Proofs

Arthur Ryman, arthur.ryman@gmail.com

December 22, 2018

Abstract

This article is a Z Notation specification for proofs and proof checking. It has been type checked by *f*UZZ. The definitions that appear here are taken from Lemmon's book, *Beginning Logic*. The purpose of this specification is to guide the development of a proof checker aimed at Z specifications

1 Introduction

For a long time I have thought that it would be extremely useful to be able to write formal proofs concerning the mathematical objects defined in Z specifications. There are some very mature proof assistants available. I know something about Coq, but unfortunately it's style of proof is very different from that one finds in mathematical papers.

A Coq proof consists of applications of so-called *tactics*. Each tactic represents a higher level aggregate of deductions. This makes sense for a proof assistant since its job is to help the user discover proofs. Tactics are like macros and as such they alleviate the user from much low-level tedium. However, the analog of a macro in normal mathematical writing is a lemma. Perhaps tactics are more useful for proving formal properties of programming languages where the mathematical objects of interest are complex, but finite, recursive structures.

While a proof assistant might make sense is some contexts, proper development of a mathematical paper consists of a gradual introduction of concepts and lemmas leading to the main results. The proofs should, in some sense, write themselves. The focus of a mathematical paper should be on explanation and clarity. The proofs should be easy to read. I'd therefore really like something that would let me write and check natural looking proofs.

In contrast to Coq, the style of proof presented by Lemmon is very clear and explicit. However, the task of checking such a proof could easily be delegated to a program, much the same way that *f*UZZ type checks Z. In fact Lemmon makes that point that, although there is no mechanical way to discover proofs, they can be mechanically checked.

I believe that the kernel of a proof checker could be very small. It is basically an engine driven by a set of deduction rules. The engine simply needs to check that each

deduction rule gets applied correctly. Even if it turns out that writing such an engine is too much work, the exercise of developing at least a simple version should give me a greater appreciation of tools like Coq and enable me to use them more productively.

My plan of attack is to formalize the concept of proof as described by Lemmon, starting with the propositional calculus, and then move on to predicate calculus.

2 Propositions

The *propositional calculus* defines a set of formal *statements* or *propositions* without being concerned about the subject matter described by those statements. The only restriction on these statements is that, in any given context, they possess a *truth value* of either *true* or *false*.

A proposition is also referred to as a *well-formed formula* or *wff* for short. This terminology stems from the traditional development of the propositional calculus in terms of a language of sentences over an alphabet with rules that prescribe when a given sentence is *well-formed*. However, here we will dispense with the language viewpoint, and its associated parsing issues, and move directly to the end result of parsing, namely the creation of *abstract syntax trees* or *ASTs* for short. This approach corresponds to Lemmon's Chapter 1. He returns to the issue of formal languages in Chapter 2.

2.1 *Prop*

Z notation has a convenient mechanism for specifying the structure of ASTs, namely that of *free types*. My first impulse was to use that mechanism to define propositions. However, the problem with free types is that they are it closed in the sense that all ways of constructing members of a free type must be specified when the free type is defined. Lemmon's book gradually introduces ways of constructing propositions, so in the interest of following his development of the subject as closely as possible, I won't define propositions that way.

On closer examination of free types in Z, it will be observed that they are merely syntactic sugar for introducing a new given set along with an exhaustive set of constructors for its elements. The constraint expressing the condition that the constructors are exhaustive is equivalent to saying that the domains of the constructors partition the set of propositions.

Therefore, given any subset of constructors, one can define the corresponding set of propositions that can be constructed from them. This allows new constructors to be gradually introduced. I'll take that approach.

Let *Prop* denote the set of all propositions.

$$[Prop]$$

2.2 *PropVar*

Separating of the form of a statement from its content with respect to any given subject matter is accomplished by the use of *propositional variables*. A propositional variable stands for an arbitrary statement that is either true or false. Let *PropVar* denote the set of all propositional variables.

$$PropVar : \mathbb{P}\, Prop$$

2.3 P \propP, Q \propQ, R \propR, S \propS, and T \propT

Traditionally, arbitrary statements are represented by single letters such as P, Q, R, S, and T which represent distinct propositions.

