SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Formalisation of Polynomials in Cubical Type Theory Using Cubical Agda

av

Carl Åkerman Rydbeck

2022 - No K13

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Formalisation of Polynomials in Cubical Type Theory Using Cubical Agda

Carl Åkerman Rydbeck

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Anders Mörtberg

2022

Formalisation of Polynomials in Cubical Type Theory Using Cubical Agda

Carl Åkerman Rydbeck

May 18, 2022

Abstract

We formalise polynomials over commutative rings in cubical type theory using Cubical Agda as proof assistant. On the basis of a formalisation of polynomials as number sequences with only a finite number of non-zero values, we use higher inductive types to formulate a list-based definition using two point constructors and two path constructors. The combinatorial explosion in proofs leads us to a redefinition: One of the path constructors is discarded, and instead we formulate a separate function-based definition. We prove equivalence of these distinct definitions, and use the function-based definition to provide a witness for the discarded path constructor. The list-based definition is then used in combination with this witness to prove that the resulting structure is itself a commutative ring.

Acknowledgements

I would like to thank Anders Mörtberg for his enthusiastic supervision and continuous feedback during the work with this thesis. Additional thanks to Evan Cavallo who in collaboration with Anders Mörtberg proved one of the results needed for the results of this thesis to be fully completed.

Contents

1 Introduction

Type theory had its inception in the early 20th century as a means of avoiding the problems of Russell's paradox[4]. Further investigations were carried out by (among others) Alonzo Church, who investigated the simply typed lambda calculus[9]. Further work was later done by Martin-Löf [11] which laid the groundwork for a type theory-renaissance in the later parts of the 20th century and early 21st century when he suggested that type theory could be used as an intuitionistic foundation for mathematics (a foundation particularly suited for computer programming). It was early understood that type theory (as compared to its competitors for the title of foundation of mathematics) had one key advantage that made it especially suited for constructive mathematics: computation. This lead to the Univalent Foundations Programme. The combination of type theory and homotopy theory (roughly the study of classification of topological spaces) yields Homotopy Type Theory (HoTT) – a foundation for mathematics with computation as one of its main selling points [14]. The underlying intuition is that equality between terms can be viewed and treated as paths between points in a space.

Taking seriously the idea that computability was the main driver of the programme, there was one problem: One important part of the Univalent Foundations Programme did not have computational content: the univalence axiom. Roughly, this axiom says that equivalence and identity are the same (more precisely, they are equivalent). To mitigate this, cubical type theory was introduced by Cohen et al. [5]. One could say somewhat simplified that this theory is what follows if one takes the intutition above regarding equalities as paths completely literally – it is not the case that equality just can be viewed as a path, it actually is a path. The adjective "cubical" is fitting, because the resulting web of equalities will be comprised of cubes of different dimensionality (roughly, there can be equalities between equalities) within the universe of types.

A more practical use of type theory is as the basis for the programming language Agda. The idea is simple: If our foundations of mathematics can be made into a working programming language, then this means that from a mathematical point of view, we can get a machine to assist us with the proofs. From a software engineering point of view, having the same specification language and programming language makes it possible to use the so-called Curry-Howard-isomorphism (which in a nutshell says that programs and proofs are one and the same) to our advantage so that the programs we write become their own correctness proofs. Their properties are then enforced by the typing system, thus avoiding the need for (and dependency of) using separate modeling-, programming- and proof languages.

The original version of Agda was described by Coquand and Coquand [6]. Later, a redesign called Agda 2 was described by Norell [13] in his doctoral thesis. Cubical Agda was introduced as a mode of the regular Agda language by Vezzosi et al. [15] in 2021. library of Agda is underway of being implemented in this cubical flavour, and that process is what we will contribute to in this thesis by supplying polynomials to the cubical mode.

1.1 Objective

The objective of this thesis is to provide the library of Cubical Agda with a formalisation of polynomials over commutative rings

1.2 Problem Statement

The library of Cubical Agda is missing a working formalisation of polynomials over commutative rings. We cannot in general decide equality in commutative rings, which makes the problem non-trivial.

1.2.1 Research Question

How do we formalize polynomials over commutative rings in Cubical Agda?

1.3 Goals

We require a working implementation of polynomials with the standard operations of addition and multiplication, as well as proofs of certain properties (more specifically, the properties consisting of the axioms of a commutative ring). For reasons which will soon become apparent, this implementation will and must be provably correct.

1.4 Approach

We will make heavy use of the fact that Cubical Agda already contains a formalisation of commutative rings. We will make use of higher inductive types (roughly types with path constructors), defining a parameterised data structure which takes a commutative ring as parameter. The formalisation will be list-based. We will then define the standard operators for polynomials (addition and multiplication) and prove, in sequence, the 8 propositions (one for each commutative ring-axiom) needed to show that the structure itself is a commutative ring. To simplify certain things, we will give two definitions. One of them will be used as the main definition for which we define operators and prove everything ring-related. The second definition will be used to prove that the structure we are working with actually is a set, which in type theory is non-trivial, since not all types are sets (In type theory, we have a specific technical definition of "set" in mind). We will also show that these definitions are equivalent. This approach might seem contrived, but the reason is one of pragmatism. The first definition being more natural to work with in a computer science setting but needing one property taken for granted, while the second definition being perhaps less natural to work with but better suited to showing the property in question.

1.5 Delimitations

We will consider specifically polynomials over commutative rings. We will formalize in Cubical Agda, entailing that the formalization will be done in cubical type theory. We will only implement the most naive algorithms for addition and multiplication, so computational efficiency will not be a priority.

2 Background

We will begin this section by presenting type theory and homotopy type theory (HoTT). We will then cover the programming language Agda. Then follows an overview of cubical type theory, leading over to a presentation of Cubical Agda. Finally, we will give a presentation of commutative rings and polynomials over commutative rings. The presentation will be kept informal and at the conceptual level that is concerned with actually using the proof assistant. Thus we will not give a formal presentation of type theory, but rather a more intuitive presentation relevant to the rather concrete activity of using Agda as a proof assistant. The following two sections (2.1, 2.2) are based on the HoTT Book [14].

2.1 Type Theory and Formalisation

What is a type? Think of set theory: In set theory we have sets and elements – the former are collections of the latter, and the latter are members of the former. In general, a set can be described by supplying a formula in first order logic, that so to speak "picks out" the elements that satisfy it, and put them together into a collection. In type theory we also have two kinds of objects at the bedrock: types and their terms (we will use the word "constructors" synonymously with terms). A type is at once an abstract description (akin to the role of a formula in set theory) of a property as well as a specification of what the terms of the type must be like (akin to a data type definition in certain programming languages). The terms are simply abstract objects that can be members of these types, i.e. satisfy the specification

Example 1. *The empty type $\mathbb{0}$ specifies that it has no terms. The unit type, $\mathbb{1}$, has only a single term, $\star$. The way to think about $\mathbb{0}$ intuitively is that it describes an impossible element – something that cannot exist. Thus, while we can express its property, we cannot create any instances of it. The case is different with $\mathbb{1}$, where the property described by the type can only be satisfied by one unique constructor.*

An important aspect of type theory is the distinction between proposition (type) and judgement. A proposition is some specification in our language of a property. We will say "A type" to indicate that A is a proposition (type). A judgement is something more concrete. We have two judgement forms (given a type A):

$$a : A,$$
$$a \equiv b : A,$$

the former expressing that the constructor a has type A and the latter expressing that the constructors a and b are of type A and in addition: equal by definition (we can check this by expanding definitions). Note that we did not qualify this statement by something along the lines of "given constructors a, b of type A ...". The constructors are introduced at this point in the theory, so it doesn't make much sense to assume them given before. Note that we are using "$\equiv$" and not "$=$". The regular equality sign, "$=$", has a special use as signifying equality types in type theory – types whose terms witness that other types are equal to each other.

In a completely formal setting, when we say that A is a type, we are actually saying that A inhabits the universe U (of discourse). This is in turn a shorthand way of referring to a cumulative hierarchy of universes:

$$U_0 : U_1 : U_2 : ...$$

so that $A : U$ means that $A : U_i$ for some $i \in \mathbb{N}$. As the notation indicates, U_i inhabits U_{i+1}. This cumulative hierarchy may seem unnecessarily complicated, but it is there for theoretical reasons in order to avoid paradoxes, not immediately relevant to this thesis. For our purposes, it suffices to keep in mind that we have some universe of discourse which our types inhabit.

We will have some use of being able to qualify statements when delving deeper into the theory, so we introduce contexts: a context is simply a list of variables and their types (you can think of them as premises for an argument), which may be needed to make sense of the construction of the type that follows (this may be thought of as the conclusion). We will use the phrase "A is true". This is simply a shorthand of $a : A$, but where we have suppressed the constructor that acts as a witness of A. At some points we might not care about what the constructor looks like, only that it exists.

In a completely formal setting one needs to specify, for each type, its formation rules (specifying how we form this kind of type from other types); introduction rules (Specifying how we introduce, or rather construct elements/witnesses of the type); elimination rules (Specifying how to operate on the elements of the type); computation rules (Specifying the relation between the elimination rules and the type, in a computational manner); (optional) uniqueness (explicitly specifying equality of elements). We will look at some selected types, but will not give a completely formal treatment where each of these rules are specified, but rather keep things informal.

2.1.1 Cartesian Product Types

Product types are analogous to products of sets in that they have terms in the form of pairs $(a, b) : A \times B$. We can project the components from a pair using projection functions as follows: $pr_1((a, b)) :\equiv a$, $pr_2((a, b)) :\equiv b$.

2.1.2 Function Types

Function types are similar to how we intuitively think of functions: given two types A and B, we can form the function type $A \to B$ that as elements have all functions f with A as domain and B as codomain. A neat way of representing such a function is using lambda abstraction:

$$\lambda(x : A).f(x)$$

which reduces to a term $f(a) : B$ if we apply f at $a : A$.

2.1.3 Product (Π) Types

A natural generalisation of a function type is the product type. The main difference between regular functions and so-called dependent functions is that the codomain can vary, depending on the argument of the domain, so we have $\Pi_{x:A}B(x)$ as a generalisation of $A \to B$. Actually, the former will be a function type when B is constant i.e. when $B(x)$ does not depend on x. In that case, we get $\Pi_{x:A}B \equiv A \to B$.

2.1.4 Dependent Pair (Σ) Types

A dependent pair $\Sigma_{x:A}B(x)$ has, as the name suggests, pairs as terms where the second components type depends on the first component. So given $a : A$ and $b : B(a)$, we have:

$$(a, b) : \Sigma_{a:A}B(a).$$

The elements of the pair can be projected analogous to the Cartesian product types. Note the difference between pairs in products and pairs in dependent pairs – in the latter the type of the second component can vary based on the first. If we let B(x) be constant, the dependent pair reduces to a Cartesian product.

2.1.5 Coproduct Types

Coproduct types $A + B$ are analogous to disjoint unions in that the terms are terms from either A or B but we retain the information from which of them they came. So we have $\mathit{inl}(a) : A + B$ and $\mathit{inr}(b) : A + B$ for $a : A$, $b : B$ (l for left, r for right).

2.1.6 Propositions as Types

We hinted above about one of the key features of type theory – its relation to formal logic. One can can use type theory as a formal deductive system, by specifying a correspondence (known as the Curry-Howard correspondence) between types and logical operators. Doing this provides us with a constructive (intuitionistic) logic in which we can do mathematics, logic and computer science. The constructive nature makes this especially fruitful, because the proofs provide algorithms for computation. The correspondence is as follows:

Propositions	**Types**
$\top$	$\mathbb{1}$
$\bot$	$\mathbb{0}$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \implies B$	$A \to B$
$A \iff B$	$(A \to B) \times (B \to A)$
$\neg A$	$A \to \mathbb{0}$
$\forall(x : A)B(x)$	$\Pi_{x:A}B(x)$
$\exists(x : A)B(x)$	$\Sigma_{x:A}B(x)$

Table 1: Propositions as types

Indeed, in many cases, it can be of great intuitive help to read types in the logical interpretation. We provide intuitive explanation for some of the logical operators of the correspondence as follows:

- $A \wedge B$ holds iff. both A and B holds. So from the constructive point of view, we need to be able to show evidence of both A and B. But this can be viewed as a pair of the two witnesses. B is not dependent on A so the Cartesian product type is suitable.
- $A \implies B$ holds iff. it is the case that if A is true, then B is also true. From the constructive point of view, we want to be able to turn evidence of A into evidence of B. But this is exactly what the function type does – it takes a witness of A and maps it to a witness of B.
- $\neg A$ holds iff. A does not hold. So we can represent this by constructing a function that takes any witness of A into a witness of the empty type, $\mathbb{0}$. We know that the latter cannot have any witnesses, and thus the former cannot either.

- $\forall(x : A)B(x)$ holds iff. it is the case that for all elements x in the type A (we can think of this as the domain of quantification), we have that $B(x)$ holds. So essentially we need to map all witnesses x of A to witnesses of $B(x)$. We need to allow $B(x)$ to be able to vary depending on the type of the element (we consider a more general kind of predicate $B : A \to U$, whose codomain type depends on the argument. Recall that our types inhabit the universe U).
- $\exists(x : A)B(x)$ Holds iff. there is some element x in A that acts as a witness for $B(x)$. In this case we simply need a pair, and again, we use the dependent pair for the same reason as above concerning the more general predicate.

One of the main differences between this logic and classical logic (due to type theory's constructive nature) is the fact that the law of excluded middle (LEM), i.e. $P \vee \neg P$ (for a proposition P) does not always hold. Intuitively this should be clear if you think of provability rather than truth: while it may be reasonable (depending on your view of the theory of knowledge) to be the case that a proposition is either true or false, it is not as reasonable to say that either we have a proof of the proposition or of its negation – After all, we might still be in the process of settling that question, trying to find a proof.

Note however, that nothing stops us from adding the law of excluded middle as an axiom to our type system, if we wish to do classical logic within it. This is so because while LEM does not hold, it is not the case that the negation of it holds by default in type theory . Thus, we see that type theory has an appealing flexibility that does not entail that we have to completely discard classical logic just because our underlying system is a constructivist one.

2.1.7 Currying

Currying is the manner in which we represent a function from multiple arguments as nested one-variable functions as follows: Given a function $f : A_1 \times A_2 \times ... \times A_n \to B$, we have an equivalent function $f' : A_1 \to A_2 \to ... \to A_n \to B$ – equivalent in the sense that application of f to the tuple $(a_1, a_1, ..., a_n)$ yields the same image as iterated application of the single elements $a_1, a_2, ..., a_n$ to f' and vice versa.

