Department of Mathematics and Computer Science University of Mas-

sachusetts Lowell
Dr. Tibor Beke
Bachelor of Science in Mathematics and Computer Science August 4, 2023

Category-Theoretic Foundations of the Curry-Howard Correspondence

Shriya Thakur

August 4, 2023

I would like to thank all those who have supported and contributed to the completion of this work.
First and foremost, I want to take a moment to extend my heartfelt thanks to my advisor Dr. Tibor Beke, whose ecouragement, guidance, valuable feedback, and expertise have been indispensable throughout this journey. .
In addition, I would also like to acknowledge Dr. Paul Downen from the department of Computer Science whose expertise and guidance have shaped my research, exposure, and focus throughout my undergraduate studies here.
I would like to acknowledge the University of Massachusetts Lowell for providing the resources and environment necessary for completing this. Access to the ACM digital library, textbooks, and papers have been a stepping stone in the successful completion of this paper.
I would like to acknowledge and give thanks to the opportunities that continue to pioneer my future and shape my passion including conferences like OPLSS'22 and POPL'23. I am grateful to the talk series, insights, teachings and notes taken during my time there and to the many researchers I met. This work also has its roots in papers from Dr. Frank Pfenning and Philip Wadler to name a few so I would like to extend my thanks to them.
Lastly, I am indebted to my family and friends for their unwavering support and encouragement all along. Their belief in me has been a source of motivation throughout.

1 Abstract

In this paper we explore an age old but comparatively new concept- The Curry-Howard Correspondence- an isomorphism that connects two important fields: Logic to Computer Science. This concept shows us that proofs in Intuitionistic logic can be translated into the Simply Typed lambda-calculus which serves as the foundation for many programming languages including Haskell and Scheme.The heart of this study is figuring out how to translate logical proofs into programs. We find out that logical ideas become types in the computer language, and the rules of logic become instructions in the language. In the end, the Curry-Howard Correspondence shows us how logic and computing are related in an un-intuitive way which furthers our understanding in both logic and Programming Languages.

Contents

2 Introduction

Category Theory is a branch of mathematics that provides a framework for studying the relationships between mathematical structures and their transformations. As mentioned in [2] It is characterized by its emphasis on formalizing the fundamental concepts of mathematics in a way that is independent of specific structures or objects. Category Theory is defined as the study of a collection of objects and a collection of arrows, also known as morphisms between such objects. It also provides a way to describe the interconnection of objects, and that includes how categories themselves relate to each other. You can relate one category to another using something called a functor.
Some examples of objects and morphisms are given in the table below:

Objects	Morphisms
Sets	Functions
Groups	Group Homomorphisms
Topological Spaces	Continuous Maps Between Spaces
Vector Spaces	Linear Transformations
Posets	Order preserving Maps
Categories	Functors

However, contrary to intuition, objects do not need to be a collections of "elements," and morphisms do not need to be functions between these elements. Thus morphisms cannot be applied to "elements" but only composed with other morphisms making function composition the primary axiom.
Any direct access to the internal structure of an object is prevented: all properties of objects must be specified by properties of morphisms. This is quite similar to considering objects as "Abstract Data Types (ADT's)" that is, data specifications that are independent of any particular implementation. Category Theory offers a language especially suited for stating abstract properties of structures and in object oriented design. In addition, the Curry Howard Correspondence relates propostions and proofs in Natural Deduction systems to Types and programs respectively in programming languages. Thus, it relates to widely used programming methodologies and provides a basis for the semantics of programming languages.

3 History of Curry Howard Isomorphism

The Curry-Howard isomorphism can be traced back with the work of mathematicians and logicians who made significant contributions to both logic and type theory. The development of this isomorphism spans several decades, and its roots can be found in the study of foundational questions in mathematics and the foundations of computation.

In the 1930's, Alonzo Church introduced the Lambda Calculus, a formal system for expressing computation based on the notion of lambda abstraction and function application. This became a foundational concept in the study of computation and later played a crucial role in the development of the Curry-Howard isomorphism.

Haskell Curry, in collaboration with others, developed combinatory logic, an alternative formal system for computation based on a set of combinators instead of functions. His work laid the foundation for later developments in lambda calculus and its connections to logic.

Later in 1958, Kurt Gödel introduced a method for translating Intuitionistic Arithmetic into Combinatory logic which established a connection between Intuitionistic logic and Combinatory Logic, which later became an essential component of the Curry-Howard isomorphism.

In the 1960's, logicians explored proof theory and the idea of Natural Deduction (described in a later section), where logical proofs are represented as trees. This work provided insight into the connections between proofs and computations.