$$\mathrm{P}, \mathrm{Q}, \mathrm{R}, \mathrm{S}, \mathrm{T} : PropVar$$
$$\mathsf{disjoint}\ \langle \{\mathrm{P}\}, \{\mathrm{Q}\}, \{\mathrm{R}\}, \{\mathrm{S}\}, \{\mathrm{T}\} \rangle$$

2.4 *PropLetter*

Let *PropLetter* denote the set of all proposition letters.

$$PropLetter == \{\mathrm{P}, \mathrm{Q}, \mathrm{R}, \mathrm{S}, \mathrm{T}\}$$

2.5 ′ \propPrime

Typical propositions contain a small number of distinct statements, in which case the letters can be used. If more statements occur then the letters are decorated with one or more primes, e.g. P', Q''. Appending a prime to a propositional variable is an injection from *PropVar* to *PropVar*.

$$\_' : PropVar \rightarrowtail PropVar$$

Example. P' *and* Q'' *are propositional variables.*

$$\mathrm{P}' \in PropVar$$

$$\mathrm{Q}'' \in PropVar$$

A propositional variable is either a letter or is primed.

$$\langle PropLetter, \mathrm{ran}(\_') \rangle\ \mathsf{partition}\ PropVar$$

2.6 $-$ `\notProp` and *Negation*

Let A be a proposition. Let $-A$ denote the *negation* of A.

$$| \quad - : \mathit{Prop} \rightarrowtail \mathit{Prop}$$

Example. $-\mathrm{P}$ *is a negation.*

$$-\mathrm{P} \in \mathit{Prop}$$

Let *Negation* denote the set of all negations.

$$\mathit{Negation} == \operatorname{ran} -$$

2.7 $\&$ `\andProp` and *Conjunction*

Let A and B be propositions. Let $A \,\&\, B$ denote the *conjunction* of A and B. The conjunction $A \,\&\, B$ is said to have A and B as its *conjuncts*.

$$| \quad \_ \,\&\, \_ : \mathit{Prop} \times \mathit{Prop} \rightarrowtail \mathit{Prop}$$

Let *Conjunction* denote the set of all conjunctions.

$$\mathit{Conjunction} == \operatorname{ran}(\_ \,\&\, \_)$$

2.8 v `\orProp` and *Disjunction*

Let A and B be propositions. Let $A \,\mathrm{v}\, B$ denote the *disjunction* of A and B. The disjunction $A \,\mathrm{v}\, B$ is said to have A and B as its *disjuncts*.

$$| \quad \_ \,\mathrm{v}\, \_ : \mathit{Prop} \times \mathit{Prop} \rightarrowtail \mathit{Prop}$$

Let *Disjunction* denote the set of all disjunctions.

$$\mathit{Disjunction} == \operatorname{ran}(\_ \,\mathrm{v}\, \_)$$

2.9 $\longrightarrow$ `\impliesProp` and *Conditional*

Let A and B be propositions. Let $A \longrightarrow B$ denote the *conditional* of A and B. The conditional $A \longrightarrow B$ is said to have A as its *antecedent* and B as its *consequent*.

$$| \quad \_ \longrightarrow \_ : \mathit{Prop} \times \mathit{Prop} \rightarrowtail \mathit{Prop}$$

Let *Conditional* denote the set of all conditionals.

$$\mathit{Conditional} == \operatorname{ran}(\_ \longrightarrow \_)$$

2.10 $\leftrightarrow$ `\equivProp` and *Biconditional*

Let A and B be propositions. Let $A \leftrightarrow B$ denote the *biconditional* of A and B.

$$\_ \leftrightarrow \_ : Prop \times Prop \rightarrowtail Prop$$

$$\forall A, B : Prop \bullet A \leftrightarrow B = (A \longrightarrow B) \,\&\, (B \longrightarrow A)$$

Let *Biconditional* denote the set of all biconditionals.

$$Biconditional == \mathrm{ran}(\_ \leftrightarrow \_)$$

3 Proofs

Lemmon has a nice way of presenting proofs. Here's an example.

1 $\mathrm{P} \longrightarrow \mathrm{Q}, \mathrm{P} \vdash \mathrm{Q}$

1	(1)	$\mathrm{P} \longrightarrow \mathrm{Q}$	A
2	(2)	P	A
1,2	(3)	Q	1,2 MPP

A proof consists of a *sequent* to be proved and a finite sequence of one or more *lines*. The lines are labelled by consecutive natural numbers, starting at 1.