2.2 Homotopy Type Theory

HoTT is type theory with a particular kind of perspective on what the equality concept is – namely, equality is considered as paths in a space. So an equality between, say, two constructors a, b of type A is a continuous path $h : [0, 1] \to A$ that has a and b as its endpoints, i.e. $h(0) = a$ and $h(1) = b$ (note that we are not using $\equiv$ here but rather $=$, because these statements are not within type theory – rather, we are talking here at the meta-level, using the more intuitive meaning of $=$, as compared to the specialized use of $=$ as equality types). We have the following visual aid, for $t \in [0, 1]$:

$$h(0) = a \xrightarrow{h(t)} b = h(1)$$

Figure 1: Equality between points

We need to have a more general view of this so that we will also be able to talk about equality between paths (One important question is if two proofs are equivalent in some sense) so more generally, a mapping between two continuous mappings $h_1 : A \to B$ and $h_2 : A \to B$ is a homotopy, if it is a map $H : A \times [0,1] \to B$ s.t. $H(x,0) = h_1(x)$ and $H(x,1) = h_2(x)$. For our paths, we have the unit interval $[0,1]$ as A and we require that $H(0,t) = h_1(t)$ and $H(1,t) = h_2(t)$ for $t \in [0,1]$ and additionally $H(0,t) = a$ and $H(1,t) = b$ (this is called endpoint-preservation). We can represent this visually using a square:

$$\begin{tikzcd}
H(0,1) = a \arrow[r, "{H(x,1)=h_2(x)}"] & b = H(1,1) \\
H(0,0) = a \arrow[r, "{H(x,0)=h_1(x)}"'] \arrow[u, "{a=H(0,t)}"] & b = H(1,0) \arrow[u, "{H(1,t)=b}"']
\end{tikzcd}$$

Figure 2: Equality between paths

So, with this in mind, we can inhabit our equality types by paths $p : x =_A y$ where $x, y : A$. But in the same manner, since $x =_A y$ is a type (the type of equalities between x and y in A) we can inhabit higher-order equality types by paths as $p' : p_1 =_{x=_A y} p_2$, and so on. We can think of this as a 2-dimensional path (square) as above, and if we iterate this construction, we get a 3-dimensional path (cube) and so on. Keep this intuition in mind – it will be used for cubical type theory in section 2.4.

Example 2. *We present some of the simplest examples of paths (omitting proofs of existence). First is reflexivity (here parameterised by the type and the element),*

$$refl_x^A : x =_A x,$$

which identifies each element with itself:

$$\begin{tikzcd}
x \arrow[loop below, "refl"]
\end{tikzcd}$$

Figure 3: reflexive equality of a point

Another simple example of a path is the reversed path: Given a path $p : x =_A y$, we reverse it to get the symmetric path:

$$p^{-1} : y =_A x,$$

visually:

$$\begin{tikzcd}
x \arrow[r, bend left, "p(t)"] & y \arrow[l, bend left, "p^{-1}(t)"]
\end{tikzcd}$$

Figure 4: symmetric equality between points

Finally, we have the concatenation of two paths that meet in the middle: Given two paths $p : x =_A y$ and $q : y =_A z$ we can construct the concatenation:

$$p \cdot q : x =_A z.$$

Visually we have:

$$\begin{tikzcd} x \arrow[r, "p"'] \arrow[rr, bend left, "p\cdot q"] & y \arrow[r, "q"'] & z \end{tikzcd}$$

Figure 5: transitive equality of two equalities

2.2.1 Univalence

There is a kind of type named equivalence. Intuitively, this is the type where the elements are special kinds of mappings, akin to isomorphisms but more general. Such mappings show that in a mathematical sense, the types which are being mapped to each other are the same, even though they might look different. Intuitively it is helpful to think in terms of representation: For example, the cyclic group of order n and the additive group $\mathbb{Z}/(n\mathbb{Z})$ are different representations of the "same" (by isomorphism) group in a sense [7]. We denote the equivalence type (given two types A and B) by

$$A \simeq B$$

As mentioned above, all types of our theory are themselves elements of a universe of discourse, U. Within this universe we will have an identity type between types (recall that U is a shorthand – in reality, we get an identity type for each respective U_i in the cumulative hierarchy):

$$A =_U B.$$

These two types seem quite similar: one of them speaks about types being equal (by transformation via paths), while the other speaks of them as being equivalent (by function mappings). This leads us to the univalence axiom, which says that these two types really are equivalent:

$$(A \simeq B) \simeq (A =_U B).$$

2.3 Agda

The following section is based on the Agda documentation [2].

Agda is a programming language/proof assistant that is based on type theory and the idea of propositions as types. Compared to many other programming languages, Agda is very close to the mathematical language upon which it is based. This has multiple advantages:

- as a mathematician, using Agda makes it possible to get an assistant that checks your proofs, and warns you if something seems to be wrong. Many of those tasks (that in some sense are mechanical) done within the theorem-proving activity, can be handled automatically by Agda so that the mathematician can concentrate on the creative and conceptual aspects of the proof.

- as a programmer, the close correspondence between programming language and specification language makes verification (at least in theory) simpler, compared with programming languages where the two languages are not as closely tied together and/or as grounded in the foundations of mathematics. In fact, when programming, the program becomes its own correctness proof, checked by the typing system.

To declare a type in Agda we simply provide a name and a type signature, separated by a colon. For example, if we want to declare a function from A to B (keep in mind the logic interpretation of this as the declaration of an implication $A \implies B$) as

$$\texttt{name} : A \to B.$$

When the type signature has been given, we need to show that it is also inhabited. This is done by setting the type of `name` equal to an expression using already instantiated other types. In our example, let's say that $A = B = \mathbb{N}$ and that we want a function that adds one to the argument. The instantiation is made as

$$\texttt{name}\ a = a + 1$$

where we assume that + and 1 already has been defined. This is in principle the way to prove statements about types, but it may also happen that we want to introduce a type definition in a more structured way. We do this using the `Data` keyword:

$$\begin{aligned} &\text{Data } \texttt{Name} : \text{Type where} \\ &\qquad \texttt{Constructor}_1 = \textit{instantiation}_1 \\ &\qquad\qquad \vdots \\ &\qquad \texttt{Constructor}_n = \textit{instantiation}_n \end{aligned}$$

whereby we define a more complicated type with many different constructors.

2.3.1 Equational Reasoning

In Agda, equational reasoning is very similar to how a mathematician works on pen and paper. The main exception is that every rewrite of a term must be explicitly motivated (akin to a proof system). The syntax is on the following form:

$$\begin{aligned} \textit{term}_1 &\equiv \langle\ \textit{justification}_1\ \rangle \\ \textit{term}_2 &\equiv \langle\ \textit{justification}_2\ \rangle \\ &\vdots \\ \textit{term}_{n-1} &\equiv \langle\ \textit{justification}_{n-1}\ \rangle \\ \textit{term}_n &\ \blacksquare \end{aligned}$$

where the equation you want to prove is $\textit{term}_1 = \textit{term}_n$ and each $\textit{justification}_i$ is an identity of the form $\textit{term}_i = \textit{term}_{i+1}$, instantiated elsewhere in the code. In a nutshell, we use equational reasoning when we need to instantiate an equality, thereby breaking down the equality into manageable steps.

2.4 Cubical Type Theory and Cubical Agda

Much of the motivation of the development of cubical type theory came from the need to provide the univalence axiom with computational content. In HoTT we simply had to postulate it as an axiom, but that means that when using it in computation, you might get stuck with the postulate, half-way so to speak. In this theory, this is not the case as univalence can now be shown rather than postulated [5].

When moving to the cubical setting, our inductively defined paths that we used in the previous setting are replaced with another path type. To define the path type we need the interval type, I, which has elements i_0, i_1 called the left and right endpoints of the interval. Then, similarly to how we approached in the homotopical case, we let $x =_A y$ mean that there exists a function $p : I \to A$ that maps our interval endpoints to the equality endpoints, as: $p(i_0) = x$ and $p(i_1) = y$ [5].

We will not go further into cubical type theory for now, but will instead present more details in the part on Cubical Agda, since this is what was used for the results.

2.4.1 Cubical Agda

The following section is based on the documentation for Cubical Agda [3]. We have simplified somewhat, leaving out certain parts of the type signatures that while technically needed, for our present purposes do more harm than good from a pedagogical perspective.

Cubical Agda is a library and mode of Agda that implements cubical type theory. The main difference is the introduction of the interval type that is used as a primitive for paths, as above. The interval type also has three operators which yield a De Morgan algebra:

```
_∧_ : I → I → I
_∨_ : I → I → I
~_ : I → I
```

$\wedge$ can be considered as the minimum over the unit interval, while $\vee$ is the maximum. The reversal of the interval is given by $\sim$ as follows: i_0 gets mapped to i_1, i_1 gets mapped to i_0 and arbitrary $i \in [0, 1]$ gets mapped to $1 - i$.

2.4.2 Higher Inductive Types

Higher inductive types are types where we allow some of the constructors to be paths. The most standard kind of constructor is a point constructor, but by enabling path constructors we can, among other things, implement quotient types with computational content, instead of simply postulating the quotient types existence.

To make a higher inductive type, simply follow the same procedure as above when defining a data type, but make sure that one of the supplied constructors is of the form $x =_A y$ where the terms $x, y : A$ are constructed by the earlier defined constructors of the type (note that these in principle can be both point and path constructors). We just have to make sure that the quotient is well-defined.

2.4.3 Basic Language Constructs

We will give an overview of the relevant language constructs (for this thesis), based on the Cubical Agda Documentation [3].

In cubical type theory, levels play the analogous role as universes. A type, A, exists at some level ℓ, and this is specified by $A : \mathsf{Type}\ \ell$. There are theoretical reasons which makes these levels needed, but for our purposes, we will only work on the lowest level. Specifying it mostly becomes a formality.

One thing to take note of is parameters for types. For a function type, we may have one or more arguments/parameters. These are either explicit, enclosed in parentheses, (A : Type), or implicit, enclosed in curly brackets, {A : Type}. The former indicate actual function arguments, while the latter only indicate conditions that the included simpler types in the complex type must fulfill.

As described above, a path in this setting is simply a function from the interval I. We have a more specific type Path that has the following type signature:

$$\mathsf{Path} : (A : \mathsf{Type}\ \ell) \to A \to A \to \mathsf{Type}\ \ell$$

in essence, you specify the type of the elements, and then the two elements, to get your path between them.

One variant is the pathP-type, which is similar, but also explicitly specifies the endpoints of the path:

$$\mathsf{PathP} : (A : \mathsf{I} \to \mathsf{Type}\ \ell) \to A\ i_0 \to A\ i_1 \to \mathsf{Type}\ \ell$$

A further related function is toPathP which we use to convert between PathP and the transport type. The transport type in turn has the type signature

$$\mathsf{transport} : A \equiv B \to A \to B,$$

so it provides a mapping of elements of the type A to elements of the type B, as long as these two types are judgementally equal.

Starting with the reflexivity path, we have the following type and instantiation:

$$\begin{aligned}&\mathsf{refl} : \{A : \mathsf{Type}\ \ell\}\{x : A\} \to \mathsf{Path}\ A\ x\ x\\&\mathsf{refl}\{A\}\{x\} = \lambda i \to x\end{aligned}$$

So we see that given some type and an element of that type, we get a function that takes every interval value $i \in \mathsf{I}$ to x.

The symmetric path is given using the path that we want to flip and applying it to the reversed interval:

$$\begin{aligned}&\mathsf{sym} : \{A : \mathsf{Type}\ \ell\}\{x, y : A\} \to x \equiv y \to y \equiv x\\&\mathsf{sym}\ p = \lambda i \to p(\sim i)\end{aligned}$$

If we want to concatenate paths, we use the concatenation operator, $\cdot$, and supply the two paths that are to be concatenated. The following is the type signature:

$$\_\cdot\_ : x \equiv y \to y \equiv z \to x \equiv z$$

The cong type corresponds to the fact that functions must respect equality – equal elements map to equal elements. So given an equality we produce a new equality between the values which result if we apply the function at the endpoints of the first equality:

$$\begin{aligned}&\mathsf{cong} : \{A : \mathsf{Type}\ \ell\}\{x, y : A\}\{B : A \to \mathsf{Type}\ \ell\}(f : (a : A) \to B\ a)(p : x \equiv y) \to\\&\qquad \mathsf{PathP}(\lambda i \to B\ (p\ i))(f\ x)(f\ y)\\&\mathsf{cong}\ f\ p\ i = f\ (p\ i)\end{aligned}$$

The isProp type has as elements only those types which are propositions, i.e. which has at most one point constructor (more precisely, any seemingly different constructors of the type can be shown equal). This conforms to our intuition of what (atomic) propositions in propositional logic are like – we only care if they are true or false, and forget the rest (In type theory this corresponds to whether they are inhabited or not):

$$\begin{aligned}&\mathsf{isProp} : \mathsf{Type}\ \ell \to \mathsf{Type}\ \ell\\&\mathsf{isProp}\ A = (x, y : A) \to x \equiv y\end{aligned}$$

isSet is similar to isProp, but indicates if a type is a set, i.e. if the equality type relating the constructors/elements is a proposition.

$$\begin{aligned}&\mathsf{isSet} : \mathsf{Type}\ \ell \to \mathsf{Type}\ \ell\\&\mathsf{isSet}\ A = (x, y : A) \to \mathsf{isProp}(x \equiv y)\end{aligned}$$

Overall there are many more useful types in Agda – more than we can cover here. Fortunately, the naming conventions are such that the meaning of the type is more or less transparent from the name. So for example, isSetΠ is a type that simply asserts that the Π-type is a set; pred-≤-pred is a type that asserts that in the theory of natural numbers, the predecessor function preserves the order relation. In such cases, the exact instantiation is not critical to understand in order to be able use the types in the proof assistant.

2.5 Polynomials Over Commutative Rings

We begin this section by first presenting commutative rings, following the presentation given by Dummit and Foote [7]:

Definition 2.1. *(commutative ring) Given a set R and two binary operators, $+$ and $\cdot$, the structure $(R, +, \cdot)$ is a commutative ring if the following holds:*

$$\textit{Ring properties}\begin{cases}\text{(i) } (R,+) \text{ is an abelian group}\\ \text{(ii) } \cdot \text{ is associative}\\ \text{(iii) } \cdot \text{ left/right-distributes over addition}\end{cases}$$
$$\text{(iv) } \cdot \text{ is commutative}$$

Perhaps the most intuitive way of viewing polynomials is as formal sums of terms consisting of a product of an element from a ring with the so-called indeterminate, x. In mathematical notation a polynomial has the form $\sum_{i=1}^{k} a_i x^i$ where $a_i \in R$ and $k \in \mathbb{N}$. While easy to work with for a human mathematician, for our purposes we will consider a more formal approach suitable for computer implementation, where polynomials are viewed as number sequences, with only a finite number of elements not being equal to the zero element of the underlying ring. We will denote these sequences using the tuple presentation $(a_0, a_1, ...)$. We now give the formal definition, following the presentation of Jacobson [8]:

Definition 2.2. *Given a commutative ring R, we define the polynomial ring as follows:*

$$R[x] = \{(a_0, a_1, ...) \mid a_i \in R \ \wedge\ a_i \neq 0 \textit{ holds for only a finite number of the } a_i\textit{'s}\}$$

Where two polynomials, $(a_0, a_1, ...), (b_0, b_1, ...)$*, are equal iff.* $a_i = b_i$ *for all* $i \in \mathbb{N}$ *and where addition is defined by*

$$(a_0, a_1, ...) + (b_0, b_1, ...) = (a_0 + b_0, a_1 + b_1, ...)$$

and multiplication is defined by

$$(a_0, a_1, ...) \cdot (b_0, b_1, ...) = (p_0, p_1, ...)$$

where each p_i *is given by*

$$p_i = \sum_{j+k=i} a_j b_k$$

The result that is central to this thesis is that the structure so defined is itself a commutative ring. We prove this in the tuple representation:

(i) Closure is immediate since the result of adding two polynomials is a new infinite sequence with only finite non-zero elements from the underlying ring (using the closure of addition of the underlying group). The identity element for addition can easily be verified to be $(0_R, 0_R, ...)$. Associativity follows immediately using componentwise associativity of the underlying group. The additive inverse of $(a_0, a_1, ...)$ can easily be checked to be $(-a_0, -a_1, ...)$, i.e. taking "component-wise" inverses. **(ii)** Follows by a tedious expansion and rewriting of both sides of the equation needed to be shown. **(iii)** As in (ii) **(iv)** Multiplicative commutativity follows by considering the commutativity of multiplication in the underlying ring and noting that the summation for a given i won't be affected by switching the two factors of the terms around.