The term "Curry-Howard Isomorphism" was coined in 1969 by William Alvin Howard in his paper [4] where he formalized the connection between proofs in Natural Deduction and lambda terms, showing that propositions in Intuitionistic logic correspond to types in the simply typed lambda calculus. This marked the formalization of the isomorphism.

Over the following decades, the Curry-Howard isomorphism continued to be explored and expanded. Researchers in Logic, Type Theory, and Computer Science explored its implications and applications in areas like functional programming, formal verification, and constructive mathematics.

The Curry-Howard correspondence has also led logicians and computer scientists to develop new logics based on the correspondence between proofs and programs such as in Coq. In these systems, rather than write a program to compute, say, the integer square root function, one would instead prove a corresponding theorem, such as that for all integers x such that $x \geq 0$, there exists a largest integer y such that $y^2 \leq x$. One could then extract the integer square root function, which given x returns the corresponding y, directly from the proof. Early systems of this sort include Automath,

developed by the logician De Bruijn in 1970.
The story continues to develop and expand even today. For instance, one might wonder whether the Curry-Howard correspondence could extend to Classical logic. Indeed it can, as shown by Timothy Griffin in 1990 in his paper [4]

Intuitionistic Logic	STLC
Proposition	Type
Proof	Programs aka Lambda Term/s
Implication Introduction	λ term(function) Function Type
Modus Ponens or Implication Elimination	Function Application- Applying a λ term representing an implication to an argument, the result is a term of the conclusion of the implication
Conjunction Introduction	Creating a pair of terms, product type
Conjunction Elimination	Extracting the individual propositions from a pair (either the first or second)
Disjunction Introduction	Sum Type
Natural Deduction	Type System for STLC

Table 1: Translating from Logic to STLC

4 Curry-Howard Isomorphism

Category theory plays a crucial role in understanding the Curry-Howard correspondence by providing a foundation to study both logic and type theory. The correspondence can be viewed as an instance of a broader categorical principle, relating the structure of logic and types to the structure of categories.
As talked about in [4] category theory, propositions can be represented as objects, and proofs can be seen as Morphisms between these objects. In a sense, category theory abstracts the common structure shared by both logic and type theory, providing a more general perspective on the Curry-Howard correspondence.
Furthermore, category theory provides tools to study the relationships between categories, Functors, and natural transformations, which have counterparts in logic and type theory.
The table 1 shown below helps understand how various rules in Intuitionistic Logic translate to The Simply Typed Lambda Calculus. In section 11 we have worked through some basic examples of translating Propositions to the Simply Typed Lambda Calculus.

5 Axioms of Category Theory:

In his notes [11] Jaap talks about the primary axioms of Category Theory which include:

1. Composition: For any two arrows $f : A \to B$ and $g : B \to C$, there exists a unique arrow.

2. Identity: Every object has an identity arrow that acts as an identity transformation for that object.

3. Associativity: The composition of arrows is associative.

The beauty of category theory lies in its ability to capture common structures and concepts across different mathematical disciplines. Regarding the connection to the algebra of computation in the lecture series [1] Thorsten explains how Category Theory has found significant applications in theoretical computer science, particularly in the study of functional programming, Type Theory, and semantics of programming languages via the Simply Typed Lambda Calculus. Some of the connections are as follows:

1. Type Theory and Type Systems: The concept of categories has been used to model and reason about various Type Systems. The Curry-Howard correspondence, for instance, establishes a connection between types and logical propositions, showing that programs and proofs can be seen as mathematical objects. This connection is based on the notion of categories and provides a foundation for proving the correctness of programs. We will dive more in detail on this in later sections.

2. Functors: Functors from category theory are used to describe structure-preserving mappings between categories. In the context of programming languages, functors have been applied to capture behaviors of data structures and their transformations.

3. Monads and Functional Programming: Monads, as mentioned earlier, are an essential concept from category theory and have become a fundamental abstraction in functional programming languages like Haskell. Monads are used to represent computations with side-effects in a pure functional way. They provide a structured approach to handling effects like state, non-determinism, and error handling, allowing programmers to reason about these effects independently of the core logic of their programs.

4. Category-Theoretic Semantics: Category theory provides a foundation for the Denotational semantics of programming languages. Denotational semantics aims to give a precise mathematical meaning to programs.

Overall, category theory has provided valuable insights into the design and semantics of programming languages, allowing for more principled and rigorous approaches

to programming. It has influenced the development of functional programming languages and the design of type systems, bringing a deeper level of abstraction and clarity to Computer Science.