The sequent contains a, possible empty, sequence of assumptions followed by a conclusion. The sequence of lines that follow show how the conclusion is derived from the assumptions using various rules of derivation.

Each line of the argument contains an inference.

Each line contains a proposition and the application of the *proof rule* used to add it to the proof. A proof rule is either an *assumption* or a *derivation*. The *rule of assumption* allows the addition of any proposition. The *rules of derivation* allow the addition of a *conclusion* derived from *premises* that have been previously added. A premise is therefore either an assumption or the conclusion of a previous deduction. Every line of the proof can therefore be traced back to a finite, possibly empty, set of assumptions upon which it ultimately *depends*.

3.1 *Sequent* and $\vdash$ `\sequent`

A *sequent* consists of a sequence of assumptions and a conclusion. Let *Sequent* denote the set of all sequents.

Sequent

$$assumptions : \mathrm{seq}\, Prop$$
$$conclusion : Prop$$

Let $\vdash$ denote the binary operator that takes assumptions and a conclusion and forms a sequent.

$$\_ \vdash \_ : \operatorname{seq} Prop \times Prop \longrightarrow Sequent$$

$$\forall\, Sequent \bullet \\ \quad assumptions \vdash conclusion = \theta Sequent$$

3.2 *RuleOfDerivation*

Let *RuleOfDerivation* denote the set of applications of rules of derviation. In the preceeding example, A and MPP are names of rules of derivation.

$$[RuleOfDerivation]$$

3.3 *Deduction*

A *deduction* consists of a proposition, an application of a rule of derivation that justifies the proposition, and a finite, possibly empty, set of assumptions on which the proposition depends. Let *Deduction* denote the set of all deduction.

$$\begin{array}{|l} \textit{Deduction} \\ \hline assumptions : \mathbb{F}\, \mathbb{N}_1 \\ prop : Prop \\ rule : RuleOfDerivation \\ \hline \end{array}$$

3.4 *DeductionTuple*

Let *DeductionTuple* denote the tuple formed by the components of a deduction.

$$DeductionTuple == \mathbb{F}\, \mathbb{N}_1 \times Prop \times RuleOfDerivation$$

3.5 tuple `\deductionTuple`

Let tuple denote the function that maps a deduction to its tuple.

$$\mathrm{tuple} == (\lambda\, Deduction \bullet (assumptions, prop, rule))$$

Remark. *The mapping from deductions to tuples is a bijection.*

$$\mathrm{tuple} \in Deduction \rightarrowtail\!\!\!\!\!\twoheadrightarrow DeductionTuple$$

3.6 prop `\deductionProp`

Let prop denote the function that maps a deduction tuple to its *prop* component.

$$\text{prop} == \{\, \mathit{Deduction} \bullet (\mathit{assumptions}, \mathit{prop}, \mathit{rule}) \mapsto \mathit{prop} \,\}$$

3.7 *Argument*

An *argument* is a finite sequence of deductions. Let *Argument* denote the set of all arguments.

$$\mathit{Argument} == \text{seq}\, \mathit{DeductionTuple}$$

3.8 *ArgumentDeduction*

When determining the soundness of an argument, we need to check of the soundness of each deduction. Each deduction within an argument is uniquely identified by its line number. Let *ArgumentDeduction* denote the set of all arguments and valid line numbers in them.

ArgumentDeduction

$$\begin{array}{l} \mathit{argument} : \mathit{Argument} \\ \mathit{lineNumber} : \mathbb{N}_1 \\ \mathit{Deduction} \\ \hline \mathit{lineNumber} \in \text{dom}\, \mathit{argument} \\ \mathit{argument}(\mathit{lineNumber}) = (\mathit{assumptions}, \mathit{prop}, \mathit{rule}) \end{array}$$

- The deduction is identified by its line number within the argument.
- The deduction has a tuple of components.