The tuple notation for the sequences are especially fitting for our purposes. From a computer scientific perspective, we can view it quite literally as a finite tuple or list, if we throw away the trailing zeros. One key problem that this entails, which we need to tackle in the results, is that if we simply choose to represent a polynomial with a list of the coefficients up to the point where they become zero – then we will need to do some work in order for equality between polynomials to be unchanged. The problem is that a single polynomial will then have infinitely many representations, where you simply append any number of trailing zeros: $[a_0, a_1, ...a_n]$ is equal to $[a_0, a_1, ...a_n, 0, 0, ..., 0]$ (modulo trailing zeros). So some form of quotient is needed to preserve the proper equality.

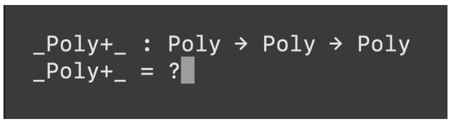
3 Method

We will begin this section with a short overview of Emacs in relation to cubical Agda. Then follows a short introduction to the libraries of Cubical Agda and their usage. Finally we outline the practical approach taken when working in Agda for this thesis.

3.1 Emacs

Emacs in Agda mode works as a code editor and proof assistant. Those things that can be mechanically inferred, such as certain type signatures and introductions of constructors and the like is suggested and can be automatically filled in.

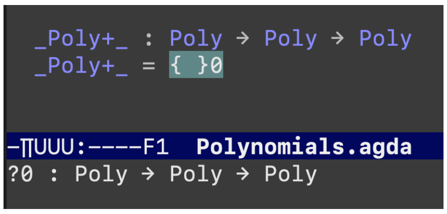
With the risk of getting ahead of ourselves, we will present the workflow and usage of the Agda proof assistant, applied on two types from the results. So let's say that we want to define addition for polynomials. We know that addition is a binary operator, so by currying, we represent it as a nested function type. We write out the type signature, and then write out the definition but provide simply a question mark as definiens:



```
_Poly+_ : Poly → Poly → Poly
_Poly+_ = ?
```

Figure 6: Defining a type

We use the command `CTRL-C CTRL-L` to load (type check) the file, resulting in a goal. Agda Provides the type signature that needs to be provided into the goal:

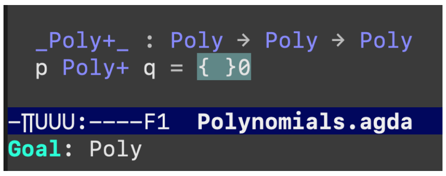


```
_Poly+_ : Poly → Poly → Poly
_Poly+_ = { }0

-∏UUU:----F1  Polynomials.agda
?0 : Poly → Poly → Poly
```

Figure 7: Type signature of an Agda hole

Now, we can provide two variables as arguments and reload the file:

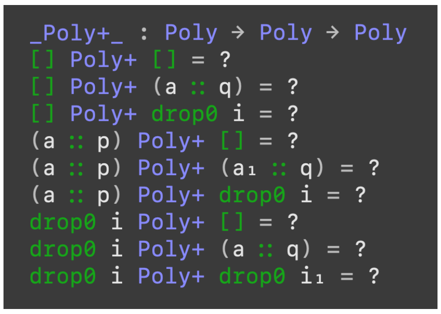


```
_Poly+_ : Poly → Poly → Poly
p Poly+ q = { }0

-∏UUU:----F1  Polynomials.agda
Goal: Poly
```

Figure 8: New type signature from partially filled hole

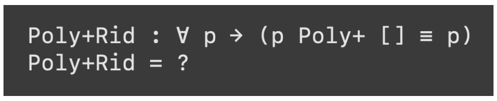
For inductive types, a good idea can be to try to do a definition on cases. So we step into the goal and hit `CTRL-C CTRL-C` for a case split. We provide p and q as the splitting variables so that we get all possible cases that we must cover, resulting in:



```
_Poly+_ : Poly → Poly → Poly
[] Poly+ [] = ?
[] Poly+ (a :: q) = ?
[] Poly+ drop0 i = ?
(a :: p) Poly+ [] = ?
(a :: p) Poly+ (a₁ :: q) = ?
(a :: p) Poly+ drop0 i = ?
drop0 i Poly+ [] = ?
drop0 i Poly+ (a :: q) = ?
drop0 i Poly+ drop0 i₁ = ?
```

Figure 9: Case split on parameters

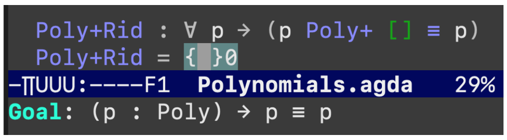
Now, let's say we have filled all holes that resulted from the previous step, so that we have a functioning addition operator. To prove a property (in this case, that the empty polynomial is the right identity) we provide the right type signature (by the Curry-Howard correspondence) and follow the same basic approach of providing a definition with a question mark as definiens:



```
Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
Poly+Rid = ?
```

Figure 10: A proposition as type

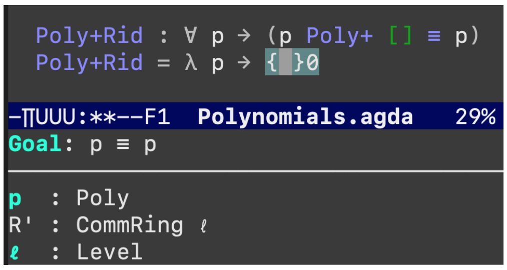
If we hit `CTRL-C CTRL-L` again to type check we get:



```
  Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
  Poly+Rid = { }0
-∏UUU:----F1  Polynomials.agda    29%
Goal: (p : Poly) → p ≡ p
```

Figure 11: Another type signature of an Agda Hole

Hit `CTRL-C CTRL-R` while the cursor is within the hole to refine (when the goal is a function type, Agda writes out lambda operators and the required parameters for us):

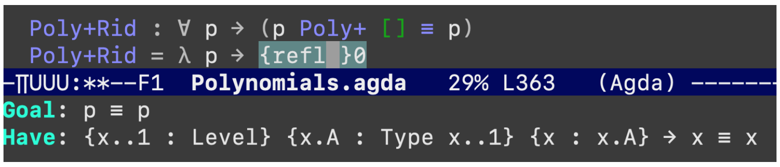


```
  Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
  Poly+Rid = λ p → { }0

-ΠUUU:**--F1  Polynomials.agda   29%
Goal: p ≡ p
————————————————————————————————————————
p   : Poly
R'  : CommRing ℓ
ℓ   : Level
```

Figure 12: Refinement of function type

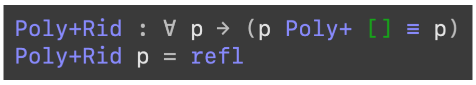
Agda informs us that we need a proof that any polynomial is equal to itself. Recall from example 2 that this is precisely what refl witnesses. So we simply type "refl" into the hole, and then hit `CTRL-C CTRL-.` to compare the supplied type with the goal type:



```
  Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
  Poly+Rid = λ p → {refl }0
-ΠUUU:**--F1  Polynomials.agda   29% L363   (Agda) -------
Goal: p ≡ p
Have: {x..1 : Level} {x.A : Type x..1} {x : x.A} → x ≡ x
```

Figure 13: Comparison of filled-in and required type

There is some additional information enclosed in curly brackets at the left hand side of the type signature. These indicate implicit information, in this case that that x must have certain properties. At this point it is not important, and we can simply hit `CTRL-C CTRL-SPACE` to make Agda fill in the hole using the supplied type (if it type checks):



```
Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
Poly+Rid p = refl
```

Figure 14: Suggested type accepted to fill the hole

This concludes the proof of right identity.

3.2 Cubical Agda Modules

Here we describe the modules from the Cubical Agda library [1] which we have used in the proof.

`Cubical.Algebra.CommRing` is the implementation of commutative rings. We use it since this is the kind of structure that the polynomials are parameterised over. Additionally, `Cubical.Algebra.Group` and `Cubical.Algebra.Ring` are used because certain mathematical propositions that are used are found in these modules – they are not repeated in `Cubical.Algebra.CommRing` (even though a commutative ring is a ring, and contains a group).

`Cubical.Data.Nat` is the implementation of natural numbers. It is used because we need natural numbers for the function-based definition (giving the cases in terms of induction over the natural numbers). `Cubical.Data.Nat.Order` is the implementation of partial orders, and we need it for a similar purpose: for the second definition, to represent the property that the function at some point only return 0. `Cubical.Data.Empty.Base` is used for a proof and `Cubical.Data.Bool` is used for a test of zero-ness (given a number, output *true* if it is zero, otherwise *false*).

We also use `Cubical.HITs.PropositionalTruncation` to squash certain types in our proofs. What this means is that we discard the more detailed information of the constructors (*how* they are true) and only care about if they are inhabited or not. `Cubical.Foundations.Prelude` contains much basic functionality for Agda which we will need. Multiple results about which types are propositions respective sets as well as the relation between propositions and sets is contained in `Cubical.Foundations.HLevels`. Sigma types are contained in `Cubical.Data.Sigma`, which we use for one of the polynomial definitions.

3.3 Formalisation Approach

We will use a list-based approach for implementing polynomials, viewing the polynomial $a_0 + a_1x + a_2x^2, ...$ as a list `[`$a_0, a_1, a_2, ...$`]`. In a functional language, this takes the form of a recursive type with the base case of the zero polynomial (empty list) and the recursive case of appending a ring element to a polynomial (a :: `[`$a_0, a_1, ...$`]`) – the ::-operator taking a similar role as the `cons`-operator of the programming language Lisp (see further Karnick [10]). To meet the challenge of undecidable (in general) equality in a ring we will need to use higher inductive types, so that we have one additional constructor that makes it possible to identify polynomials that are the same except for possibly trailing zeros. So for example $a_0 + a_1x + a_2x^2 + 0x^3$ is equal to $a_0 + a_1x + a_2x^2$, but in our formalisation we need to have the explicit equality constructor that enable us to drop zeros, so that `[`$a_0, a_1, a_2, 0$`]` is equal to `[`a_0, a_1, a_2`]`. The standard operators for polynomials will be implemented and we will then prove that the mathematical structure that we get by forming polynomials over a commutative ring is itself a commutative ring, by showing those properties needed as per definition 2.1. Specifically, the following checklist is provided in the `CommRing`-module of the Cubical Agda library:

- The additive identity.
- The multiplicative identity.
- The additive operator.
- The multiplicative operator.
- The inverse operator for addition.
- A proof that the type is a set (i.e. that the equality type of the type is a proposition).
- A proof of associativity of addition.
- A proof of right identity for addition.
- A proof of inverses for addition.
- A proof of commutativity for addition.
- A proof of associativity of multiplication.

- A proof of right identity of multiplication.
- A proof of left distributivity of multiplication over addition.
- A proof of commutativity of multiplication.

4 Results

This section presents the key parts of the formalisation, provided in full in the appendix (and also available in the form of a Git commit [16]). All important results will have the code duplicated in this section, but the reader is encouraged to look at the appendix for the less important lemmas, perhaps even reading the results and the appendix side by side. The expressions given below for explanatory purposes will be closer to mathematical notation as compared to the code. We will present most of the relevant theorems in an informal way (in addition to the formal proofs in Cubical Agda) but give a more detailed presentation on some of them, providing the reader with examples of how one reads the formal proofs. The order of the theorems presented in this chapter is slightly different from the order of the theorems in the code, since we group them under their respective operator(s). The order in the code was chosen pragmatically during the course of the work, on the basis of which theorems needed which results to be shown.

4.1 Definitions

4.1.1 The Polynomial Data types

The representation of polynomials as lists was defined as follows:

```
data Poly : Type ℓ where
  [] : Poly
  _::_ : (a : R) → (p : Poly) → Poly
  drop0 : 0r :: [] ≡ []
```

Code snippet 1: List-based definition of polynomials

So we have as a basis the empty polynomial (one of the representations of the zero polynomial) and as the inductive case we take a ring element as coefficient for the next degree term of a given polynomial. The equality constructor drop0 enable us to identify different equal representations of the same polynomial with trailing zero coefficients.

After some initial proofs we noted that there were duplication of code in the proofs around the interesting parts. In order to avoid this and simplify, we defined two eliminator modules (the latter being a specialisation of the former, being specifically for propositions). Without these modules, one has to duplicate certain parts of proofs for every statement proved. This way, we simply need to present the ElimProp.f-function with four arguments: First, the proposition (as a type); Second, the base case; Third, the induction case; Fourth, The proof that the type actually is a proposition:

```
module Elim (B      : Poly → Type ℓ')
            ([]*    : B [])
            (cons* : (r : R) (p : Poly) (b : B p) → B (r :: p))
            (drop0* : PathP (λ i → B (drop0 i)) (cons* 0r [] []*) []*) where

  f : (p : Poly) → B p
  f [] = []*
  f (x :: p) = cons* x p (f p)
  f (drop0 i) = drop0* i


module ElimProp (B : Poly → Type ℓ')
```

```
                    ([]* : B [])
                    (cons* : (r : R) (p : Poly) (b : B p) → B (r :: p))
                    (BProp : {p : Poly} → isProp (B p)) where
  f : (p : Poly) → B p
  f = Elim.f B []* cons* (toPathP (BProp (transport (λ i → B (drop0 i)) (cons* 0r [] []*)) []*))
```

Code snippet 2: Eliminator module

In the first iteration of the project, we had an additional constructor in the definition, named trunc and of type isSet Poly. While straightforward at first, this had the later consequence of unnecessary proof complexity, where the proof cases involving trunc had a tendency to become large and unwieldy. To avoid this, we completely removed the constructor as a part of the basic definition, and instead took the following approach: We created another implementation of polynomials for which it would be easier to prove this property (the proof went outside of scope of this thesis, but was proved separately by Cavallo and Mörtberg). Then we set up a proof that these two different definitions were equivalent, thus enabling us to use the isSetPoly type, fulfilling the same role as the trunc term (i.e. as a witness of the set property) in our proofs – now being a theorem rather than a constructor. The second (function-based) definition, inspired by definition 2.2 took the following form:

```
PolyFun : Type ℓ
PolyFun = Σ[ p ∈ (ℕ → R) ] (∃[ n ∈ ℕ ] ((m : ℕ) → n ≤ m → p m ≡ 0r))
```