6 Category Theory and The Curry Howard Correspondence

Category theory, with its emphasis on the study of objects, arrows, and their relationships, has proven to be a powerful and unifying framework in various branches of mathematics and computer science. It provides a versatile language to explore the common structures and connections between mathematical concepts. But what is even more fascinating is how category theory bridges the gap between seemingly disparate fields, leading to various insights and unexpected correspondences as shown in [10] such as the Curry Howard Correspondence as mentioned in later sections.

One of the most striking examples of this bridge is the Curry-Howard isomorphism, a remarkable correspondence that establishes a deep connection between logic and type theory. The Curry-Howard correspondence (in its simplest form) is between proofs in Intuitionistic Minimal logic and terms of the Untyped Lambda Calculus.
Traditionally, the correspondence identified by Curry and formalized by Howard relates Intuitionistic logic and the Simply-Typed λ-calculus. As is well-known, though, this correspondence holds in other contexts, too. Indeed, in the last fifty years more sophisticated Type Systems have been put in relation with expressive logical formalisms: from polymorphism to various forms of session typing,including dependent types. In order to gain a deeper understanding, we will go over some basic Natural Deductions but mainly focusing on Intuitionistic logic and Typed Lambda Calculus.

7 Natural Deductions: Intuitionistic Logic

Natural deduction, as introduced to me during OPLSS in the lecture series [9], is a method of formal reasoning in logic, introduced by Gerhard Gentzen in the 1930's. It is characterized by its intuitive and constructive approach to proving propositions. In natural deduction, logical proofs are represented as structured trees, where each node in the tree represents a step in the proof. The rules of natural deduction allow for the introduction and elimination of logical connectives such as conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and negation ($\neg$).
For the scope of this paper, we will follow in the footsteps of the paper [7] and thus limit ourselves to Intuitionistic Logic since this form of logic is associated with the CHC due to its constructive nature and excluding some of the laws of Classical Logic in order for successful computations. Intuitionistic Logic excludes concepts like:

1. Law of Excluded Middle: The principle that states that every proposition P is either true or its negation is true aka there is no third choice.
   P is True $\vee$ $\neg$P is True

2. Double Negation Elimination: The principle that states that if $\neg(\neg P)$ is true, then P is true. Thus in Intuitionistic Logic, as against in Classical Logic, the double negation of a proposition cannot be eliminated.

Since these statements lead to non-constructive proofs. Such proofs do not provide direct evidence from the previous steps to obtain the results of a given computation and thus cannot be used to infer types in a Type System of a Programming Language. The Curry Howard isomorphism is a correspondence between proofs in natural deduction systems and terms in Simply Typed lambda calculus. A proof is thought of as establishing that the conclusion follows from the set of assumptions that remain true (called the Context). In "Sequent style" natural deduction, each Sequent $\Gamma \vdash$ A in the proof records the set of assumptions that a formula follows at every step in the proof. So an assumption by itself is recorded as A $\vdash$ A, and generally a proof with end-formula A from open assumptions Γ would be represented, in Sequent style natural deduction, as a tree of sequents with end sequent $\Gamma \vdash$ A.
In this, the Inference Rules are as follows:

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \ldots 1$$

The proof tree 7 represents an Inference Rule. A and B are propositions and A actually is an assumption whereas B is the proposition we derive under that assumption. Γ is the context which consists of the list of propositions that are true.Lastly, the $\vdash$ represents the Turnstyle which indicates that we are proving a proposition.
This is saying that if we have a way to prove B from the assumption A and the context Γ, then we can claim that A $\rightarrow$ B holds in Γ. The implication introduction rule captures the idea that if we have a valid argument for B based on the assumption A, then we can conclude that A $\rightarrow$ B is a valid proposition in the given context.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \ldots 2$$

This next proof tree 7 shown above represents the Inference Rule for Conjunction. In everyday terms we say that if A is true and B is also true we can assume that A $\wedge$ B is also true. Using the meaning of symbols mentioned in the first proof tree we can see that A and B are premises in the propositions which are true in the context Γ. This tells us that since we have separate proofs for both A and B in Γ, then we can combine these proofs to conclude that the conjunction A $\wedge$ B is also true in the same context.

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \ldots 3$$

The proof tree 7 shown above represents an Inference rule for Implication $\to$(translates to function types in the STLC as discussed in the next section). It tells us that if we have a proof that A $\to$ B (The premises are the propositions A $\to$ B and A, which are assumed to be true in Γ), and we also have a proof that A is true, then we can conclude that B is true as well.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \ldots 4$$

The proof tree 7 shown above represents the Inference Rule for Conjunction $\wedge$ elimination. This tells us that if A $\wedge$ B is known to be true in Γ then we can conclude that A is true. (translates to the Product Type (A, B) in the STLC)

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \ldots 5$$

Just like in the example that came before, this proof tree also represents the Inference Rule for Conjunction Elimination. If in the context we know that A $\wedge$ B is true then we can conclude that B is true.

