Curry-Howard Correspondence

Wesley Jenkins
MATH 400: Mathematical Connections
wljenkins@wm.edu

***Abstract*—This paper is about the Curry-Howard Correspondence and how it demonstrates the strong connection between abstract mathematics and concrete computer science.**

I. Introduction

The Curry-Howard correspondence is a connection between two very important fields of study: mathematics and computer science. It connects many features of mathematics and computer science in a very surprising way, which can help in providing a new perspective to both. Thus, it has important consequences in both mathematics and computer science, and it very useful both theoretically and in actual use.

II. Proofs

First comes the question of what is a proof? While all math students have encountered proofs, it can be difficult to exactly define what a proof is. The simplest, and most general, definition is that a proof is "a thing which shows something to be true." Every proof must exist within some framework, given some base assumptions; for most modern proofs, this is Zermelo-Franken set theory (with or without the axiom of choice).

But the most important thing is that proofs are normal objects, and one can create a set of them:

$$P(X) = (\text{Set of proofs of fact X})$$

And then one can use standard set theory operations to manipulate this set. The standard logical operations translate to set theory as so:

Logical Operations	Set theory
X is true	$P(X)$ is inhabited
$X \wedge Y$	$P(X) \times P(Y)$
$X \vee Y$	$P(X) \cup P(Y)$
$(X \implies Y)$	$P(X) \to P(Y)$
$(\forall (n \in N) X(n)$	$(n : N) \to P(X(n))$
$(\exists (n \in N) X(n)$	$(n : N) \times P(X(n))$

This requires the addition of the peculiar dependent product and sum types. These objects are similar to normal functions and cartesian products, except the second set depends on the value present from the first. Thus, it is a dependent type.

And here are a few examples,

- The first is a proof that if 2 is even, then 4 is even. This can be represented as $P(2 \text{ is even}) \to P(4 \text{ is even})$. The domain of this function is proofs that 2 is even, and the codomain is proofs that 4 is even. Thus, it can be thought of in the way, "given proof that 2 is even, this function can give you proof that 4 is even." Of course since 4 is trivially proven as even, this proof is trivial.
- The second is proof that all integers are real numbers. This can be represented as $(n : \mathbb{Z}) \to P(\text{n is real})$. Here, while the domain is Z, the co-domain is the union of proofs for all possible integers. This function is only valid if it is well-defined, and thus is has a proof for all possible Z, which is why it is equivalent to a for-all statement.
- The final is proof that there exists an integer less than 5. This can be represented as $(n : \mathbb{Z}) \times P(n < 5)$. Here, there must only exist a proof $P(n < 5)$ for at least one n for this object to be inhabited.

III. Programs

Next comes the question of what is a program? Yet again, it can be difficult to nail down an exact definition for this, especially one that everyone will agree to. However, a suitable response is that a program is a sequence of steps to perform some computation. There will be some form of input and some form of output.

But the exact steps can vary, because there are multiple abstract models of computation. There are multiple Turing machines like standard Turing machines, random access Turing machines, quantum Turing machines, etc. All Turing machines can solve the same set of problems, called Turing computable, but the complexity of problems can vary (For example, on a standard Turing machine, prime factorization runs in nearly exponential time, while it can run in linear time on a quantum Turing machine). There are also multiple different forms of lambda calculus, which were proven by Turing to be equivalent to Turing machines as well.

But moving away from the theoretical, there are many programming languages which are used to write actual programs to execute on physical computers. These programming languages are said to be Turing complete if they can theoretically perform any calculation that a Turing machine can, except they obviously are going to be limited by physical resources like memory and time constraints unlike a theoretical machine.

IV. Type Theory

As said before, a program can be thought of as a sequence of steps to perform some computation. These steps must operate on some values, and these values are commonly given types. When manipulating values, its type details how to perform said operations. For example, adding two integers must be performed differently from adding two vectors; the value's types can explain exactly what addition means for that specific

value, and how to implement it. But types are also equivalent to sets containing all possible values of that type.

Programming languages vary widely in their usage of types, from being completely untyped to being statically typed, meaning all types must be known at compile time. A statically-typed language is equipped with a type-checker which verifies that all statements are of the correct expected type. This is equivalent to proving that a statement evaluates to a given type.

As an example, Python is dynamically typed. Every object in Python has a type, but this type doesn't have to be known beforehand, and variables can change type at any time. This can be contrasted with C++, where all variables must have their types given, which cannot change during the course of the program.

V. Proof Types

As said before, $P(X)$ is the set of proofs of some fact X. And since this set is equivalent to some type in a program, one can write a program to simulate the manipulations presented earlier. And this isn't just an interesting way of writing proofs: the type checker becomes equivalent to a proof checker. Because by proving that an expression evaluates to the expected type (which represents a proof), that proves that it actually constructs a proof.