Code snippet 3: Function-based definition of polynomials

In contrast to the list-based definition, in this case we represent the polynomial as a function (specifically a sequence), that given a natural number n provide us with the coefficient of degree n. Note also how the equality modulo trailing zeros is avoided simply by letting this function always return zeros for large enough n. The actual function, p, that is used as a parameter can be constructed from a polynomial in list-form. The definition of the function is given by:

```
Poly→Fun : Poly → (ℕ → R)
Poly→Fun [] = (λ _ → 0r)
Poly→Fun (a :: p) = (λ n → if isZero n then a else Poly→Fun p (predℕ n))
Poly→Fun (drop0 i) = lemma i
  where
  lemma : (λ n → if isZero n then 0r else 0r) ≡ (λ _ → 0r)
  lemma i zero = 0r
  lemma i (suc n) = 0r
```

Code snippet 4: Construction of function given a list-based polynomial

For the proof of equivalence between these two definitions of polynomials (including the proof of the second having the set property), the reader is directed to the appendix. For our purposes, the main part is the instantiation of the isSetPoly-type with type signature isSet Poly. We do this by means of the isSetRetract-function: requiring two mappings between the two types (both directions); a proof that these mappings in composition is the identity mapping; and finally a proof that the second definition is a set:

```
isSetPoly = isSetRetract Poly→PolyFun
                         PolyFun→Poly
                         PolyFun→Poly→PolyFun
                         isSetPolyFun
```

Code snippet 5: Proof that the list-based type is a set

4.1.2 Operations

Now, continuing to work with the first list-based definition, we defined operations on polynomials. The first operation is negation, i.e. the inverse operator of addition. We define this before addition simply because of pedagogic reasons – A 1-ary function being easier to handle than a 2-ary function:

```
Poly- : Poly → Poly
Poly- [] = []
Poly- (a :: p) = (- a) :: (Poly- p)
Poly- (drop0 i) = (cong (_:: []) (inv1g) · drop0) i
```

Code snippet 6: Negation of a polynomial

Next, we have addition of polynomials. Evidently, we get quite a few cases to take care of when moving to 2-airy functions – where each one of the cases involving the drop0-constructor requires some use of ElimProp, since we need a lemma involving left identities for addition to show the required identity. For the final case of the two drop0-constructors, we essentially have a square to work with, since there are two equalities at play. We also have application of the two interval variables i, j at each equality. We work these equalities into a square using a sort of helper function (a filler – which helps us fill in the complete square from the two sides/equalities) in combination with properties about addition from the underlying ring:

```
_Poly+_ : Poly → Poly → Poly
p Poly+ [] = p
[] Poly+ (drop0 i) = drop0 i
[] Poly+ (b :: q) = b :: q
(a :: p) Poly+ (b :: q) = (a + b) :: (p Poly+ q)
(a :: p) Poly+ (drop0 i) = +Rid a i :: p
(drop0 i) Poly+ (a :: q) = lem q i where
                                  lem : ∀ q → (0r + a) :: ([] Poly+ q) ≡ a :: q
                                  lem = ElimProp.f (λ q → (0r + a) :: ([] Poly+ q) ≡ a :: q)
                                                   (λ i → (+Lid a i :: []))
                                                   (λ r p _ → λ i → +Lid a i :: r :: p )
                                                   (isSetPoly _ _)
(drop0 i) Poly+ (drop0 j) = isSet→isSet' isSetPoly (cong ([_] ) (+Rid 0r)) drop0
                                                   (cong ([_] ) (+Lid 0r)) drop0 i j
```

Code snippet 7: Addition of two polynomials

There are some things to note: Most striking is the difference compared to the standard mathematical presentation when it comes to the cases in the definitions for the drop0-equality. What this amounts to is to show that this equality (modulo trailing zeros) is preserved by the operation. If this were not the case, we would be in a situation where the mapping takes equal elements into unequal elements, breaking functionality. Another thing to note is the straightforward way in which we use pattern matching to give the recursive definitions concerning the empty polynomial ([]) as base and the ::-operator as the recursive step. Also note the combinatoric explosion that multiple constructors give rise to (this is part of the reason why the removed trunc-constructor gave rise to so much complexity), for k constructors we get 2^k cases to take care of in the worst case.

It might seem a bit unintuitive to consider the case where the drop0-equality is combined with a polynomial p in the operation, but what this amounts to is to consider the applications of the equality's/path's endpoints, rather than the equality itself.

Before we define multiplication, we define a helper-operator which is constant multiplication where the first argument ranges over the underlying ring, while the second argument ranges over polynomials. This simplifies things later. Note the use of the result 0RightAnnihilates in the last case, taken from the Ring module of the library. We need this property of the ring elements to show that the equality is preserved.

```
_PolyConst*_ : (R) → Poly → Poly
r PolyConst* [] = []
r PolyConst* (a :: p) = (r · a) :: (r PolyConst* p)
r PolyConst* (drop0 i) = lem r i where
                                    lem : ∀ r → [ r · 0r ] ≡ []
                                    lem = λ r → [ r · 0r ] ≡⟨ cong (_:: []) (0RightAnnihilates r) ⟩
                                             [ 0r ] ≡⟨ drop0 ⟩
                                             [] ∎
```

Code snippet 8: Constant multiplication of a polynomial with a ring element

Finally, we define general multiplication where both arguments are polynomials. At this point, we don't need to handle as many cases as for addition, because of the annihilating character of the base case. We use two lemmas in our definition: 0rLeftAnnihilatesPoly is the proposition that the zero in the underlying ring annihilates any polynomial when PolyConst* is applied to them. 0Idempotent is a property of the underlying ring that the zero element is idempotent with respect to addition. Note the use of _PolyConst*_ in the second case, where we split the first argument so that the first term is concerned with the ring element, while the second term is concerned with the inductive step of two polynomials.

```
_Poly*_ : Poly → Poly → Poly
[] Poly* q = []
(a :: p) Poly* q = (a PolyConst* q) Poly+ (0r :: (p Poly* q))
(drop0 i) Poly* q = lem q i where
                              lem : ∀ q → (0r PolyConst* q) Poly+ [ 0r ] ≡ []
                              lem = λ q → ((0r PolyConst* q) Poly+ [ 0r ]) ≡⟨ cong ( _Poly+ [ 0r ] ) (0rLeftAnnihilatesPoly q)⟩
                                        ([ 0r ] Poly+ [ 0r ]) ≡⟨ cong (_:: []) 0Idempotent ∙ drop0 ⟩
                                        [] ∎
```

Code snippet 9: Multiplication of two polynomials

The PolyConst*-function facilitates the definition of the regular polynomial multiplication of Poly*: it is used in the recursive step and in the drop0 case. Indeed, the multiplication of a ring element and a polynomial using the PolyConst* operator can be expressed as the multiplication of a singleton polynomial:

$$a \text{ PolyConst* } p = [a] \text{ Poly* } p.$$

This is in fact a lemma that we need later.

Note now that the unit for Poly+ is the empty polynomial (modulo trailing zeros), [], and the unit for Poly* is the polynomial consisting of the multiplicative unit of the underlying ring, $[1_R]$. We denote these as 0P and 1P respectively.

With these definitions, we will now prove the properties that make up the properties of a commutative ring (following the checklist of section 3.3).

4.2 Theorems

The relevant theorems that we will look closer at are the ones needed to prove that polynomials over a commutative ring is itself a commutative ring. We look first at the properties of addition, and then at the properties of multiplication. It is worth to note that the order of lemmas and theorems in the appendix came about as the work proceeded, in order to be somewhat economic – some lemmas as well as theorems are reused.

4.2.1 Additive Properties

Right identity The first theorem we prove is that of existence of a right identity for polynomials.

```
Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
Poly+Rid p = refl
```

Code snippet 10: Right identity for addition

The proof is very simple – just a matter of using refl. This is the case, because as we defined addition, Agda will reduce p Poly+ [] into p without any additional input from the programmer.

associativity The proof of associativity is straight-forward:

```
Poly+Assoc : ∀ p q r → p Poly+ (q Poly+ r) ≡ (p Poly+ q) Poly+ r
Poly+Assoc =
  ElimProp.f (λ p → (∀ q r → p Poly+ (q Poly+ r) ≡ (p Poly+ q) Poly+ r))
             (λ q r → Poly+Lid (q Poly+ r) · cong (_Poly+ r) (sym (Poly+Lid q)))
             (λ a p prf → ElimProp.f ((λ q → ∀ r → ((a :: p) Poly+ (q Poly+ r)) ≡
                                                    (((a :: p) Poly+ q) Poly+ r)))
                                      (λ r → cong ((a :: p) Poly+_) (Poly+Lid r))
                                      (λ b q prf2 →
                                      ElimProp.f
                                        (λ r → ((a :: p) Poly+ ((b :: q) Poly+ r)) ≡
                                                  ((a + b :: (p Poly+ q)) Poly+ r))
                                        refl
                                        (λ c r prfp → cong ((a + (b + c))::_)
                                                           (prf q r) ·
                                                           (cong (_:: ((p Poly+ q) Poly+ r))
                                                                 (+Assoc a b c)))
                                        (isSetPoly _ _))
                                      λ x y i r → isSetPoly (x r i0) (y r i1) (x r) (y r) i)
             λ x y i q r → isSetPoly _ _ (x q r) (y q r) i
```

Code snippet 11: Associativity of addition

We use left inverses and the fact that addition in the underlying ring is associative. One thing to note is the nested use of ElimProp.f. To understand what is going on, note that for each use of ElimProp.f, we eliminate one argument. In the case of associativity, we have three universally quantified arguments, p, q, r such that:

$$p \text{ Poly+ } (q \text{ Poly+ } r) \equiv (p \text{ Poly+ } q) \text{ Poly+ } r,$$

so after the first elimination, we have a new proposition with two universally quantified arguments q, r:

$$((a :: p) \text{ Poly+ } (q \text{ Poly+ } r)) \equiv (((a :: p) \text{ Poly+ } q) \text{ Poly+ } r))$$

$(a :: p)$ can in this case be considered as a constant term, and we again apply *ElimProp*, in a nested fashion, to eliminate q and finally r.

inverses The next result we need is that of inverses for addition:

```
Poly+Inverses : ∀ p → p Poly+ (Poly- p) ≡ []
Poly+Inverses = ElimProp.f ( λ p → p Poly+ (Poly- p) ≡ [])
                                                refl
                           (λ r p prf → cong (r + - r ::_) prf ∙
                                        (cong (_:: []) (+Rinv r) ∙ drop0))
                           (isSetPoly _ _)
```

Code snippet 12: Inverses for addition

The existence of inverses is proven by utilising Poly-, which as we have defined it gives the additive inverses of a polynomial. We give a detailed proof of this to showcase the procedure taken. The first step is to represent the property as a type using the Curry-Howard isomorphism. The statement we are after is that for any polynomial, there is an additive inverse. However, we can avoid the nested quantifiers if we give some thought into what the additive inverse actually should look like (it should just be the negation of the polynomial). Thus we get the following type signature:

$$\forall p \to p \text{ Poly+ } (\text{Poly- } p) \equiv []$$

We use ElimProp.f applied to this expression. and thus do a proof by cases (one case for each constructor of the data definition). The first case we need to prove is:

$$[] \text{ Poly+ } (\text{Poly- } []) \equiv []$$

But by the definitions of Poly+ and Poly-, Agda simplifies the goal into

$$[] \equiv []$$

and this can be filled with refl.

The second case is a bit more complicated. Given a ring element r, a polynomial p, and an inductive hypothesis, $prf : (p \text{ Poly+ Poly- } p) \equiv []$ The goal is:

$$r + -r :: (p \text{ Poly+ Poly- } p) \equiv []$$

We see that *prf* almost has the right form. We only need to use *cong* to append the expression $(r + -r)$ to both sides of the equality, and then show that the RHS will still be equal to $[]$. The concatenation function, $\cdot$, is used to bind together the two different equalities with the same endpoints. So we get the following goal:

$$[r + -r] \equiv []$$

which we can prove by using the right inverses for groups to get $[0r] \equiv$ [] and finally, we use concatenation again with the drop0-equality.

Finally, we need to show for the last case that the type is a proposition. We do this by using isSetPoly and ignoring the first two arguments. At this point, the reader might ask himself why in this case we don't need to give more arguments, as in the above examples using isSetPoly. The reason is that Agda can do the work for us, and infer what must be the case for the arguments. This does not work every time though, so sometimes we need to be more explicit.

Commutativity The commutativity property of Poly+ is proven by means of left identity and the commutativity of addition in the underlying ring – no surprises here:

```
Poly+Comm : ∀ p q → p Poly+ q ≡ q Poly+ p
Poly+Comm = ElimProp.f (λ p → (∀ q → p Poly+ q ≡ q Poly+ p))
                       (λ q → Poly+Lid q)
                       (λ a p prf → ElimProp.f (λ q → ((a :: p) Poly+ q) ≡ (q Poly+ (a :: p)))
                                               refl
                                               (λ b q prf2 → cong (_:: (p Poly+ q)) (+Comm a b) ∙
                                                             cong ((b + a) ::_) (prf q))
                                               (isSetPoly _ _)
                       )
                       (λ {p} → isPropΠ (λ q → isSetPoly (p Poly+ q) (q Poly+ p)))
```

Code snippet 13: Commutativity of addition

We are now finished with the additive properties.

4.2.2 Multiplicative Properties

Commutativity The first theorem regarding multiplication that we prove is that of commutativity of multiplication. We will give a detailed, more mathematical presentation of this in addition to the formal proof in Agda. We will use the following notation:

- $+_R$ is used for addition in the underlying ring
- $\cdot_R$ is used for multiplication in the underlying ring
- $+$ is used for addition of polynomials
- $*$ is used for constant multiplication of a ring element with a polynomial
- $\cdot$ is used for polynomial multiplication
- the singleton polynomials consisting only of a constant ring element, a, will be denoted by $[a]$.