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \ldots 6$$

The proof tree mentioned above tells us that if in Γ we know that $A \to B$ and A are true then we can conclude B is True (translates to Lambda Abstraction in the STLC) Note: We're not keeping track of which assumptions are discharged where.

7.1 Intuitionistic Minimal Logic

"Intuitionistic Minimal Logic" only contains propositional variables, the connective ($\rightarrow$) and parentheses. It's sometimes also called "Intuitionistic Implicational Logic" since the implication connective is its primary operator.
In order to gain a clearer understanding we will now explore some axioms of logic:

1. **Modus Ponens:** If we proved X $\rightarrow$Y and X then we've proved Y.

2. $A \rightarrow (B \rightarrow A)$
   This can be represented as the proof tree given below:

$$\dfrac{\dfrac{\dfrac{\overline{A \vdash A}}{A, B \vdash A}}{A \vdash B \rightarrow A}}{\vdash A \rightarrow (B \rightarrow A)}$$

   (a) Lambda term has type $a \rightarrow b \rightarrow a$ if x:a and other lambda has type $b \rightarrow a$

   (b) The other term has type $b \rightarrow a$ if in Γ x: a, y: b, x has type a

   (c) Thus by identity the original Lambda Abstraction is valid in STLC

   In everyday language this could be thought of as follows:

   (a) A: "The sun is shinning"

   (b) B: "People will be happy."

   (c) If the Sun is shining (A is true), then if people will be happy (B is true) then it will mean that the Sun is still shinning.

3. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$\dfrac{\dfrac{\dfrac{A, B \vdash C}{A \vdash (B \rightarrow C)}}{A \rightarrow (B \rightarrow C), A} \quad \dfrac{A \vdash B}{A \rightarrow B}}{\vdash (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))}$$

1. The first line states that if we have two statements "A" and "B" together, then we can conclude another statement "C." In other words, when both "A" and "B" are true, we can infer that "C" must also be true.

2. The second line introduces the concept of implication. In the context "A" is true, we can deduce a new statement "$B \rightarrow C$." This means that if "A" is true, then "B" being true would lead to "C" being true as well.

3. The third line combines the two previous statements. It tells us that if we have a statement "$A \rightarrow (B \rightarrow C)$" and also know that "A" is true, then we can deduce that "$B \rightarrow C$" is true.

4. The fourth line introduces another statement: "$A \to B$" This means that if in context "A" is true, then "B" is also true.

5. Finally, the last line combines the information from the third and fourth lines. It says that if we have a statement "$A \to (B \to C)$" and also know that "$A \to B$" is true, then we can further deduce that "$A \to C$" is true.

In everyday language this could be thought of as:

1. A: The weather is sunny

2. B: We will have a beach picnic

3. C: We will have fun

4. If the weather is sunny then we will have fun given that we have a beach picnic. Supposing we have a beach picnic if the weather is sunny, then we will have fun if the weather is sunny.

8 Simply Typed Lambda Calculus

A type is a collection of computational entities that share some common property. For example, the type int represents all expressions that evaluate to an integer, and the type int $\rightarrow$ int represents all functions from integers to integers. The Pascal subrange type [1..100] represents all integers between 1 and 100. Uses of type systems include: naming and organizing useful concepts; providing information (to the compiler or programmer) about data manipulated by a program; and ensuring that the run-time behavior of programs meet certain criteria.
As defined and developed on in [6] the Simply Typed Lambda Calculus (STLC) is a formal system bridging Mathematical Logic and Computer Science. It is an extension of the Untyped Lambda Calculus, introduced by Alonzo Church in the 1930's. The STLC introduces type annotations to the lambda calculus, providing a more structured and well-behaved system for modeling computations and proofs.
Although the syntax and reduction relations for the λ-calculus are minimal, it is closely related to practical languages like Haskell, Scheme and ML. In those languages, functions not only manipulate Booleans, integers, and pairs (product types as described), but also other functions. In other words, functions are values. For example, the map function consumes a function and a list of elements, and applies the function to each element of the list.
Lambda expressions are defined as follows:

1. Variable: a, b, c, . . . are lambda expressions.

2. Abstraction: If b is a lambda expression then $\lambda x.b$ is a lambda expression for any variable x. (It is a function taking one argument x and with body b.) When we have multiple lambdas in a row we abbreviate as follows, $\lambda xy.b := \lambda x.\lambda y.b$.

3. Application: If x and y are lambda expressions then $(x\ y)$ is a lambda expression. (It is applying y as an argument to x.).

