A Polynomial Testing Principle

Conor McBride

Mathematically Structured Programming Group, University of Strathclyde
Livingstone Tower, 26 Richmond Street, Glasgow, G1 1XH, Scotland
`conor.mcbride@strath.ac.uk`

Abstract

Two polynomial functions of degree at most n agree on all inputs if they agree on $n+1$ different inputs, e.g., on $\{0, 1, 2, \ldots, n\}$. This fact gives us a simple procedure for testing equivalence in a language of polynomial expressions. Moreover, we may readily extend this language to include a summation operator and test standard results which are usually established inductively.

I present a short proof of this testing principle which rests on the construction and correctness of a syntactic forward-difference operator. This paper is a literate Agda file, containing just one

```
module TLCAPoly where
```

nothing is imported and there are 250 non-blank lines of code, all of which I show here.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases finite difference methods, testing coverage, mechanical theorem proving, dependently typed programming

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 In(tro)duction

Once upon a time in the late 1990s, I was a PhD student working for Rod Burstall at the Laboratory for Foundations of Computer Science in Edinburgh. I supplemented my income by working as a teaching assistant for the Computer Science department, and I supplemented my income supplement by moonlighting as a teaching assistant for the Mathematics department, working in particular with classes taught by mathematicians to first year undergraduate computer scientists. The mathematicians often asked the students to "show" things which was, for many of our students, an unfamiliar game. Old chestnuts like

$$\sum_{0\le i\le n} i = \frac{1}{2}n(n+1)$$

were a cause of panic and terror.

Learning to prove via weekly assignments is rather like learning chess by post: on Monday you send a card stating your move, and on Friday you receive the reply "You can't do that with a Bishop!".

My PhD research involved working with interactive proof assistants (specifically, LEGO [4]), so it seemed obvious to me that we should consider the use of machines to shorten the undergraduate feedback loop. At the time, Rod was working on a more basic proof editor, PROVEEASY [2], with which he was successfully teaching programming language semantics to third year undergraduates. I wondered if PROVEEASY might help the first years grapple with induction and proposed to demonstrate Rod's system to my mathematical employers. Carefully avoiding vulgar fractions, I drove the machine to deliver an interactive proof of

$$2\sum_{0\le i\le n} i = n(n+1)$$

They were unimpressed. I became depressed.

To cheer me up, Rod attempted cynicism: ‘But Conor, aren’t most of these students going to work in banks? Do they really to prove that? Don’t they just need to bung in three numbers and see that it works?’. ‘Rod,’ I replied, ‘you’re absolutely right. The formulae on both sides of the equation are manifestly quadratic, so bunging in three numbers to see that it works *is* a proof!’. Of course, it takes rather more mathematics to see why the result is that easy to *test* than to *prove* it directly. Indeed, I intended my remark merely as a throwaway absurdity, but it stuck with me. Once in a while I would idly fiddle with representations of polynomials, looking for an easy way to prove a testing lemma, but somehow things always became tricky very quickly.

Little did I know in the late 1990s that Marko Petkovšek, Herbert Wilf and Doron Zeilberger had given a fascinating treatment of this and other ‘proof by testing’ methods in their wondrous compendium, $A{=}B$ [10]. Their tools are powerful but informal computer algebra packages: mine is Martin-Löf’s intensional type theory [5], which is a formal discipline for proof where the division of labour between people and machines is negotiable by the act of programming. Recently, I found a way to prove the result in 250 lines of Agda [9], without recourse to any library functionality, and remaining entirely within the theory of the natural numbers. I suspect this means I have learned something in the intervening years. This paper is an attempt to communicate it.

2 Polynomials and testing

Polynomial functions are just those generated by constants and the identity, closed under pointwise addition and multiplication.

$$\underline{n}\,x = n \qquad \iota\,x = x \qquad (p \oplus q)\,x = p\,x + q\,x \qquad (p \otimes q)\,x = p\,x \times q\,x$$

We can define the *degree*, $|p|$, of a polynomial recursively:

$$|\underline{n}| = 0 \qquad |\iota| = 1 \qquad |p \oplus q| = |p| \vee |q| \qquad |p \otimes q| = |p| + |q|$$

The *testing principle* is just that if $|p| \leq n$, $|q| \leq n$ and $p\,i = q\,i$ for $i \in \{0, 1, \ldots, n\}$, then $p\,x = q\,x$ for all natural numbers x.

One way to see that it holds is to construct the *forward-difference* operator, Δ, satisfying the specification

$$\Delta p\,x = p\,(1 + x) - p\,x$$

and check that

- if $|p| = 0$ then $p = \underline{p\,0}$ and $\Delta p = \underline{0}$
- if $|p| = 1 + n$ then $|\Delta p| = n$.

The specification of Δ gives us an easy proof that

$$p\,x = p\,0 + \sum_{i<x} \Delta p\,i$$

As a consequence we should gain an easy inductive argument for the testing principle: if p and q have degree at most $1 + n$ and agree on $\{0, 1, \ldots, 1 + n\}$, then $p\,0 = q\,0$ and Δp and Δq agree on $\{0, 1, \ldots, n\}$; inductively, $\Delta p = \Delta q$, so replacing equal for equal in the above summation, $p\,x = q\,x$.