The formal proof is as follows:

```
Poly*Commutative : ∀ p q → p Poly* q ≡ q Poly* p
Poly*Commutative =
  ElimProp.f (λ p → ∀ q → p Poly* q ≡ q Poly* p)
             (λ q → sym (0PLeftAnnihilates q))
             (λ a p prf q → (a PolyConst* q) Poly+ (0r :: (p Poly* q)) ≡⟨
                cong ((a PolyConst* q) Poly+_)
                     (cong (0r ::_) (prf q)) ⟩
((a PolyConst* q) Poly+ (0r :: (q Poly* p))) ≡⟨
                                   cong ((a PolyConst* q) Poly+_)
                                   (sym (0r::LeftAssoc q p))
                                   ⟩
((a PolyConst* q) Poly+ ((0r :: q) Poly* p)) ≡⟨
                cong (_Poly+ ((0r PolyConst* p) Poly+ (0r :: (q Poly* p))))
                     (PolyConst*r=Poly*[r] a q) ⟩
((q Poly* [ a ]) Poly+ ((0r :: q) Poly* p)) ≡⟨
                                 cong ((q Poly* [ a ]) Poly+_)
                                      (0r::Comm q p)
                                 ⟩
((q Poly* [ a ]) Poly+ (q Poly* (0r :: p))) ≡⟨
                                 sym (Poly*LDistrPoly+ q [ a ] (0r :: p))
                                 ⟩
(((q Poly* ([ a ] Poly+ (0r :: p))))) ≡⟨
                            cong (q Poly*_)
                                 (Poly+Comm [ a ] (0r :: p))
                            ⟩
((q Poly* ((0r :: p) Poly+ [ a ]))) ≡⟨
                           refl
                           ⟩
(q Poly* ((0r + a) :: p)) ≡⟨ cong (q Poly*_)
                                 (cong (_:: p) (+Lid a))
                        ⟩
(q Poly* (a :: p)) ■)
(λ x y i q → isSetPoly _ _ (x q ) (y q) i)
```

Code snippet 14: Commutativity of multiplication

There is much groundwork necessary before this can be done, in the form of lemmas. of special note are the lemmas relating PolyConst* to Poly* (which are needed since Poly* is defined in terms of PolyConst*). The base case of commutativity is proved using an annihilation lemma (of the equality $0_P * p = 0_P$, where 0_P is the zero polynomial), which we reverse order on using sym. The inductive step uses multiple properties. What we need to prove is

$$(a * q) +_P (0_R :: (p \cdot q)) \equiv q \cdot (a :: p)$$

or in a more familiar notation:

$$(a * q) + (p \cdot q)x \equiv q \cdot (a + px)$$

So starting from $(a * q) + (p \cdot q)x$, our first rewrite is simply an application of the inductive hypothesis, resulting in:

$$(a * q) + (q \cdot p)x$$

For the next step we need the lemma 0_R::LeftAssoc witnessing the following identity:

$$(0r :: p) \cdot q \equiv 0_R :: (p \cdot q)$$

i.e.

$$(px) \cdot q \equiv (p \cdot q)x.$$

We reverse the expression we were left with and use cong to get the result

$$(a * q) + ((qx) \cdot p).$$

Next, we need the lemma PolyConst∗r=Poly∗[r], giving us the identity

$$a * p \equiv p \cdot [a],$$

essentially relating the constant multiplication of a polynomial with a ring element with the corresponding polynomial multiplication of a polynomial with a constant polynomial. One can view it as the lemma which gives the embedding of the underlying ring within the polynomial ring. Again, we use cong which results in the expression

$$(q \cdot [a]) + ((qx) \cdot p).$$

Next, we need the lemma 0r::Comm giving us the identity

$$(0_R :: p) \cdot q \equiv p \cdot (0_R :: q),$$

or in a more familiar notation,

$$(px) \cdot q \equiv p \cdot (qx).$$

cong-ing the result we were left with yields

$$(q \cdot [a]) + (q \cdot (px))$$

Next rewrite is a matter of reversing the equality provided by the theorem Poly*LDistrPoly+ (presented in section 4.2.3), providing us with the identity

$$p \cdot (q + r) \equiv (p \cdot q) + (p \cdot r).$$

We are now at the expression:

$$q \cdot ([a] + (px)).$$

Next, we use the theorem Poly+Comm from section 4.2.1 to reverse the inner addition:

$$q \cdot ((px) + [a]).$$

For the next step, we need only to use refl. This step is simple, because it is only a matter of rewriting on the basis of the definition of addition. We get:

$$q \cdot ((0r +_R a) + px).$$

We are now at the final step, and by reference to the left additive identity of the underlying ring (+Lid), we finally get our sought after expression:

$$q \cdot (a + px).$$

Right identity We also prove the existence of a right identity for polynomial multiplication by first proving the existence of a left identity and then applying the commutativity property of multiplication. The proof for the existence of a left identity is as follows:

```
Poly*Lid : ∀ q → 1P Poly* q ≡ q
Poly*Lid =
  ElimProp.f (λ q → 1P Poly* q ≡ q)
             drop0
             (λ r p prf → lemma r p)
```

```
(λ x y → isSetPoly _ _ x y)
  where
  lemma : ∀ r p → 1r · r + 0r :: (1r PolyConst* p) ≡ r :: p
  lemma =
    λ r p → 1r · r + 0r :: (1r PolyConst* p) ≡⟨ cong (_:: (1r PolyConst* p) )
                                                  (+Rid (1r · r)) ⟩
            1r · r :: (1r PolyConst* p) ≡⟨ cong (_:: 1r PolyConst* p) (·Lid r) ⟩
            r :: (1r PolyConst* p) ≡⟨ cong (r ::_) (PolyConst*Lid p) ⟩
            r :: p ∎
```

Code snippet 15: Left identity for multiplication

We use a local lemma for the inductive part, using the additive right identities and multiplicative left identities of the underlying ring, as well as the lemma PolyConst*Lid, providing us with the identity

$$1_R * q \equiv q$$

again making explicit the relation between the embedding of the underlying ring within the polynomials.

Associativity Finally we prove the associativity of polynomial multiplication:

```
Poly*Associative : ∀ p q r → p Poly* (q Poly* r) ≡   (p Poly* q) Poly* r
Poly*Associative =
  ElimProp.f (λ p → ∀ q r → p Poly* (q Poly* r) ≡ (p Poly* q) Poly* r )
             (λ _ _ → refl)
             (λ a p prf q r →
               ((a :: p) Poly* (q Poly* r)) ≡⟨
                                              cong (_Poly+ (0r :: (p Poly* (q Poly* r))))
                                                   (PolyConst*AssocPoly* a q r)
                                              ⟩
               (((a PolyConst* q) Poly* r) Poly+
                 (0r :: (p Poly* (q Poly* r)))) ≡⟨
                                                 sym (cong (((a PolyConst* q) Poly* r) Poly+_)
                                                          (cong (_:: (p Poly* (q Poly* r)))
                                                                (+Lid 0r)))
                                                 ⟩
               (((a PolyConst* q) Poly* r) Poly+
                 (0r + 0r :: (p Poly* (q Poly* r)))) ≡⟨
                                                 cong (((a PolyConst* q) Poly* r) Poly+_)
                                                      (cong (0r + 0r ::_)
                                                            (sym (Poly+Lid (p Poly* (q Poly* r)))))
                                                      ⟩
               (((a PolyConst* q) Poly* r) Poly+
                 (0r + 0r :: ([] Poly+ (p Poly* (q Poly* r))))) ≡⟨
                                                        cong (((a PolyConst* q) Poly* r) Poly+_)
                                                             (cong (0r + 0r ::_)
                                                                   (cong ([] Poly+_)
                                                                         (prf q r)))
                                                                   ⟩
               (((a PolyConst* q) Poly* r) Poly+
                 (0r + 0r :: ([] Poly+ ((p Poly* q) Poly* r)))) ≡⟨
                                                   cong (((a PolyConst* q) Poly* r) Poly+_)
                                                        (cong (_Poly+ (0r :: ((p Poly* q) Poly* r)))
                                                              (sym (0rLeftAnnihilatesPoly r)))
                                                                   ⟩
               (((a PolyConst* q) Poly* r) Poly+
                 ((0r PolyConst* r) Poly+ (0r :: ((p Poly* q) Poly* r)))) ≡⟨
                                                             sym (Poly*RDistrPoly+ (a PolyConst* q)
                                                                                   (0r :: (p Poly* q)) r)
                                                                                   ⟩
               ((((a :: p) Poly* q) Poly* r)) ∎)
             (λ x y i q r → isSetPoly _ _ (x q r) (y q r) i)
```

Code snippet 16: Associativity of multiplication

The proof uses the associativity of constant multiplication in relation to polynomial multiplication, as well as additive left identities both for the underlying ring and for the polynomial ring. We also use a lemma for right distributivity of multiplication over addition (which in turn is proved using the corresponding left distributivity shown in the next section) for the final step.

4.2.3 Properties Relating Addition and Multiplication

Left Distributivity of Multiplication Over Addition The left distributivity of polynomial multiplication over polynomial addition is the property we need that relate multiplication and addition. We will provide a detailed presentation in addition to the formal proof:

```
Poly*LDistrPoly+ : ∀ p q r → p Poly* (q Poly+ r) ≡ (p Poly* q) Poly+ (p Poly* r)
Poly*LDistrPoly+ =
  ElimProp.f
    (λ p → ∀ q r → p Poly* (q Poly+ r) ≡ (p Poly* q) Poly+ (p Poly* r))
    (λ _ _ → refl)
    (λ a p prf q r → ((a PolyConst* (q Poly+ r)) Poly+
                       (0r ::(p Poly*(q Poly+ r)))) ≡⟨
                                              cong (_Poly+ (0r :: (p Poly* (q Poly+ r))))
                                                   (PolyConst*LDistrPoly+ a q r)
                                              ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: (p Poly* (q Poly+ r)))) ≡⟨
                                 cong (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+_)
                                      (cong (0r ::_) (prf q r))
                                 ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: ((p Poly* q) Poly+ (p Poly* r)))) ≡⟨
                                          cong (((a PolyConst* q) Poly+
                                                 (a PolyConst* r)) Poly+_)
                                          (XLDistrPoly+ (p Poly* q) (p Poly* r))
                                          ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      ((0r :: (p Poly* q)) Poly+ (0r :: (p Poly* r)))) ≡⟨
                                                 Poly+Assoc ((a PolyConst* q) Poly+
                                                             (a PolyConst* r))
                                                            (0r :: (p Poly* q))
                                                            (0r :: (p Poly* r))
                                                 ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: (p Poly* q))) Poly+ (0r :: (p Poly* r)) ≡⟨ cong (_Poly+ (0r :: (p Poly* r)))
                                                      (sym (Poly+Assoc (a PolyConst* q)
                                                                       (a PolyConst* r)
                                                                       (0r :: (p Poly* q))))
                                                ⟩
(((a PolyConst* q) Poly+ ((a PolyConst* r) Poly+
  (0r :: (p Poly* q)))) Poly+ (0r :: (p Poly* r))) ≡⟨
                                         cong (_Poly+ (0r :: (p Poly* r)))
                                           (cong ((a PolyConst* q) Poly+_)
                                              (Poly+Comm (a PolyConst* r)
                                                    (0r :: (p Poly* q))))
                                         ⟩
(((a PolyConst* q) Poly+ ((0r :: (p Poly* q)) Poly+
  (a PolyConst* r))) Poly+ (0r :: (p Poly* r))) ≡⟨
                                        cong (_Poly+ (0r :: (p Poly* r)))
                                          (Poly+Assoc (a PolyConst* q)
                                                 (0r :: (p Poly* q))
                                                 (a PolyConst* r))
                                          ⟩
((((a PolyConst* q) Poly+ (0r :: (p Poly* q))) Poly+
  (a PolyConst* r)) Poly+ (0r :: (p Poly* r))) ≡⟨
                                        sym (Poly+Assoc ((a PolyConst* q) Poly+
                                                  (0r :: (p Poly* q)))
                                                 ((a PolyConst* r))
                                                 ((0r :: (p Poly* r))))
                                        ⟩
((a PolyConst* q) Poly+ (0r :: (p Poly* q))) Poly+
  ((a PolyConst* r) Poly+ (0r :: (p Poly* r))) ■)
(λ x y i q r → isSetPoly _ _ (x q r) (y q r) i)
```

Code snippet 17: Left distributivity of multiplication over addition

The proof relies on using the earlier proved properties of associativity and commutativity for polynomial addition. We need an additional lemma, XLDistrPoly+, Which provides the following equality:

$$0_R :: (p +_P q) \equiv (0_R :: p) +_P (0_R :: q),$$

or in more familiar notation:

$$(p + q)x \equiv (px) + (qx).$$

Interestingly enough, this proof was relatively easy, even though one might expect that relating the two operations would present the most difficulty. The type we want to instantiate is

$$\forall\ p\ q\ r \to p\ \text{Poly*}\ (q\ \text{Poly+}\ r) \equiv (p\ \text{Poly*}\ q)\ \text{Poly+}\ (p\ \text{Poly*}\ r)$$

We will do this by one usage of ElimProp.f – eliminating on p, then using the standard equational reasoning.

The case for [] is simple enough: the type needed is $(q, r : \text{Poly}) \to [] \equiv []$. We see that the two polynomial arguments are irrelevant to the equality needed, and that the equality itself is just reflexivity. So we simply discard the arguments and supply refl:

$$\lambda\ \_\ \_ \to \text{refl}$$

The second (inductive) case is more difficult. Given $a \in R$ and polynomials p, q, r and an inductive hypothesis *prf*, we want to show

$$(a * (q + r)) + (p \cdot (q + r))x = ((a * q) + ((p \cdot q)x)) + ((a * r) + (p \cdot r)x)$$

The first step is to left distribute a over $(q+r)$. Note that this is not circular, since technically, polynomial multiplication and constant multiplication are two separate operations in our formalisation. We proved an additional lemma, PolyConst*LDistrPoly+, showing that constant multiplication indeed left distributes over addition, so we get:

$$((a * q) + (a * r)) + (p \cdot (q + r))x.$$

Now, at this stage we can invoke the inductive hypothesis to distribute p over $q + r$:

$$((a * q) + (a * r)) + ((p \cdot q) + (p \cdot r))x$$

The next step requires an additional lemma, XLDistrPoly+, which says that we can distribute multiplication with the indeterminate over addition (so again, we avoid circularity for technical reasons. In the formalisation, multiplication with a polynomial and multiplication with the indeterminate are two different things – the first being $Poly*$, the second being the usage of appending 0_R using the ::-operator). If we proceed with doing this on the second term, we get

$$((a * q) + (a * r)) + ((p \cdot q)x + (p \cdot r)x)$$

Now we are almost done. We will need to apply commutativity of addition to $(a * r)$ and $(p \cdot q)x$ to switch them around. A slight complication is the explicit handling of parentheses that must be done. Two applications of associativity give us

$$((a * q) + ((a * r) + ((p \cdot q)x))) + (p \cdot r)x$$

so now we can apply commutativity, yielding

$$((a * q) + (((p \cdot q)x) + (a * r))) + (p \cdot r)x,$$

and the only thing left is to reverse the two applications of associativity that we just performed.

The third and final case to show is that the type is a proposition. We have a few parameters: the polynomials p, q, r, an interval i and finally two equalities with two polynomial parameters, q', r', yielding the equality type we are trying to prove:

$$(p \cdot (q' + r')) \equiv ((p \cdot q') + (p \cdot r'))$$

We use the same approach as always to do this, by means of isSetPoly, using Agda's proof assistance. We discard two elements and provide the two equalities applied to q and r as the two proofs we want to show equal. We get:

$$x\ r\ q \equiv y\ r\ q$$

so we only need to apply to the interval element i to get it to get the proper form.

4.2.4 Putting It All Together

Finally, we prove that the resulting data type is a commutative ring by using the constructor for commutative rings, CommRingStr, and then simply plugging in the corresponding types, elements, operations and propositions:

```
Poly : (CommRing ℓ) → CommRing ℓ
Poly R = (PolyMod.Poly R) , str
  where
    open CommRingStr
    str : CommRingStr (PolyMod.Poly R)
    0r str = PolyMod.0P R
    1r str = PolyMod.1P R
    _+_ str = PolyMod._Poly+_ R
    _·_ str = PolyMod._Poly*_ R
    - str = PolyMod.Poly- R
    isCommRing str = makeIsCommRing (PolyMod.isSetPoly R)
                                    (PolyMod.Poly+Assoc R)
                                    (PolyMod.Poly+Rid R)
                                    (PolyMod.Poly+Inverses R)
                                    (PolyMod.Poly+Comm R)
                                    (PolyMod.Poly*Associative R)
                                    (PolyMod.Poly*Rid R)
                                    (PolyMod.Poly*LDistrPoly+ R)
                                    (PolyMod.Poly*Commutative R)
```

Code snippet 18: Instantiation of polynomials as a commutative ring

5 Discussion And Conclusion

The formalisation was successful, and so the last hole that we fill is that of polynomials over commutative rings in the library of Cubical Agda. There were some initial difficulties with the list-based approach that made the proofs far more extensive than they needed to be. Indeed, this can be seen a priori, since if we have a definition of an n-ary operation involving k operands, we get k^n cases in the worst case. The use of ElimProp.f mitigates the combinatoric explosion somewhat. In our case, we had a 2-ary operation with 4 constructors, giving rise to 16 cases.