3.9 *SoundDeduction*

A deduction within an argument is sound if it adheres to one of the rules of derivation. The rules of derivation will be described below. Let *SoundDeduction* denote the set of all sound deductions.

$$\mathit{SoundDeduction} : \mathbb{P}\, \mathit{ArgumentDeduction}$$

3.10 *SoundArgument*

An argument is sound precisely when all of its deductions are sound. Let *SoundArgument* denote the set of all sound arguments.

$$\mathit{SoundArgument} : \mathbb{P}\, \mathit{Argument}$$

3.11 sequent `\deductionSequent`

Every deduction in an argument defines a sequent. Let sequent denote the mapping from deductions to sequents.

$$\text{sequent} : \mathit{ArgumentDeduction} \longrightarrow \mathit{Sequent}$$

$$\forall\, \mathit{ArgumentDeduction} \bullet \\ \quad \text{sequent}(\theta \mathit{ArgumentDeduction}) = \\ \quad\quad \mathit{assumptions} \upharpoonright (\mathit{argument} \fatsemi \text{prop}) \vdash \mathit{prop}$$

3.12 *ArgumentProvesSequent*

A sound argument proves a sequent if the final deduction of the argument defines the given sequent. Let *ArgumentProvesSequent* denote this situation.

ArgumentProvesSequent

$$\mathit{ArgumentDeduction}$$
$$s : \mathit{Sequent}$$

$$\mathit{argument} \in \mathit{SoundArgument}$$
$$\mathit{lineNumber} = \#\mathit{argument}$$
$$s = \text{sequent}(\theta \mathit{ArgumentDeduction})$$

3.13 *Proof*

A proof is a sequent and a sound argument whose last deduction proves the sequent. Let *Proof* denote the set of proofs.

$$\mathit{Proof} : \mathit{Sequent} \leftrightarrow \mathit{SoundArgument}$$

$$\mathit{Proof} = \{\, \mathit{ArgumentDeduction} \mid \\ \quad (\mathit{lineNumber} = \#\mathit{argument} \land \\ \quad \mathit{argument} \in \mathit{SoundArgument}) \bullet \\ \quad\quad \text{sequent}(\theta \mathit{ArgumentDeduction}) \mapsto \mathit{argument} \,\}$$

3.14 A `\ruleA`

Let A denote the rule of assumption. The rule of assumption is used to introduce an arbitrary assumption at any point in the argument.

$$\text{A} : \mathit{RuleOfDerivation}$$

3.15 *RuleOfAssumptionDetail*

A deduction uses the rule of assumption A in an argument is sound if it introduces some arbitrary proposition P and it depends only on itself. Let *RuleOfAssumptionDetail* denote the set of all deductions, along with their details, that use the rule of assumption soundly.

RuleOfAssumptionDetail

$$\begin{array}{l} ArgumentDeduction \\ P : Prop \\ \hline assumptions = \{lineNumber\} \\ prop = P \\ rule = \text{A} \end{array}$$

- The deduction depends only on itself.
- The deduction introduces an arbitrary proposition P.
- The deduction is justified by the rule of assumption A.

3.16 *RuleOfAssumption*

Let *RuleOfAssumption* denote the set of all deductions that use the rule of assumption with the detail hidden.

$$RuleOfAssumption \mathrel{\widehat=} RuleOfAssumptionDetail \upharpoonright ArgumentDeduction$$

The rule of assumption is sound.

$$RuleOfAssumption \subset SoundDeduction$$

3.17 MPP `\ruleMPP`

Let MPP denote the MPP rule. The MPP rule is used to deduce the consequent of a conditional given its antecedent. The rule specifies the conditional and antecedent as its premises.

$$\text{MPP} : \mathbb{N}_1 \times \mathbb{N}_1 \longrightarrow RuleOfDerivation$$

3.18 *RuleOfMPPDetail*

A deduction that uses the rule MPP(i, j) in an argument is sound if i and j are lines that precede it, the proposition on line i is an implication, the proposition on line j is the antecedent of the implication, the proposition of the proof line is the consequent of the implication, and the proof line's assumptions is the union of the assumptions of lines i and j. Let *RuleOfMPPDetail* denote the set of all deductions, along with their detail, that use the rule of MPP soundly.