So, can we construct Δ and make this argument formal? McBrides have been in the business of constructing symbolic differential operators in functional programming languages for nearly half a century [8], so it will be something of a shame for me if we cannot. It is, however, tricky in parts.

3 Polynomials by degree

We can define natural numbers and the basic arithmetic operators straightforwardly. The ‘BUILTIN’ pragma allows us to use decimal notation.

```
data Nat : Set where
  ze :       Nat
  su : Nat → Nat
 {-# BUILTIN NATURAL Nat #-}
_+_ : Nat → Nat → Nat
ze   + y = y
su x + y = su (x + y)
infixr 5 _+_
_×_ : Nat → Nat → Nat
ze × x = ze
su n × x = n × x + x
infixr 6 _×_
```

Addition and multiplication both associate rightward, the latter binding more tightly. For convenience in writing polynomials, let us also define exponentiation as iterated multiplication.

```
_^_ : Nat → Nat → Nat
x ^ ze   = 1
x ^ su n = (x ^ n) × x
```

In order to capture *degree*, one might be tempted to define

```
_∨_ : Nat → Nat → Nat
ze   ∨ y    = y
su x ∨ ze   = su x
su x ∨ su y = su (x ∨ y)
```

and then give a type of polynomials *indexed by degree*.

```
data Poly : Nat → Set where
  κ     :                       Nat →    Poly 0
  ι     :                                Poly 1
  _⊕_ : ∀ {l m} → Poly l → Poly m → Poly (l ∨ m)
  _⊗_ : ∀ {l m} → Poly l → Poly m → Poly (l + m)
⟦_⟧ : ∀ {n} → Poly n → Nat → Nat
⟦ κ n ⟧     x = n
⟦ ι ⟧       x = x
⟦ p ⊕ r ⟧ x = ⟦ p ⟧ x + ⟦ r ⟧ x
⟦ p ⊗ r ⟧ x = ⟦ p ⟧ x × ⟦ r ⟧ x
```

We might attempt to define Δ reducing degree, as follows

```
Δ : ∀ {n} → Poly (su n) → Poly n
Δ ι         = ?
Δ (p ⊕ r) = ?
Δ (p ⊗ r) = ?
```

but these patterns do not typecheck, because it is not obvious how to unify su n with $l \vee m$ or $l + m$. The trouble is that our Δ function has a *prescriptive* index pattern in its argument type—not just a universally quantified variable—whilst our constructors have *descriptive* indices, imposing constraints by instantiation. For pattern matching to work, these indices must unify, but our use of defined functions (which I habitually colour green), $l \vee m$ and $l + m$, have taken us outside the constructor-form language where unification is straightforward. We have learned an important design principle.

▶ Principle 1 (Don't touch the green slime!). [1] When combining prescriptive and descriptive indices, ensure both are in constructor form. Exclude defined functions which yield difficult unification problems.

One way to avoid green slime without losing precision is to recast the datatype definition equationally, making use of the *propositional equality*:

```
data _≡_ {A : Set} (a : A) : A → Set where refl : a ≡ a
infix 0 _≡_
```

We may now replace non-constructor descriptive indices by equations and get to work:

```
data Poly : Nat → Set where
  κ     :                                Nat →                    Poly 0
  ι     :                                                          Poly 1
  plus  : ∀ {l m n} → Poly l → Poly m → (l ∨ m) ≡ n → Poly n
  times : ∀ {l m n} → Poly l → Poly m → (l + m) ≡ n → Poly n

Δ : ∀ {n} → Poly (su n) → Poly n
Δ ι              = κ 1
Δ (plus  p r q)  = ?
Δ (times p r q)  = ?
```

To make further progress, we shall need somehow to exploit the fact that in both cases, at least one of p and r has non-zero degree. This is especially galling as, morally, we should have something as simple as $\Delta\ (p \oplus r) = \Delta\ p \oplus \Delta\ r$. By being precise about the degree of polynomials, we have just made trouble for ourselves. We have just encountered another design principle of dependent types.

▶ Principle 2 (Jagger/Richards). [2] Be minimally prescriptive, not maximally descriptive.

In order to establish our testing principle, we do not need precise knowledge of the degree of a polynomial: an *upper bound* will suffice. We can see the index of our type of polynomials not as a degree measure being propagated outwards, but as a degree requirement being propagated *inwards*. The very syntax of datatype declarations is suggestive of descriptions propagated outwards, but it is usually a mistake to think in those terms.

We can get a little further, observing that polynomials of all degrees are closed under constants and addition, whilst the identity polynomial requires the degree to be positive. It is only in the case of products that we must precise about how we apportion our multiplicative resources.

[1] With the advent of colour television, the 1970s saw a great deal of science fiction in which green cleaning gel Swarfega was portrayed as a noxious substance, e.g., *Doctor Who: The Green Death* by Robert Sloman and Barry Letts.