Some points to note include:

1. Application associates to the left. eg: $M_1\ M_2\ M_3$ means $((M_1 M_2) M_3)$

2. Application is stronger than abstraction. eg: $\lambda X.M_1 M_2$ means $(\lambda X.(M1M2))$

3. Consecutive lambdas can be collapsed. eg: $\lambda XYZ.M$ means $(\lambda X.(\lambda Y.(\lambda Z.M)))$

In the STLC, every term is associated with a specific type. Types represent the classification of expressions, and they specify what kind of values the terms can produce. Each lambda (λ) abstraction (function) in the STLC is annotated with a type for its parameter, and every application of a function is checked to ensure the types of the arguments match the expected types.
The STLC is type-safe, meaning that well-typed programs (those with correct type annotations) will never encounter type errors during evaluation. Type safety is a crucial property that guarantees the absence of certain run-time errors.

The syntax of the λ-calculus provides a simple, regular method for writing down functions for application, and also as the inputs and outputs of other functions. The specification of such functions in the λ-calculus concentrates on the rule for going from an argument to a result, and ignores the issues of naming the function and its domain and range.

8.1 Syntax of λ terms:

Let's take the identity function (id) into account.
The specification of such functions in the λ-calculus concentrates on the rule for going from an argument to a result, and ignores the issues of naming the function and its domain and range. Mathematically the identity function on some set A is represented as:

$$\forall x \in A, f(x) = x \tag{1}$$

In the λ-Calculus syntax this is represented as,

$$(\lambda x.x)$$

So this function outputs the datum that it inputs.
To write down the application of a function f to an argument a, the λ-calculus uses ordinary mathematical syntax, modulo the placement of parentheses:

$$(fa)$$

The expression representing the application of the identity function to a is:

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

Lastly, a possible argument for the identity function is the identity function itself as follows:

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

Here is an expression representing a function that takes an argument, ignores it, and returns the identity function:

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

Here is an expression representing a function that takes an argument and returns a function; the returned function ignores its own argument, and returns the argument of the original function:

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

The λ-calculus supports only single-argument functions, but this last example shows how a function can effectively take two arguments, x and y, by consuming the first argument, then returning another function to get the second argument; A technique called **currying**.
In conventional notation, f(a) can be "simplified" by taking the expression for the definition of f, and substituting "a" everywhere for f's argument. For example, given

$f(x) = x$, f(a) can be simplified to a. Simplification of λ-calculus terms is similar:

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

can be simplified to a, which is the result of taking the body (the part after the dot) of ($\lambda x.\ x$), and replacing the argument x (the part before the dot) with the actual argument a. Here are some such examples:

Lambda Term	Result	Explanation
$((\lambda x.x)a)$	a	replacing x with a in x
$((\lambda x.x)(\lambda y.y))$	$\lambda y.y$	replacing x with (λy.y) in x
$((\lambda y.(\lambda x.y))a)$	$(\lambda x.a)$	replacing y with a in (λx.y)

Now we will introduce the Booleans - True and False:

$$False := \lambda x.\lambda y.y$$

$$True := \lambda x.\lambda y.x$$

Using this we can introduce If-Then-Else conditions in the STLC

$$\lambda a.\lambda x.\lambda y.axy$$

Here 'a' is the condition (a Church Boolean), 'x' is the expression to be evaluated if the condition is True, and 'y' is the expression to be evaluated if the condition is False.

8.2 α-Equivalence

As explained in the book [8] the α-equivalence allows us to reason about λ-calculus expressions without being concerned about the specific names of bound variables, simplifying the process of Reduction to compute a result. It helps with reduction by ensuring that we can perform β-reduction without worrying about unintended variable clashes or capture. The α relation renames a formal argument also known as a bound variable. It encodes the fact that functions like (λx.x) and (λy.y) are the same function, simply expressed with different names for the argument.
In simple terms, this equivalence allows us to freely rename bound variables as long as we do so consistently within the same expression as so:

$$(\lambda x.xy) \equiv (\lambda z.zy)$$

In the former expression above λx is a binding site or abstraction for the variable x with a function body (x y). Now we can use α-equivalence to consistently rename the bound variable x to another name, such as z as shown in the latter lambda expression. It is also essential to note that Alpha equivalence does not apply to free variables, which are variables that are not bound by a lambda abstraction also called a binding site. In the example given above, y, is the free variable.
Let's work through another example of such an equivalence.

8.3 Substitution

Substituting allows us to replace a bound variable's name with an expression. The syntax for substitution is:
Expression [variableName → otherExpression].