But this requires that the expression actually evaluate at all. There are multiple problems with this.

First of all, many programming languages have "escape routes." That is, it's possible for an expression to never actually evaluate to anything. For example, a C expression could call the exit function, and then the program would simply end! Alternatively, a program could be infinitely recursive, meaning it will never end (Until it eventually crashes).

Therefore, for this to work, there need to be some restrictions on the program itself. Every expression must finish evaluating: it cannot secretly exit the program and cannot infinitely recurse. By making these restrictions, it now becomes possible to provide guarantees on the types of expressions, and thus the generation of proofs. But what are the consequences of these restrictions exactly? That will be discussed later.

There have been multiple attempts at creating languages which obey these rules. A good example of this is Agda, which has a complex enough type system to allow the creation and verification of real proofs. It also disallows any kinds of escape routes like ending the program early or recursing infinitely. However, due to its complexity, it has had problems finding usage outside of the math community.

VI. Curry-Howard Correspondence

And thus, it becomes clear the connection between proofs and programs. A proof which relies on some assumptions and produces proof of some fact can be embedded inside of a program which relies on some input and produces some output. And type checking becomes equivalent to proof checking under this embedding.

But the Curry-Howard correspondence goes further than this, stating that proof systems and systems of computation are isomorphic to one another. They are different ways of describing the same set of rules. And all valid logical arguments can be turned into a runnable program, while all runnable programs can be turned into logical arguments.

This correspondence connects computation and logic and also unifies certain parts of mathematics and foundational computer science. It turns abstract logical arguments into actualizable programs, which has potential philosophical considerations as well.

VII. Use in Mathematics

As stated before, there have been multiple attempts at creating real programming languages which obey the rules required to make the Curry-Howard correspondence work. However, one of the rules mentioned before is that expressions cannot contain any kind of "escape route." That is, it must be able to prove that given the specific proof inputs, the expression will successfully terminate. Of course, this is in violation of Turing's halting problem, which states that it is impossible to determine for all programs whether they will terminate or not.

Therefore, there must exist programs which will terminate and yet will not be accepted by the type checker. And thus, there must be facts which cannot be proven, because any program which proves them will not be accepted as terminating.

This sounds remarkably similar to the common problem in mathematics, Gödel's incompleteness theorem. And indeed they are intimately related. If all programs could be proven as terminating or not, then this would allow for the proving of all facts by the correspondence, thus violating Gödel's incompleteness theorem. And thus, the halting problem implies Gödel's incompleteness theorem.

Therefore, the correspondence has a lot of use in understanding a new perspective on either problem. For many, Gödel's incompleteness theorem is complicated, yet the halting problem can be much easier to explain. You can always think that the halting problem is what's truly at the heart of Gödel's incompleteness theorem: The reason some facts can't be proven is because it's impossible to write a computer program proving it that will always terminate!

VIII. Use in Programming

But it also has a lot of use in programming as well.

By writing proofs into programs, one can write programs with guaranteed behavior. Once a program's behavior has been proven, the program itself can be extracted, which is guaranteed to be correct (Assuming there are no bugs in the compilation process).

This can be utilized in a variety of ways. For example, Microsoft is currently using their proof language F* to write encryption and decryption functions that have guaranteed behavior, where proving the security of implementations is usually extremely difficult. Bugs in these kinds of software are scarily common and can undermine the security of the

entire encryption algorithm, and therefore there is a lot of use in having proven versions of these primitives.

Yet, most user programs are unlikely to be proven due to the sheer amount of effort required in writing proven code, which is why it is best reserved for small extremely-important software libraries or programs. And hopefully, as this use case matures, more and more of these sensitive softwares will be proven correct, which would drastically improve computer and network security globally. All that's needed is more people willing to write the software that's required.

IX. Conclusion

And so, the Curry-Howard correspondence is an extremely interesting and useful correspondence. It has both theoretical and actual use cases, from a better understanding of abstract mathematics to improving various computer programs' security.

But it requires a lot more study. The best method of converting between programs and proofs, given how complicated Turing machines programs can be, is still uncertain. There are multiple different methods currently in use, many borrowing concepts from category theory like monads and functors, but the most concise and correct method is still uncertain.

And there is still work to be done on creating the complex type systems required to embed all required mathematical logic in a succinct way. Attempts like Agda work well, but are very limited in how they determine whether a program terminates or not, which means a lot of facts can't be easily proven.

But hopefully this will improve; after all, applications like this are still in their infancy and are evolving quickly.