly Typed Programming Language What Is.

[STWT] Magic lies here - Statically vs Dynamically Typed Languages AndroidPub.

[TC] Type Conversion in C Geeks for Geeks.

[WK] Caesar cipher Wikipedia.

[WP] Simply typed lambda calculus Wikipedia.

[WT] Weakly Typed Languages.