Now that we know this, let's substitute x in an expression which adds 1 to the number passed to it. In this case, we substitute x by 2:

$$(\lambda x. + x\ 1)[x \to 2]$$

$$(+2\ 1)$$

As might have you noticed in the example above, a substitution is what happens when an expression is evaluated: when you *apply* something to it. In the previous example that substitution is what would have happened if we applied 2 to that expression:

$$(\lambda x. + x\ 1)2$$

$$(\lambda x. + x\ 1)[x \to 2]$$

$$(+2\ 1)$$

To demonstrate this better, let's use an expression that's a little bit more complicated:

$$\lambda x.\lambda y. + y(+\ x\ 1)$$

Applying 4 to the given expression we get,

$$(\lambda x.\lambda y. + y(+\ x\ 1))4$$

$$(\lambda x.\lambda y. + y(+\ 4\ 1))[x \to 4]$$

$$(\lambda y. + y(+\ 4\ 1))$$

Finally we can also apply 2 to the resulting expression, which will then result in:

$$(\lambda y. + y(+\ 4\ 1))2$$

$$(\lambda y. + y(+\ 4\ 1))[y \to 2]$$

$$(+\ 2(+\ 4\ 1))[y \to 2]$$

$$(+\ 2(+\ 4\ 1))$$

$$(+\ 2\ 5)$$

$$(7)$$

Note: One cannot substitute free variables in a lambda expression.

8.4 Beta-Reduction

The β-relation is the main reduction relation, encoding function application. All beta reductions follow the rule:

$$\frac{(\lambda x.a)b}{a[\frac{b}{x}]}$$

which means if we have a lambda expression (or sub-expression) of the form $(\lambda x.a)b$, then we can reduce that expression or sub-expression to $a[b/x]$. The meaning of this is take the term b and wherever x appears as a free variable in a, replace x with b. We say an expression is in normal form if no more reductions can be performed on any expression or sub-expression. Evaluating a lambda expression is the process of performing beta reductions.
Let's take an example of a beta reduction:

$$((\lambda x.\lambda y.y)((\lambda z.z)(\lambda y.\lambda x.y)))a$$
$$((\lambda x.\lambda y.y)(\lambda y.\lambda x.y))a$$
$$(\lambda y.y)a$$
$$a$$

Under the Curry-Howard correspondence, the STLC provides a way to represent and reason about logical propositions and proofs as types and terms, respectively. This correspondence demonstrates the deep connection between logic and computation.

Specifically, as mentioned in [3], in the Curry-Howard correspondence:

1. Types as Propositions: Types in the STLC correspond to logical propositions in constructive logics like Intuitionistic. For example, the type 'Bool' might correspond to the logical proposition "true or false," and the function type 'Bool $\rightarrow$ Bool' might correspond to the logical proposition "a proof that, given a proof of true, produces a proof of true."

2. Terms as Proofs: Terms (Λ expressions) in the STLC correspond to proofs of the corresponding logical propositions. For instance, a lambda expression that takes a boolean argument and returns the same boolean could represent a proof of the tautology "A $\rightarrow$ A".

3. Type Inference and Proof Checking: The process of type inference in the STLC corresponds to checking the validity of logical proofs. The type checker ensures that the terms are constructed in accordance with the logical structure of the propositions.

By establishing this correspondence, the Simply Typed Lambda Calculus provides a foundational framework for representing and manipulating proofs using lambda expressions in programming languages.

9 Propositions Are Types

The proposition-as-types principle states that logical propositions can be viewed as types in a programming language. In his paper [13] Wadler explains that for every proposition in logic, there exists a corresponding type in the type system of a suitable programming language, and vice versa.
Given a type a → b, simply means that a → b. Of course, this only makes sense if a and b are types which can further be interpreted in symbolic logic. This is the essence of CHC. Furthermore, as we mentioned before, a → b is a theorem iff a → b is an inhabited type.
Let's see this using one of the simplest of Haskell functions.

```
const :: a -> b -> a
```

Translated into logic, we have that a → b → a. This must be a theorem, as the type a → b → a is inhabited by the value const. Now, another way of expressing it is 'If we assume a is true, then b must be true.' So a → b → a means that if we assume a is true, then if we further assume that b is true, then we can conclude a is true.

10 Proofs Are Programs

This principle reflects the observation that certain formal systems of logic can be interpreted as type systems in programming languages. As a consequence, proofs in logic can be seen as constructive programs that produce specific computational results. The Curry-Howard correspondence allows us to view programming with lambda calculus as a form of constructive logic, where programs are the constructive proofs of logical propositions. The fundamental reasons why proofs are considered programs are enumerated in the paper [12] and are as follows:

1. Constructive Nature of Proofs: The Curry-Howard correspondence is particularly relevant in constructive logic, where a proof demonstrates the constructive method of establishing the truth of a proposition. Constructive logic rejects the Law of Excluded Middle (i.e., the proposition is either true or false) and focuses on providing explicit evidence or a constructive procedure for the truth of a statement. In this setting, a proof can be seen as a step-by-step method or algorithm to construct the evidence for a proposition's truth.

