Notes on Proofs as Programs

Arthur Ryman

1 Introduction

The fascinating correspondence between proofs and programs has become a central aspect of modern computer science. These notes explain the correspondence.

The usual approach to formal logic is that first one is given some formal language in which one can make statements. The language has a grammar that lets us use logical connectives to combine statements into new statements. The specifics of the language vary depending on the desired expressive power. For example, propositional logic involves simple true-false statements, e.g. Alice loves Bob, whereas predicate logic allows one to make complex statements about objects drawn from some, possibly infinite, universe of discourse, e.g. Alice loves everyone.

It is further supposed that statements are either true or false in any given situation. A situation is also called an interpretation. For example Alice may or may not love Bob, but we make the simplifying assumption that exactly one of the two alternatives hold. A statement that is either true or false is called a proposition.

One is then given rules of sound deduction that specify how to derive new statements from given statements. The given statements are called assumptions, antecedents, premises, or hypotheses. The derived statement is called the conclusion or consequence. The rules are sound in the sense that if the assumptions are true then the conclusion is true. A chain of applications of the sound rules of deduction is called a proof.

Historically, proofs were not treated as formal mathematics objects. The correspondence of proofs with programs begins by treating proofs as mathematical objects.

Now suppose that proofs are mathematical objects and it makes sense to talk about the set of all proofs and its various subsets. Given a proposition P we can consider the set of all proofs of P. We refer to the set of all proofs of a given proposition as the type associated with the proposition. This is the *propositions as types* viewpoint. Suppose that x is a proof of P. We denote this as follows:

$$x : P$$

The fascinating thing about this viewpoint is that the logical connectives on propositions correspond to familiar constructors on the corresponding types!

1.1 Falsity

Let F denote a proposition that is false in all situations. Since F is never true, it has no proof. The type corresponding to F is therefore the empty set.

$$F \sim \emptyset$$

I am tempted to say that we have the following entailment:

$$x : F \vdash y : P$$

1.2 Truth

Let T be a proposition that is true in all situations. In a sense, T is self-evident so it requires no proof. However, we want to have at least one proof for each true proposition so we'll introduce the symbol I for this purpose. We only need one proof of T so we can regard the type of T as the singleton set whose unique member is I.

$$T \sim \{I\}$$

We have the following entailment:

$$\vdash I : T$$

1.3 Conjunction

Let P and Q be propositions and consider their conjunction $P \wedge Q$. What constitutes a proof of the conjunction? Clearly, if we can prove each proposition then we have proved the conjunction. Let $x : P$ and $y : Q$. It is natural to regard the ordered pair (x, y) as a proof of $P \wedge Q$. Mathematically, an ordered pair is an element of the Cartesian product of two sets. Therefore, the type of a conjunction is the Cartesian product of the types of the conjuncts:

$$P \wedge Q \sim P \times Q$$

We have the following entailment:

$$x : P, y : Q \vdash (x, y) : P \wedge Q$$

$P \times Q$ corresponds to a *record* or *struct* data type in computer programming languages.

As a sanity check, let's see if our correspondences for falsity, truth, and conjunction behave sensibly. For example, $F \wedge P \Leftrightarrow F$ so we should have $F \times P \cong F$.

$$\begin{aligned} F \wedge P &\sim F \times P \\ &= \emptyset \times P \\ &\cong \emptyset \\ &\sim F \end{aligned}$$

Similarly $T \wedge P \Leftrightarrow P$ so we should have $T \times P \cong P$.

$$\begin{aligned} T \times P &\sim T \times P \\ &= \{I\} \times P \\ &\cong P \end{aligned}$$

This last result confirms that it makes sense to require the type corresponding to T to be a singleton.

1.4 Disjunction

Following similar reasoning, we have a proof of the disjunction $P \vee Q$ if we are given a proof of either P or Q. Mathematically, the set of elements from P and Q, along with an indicator of which set they came from, is called their disjoint union or sum and is denoted $P + Q$. One can think of the sum $P + Q$ as $\{1\} \times P \cup \{2\} \times Q$.

$$P \vee Q \sim P + Q$$

We have the following entailments:

$$\begin{aligned} x : P &\vdash (1, x) : P \vee Q \\ y : Q &\vdash (2, y) : P \vee Q \end{aligned}$$

$P + Q$ corresponds to a *discriminated union* or *variant record* data type in computer programming languages.

Now consider $T \vee P \equiv P$. Does this carry over to the corresponding types?

$$\begin{aligned} T \vee P &\sim T + P \\ &= \{I\} + P \\ &\neq P \end{aligned}$$

Looks like the correspondence is not exact. Perhaps we need to define a notion of equivalence of proofs. Perhaps we can have more than one proof of T but regard them all as equivalent. Time to watch more videos.

1.5 Implication

If we have proofs of P and $P \Rightarrow Q$ then we have a proof of Q. It is therefore natural to regard a proof of $P \Rightarrow Q$ as a program or mapping or function that takes as input any proof of P and produces as output a proof of Q.

$$P \Rightarrow Q \sim P \to Q$$

We have the following entailment:

$$x : P, f : P \Rightarrow Q \vdash f(x) : Q$$

1.6 Biconditional

The biconditional logical connective is defined in terms of implication and conjunction, so we can compute its corresponding type.

$$\begin{aligned} P \Leftrightarrow Q &\equiv (P \implies Q) \wedge (Q \implies P) \\ &\sim (P \to Q) \times (Q \to P) \end{aligned}$$

1.7 Negation