The way to tackle this was to introduce a completely separate, functional definition of polynomials, prove the required property for this structure, and then prove the equivalence of the two definitions – thus enabling us to completely remove one of the constructors. This turned out to be somewhat more complicated than perhaps initially thought, and while most of the statements needed were simple enough to be proved by the author of this thesis, the polyEq-proposition was too complicated. Fortunately, the thesis supervisor Anders Mörtberg agreed that this part went out of scope of a bachelors thesis, and proved it himself together with Evan Cavallo.

The proofs about the ring properties were in general not hard, as long as some care was taken in dividing and conquering the type goals into separate lemmas when appropriate. One could have looked closer at proofs presented in mathematical texts, and followed those closely. But the approach taken here was to only use (Cubical) Agda as a guide. The key component of this approach was the eliminator module suggested by Anders Mörtberg – much of the initial confusion could then be dispelled, since the approach now closely mirrored what you would do on pen and paper, the main difference being the handling of the drop0-constructor, which required some perhaps more abstract thought. Though as indicated above, many of the details of those cases were fortunately taken care of by Agda itself.

One of the most interesting aspects of working with Agda is the proof assistance that one gets as described in section 3.1. In some sense, one doesn't need to understand much type theory before starting with programming in Agda – As long as one understands some of the basic functionality (such as cong, sym, how to handle type holes and the equational reasoning-operators) a programmer can tackle a problem in a sort of gamified way. Even though one does not perhaps completely understand type theory, the mind is able to tackle these problems using common sense in combination with the proof assistant and its documentation. And while doing so, you get more and more intuition that helps you understand the deeper aspects of (in this case, cubical) type theory.

For programming purposes, the above remark highlights some potential: A supervisor can provide code skeletons in the form of type signatures, and let juniors work on the specific cases, providing assistance, explanation, and the occasional implementation (when a case is deemed too hard for the junior programmer). Of course, the main charm of Cubical Agda is the constructivity and provable correctness that follows from the approach of cubical type theory. In practice one will have to deal with side effects, but taking a structured approach, with the ambition of writing everything in Cubical Agda that can be written in Cubical Agda, would likely make for quite robust code – without the added work of applying a separate modelling and/or proof language and verifying the code properties by more traditional approaches. This seems to be a key advantage of having the programming language and specification language essentially being one and the same. Additionally, the fact that the syntax of the programming language so closely mirrors a mathematical foundational theory gives a certain conceptual clarity when programming that is hard to achieve using more conventional programming languages. Indeed, this also minimises the mental effort needed for "switching contexts" in the sense of switching between code and your sketches (on paper or otherwise). So programming becomes a matter of filling in the details, avoiding the additional translation needed between different theories/notations.

5.1 Further Research

All definitions of the operators follow the naïve approach and as such, they have unnecessarily high time complexity. To facilitate more practical usage, other algorithms should be implemented for the cubical mode (such as the Karatsuba algorithm for multiplication, see further Meshveliani [12] for an implementation in regular Agda) and then proven to be equivalent to the naïve definitions given here. This is also necessary for the wider adoption of the language.

Some investigation into the alternative function-based definition could be carried out in the form of implementing the file using it directly, and then comparing the difficulties of the proofs to the ones done here. It might seem redundant, but perhaps we can learn something about the perceived difficulty of alternative definitions – an experience that would be of use in the future when deciding on approaches for implementations of other mathematical structures for the library.

The dramatic decrease in difficulty that ElimProp.f provided suggests that perhaps it could be a good idea, if possible, to create a more generalised variant for the library, so that we don't need to recreate eliminators from scratch for each data type. If not possible in general, then perhaps it is at least possible for certain fragments.

Some pedagogical research could also be carried out regarding the perceived difficulty/intuitiveness of using type theory as your mathematical foundation. Set theory can perhaps feel quite intuitive in comparison, but then again most who encounter type theory have already been exposed to set theory from an earlier age. What would the experience be if one encounters type theory and lambda calculus earlier in the curriculum? Perhaps one could even present simplified versions (focusing on the more concrete and mechanical aspects) in high school or earlier.

5.2 Conclusion

We have formalised polynomials over commutative rings by means of a list-based approach, using higher inductive types. We have provided an alternative function-based formalisation. Using the list-based formalisation, we have defined polynomial addition and multiplication, and proved multiple statements about these, culminating in the result of showing that the mathematical structure resulting from these operators and the set of polynomials over a commutative ring, is itself a commutative ring. Everything is proven correct and type-checked and the final version of the code has been provided to the cubical library on Github [16].

References

[1] Cubical agda github repository. `https://github.com/agda/cubical`, 2021.

[2] Agda documentation. `https://agda.readthedocs.io/en/v2.6.2.1/`, 2022.

[3] Cubical agda documentation. `https://agda.readthedocs.io/en/v2.6.0/language/cubical.html`, 2022.

[4] S. Awodey, N. Gambino, and E. Palmgren. Introduction – from type theory and homotopy theory to univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1005–1009, 2015. doi: 10.1017/S0960129514000474.

[5] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. 2016. doi: 10.48550/ARXIV.1611.02108. URL `https://arxiv.org/abs/1611.02108`.

[6] C. Coquand and T. Coquand. Structured type theory. *In Proceedings Workshop on Logical Frameworks and Meta-languages (LFM'99)*, 1999.

[7] D. S. Dummit and R. Foote. *Abstract algebra.* Wiley, Hoboken, N.J., 3. ed.. edition, 2004. ISBN 978-0-471-43334-7.

[8] N. Jacobson. *Basic Algebra I: Second Edition.* Dover Books on Mathematics. Dover Publications, 2012. ISBN 9780486135229.

[9] F. Kamareddine. *A Modern Perspective on Type Theory From its Origins until Today.* Applied Logic Series, 29. 1st ed. 2005.. edition, 2005. ISBN 1-280-61676-8.

[10] H. Karnick. Lisp. *Resonance*, 19(3):208–221, 2014. ISSN 0971-8044.

[11] P. Martin-Löf. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[12] S. D. Meshveliani. A certified program for the karatsuba method to multiply polynomials. *Programming and computer software*, 48(1):1–18, 2022. ISSN 0361-7688.

[13] U. Norell. Towards a practical programming language based on dependent type theory. Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie Nr 2677, 2007. ISBN 978-91-7291-996-9.

[14] the HoTT Book. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[15] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3 (ICFP), jul 2019. doi: 10.1145/3341691. URL `https://doi.org/10.1145/3341691`.

[16] C. Åkerman Rydbeck. Git commit: Polynomials over commutative rings added. `https://github.com/agda/cubical/commit/e93eec912fe75e8242b47c3336410d10baa9bbaa`, 2022.