2. Correspondence between Propositions and Types: The Curry-Howard correspondence establishes a bijection between propositions in logic and types in a programming language. Each proposition corresponds to a type, and vice versa.

3. Logical Connectives as Type Constructors: Logical connectives (e.g., Conjunction, Disjunction, Implication) allow us to combine propositions. Under the Curry-Howard correspondence, these logical connectives correspond to type constructors in the programming language. Some of which are mentioned in 4

4. Programs as Computational Evidence: From the perspective of the programming language, a program represents a computation that produces a particular result. Under the Curry-Howard correspondence, a proof of a proposition is viewed as a program that computes the evidence for the truth of the proposition.

Lets take an example.
In Haskell, we can represent logical propositions as types and proofs of those propositions as functions that compute evidence for the truth of the propositions.
Let's consider the logical proposition A $\wedge$ B $\rightarrow$ A and demonstrate how a proof of this proposition can be represented as a Haskell function.

– Logical proposition A $\wedge$ B $\rightarrow$ A represented as a Haskell type

```
$ type AAndBImpliesA = (Bool, Bool) -> Bool
```

Now let's consider this in the STLC:
We define types using a special type **Type**. For this example, let's assume that A and B are two types, and AAndBImpliesA represents the function type that takes a pair of types (A, B) and returns a value of type A.

```
AAndBImpliesA = (A, B) -> A
```

Now, we can represent this function proofAAndBImpliesA in the STLC as follows:

$$proofAAndBImpliesA = \lambda p : AAndBImpliesA.\lambda x : A.\lambda y : B.p(x, y)$$

Note: For convenience we have simplified the STLC notation.
Here, we define a lambda abstraction for proofAAndBImpliesA that takes an argument p of type AAndBImpliesA. The lambda abstraction then takes two additional arguments x and y, representing elements of types A and B, respectively. Finally, the lambda abstraction applies the function p to the pair (x, y) to return a value of type A.

The Haskell equivalent of the STLC function is written below.

```
-- Proof as a Haskell function
proofAAndBImpliesA :: AAndBImpliesA
proofAAndBImpliesA (a, _) = a
```

In this example, we define the type AAndBImpliesA to represent the logical proposition "A $\wedge$ B $\rightarrow$ A". The type takes a pair of Booleans (Bool, Bool) as input and returns a Boolean Bool as output. This type corresponds to a function that takes two Boolean arguments A and B and returns the Boolean A.
This function serves as a proof of the logical proposition, demonstrating that if A and B are both True, then the result is True, which aligns with the truth of the logical implication.

```
-- Example usage of the proof function
result1 :: Bool
result1 = proofAAndBImpliesA (True, False) -- Output: True

result2 :: Bool
result2 = proofAAndBImpliesA (True, True)  -- Output: True
```

The Haskell function proofAAndBImpliesA serves as a constructive program that corresponds to a proof of the logical proposition "$A \wedge B \rightarrow A$". The idea that proofs are programs provides a powerful unifying principle, thus bridging the two. It allows us to use formal proofs to establish the correctness of programs, and it enables us to use the tools of Type Theory to reason about logical propositions.
In the STLC, we use Lambda abstractions to represent functions, and the application of a function to its arguments is achieved through Beta-reduction.
In the following section we will see some more examples of this correspondence to see how the proofs in Mathematics correspond to Programs In the STLC.

11 Examples

These examples further illustrate how logical propositions can be encoded as types, and corresponding proofs can be constructed as lambda terms in the STLC.
In this representation, we'll use a single lambda abstraction to directly define the function without explicitly naming the arguments.

1. **Logical Proposition:**"A implies A"; (A $\rightarrow$ A)
   STLC: $\lambda x : A.x$
   In the given example the lambda term λ **x: A. x** is a proof that, given any value of type A (represented by the bound variable x), we can return that same value, thus proving our proposition "A implies A"

2. **Logical proposition:** "A implies B implies A"; (A $\rightarrow$ (B $\rightarrow$ A))
   STLC: λx: A. λ y: B. x
   The lambda term λ**x: A.** λ**y: B. x** is a proof that, given any value x of type A, and any value y of type B, we can ignore the value y and return the value x, thus proving our original proposition.

3. Prove Equivalence between $(p \wedge q) \rightarrow r$ and $p \rightarrow (q \wedge r)$ (They are not equivalent!)

   (a) **Types in STLC:**

      i. A = p
      ii. B = q
      iii. C = r

   (b) **Proposition:** "$p \wedge q \rightarrow r$" (corresponding to (p, q) $\rightarrow$ r in the STLC)
   STLC:
   $$\lambda p : A.\lambda q : B.\lambda r : (A \wedge B) \rightarrow C$$
   It signifies the logical relationship that if both A and B are true, then C must also be true.