RuleOfMPPDetail

$$
\begin{array}{l}
ArgumentDeduction \\
i, j : \mathbb{N}_1 \\
Deduction_1 \\
Deduction_2 \\
P, Q : Prop \\
\hline
rule = \text{MPP}(i, j) \\
i < lineNumber \land j < lineNumber \\
argument(i) = (assumptions_1, P \longrightarrow Q, rule_1) \\
argument(j) = (assumptions_2, P, rule_2) \\
assumptions = assumptions_1 \cup assumptions_2 \\
prop = Q
\end{array}
$$

- The rule of derivation is MPP which specifies the premise line numbers i of a conditional and j of its antecedent.
- The line numbers of the conditional and its antecedent must precede the deduction in the argument.
- Line number i contains a conditional of the form $P \longrightarrow Q$.
- Line number j contains the antecedent P of the conditional.
- The deduction depends on the union of the assumptions that the premises depend on.
- The deduction introduces the consequent Q as the conclusion of the premises.

3.19 *RuleOfMPP*

Let *RuleOfMPP* denote the set of all deductions that use the rule of MPP with the detail hidden.

$RuleOfMPP \mathrel{\widehat{=}} RuleOfMPPDetail \upharpoonright ArgumentDeduction$

The rule of MPP is sound.

$RuleOfMPP \subset SoundDeduction$

3.20 $argument_1$

Let $argument_1$ denote the argument defined by proof **1**.

$$argument_1 == \\ \quad \{1 \mapsto (\{1\}, \mathrm{P} \longrightarrow \mathrm{Q}, \mathrm{A}), \\ \quad 2 \mapsto (\{2\}, \mathrm{P}, \mathrm{A}), \\ \quad 3 \mapsto (\{1, 2\}, \mathrm{Q}, \mathrm{MPP}(1, 2))\}$$

Example. *$argument_1$ is an argument.*

$argument_1 \in Argument$

Example. *Line 3 of proof* **1** *corresponds to the following deduction tuple.*

$(\{1, 2\}, \mathrm{Q}, \mathrm{MPP}(1, 2)) \in DeductionTuple$

3.21 $proof_2$

Here is Lemmon's proof **2**.

2 $-\mathrm{Q} \longrightarrow (-\mathrm{P} \longrightarrow \mathrm{Q}), -\mathrm{Q} \vdash -\mathrm{P} \longrightarrow \mathrm{Q}$

1	(1)	$-\mathrm{Q} \longrightarrow (-\mathrm{P} \longrightarrow \mathrm{Q})$	A
2	(2)	$-\mathrm{Q}$	A
1,2	(3)	$-\mathrm{P} \longrightarrow \mathrm{Q}$	1,2 MPP

Let $sequent_2$ denote the sequent of this proof.

$$sequent_2 == \langle -\mathrm{Q} \longrightarrow (-\mathrm{P} \longrightarrow \mathrm{Q}), -\mathrm{Q} \rangle \vdash -\mathrm{P} \longrightarrow \mathrm{Q}$$

Let $argument_2$ denote the argument of this proof.

$$argument_2 == \\ \quad \{1 \mapsto (\{1\}, -\mathrm{Q} \longrightarrow (-\mathrm{P} \longrightarrow \mathrm{Q}), \mathrm{A}), \\ \quad 2 \mapsto (\{2\}, -\mathrm{Q}, \mathrm{A}), \\ \quad 3 \mapsto (\{1, 2\}, -\mathrm{P} \longrightarrow \mathrm{Q}, \mathrm{MPP}(1, 2))\}$$

Let $proof_2$ denote this proof.

$proof_2 == (sequent_2, argument_2)$

Example. *$proof_2$ is a proof.*

$proof_2 \in Proof$

4 The Curry-Howard Correspondence

The *Curry-Howard Correspondence* is an interpretation of propositions and proofs in terms of types. To each proposition there correspondences a type. The inhabitants of the type that corresponds to a proposition are proofs of that proposition. The logical connectives that build up propositions correspond to type constructors.

To illustrate the correspondence, consider the implication logical connective $P \Rightarrow Q$. It corresponds to the type constructor $P \longrightarrow Q$. A proof f of $P \Rightarrow Q$ corresponds to a function that maps any proof of P to some proof of Q.

Of course, $P \longrightarrow Q$ is not a type in Z. It is a subset of the type $\mathbb{P}(P \times Q)$. Nevertheless, we'll press on and pretend that it is a Z type for now.

The rules of derivation correspond to the construction of a proof from assumptions. For example the rule of MPP corresponds to function application. Consider $Proof_1$. It corresponds to the following.

CHProof 1[P, Q]

$f : P \longrightarrow Q$
$x : P$
$y : Q$

$y = f(x)$