Appendix Polynomials.agda

```
{-A Polynomials over commutative rings-}
{-# OPTIONS --safe #-}

----------------------------------

module Cubical.Algebra.Polynomials where

open import Cubical.HITs.PropositionalTruncation
open import Cubical.Foundations.Prelude
open import Cubical.Foundations.HLevels

open import Cubical.Data.Sigma
open import Cubical.Data.Nat renaming (_+_ to _Nat+_; _·_ to _Nat·_) hiding (·-comm)
open import Cubical.Data.Nat.Order
open import Cubical.Data.Empty.Base renaming (rec to ⊥rec )
open import Cubical.Data.Bool

open import Cubical.Algebra.Group hiding (Bool)
open import Cubical.Algebra.Ring
open import Cubical.Algebra.CommRing

------------------------------------------------------------------------------------

private
  variable
    ℓ ℓ' : Level
    A : Type ℓ

module PolyMod (R' : CommRing ℓ) where
  private
    R = fst R'
  open CommRingStr (snd R') public

-------------------------------------------------------------------------------------------
-- First definition of a polynomial.
-- A polynomial a₁ +  a₂x + ... + aⱼxʲ of degree j is represented as a list [a₁, a₂, ...,aⱼ]
-- modulo trailing zeros.
-------------------------------------------------------------------------------------------

  data Poly : Type ℓ where
    [] : Poly
    _::_ : (a : R) → (p : Poly) → Poly
    drop0 : 0r :: [] ≡ []

  infixr 5 _::_


  module Elim (B       : Poly → Type ℓ')
              ([]*     : B [])
              (cons*   : (r : R) (p : Poly) (b : B p) → B (r :: p))
              (drop0* : PathP (λ i → B (drop0 i)) (cons* 0r [] []*) []*) where

    f : (p : Poly) → B p
    f [] = []*
    f (x :: p) = cons* x p (f p)
    f (drop0 i) = drop0* i


  -- Given a proposition (as type) Φ ranging over polynomials, we prove it by:
  -- ElimProp.f Φ ⌜proof for base case []⌝ ⌜proof for induction case a :: p⌝
  --           ⌜proof that Φ actually is a proposition over the domain of polynomials⌝
  module ElimProp (B : Poly → Type ℓ')
                  ([]* : B [])
                  (cons* : (r : R) (p : Poly) (b : B p) → B (r :: p))
                  (BProp : {p : Poly} → isProp (B p)) where
    f : (p : Poly) → B p
  f = Elim.f B []* cons* (toPathP (BProp (transport (λ i → B (drop0 i)) (cons* 0r [] []*)) []*))

 module Rec (B : Type ℓ')
            ([]* : B)
            (cons* : R → B → B)
            (drop0* : cons* 0r []* ≡ []*)
            (Bset : isSet B) where
  f : Poly → B
  f = Elim.f (λ _ → B) []* (λ r p b → cons* r b) drop0*


 module RecPoly ([]* : Poly) (cons* : R → Poly → Poly) (drop0* : cons* 0r []* ≡ []*) where
  f : Poly → Poly
  f [] = []*
  f (a :: p) = cons* a (f p)
  f (drop0 i) = drop0* i


-----------------------------------------------------------------------------------------------------
-- Second definition of a polynomial. The purpose of this second definition is to
-- facilitate the proof that the first definition is a set. The two definitions are
-- then shown to be equivalent.
-- A polynomial a₀ +  a₁x + ... + aⱼxʲ of degree j is represented as a function f
-- such that for i ∈ ℕ we have  f(i) = aᵢ if i ≤ j, and 0 for i > j
-----------------------------------------------------------------------------------------------------

 PolyFun : Type ℓ
 PolyFun = Σ[ p ∈ (ℕ → R) ] (∃[ n ∈ ℕ ] ((m : ℕ) → n ≤ m → p m ≡ 0r))


 isSetPolyFun : isSet PolyFun
 isSetPolyFun = isSetΣSndProp (isSetΠ (λ x → isSetCommRing R')) λ f x y → squash x y



 --construction of the function that represents the polynomial
 Poly→Fun : Poly → (ℕ → R)
 Poly→Fun [] = (λ _ → 0r)
 Poly→Fun (a :: p) = (λ n → if isZero n then a else Poly→Fun p (predℕ n))
 Poly→Fun (drop0 i) = lemma i
  where
  lemma : (λ n → if isZero n then 0r else 0r) ≡ (λ _ → 0r)
  lemma i zero = 0r
  lemma i (suc n) = 0r

 Poly→Prf : (p : Poly) → ∃[ n ∈ ℕ ] ((m : ℕ) → n ≤ m → (Poly→Fun p m ≡ 0r))
 Poly→Prf = ElimProp.f (λ p → ∃[ n ∈ ℕ ] ((m : ℕ) → n ≤ m → (Poly→Fun p m ≡ 0r)))
                       ∣ 0 , (λ m ineq → refl) ∣
                       (λ r p → map ( λ (n , ineq) → (suc n) ,
                                      λ { zero h → ⊥rec (znots (sym (≤0→≡0 h))) ;
                                          (suc m) h → ineq m (pred-≤-pred h)
                                        }
                                    )
                       )
                       squash

 Poly→PolyFun : Poly → PolyFun
 Poly→PolyFun p = (Poly→Fun p) , (Poly→Prf p)


----------------------------------------------------
-- Start of code by Anders Mörtberg and Evan Cavallo


 at0 : (ℕ → R) → R
 at0 f = f 0

 atS : (ℕ → R) → (ℕ → R)
atS f n = f (suc n)

polyEq : (p p' : Poly) → Poly→Fun p ≡ Poly→Fun p' → p ≡ p'
polyEq [] [] _ = refl
polyEq [] (a :: p') α =
  sym drop0 ∙∙ cong₂ _::_ (cong at0 α) (polyEq [] p' (cong atS α)) ∙∙ refl
polyEq [] (drop0 j) α k =
  hcomp
    (λ l → λ
      { (j = i1) → drop0 l
      ; (k = i0) → drop0 l
      ; (k = i1) → drop0 (j ∧ l)
      })
    (is-set 0r 0r (cong at0 α) refl j k :: [])
polyEq (a :: p) [] α =
  refl ∙∙ cong₂ _::_ (cong at0 α) (polyEq p [] (cong atS α)) ∙∙ drop0
polyEq (a :: p) (a₁ :: p') α =
  cong₂ _::_ (cong at0 α) (polyEq p p' (cong atS α))
polyEq (a :: p) (drop0 j) α k =
  hcomp -- filler
    (λ l → λ
      { (k = i0) → a :: p
      ; (k = i1) → drop0 (j ∧ l)
      ; (j = i0) → at0 (α k) :: polyEq p [] (cong atS α) k
      })
    (at0 (α k) :: polyEq p [] (cong atS α) k)
polyEq (drop0 i) [] α k =
  hcomp
    (λ l → λ
      { (i = i1) → drop0 l
      ; (k = i0) → drop0 (i ∧ l)
      ; (k = i1) → drop0 l
      })
    (is-set 0r 0r (cong at0 α) refl i k :: [])
polyEq (drop0 i) (a :: p') α k =
  hcomp -- filler
    (λ l → λ
      { (k = i0) → drop0 (i ∧ l)
      ; (k = i1) → a :: p'
      ; (i = i0) → at0 (α k) :: polyEq [] p' (cong atS α) k
      })
    (at0 (α k) :: polyEq [] p' (cong atS α) k)
polyEq (drop0 i) (drop0 j) α k =
  hcomp
    (λ l → λ
      { (k = i0) → drop0 (i ∧ l)
      ; (k = i1) → drop0 (j ∧ l)
      ; (i = i0) (j = i0) → at0 (α k) :: []
      ; (i = i1) (j = i1) → drop0 l
      })
    (is-set 0r 0r (cong at0 α) refl (i ∧ j) k :: [])


PolyFun→Poly+ : (q : PolyFun) → Σ[ p ∈ Poly ] Poly→Fun p ≡ q .fst
PolyFun→Poly+ (f , pf) = rec lem (λ x → rem1 f (x .fst) (x .snd) ,
                                              funExt (rem2 f (fst x) (snd x))
                                   ) pf
  where
  lem : isProp (Σ[ p ∈ Poly ] Poly→Fun p ≡ f)
  lem (p , α) (p' , α') =
    ΣPathP (polyEq p p' (α ∙ sym α'), isProp→PathP (λ i → (isSetΠ λ _ → is-set) _ _) _ _)

  rem1 : (p : ℕ → R) (n : ℕ) → ((m : ℕ) → n ≤ m → p m ≡ 0r) → Poly
  rem1 p zero h = []
  rem1 p (suc n) h = p 0 :: rem1 (λ x → p (suc x)) n (λ m x → h (suc m) (suc-≤-suc x))

  rem2 : (f : ℕ → R) (n : ℕ) → (h : (m : ℕ) → n ≤ m → f m ≡ 0r) (m : ℕ) →
                                                            Poly→Fun (rem1 f n h) m ≡ f m
    rem2 f zero h m = sym (h m zero-≤)
    rem2 f (suc n) h zero = refl
    rem2 f (suc n) h (suc m) = rem2 (λ x → f (suc x)) n (λ k p → h (suc k) (suc-≤-suc p)) m

  PolyFun→Poly : PolyFun → Poly
  PolyFun→Poly q = PolyFun→Poly+ q .fst

  PolyFun→Poly→PolyFun : (p : Poly) → PolyFun→Poly (Poly→PolyFun p) ≡ p
  PolyFun→Poly→PolyFun p = polyEq _ _ (PolyFun→Poly+ (Poly→PolyFun p) .snd)



--End of code by Mörtberg and Cavallo
-------------------------------------

  isSetPoly : isSet Poly
  isSetPoly = isSetRetract Poly→PolyFun
                           PolyFun→Poly
                           PolyFun→Poly→PolyFun
                           isSetPolyFun

-------------------------------------------------
-- The rest of the file uses the first definition
-------------------------------------------------


  open CommRingTheory R'
  open RingTheory (CommRing→Ring R')
  open GroupTheory (Ring→Group (CommRing→Ring R'))


  pattern [_] x = x :: []


---------------------------------------
-- Definition
-- Identity for addition of polynomials
---------------------------------------

  0P : Poly
  0P = []


  --ReplicatePoly(n,p) returns 0 :: 0 :: ... :: [] (n zeros)
  ReplicatePoly0 : (n : ℕ)      → Poly
  ReplicatePoly0 zero       = 0P
  ReplicatePoly0 (suc n) = 0r :: ReplicatePoly0 n


  --The empty polynomial has multiple equal representations on the form 0 + 0x + 0 x² + ...
  replicatePoly0Is0P : ∀ (n : ℕ) → ReplicatePoly0 n ≡ 0P
  replicatePoly0Is0P zero = refl
  replicatePoly0Is0P (suc n) = (cong (0r ::_) (replicatePoly0Is0P n)) ∙ drop0


-----------------------------
-- Definition
-- subtraction of polynomials
-----------------------------

  Poly- : Poly → Poly
  Poly- [] = []
  Poly- (a :: p) = (- a) :: (Poly- p)
  Poly- (drop0 i) = (cong (_:: []) (inv1g) ∙ drop0) i


  -- Double negation (of subtraction of polynomials) is the identity mapping
  Poly-Poly- : (p : Poly) → Poly- (Poly- p) ≡ p
  Poly-Poly- = ElimProp.f (λ x → Poly- (Poly- x) ≡ x)
```

```
                                    refl
                                    (λ a p e → cong (_:: (Poly- (Poly- p)))
                                                     (-Idempotent a) · cong (a ::_ ) (e))
                                    (isSetPoly _ _)


----------------------------
-- Definition
-- addition for polynomials
----------------------------

  _Poly+_ : Poly → Poly → Poly
  p Poly+ [] = p
  [] Poly+ (drop0 i) = drop0 i
  [] Poly+ (b :: q) = b :: q
  (a :: p) Poly+ (b :: q) = (a + b) :: (p Poly+ q)
  (a :: p) Poly+ (drop0 i) = +Rid a i :: p
  (drop0 i) Poly+ (a :: q) = lem q i      where
                              lem : ∀ q → (0r + a) :: ([] Poly+ q) ≡ a :: q
                              lem = ElimProp.f (λ q → (0r + a) :: ([] Poly+ q) ≡ a :: q)
                                                   (λ i → (+Lid a i :: []))
                                                   (λ r p _ → λ i → +Lid a i :: r :: p )
                                                   (isSetPoly _ _)
  (drop0 i) Poly+ (drop0 j) =  isSet→isSet' isSetPoly        (cong ([_] ) (+Rid 0r)) drop0
                                                              (cong ([_] ) (+Lid 0r)) drop0 i j

  -- [] is the left identity for Poly+
  Poly+Lid : ∀ p → ([] Poly+ p ≡ p)
  Poly+Lid =  ElimProp.f (λ p → ([] Poly+ p ≡ p) )
                          refl
                          (λ r p prf → refl)
                          (λ x y → isSetPoly _ _ x y)


  -- [] is the right identity for Poly+
  Poly+Rid : ∀ p → (p Poly+ [] ≡ p)
  Poly+Rid p = refl



  --Poly+ is Associative
  Poly+Assoc : ∀ p q r → p Poly+ (q Poly+ r) ≡ (p Poly+ q) Poly+ r
  Poly+Assoc =
    ElimProp.f (λ p → (∀ q r → p Poly+ (q Poly+ r) ≡ (p Poly+ q) Poly+ r))
               (λ q r → Poly+Lid (q Poly+ r) · cong (_Poly+ r)       (sym (Poly+Lid q)))
               (λ a p prf → ElimProp.f ((λ q → ∀ r → ((a :: p) Poly+ (q Poly+ r)) ≡
                                                  (((a :: p) Poly+ q) Poly+ r)))
                                        (λ r → cong ((a :: p) Poly+_) (Poly+Lid r))
                                        (λ b q prf2 →
                                        ElimProp.f
                                          (λ r → ((a :: p) Poly+ ((b :: q) Poly+ r)) ≡
                                                 ((a + b :: (p Poly+ q)) Poly+ r))
                                          refl
                                          (λ c r prfp → cong ((a + (b + c))::_)
                                                              (prf q r) ·
                                                              (cong (_:: ((p Poly+ q) Poly+ r))
                                                                     (+Assoc a b c)))
                                          (isSetPoly _ _))
                                        λ x y i r → isSetPoly (x r i0) (y r i1) (x r) (y r) i)
               λ x y i q r    → isSetPoly _ _ (x q r) (y q r) i

  -- for any polynomial, p, the additive inverse is given by Poly- p
  Poly+Inverses : ∀ p → p Poly+ (Poly- p) ≡ []
  Poly+Inverses = ElimProp.f ( λ p → p Poly+ (Poly- p) ≡ [])
                              refl --(Poly+Lid (Poly- []))
                              (λ r p prf → cong (r + - r ::_) prf ·
                                               (cong (_:: [])       (+Rinv r) · drop0))
                              (isSetPoly _ _)
```

```
 --Poly+ is commutative
 Poly+Comm : ∀ p q → p Poly+ q ≡ q Poly+ p
 Poly+Comm = ElimProp.f (λ p → (∀ q → p Poly+ q ≡ q Poly+ p))
                        (λ q → Poly+Lid q)
                        (λ a p prf → ElimProp.f (λ q → ((a :: p) Poly+ q) ≡ (q Poly+ (a :: p)))
                                                refl
                                                (λ b q prf2 → cong (_:: (p Poly+ q)) (+Comm a b) ·
                                                              cong ((b + a) ::_) (prf q))
                                                (isSetPoly _ _)
                        )
                        (λ {p} → isPropΠ (λ q → isSetPoly (p Poly+ q) (q Poly+ p)))

----------------------------------------------------------------
-- Definition
-- multiplication of a polynomial by a (constant) ring element
----------------------------------------------------------------

 _PolyConst*_ : (R) → Poly → Poly
 r PolyConst* [] = []
 r PolyConst* (a :: p) = (r · a) :: (r PolyConst* p)
 r PolyConst* (drop0 i) = lem r i where
                               lem : ∀ r → [ r · 0r ] ≡ []
                               lem =     λ r → [ r · 0r ] ≡⟨ cong (_:: []) (0RightAnnihilates r) ⟩
                                         [ 0r ] ≡⟨ drop0 ⟩
                                         [] ■

 -- For any polynomial p we have: 0 _PolyConst*_ p = []
 0rLeftAnnihilatesPoly : ∀ q → 0r PolyConst* q ≡ [ 0r ]
 0rLeftAnnihilatesPoly = ElimProp.f (λ q → 0r PolyConst* q ≡ [ 0r ])
                                    (sym drop0)
                                    (λ r p prf → cong ((0r · r) ::_) prf ·
                                                 cong (_:: [ 0r ]) (0LeftAnnihilates r) ·
                                                 cong (0r ::_) drop0 )
                                    λ x y → isSetPoly _ _ x y


 -- For any polynomial p we have: 1 _PolyConst*_ p = p
 PolyConst*Lid : ∀ q → 1r PolyConst* q ≡ q
 PolyConst*Lid = ElimProp.f (λ q → 1r PolyConst* q ≡ q ) refl
                            (λ a p prf → cong (_:: (1r PolyConst* p)) (·Lid a) ·
                                         cong (a ::_) (prf) )
                            λ x y → isSetPoly _ _ x y


--------------------------------
-- Definition
-- Multiplication of polynomials
--------------------------------

 _Poly*_ : Poly → Poly → Poly
 [] Poly* q = []
 (a :: p) Poly* q = (a PolyConst* q) Poly+ (0r :: (p Poly* q))
 (drop0 i) Poly* q = lem q i where
                           lem : ∀ q → (0r PolyConst* q) Poly+ [ 0r ] ≡ []
                           lem = λ q → ((0r PolyConst* q) Poly+ [ 0r ]) ≡⟨ cong ( _Poly+ [ 0r ] ) (0rLeftAnnihilatesPoly q)⟩
                                       ([ 0r ] Poly+ [ 0r ]) ≡⟨ cong (_:: []) 0Idempotent · drop0 ⟩
                                       [] ■

--------------------
--Definition
--Identity for Poly*
--------------------

 1P : Poly
 1P = [ 1r ]
```

```
-- For any polynomial p we have: p Poly* [] = []
0PRightAnnihilates : ∀ q → 0P Poly* q ≡ 0P
0PRightAnnihilates = ElimProp.f (λ q → 0P Poly* q ≡ 0P)
                                refl
                                (λ r p prf → prf)
                                λ x y → isSetPoly _ _ x y


-- For any polynomial p we have: [] Poly* p = []
0PLeftAnnihilates : ∀ p → p Poly* 0P ≡ 0P
0PLeftAnnihilates = ElimProp.f (λ p → p Poly* 0P ≡ 0P )
                               refl
                               (λ r p prf → cong (0r ::_) prf ∙ drop0)
                               λ x y → isSetPoly _ _ x y


-- For any polynomial p we have: p Poly* [ 1r ] = p
Poly*Lid : ∀ q → 1P Poly* q ≡ q
Poly*Lid =
  ElimProp.f (λ q → 1P Poly* q ≡ q)
             drop0
             (λ r p prf → lemma r p)
             (λ x y → isSetPoly _ _ x y)
               where
               lemma : ∀ r p → 1r · r + 0r :: (1r PolyConst* p) ≡ r :: p
               lemma =
                 λ r p → 1r · r + 0r :: (1r PolyConst* p) ≡⟨ cong (_:: (1r PolyConst* p) )
                                                              (+Rid (1r · r)) ⟩
                         1r · r :: (1r PolyConst* p) ≡⟨ cong (_:: 1r PolyConst* p) (·Lid r) ⟩
                         r :: (1r PolyConst* p) ≡⟨ cong (r ::_) (PolyConst*Lid p) ⟩
                         r :: p ∎


-- Distribution of indeterminate: (p + q)x = px + qx
XLDistrPoly+ : ∀ p q → (0r :: (p Poly+ q)) ≡ ((0r :: p) Poly+ (0r :: q))
XLDistrPoly+ =
  ElimProp.f (λ p → ∀ q → (0r :: (p Poly+ q)) ≡ ((0r :: p) Poly+ (0r :: q)) )
             (λ q → (cong (0r ::_) (Poly+Lid q)) ∙
                     cong (0r ::_) (sym (Poly+Lid q)) ∙
                     sym (cong (_:: [] Poly+ q) (+Lid 0r)))
             (λ a p prf → ElimProp.f (λ q → 0r :: ((a :: p) Poly+ q) ≡
                                         ((0r :: a :: p) Poly+ (0r :: q)))
                                       (cong (_:: a :: p ) (sym (+Lid 0r)))
                                       (λ b q prf2 → cong (_:: a + b :: (p Poly+ q)) (sym (+Lid 0r)))
                                       (λ x y i → isSetPoly (x i0) (x i1) x y i))
             (λ x y i q → isSetPoly (x q i0) (x q i1) (x q) (y q) i)


-- Distribution of a constant ring element over added polynomials p, q: a (p + q) = ap + aq
PolyConst*LDistrPoly+ : ∀ a p q → a PolyConst* (p Poly+ q) ≡
                                  (a PolyConst* p) Poly+ (a PolyConst* q)
PolyConst*LDistrPoly+ =
  λ a → ElimProp.f (λ p → ∀ q → a PolyConst* (p Poly+ q) ≡