   (c) **Proposition:** "$p \rightarrow (q \wedge r)$" (corresponding to p $\rightarrow$ (q, r) in the STLC)
   STLC:
   $$\lambda p : A.\lambda q : B.\lambda r : A \rightarrow (B \wedge C)$$
   It signifies the logical relationship that if A is true, then both B and C must also be true.

   (d) As the final step, we will compare the given Lambda terms.

      i. The first type is $(A \rightarrow (B \rightarrow ((A \wedge B) \rightarrow C)))$
      ii. The second type is $(A \rightarrow (B \rightarrow (A \rightarrow (B \wedge C))))$

   (e) By comparing the types of both lambda terms we observe that they have different types. As per the Curry-Howard correspondence, the type of a lambda term corresponds to a logical proposition. Since both lambda terms have different types, they represent different logical propositions. Therefore, we have demonstrated that the proposition "$p \wedge q \rightarrow r$" is **not**

equivalent to "$p \to (q \wedge r)$" using the Curry-Howard correspondence in the Simply Typed Lambda Calculus.

12 Conclusion

Thus The CHC can be summarised as follows:

1. Logical Propositions $\leftrightarrow$ Types:
   (a) In logic, propositions are statements that can be assumed to be true or false. In type theory, types classify values or expressions.
   (b) Under the Curry-Howard isomorphism, each logical proposition corresponds to a type in the type system of a programming language. For example, "A $\wedge$ B" (A and B) corresponds to a product type in the Simply Typed Lambda Calculus (STLC), representing the combination of two values of types A and B.
2. Proofs $\leftrightarrow$ Programs:
   (a) In logic, proofs are structured arguments that establish the truth of propositions. In the STLC, programs are sequences of instructions that compute results.
   (b) Under the Curry-Howard isomorphism, each proof of a logical proposition corresponds to a program of the corresponding type in the STLC as mentioned in [14]. Proofs demonstrate the existence of values of certain types, while programs compute those values.
3. Logical Connectives $\leftrightarrow$ Type Constructors:
   (a) Logical connectives, such as conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and negation ($\neg$), correspond to type constructors in STLC.
   (b) For instance, conjunction corresponds to product types, disjunction corresponds to sum (or union) types, and implication corresponds to function types.

The Curry-Howard isomorphism is a correspondence between logic and programming languages, specifically between proof theory and type theory. It establishes a relationship between mathematical proofs and computer programs, linking logical propositions with types and proofs with programs. Due to this property the CHC finds applications in Type Systems, Functional Programming Languages like Haskell, proof assistants like Coq and Agda, and in Program Verification.

13 Future work: Can the CHC be extended to Classical Logic efficiently?

As shown in previous sections, the CHC establishes a connection between logical proofs and typed lambda terms as shown in section 11, showing that propositions in logic correspond to types and proofs of propositions correspond to lambda terms with those types.
As noticed in 7 we limited our scope to Intuitionistic Logic since there are some subtleties and differences between Classical logic and Intuitionistic Logic. For example, the CHC typically holds for Constructive and Intuitionistic logic, where the law of excluded middle and double negation elimination are not assumed. If the Double Negation Elimination was taken into account it could potentially lead to non-termination or infinite loops in programs.
If the CHC was not limited to Intuitionistic Logic, exploring the behavior and proving soundness and correctness of such programs would pose a significant challenge for further development.

References

[1] Thorsten Altenkirch. Lecture notes in oplss 2022 introduction to type theory, June 2022.

[2] Andrea Asperti and Giuseppe Longo. *Categories, types, and structures: an introduction to category theory for the working computer scientist*. MIT press, 1991.

[3] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[4] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[5] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[6] Ralph Loader. *Notes on simply typed lambda calculus*. University of Edinburgh, 1998.

[7] Per Martin-Löf. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982.

[8] John C Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.

[9] Frank Pfenning. Lecture notes in proof theory, June 2022.

[10] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[11] Jaap Van Oosten. *Basic category theory*. Citeseer, 1995.

[12] Philip Wadler. Proofs are programs: 19th century logic and 21st century computing. 2000.

[13] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[14] Richard Zach. The significance of curry-howard isomorphism. In *Philosophy of Logic and Mathematics. Proc. of the 41st International Ludwig Wittgenstein Symposium/Eds.: GM Mras, P. Weingartner, B. Ritter. Berlin: DeGruyter*, pages 313–326, 2019.