The Curry-Howard Correspondance

WESLEY JENKINS

Proofs

- Proofs are common to all of us.
- What is a proof?
  - A "thing" that somehow shows that a conclusion is true.
- Can create a set of proofs that prove some fact:

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

- Not a set of all proofs, just a set of all proofs for X specifically.

Proofs and Set Theory

Proof ideas need to be rewritten into set theoretic notation

- X is true if and only if P(X) is inhabited (has at least one element).
- $P(X \text{ and } Y) \cong P(X) \times P(Y)$
- $P(X \text{ or } Y) \cong P(X) \cup P(Y)$
- $P(X \text{ implies } Y) \cong P(X) \longrightarrow P(Y)$
  - Given proof of X, this function will return proof of Y.
  - The function must be well-defined.
- $P\big(\forall (n \in N)\, X(n)\big) \cong (n : N) \longrightarrow P(X(n))$
  - Confusing notation
  - Dependent product type
- $P\big(\exists (n \in N) X(n)\big) \cong (n{:}\, N) \times P(X(n))$
  - Dependent sum type
  - Confusing because it uses a product

Examples

- Proof that if 2 is even, then 4 is also even.
  - $\phi : P(2 \text{ is even}) \longrightarrow P(4 \text{ is even})$
  - Given proof that 2 is even, it will return proof that 4 is even.
- Proof that all integers are real numbers.
  - $\phi : (n : \mathbb{Z}) \longrightarrow P(n \text{ is real})$
  - Given a value like 3, $\phi(3) : P(3 \text{ is real})$
  - Codomain is the union of all $P(n \text{ is real})$.
  - Only true if such a function can be constructed!
- Proof that there exists an integer less than 5
  - $\phi : (n : \mathbb{Z}) \times P(n < 5)$
  - Second set depends on a value from the first.

Programming

- Everyone is also familiar with programming and programs.
- What is a program?
  - A sequence of steps to perform some computation.
- There are many abstract models of computation
  - Turing machines, lambda calculus, etc.
- And many programming languages, which are considered "turing-complete,"
  - C++, Python, Java, etc.

Language Typing

- Values in most programming languages have "types."
  - Equivalent to sets of objects.
- Statically-typed languages make requirements on the type of a variable.
  - Equipped with a type checker, which ensures the types are correct.
  - Cannot use a value where it is unexpected, e.g. real number in place of integer

Python: Dynamic typing

```
def addValues(x, y):
    z = x + y
    return z
```

C++: Static typing

```
int addValues(int x, int y) {
    int z = x + y;
    return z;
}
```

Proof Types

- Since P(X) is a set, it could also be used as a type in a programming language.
- The program could then manipulate these "proof objects" to generate new proof objects.
- The static type checker could verify that the proof is correct.

$P(2\ is\ even) \longrightarrow P(4\ is\ even)$

```
proof_4_is_even proveEven(proof_2_is_even proof) {
    ???
}
```

- But will it work?
  - No. Why?

Type Systems

- C++ isn't sufficient
  - Not a complex enough type system.
  - Has a lot of "outs" like exceptions and the "exit" function which ends the program.
  - Functions aren't pure (So, not even real mathematical functions).
  - Not actually strictly typed! Allows casting!
- Need a more suitable language.

Agda

- Agda is designed for this purpose.
  - Much more complicated type system.
    - Hindley-Milner type system with support for dependent types.
  - All functions are total and pure, meaning there are no escape routes like in C++.
  - Can't create values of any type out of thin air.
- Functional language closely related to Haskell.

```
proveEven : proof_2_even -> proof_4_even
proveEven proof = ???
```

Automated Proof Checking

- Agda programs themselves act as proofs.
- The Agda compiler checks the program.
  - If the Agda compiler accepts the program, then it must be a valid proof.
    - (Unless there is a bug in the Agda compiler)

```
data Nat : Set where
    zero : Nat
    suc : Nat -> Nat

data _==_ : Nat -> Nat -> Set where
    refl : (n : Nat) -> n == n

eqTrans : {n m l : Nat} -> n == m -> m == l -> n == l
eqTrans (refl x) (refl x) = refl x
```

Proven Programs

- Agda programs, after being verified, can be exported to Haskell.
- These programs have guaranteed behavior, because their behavior has been proven already.
  - Can be used in critical applications where the validity of code is crucial: nuclear reactors, space probes, etc.
  - Assuming there are no hardware faults, the program cannot contain bugs.
- Can provide certain security guarantees at compile time rather than runtime.
  - Buffer overrun checks, array out of bounds checks, etc.
- So why aren't all programs proven?
  - Because it's very difficult and time-consuming to write a proven program.

Curry-Howard Correspondence

- So, formal logic can be embedded inside of programming.
- And type checking can then be used to prove such logic is valid.
- The Curry-Howard correspondence states that proof systems and systems of computation are isomorphic to one another.
  - They describe the same set of rules in a different way.
- And so, this ties together computation and logic.
  - Every valid logical argument can be turned into a runnable program.
  - And every runnable program can be turned into a logical argument.

And its Consequences

- There are a lot of consequences of this correspondence.
  - Unifies certain parts of mathematics and foundational computer science.
  - Gives new understandings of both proofs and programs.
  - And creates new methods of creating proofs as programs.
  - Potential philosophical interpretations on the nature of logic and computation.

Questions?