                                 (a PolyConst* p) Poly+ (a PolyConst* q))
                   (λ q → cong (a PolyConst*_) (Poly+Lid q) ∙
                          (sym (Poly+Lid (a PolyConst* q))))
                   (λ b p prf → ElimProp.f (λ q → (a PolyConst* ((b :: p) Poly+ q)) ≡
                                       (a PolyConst* (b :: p)) Poly+ (a PolyConst* q))
                                refl
                                (λ c q prf2 → cong (_:: (a PolyConst* (p Poly+ q)))
                                                  (·Rdist+ a b c) ∙
                                              cong (a · b + a · c ::_) (prf q))
                                (isSetPoly _ _))
                   (λ x y i q    → isSetPoly (x q i0) (x q i1) (x q) (y q) i)


--Poly* left distributes over Poly+
Poly*LDistrPoly+ : ∀ p q r → p Poly* (q Poly+ r) ≡ (p Poly* q) Poly+ (p Poly* r)
Poly*LDistrPoly+ =
  ElimProp.f
    (λ p → ∀ q r → p Poly* (q Poly+ r) ≡ (p Poly* q) Poly+ (p Poly* r))
    (λ _ _ → refl)
    (λ a p prf q r → ((a PolyConst* (q Poly+ r)) Poly+
                       (0r ::(p Poly*(q Poly+ r)))) ≡⟨
                                                cong (_Poly+ (0r :: (p Poly* (q Poly+ r))))
                                                     (PolyConst*LDistrPoly+ a q r)
                                             ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: (p Poly* (q Poly+ r)))) ≡⟨
                                      cong (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+_)
                                           (cong (0r ::_) (prf q r))
                                    ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: ((p Poly* q) Poly+ (p Poly* r)))) ≡⟨
                                               cong (((a PolyConst* q) Poly+
                                                      (a PolyConst* r)) Poly+_)
                                               (XLDistrPoly+ (p Poly* q) (p Poly* r))
                                              ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      ((0r :: (p Poly* q)) Poly+ (0r :: (p Poly* r)))) ≡⟨
                                                        Poly+Assoc ((a PolyConst* q) Poly+
                                                                   (a PolyConst* r))
                                                                  (0r :: (p Poly* q))
                                                                  (0r :: (p Poly* r))
                                                        ⟩
    (((a PolyConst* q) Poly+ (a PolyConst* r)) Poly+
      (0r :: (p Poly* q))) Poly+ (0r :: (p Poly* r)) ≡⟨ cong (_Poly+ (0r :: (p Poly* r)))
                                                          (sym (Poly+Assoc (a PolyConst* q)
                                                                           (a PolyConst* r)
                                                                           (0r :: (p Poly* q))))
                                                     ⟩
    (((a PolyConst* q) Poly+ ((a PolyConst* r) Poly+
      (0r :: (p Poly* q)))) Poly+ (0r :: (p Poly* r))) ≡⟨
                                                        cong (_Poly+ (0r :: (p Poly* r)))
                                                             (cong ((a PolyConst* q) Poly+_)
                                                                  (Poly+Comm (a PolyConst* r)
                                                                             (0r :: (p Poly* q))))
                                                        ⟩
    (((a PolyConst* q) Poly+ ((0r :: (p Poly* q)) Poly+
      (a PolyConst* r))) Poly+ (0r :: (p Poly* r))) ≡⟨
                                                     cong (_Poly+ (0r :: (p Poly* r)))
                                                          (Poly+Assoc (a PolyConst* q)
                                                                      (0r :: (p Poly* q))
                                                                      (a PolyConst* r))
                                                          ⟩
    ((((a PolyConst* q) Poly+ (0r :: (p Poly* q))) Poly+
      (a PolyConst* r)) Poly+ (0r :: (p Poly* r))) ≡⟨
                                                    sym (Poly+Assoc ((a PolyConst* q) Poly+
                                                                    (0r :: (p Poly* q)))
                                                                   ((a PolyConst* r))
                                                                   ((0r :: (p Poly* r))))
                                                   ⟩
    ((a PolyConst* q) Poly+ (0r :: (p Poly* q))) Poly+
      ((a PolyConst* r) Poly+ (0r :: (p Poly* r))) ∎)
    (λ x y i q r → isSetPoly _ _ (x q r) (y q r) i)

-- The constant multiplication of a ring element, a, with a polynomial, p, can be
-- expressed by polynomial multiplication with the singleton polynomial [ a ]
PolyConst*r=Poly*[r] : ∀ a p → a PolyConst* p ≡ p Poly* [ a ]
PolyConst*r=Poly*[r] =
  λ a → ElimProp.f (λ p → a PolyConst* p ≡ p Poly* [ a ])
                   refl
                   ( λ r p prf →      a · r :: (a PolyConst* p) ≡⟨
                                                         cong (a · r ::_) prf
                                                         ⟩
```

```
                  a · r :: (p Poly* [ a ]) ≡⟨
                                          cong (a · r ::_)
                                               (sym (Poly+Lid (p Poly* [ a ])))
                                      ⟩
                  a · r :: ([] Poly+ (p Poly* [ a ])) ≡⟨
                                                     cong (_:: ([] Poly+ (p Poly* [ a ])))
                                                          (·Comm a r )
                                                    ⟩
                  r · a :: ([] Poly+ (p Poly* [ a ])) ≡⟨
                                                     cong (_:: ([] Poly+ (p Poly* [ a ])))
                                                          (sym (+Rid (r · a)))
                                                    ⟩
                  r · a + 0r :: ([] Poly+ (p Poly* [ a ])) ■)
               ( λ x y i → isSetPoly (x i0) (x i1) x y i)


-- Connection between the constant multiplication and the multiplication in the ring
PolyConst*Nested· : ∀ a b p → a PolyConst* (b PolyConst* p) ≡ (a · b) PolyConst* p
PolyConst*Nested· =
  λ a b → ElimProp.f (λ p → a PolyConst* (b PolyConst* p) ≡ (a · b) PolyConst* p)
                      refl
                      (λ c p prf → cong ((a · (b · c)) ::_) prf ·
                                   cong (_:: ((a · b) PolyConst* p)) (·Assoc a b c))
                      (isSetPoly _ _)


-- We can move the indeterminate from left to outside: px * q = (p * q)x
0r::LeftAssoc : ∀ p q → (0r :: p) Poly* q ≡ 0r :: (p Poly* q)
0r::LeftAssoc =
  ElimProp.f (λ p → ∀ q → (0r :: p) Poly* q ≡ 0r :: (p Poly* q))
             (λ q → cong (_Poly+ [ 0r ])((cong (_Poly+ []) (0rLeftAnnihilatesPoly q))) ·
                    cong (_:: []) (+Lid 0r))
             (λ r p b q → cong (_Poly+ (0r :: ((r PolyConst* q) Poly+ (0r :: (p Poly* q)))))
                               ((0rLeftAnnihilatesPoly q) · drop0))
             (λ x y i q → isSetPoly _ _ (x q) (y q) i)


--Associativity of constant multiplication in relation to polynomial multiplication
PolyConst*AssocPoly* : ∀ a p q → a PolyConst* (p Poly* q) ≡ (a PolyConst* p) Poly* q
PolyConst*AssocPoly* =
  λ a → ElimProp.f (λ p → ∀ q → a PolyConst* (p Poly* q) ≡ (a PolyConst* p) Poly* q)
                   (λ q → refl)
                   (λ b p prf q → a PolyConst* ((b PolyConst* q) Poly+
                                  (0r :: (p Poly* q))) ≡⟨
                                                         PolyConst*LDistrPoly+ a
                                                                               (b PolyConst* q)
                                                                               (0r :: (p Poly* q))
                                                        ⟩
                   (a PolyConst* (b PolyConst* q)) Poly+
                     (a PolyConst* (0r :: (p Poly* q))) ≡⟨
                                             cong (_Poly+ (a · 0r :: (a PolyConst* (p Poly* q))))
                                                  (PolyConst*Nested· a b q)
                                                        ⟩
                   ((a · b) PolyConst* q) Poly+
                     (a PolyConst* (0r :: (p Poly* q))) ≡⟨
                                                  cong (((a · b) PolyConst* q) Poly+_)
                                                        (cong (a · 0r      ::_)
                                                              (PolyConst*r=Poly*[r] a
                                                                                    (p Poly* q)))
                                                        ⟩
                   ((a · b) PolyConst* q) Poly+
                     (a · 0r :: ((p Poly* q) Poly* [ a ])) ≡⟨
                                                        cong (((a · b) PolyConst* q) Poly+_)
                                                             (cong (_:: ((p Poly* q) Poly* [ a ]))
                                                                   (0RightAnnihilates a)) ⟩
                   ((a · b) PolyConst* q) Poly+
                     (0r :: ((p Poly* q) Poly* [ a ])) ≡⟨
                                           cong (((a · b) PolyConst* q) Poly+_)
```

```
                                                  (cong (0r ::_)
                                                        (sym (PolyConst*r=Poly*[r] a (p Poly* q))))
                                                          ⟩
                    ((a · b) PolyConst* q) Poly+
                      (0r :: (a PolyConst* (p Poly* q))) ≡⟨
                                                           cong (((a · b) PolyConst* q) Poly+_)
                                                                (cong (0r ::_) (prf q))
                                                           ⟩
                    ((a · b) PolyConst* q) Poly+
                      (0r :: ((a PolyConst* p) Poly* q)) ∎)
                    (λ x y i q → isSetPoly (x q i0) (x q i1) (x q) (y q) i)


-- We can move the indeterminate from left to outside: p * qx = (p * q)x
0r::RightAssoc : ∀ p q → p Poly* (0r ::     q) ≡ 0r :: (p Poly* q)
0r::RightAssoc =
  ElimProp.f (λ p → ∀ q → p Poly* (0r ::     q) ≡ 0r :: (p Poly* q))
             (λ q → sym drop0)
             (λ a p prf q → ((a :: p) Poly* (0r :: q)) ≡⟨
                                                       cong ( a · 0r + 0r ::_)
                                                            (cong ((a PolyConst* q) Poly+_ )
                                                                  (prf q))
                                                       ⟩
             a · 0r + 0r :: ((a PolyConst* q) Poly+
               (0r :: (p Poly* q))) ≡⟨
                                     cong (_:: ((a PolyConst* q) Poly+ (0r :: (p Poly* q))))
                                          ((+Rid (a · 0r)))
                                    ⟩
             a · 0r :: ((a PolyConst* q) Poly+
               (0r :: (p Poly* q))) ≡⟨
                                    cong (_:: ((a PolyConst* q) Poly+ (0r :: (p Poly* q))))
                                         (0RightAnnihilates a) ⟩
             0r :: ((a PolyConst* q) Poly+ (0r :: (p Poly* q))) ∎)
             (λ x y i q     → isSetPoly (x q i0) (x q i1) (x q) (y q) i)


-- We can move the indeterminate around: px * q = p * qx
0r::Comm : ∀ p q → (0r :: p) Poly* q ≡ p Poly* (0r :: q)
0r::Comm = ElimProp.f (λ p → ∀ q → (0r :: p) Poly* q ≡ p Poly* (0r :: q))
                        (λ q → (cong ((0r PolyConst* q) Poly+_) drop0) ·
                                      0rLeftAnnihilatesPoly q ·
                                                           drop0   )
                        (λ a p prf q → ((0r :: a :: p) Poly* q) ≡⟨ 0r::LeftAssoc (a :: p) q ⟩
                                          0r :: ((a :: p) Poly* q) ≡⟨ sym (0r::RightAssoc (a :: p) q) ⟩
                                       ((a :: p) Poly* (0r :: q)) ∎
                      )
                      λ x y i q → isSetPoly (x q i0) (x q i1) (x q) (y q) i



--Poly* is commutative
Poly*Commutative : ∀ p q → p Poly* q ≡ q Poly* p
Poly*Commutative =
  ElimProp.f (λ p → ∀ q → p Poly* q ≡ q Poly* p)
             (λ q → sym (0PLeftAnnihilates q))
             (λ a p prf q → (a PolyConst* q) Poly+ (0r :: (p Poly* q)) ≡⟨
                                     cong ((a PolyConst* q) Poly+_)
                                          (cong (0r ::_) (prf q)) ⟩
             ((a PolyConst* q) Poly+ (0r :: (q Poly* p))) ≡⟨
                                                           cong ((a PolyConst* q) Poly+_)
                                                           (sym (0r::LeftAssoc q p))
                                                           ⟩
             ((a PolyConst* q) Poly+ ((0r :: q) Poly* p)) ≡⟨
                                    cong (_Poly+ ((0r PolyConst* p) Poly+ (0r :: (q Poly* p))))
                                         (PolyConst*r=Poly*[r] a q) ⟩
             ((q Poly* [ a ]) Poly+ ((0r :: q) Poly* p)) ≡⟨
                                                          cong ((q Poly* [ a ]) Poly+_)
                                                               (0r::Comm q p)
```

```
                                                    ⟩
          ((q Poly* [ a ]) Poly+ (q Poly* (0r :: p))) ≡⟨
                                                    sym (Poly*LDistrPoly+ q [ a ] (0r :: p))
                                                    ⟩
          (((q Poly* ([ a ] Poly+ (0r :: p))))) ≡⟨
                                              cong (q Poly*_)
                                                   (Poly+Comm [ a ] (0r :: p))
                                              ⟩
          ((q Poly* ((0r :: p) Poly+ [ a ]))) ≡⟨
                                            refl
                                            ⟩
          (q Poly* ((0r + a) :: p)) ≡⟨ cong (q Poly*_)
                                          (cong (_:: p) (+Lid a))
                                   ⟩
          (q Poly* (a :: p)) ■)
          (λ x y i q → isSetPoly _ _ (x q ) (y q) i)

--1P is the right identity of Poly*.
Poly*Rid : ∀ p → p Poly* 1P ≡ p
Poly*Rid = λ p → (Poly*Commutative p 1P ∙ Poly*Lid p)


--Polynomial multiplication right distributes over polynomial addition.
Poly*RDistrPoly+ : ∀ p q r → (p Poly+ q) Poly* r ≡ (p Poly* r) Poly+ (q Poly* r)
Poly*RDistrPoly+ = λ p q r → sym (Poly*Commutative r (p Poly+ q)) ∙
                                  Poly*LDistrPoly+ r p q ∙
                                  cong (_Poly+ (r Poly* q)) (Poly*Commutative r p) ∙
                                  cong ((p Poly* r) Poly+_) (Poly*Commutative r q)


--Polynomial multiplication is associative
Poly*Associative : ∀ p q r → p Poly* (q Poly* r) ≡       (p Poly* q) Poly* r
Poly*Associative =
  ElimProp.f (λ p → ∀ q r → p Poly* (q Poly* r) ≡   (p Poly* q) Poly* r )
          (λ _ _ → refl)
          (λ a p prf q r    →
            ((a :: p) Poly* (q Poly* r)) ≡⟨
                                          cong (_Poly+ (0r :: (p Poly* (q Poly* r))))
                                               (PolyConst*AssocPoly* a q r)
                                          ⟩
            (((a PolyConst* q) Poly* r) Poly+
              (0r :: (p Poly* (q Poly* r)))) ≡⟨
                                            sym (cong (((a PolyConst* q) Poly* r) Poly+_)
                                                      (cong (_:: (p Poly* (q Poly* r)))
                                                            (+Lid 0r)))
                                            ⟩
            (((a PolyConst* q) Poly* r) Poly+
              (0r + 0r :: (p Poly* (q Poly* r)))) ≡⟨
                                            cong (((a PolyConst* q) Poly* r) Poly+_)
                                                 (cong (0r + 0r ::_)
                                                       (sym (Poly+Lid (p Poly* (q Poly* r)))))
                                                 ⟩
            (((a PolyConst* q) Poly* r) Poly+
              (0r + 0r :: ([] Poly+ (p Poly* (q Poly* r))))) ≡⟨
                                                  cong (((a PolyConst* q) Poly* r) Poly+_)
                                                       (cong (0r + 0r ::_)
                                                             (cong ([] Poly+_)
                                                                   (prf q r)))
                                                       ⟩
            (((a PolyConst* q) Poly* r) Poly+
              (0r + 0r :: ([] Poly+ ((p Poly* q) Poly* r)))) ≡⟨
                                            cong (((a PolyConst* q) Poly* r) Poly+_)
                                                 (cong (_Poly+ (0r :: ((p Poly* q) Poly* r)))
                                                       (sym (0rLeftAnnihilatesPoly r)))
                                                       ⟩
            (((a PolyConst* q) Poly* r) Poly+
              ((0r PolyConst* r) Poly+ (0r :: ((p Poly* q) Poly* r)))) ≡⟨
                                             sym (Poly*RDistrPoly+ (a PolyConst* q)
                                                                              (0r :: (p Poly* q)) r)
                                                                              ⟩
            ((((a :: p) Poly* q) Poly* r)) ■)
          (λ x y i q r      → isSetPoly _ _ (x q r) (y q r) i)
```