[2] You can't always get what you want, but if you try sometime, you just might find that you get what you need.

```
data Poly : Nat → Set where
  κ     : ∀ {n} →                    Nat →                 Poly n
  ι     : ∀ {n} →                                          Poly (su n)
  _ ⊕ _ : ∀ {n} →        Poly n → Poly n →                 Poly n
  times : ∀ {l m n} → Poly l → Poly m → (l + m) ≡ n → Poly n
Δ : ∀ {n} → Poly (su n) → Poly n
Δ (κ _)          = κ 0
Δ ι              = κ 1
Δ (p ⊕ r)        = Δ p ⊕ Δ r
Δ (times p r q)  = ?
```

We can use the specification of Δ to try to figure out how to deal with multiplication. Firstly, let us eliminate subtraction by rephrasing

$$p\,(1+x) = p\,x + \Delta p\,x$$

and then we may reason as follows:

$$\begin{array}{lll}
 & (p \otimes r)\,(1+x) & \text{definition of } \otimes \\
= & p\,(1+x) \times r\,(1+x) & \text{spec of } \Delta p \\
= & (p\,x + \Delta p\,x) \times r\,(1+x) & \text{distribution} \\
= & p\,x \times r\,(1+x) + \Delta p\,x \times r\,(1+x) & \text{spec of } \Delta r \\
= & p\,x \times (r\,x + \Delta r\,x) + \Delta p\,x \times r\,(1+x) & \text{distribution} \\
= & p\,x \times r\,x + p\,x \times \Delta r\,x + \Delta p\,x \times r\,(1+x) & \text{algebra} \\
= & (p\,x \times r\,x) + (\Delta p\,x \times r\,(1+x) + p\,x \times \Delta r\,x) & \text{definition of } \otimes \\
= & (p \otimes r)\,x + (\Delta p\,x \times r\,(1+x) + p\,x \times \Delta r\,x) &
\end{array}$$

So, we need

$$\Delta(p \otimes r) = \Delta p \otimes (r \cdot (1+)) \;\oplus\; p \otimes \Delta r$$

If we have eyes to see, we can take some useful hints here:

- we shall need to account for *shifting*, as in $r \cdot (1+)$;
- we shall need to separate the degenerate cases where p or r have degree ze from the general case when both have non-zero degree, in order to ensure that Δ is recursively applicable;
- we shall need to make it obvious that ze $+\ n = n = n\ +$ ze and $l\ +$ su $m =$ su $l\ +\ m$.

Shifting preserves degree, so we may readily add it to the datatype of polynomials, at the cost of introducing some redundancy. Meanwhile, the case analysis we need on our sum of degrees is not the one which arises naturally from the recursive definition of $+$. Some theorem proving will be necessary, but a good start is to write down the case analysis we need, relationally. Indeed, this is a key lesson which I learned from James McKinna and which sits at the heart of our work on Epigram [7]:

▶ Principle 3 (Structure by Construction). If you need to observe a particular inductive structure, work with the type which is defined to have that inductive structure.

```
data Add : Nat → Nat → Nat → Set where
  zel  : ∀ {n} →                     Add ze     n      n
  zer  : ∀ {n} →                     Add n      ze     n
  susu : ∀ {l m n} → Add l m n →  Add (su l) (su m) (su (su n))
```

We can now define polynomials in a workable way.

```
data Poly : Nat → Set where
  κ      : ∀ {n} →                       Nat →              Poly n
  ι      : ∀ {n} →                                          Poly (su n)
  ↑      : ∀ {n} →                       Poly n →           Poly n
  _⊕_    : ∀ {n} →            Poly n →   Poly n →           Poly n
  times  : ∀ {l m n} → Poly l → Poly m → Add l m n →        Poly n
```

The evaluator for polynomials can safely ignore the informatin about adding degrees.

```
⟦_⟧ : ∀ {n} → Poly n → Nat → Nat
⟦ κ n ⟧          x = n
⟦ ι ⟧            x = x
⟦ ↑p ⟧           x = ⟦ p ⟧ (su x)
⟦ p ⊕ r ⟧        x = ⟦ p ⟧ x + ⟦ r ⟧ x
⟦ times p r a ⟧ x = ⟦ p ⟧ x × ⟦ r ⟧ x
```

However, when we come to define Δ, we can now make substantial progress, splitting the multiplication into the three cases which matter.

```
Δ : ∀ {n} → Poly (su n) → Poly n
Δ (κ _)                 = κ 0
Δ ι                     = κ 1
Δ (↑p)                  = ↑(Δ p)
Δ (p ⊕ r)               = Δ p ⊕ Δ r
Δ (times p r zel)       = times (κ (⟦ p ⟧ 0)) (Δ r) zel
Δ (times p r zer)       = times (Δ p) (κ (⟦ r ⟧ 0)) zer
Δ (times p r (susu a))  = times (Δ p) (↑r) ? ⊕ times p (Δ r) ?
```

The proof obligations, thus identified, are easily discharged:

```
sul : ∀ {l m n} → Add l m n → Add (su l) m (su n)
sul (zel {ze})      = zer
sul (zel {su n})    = susu zel
sul zer             = zer
sul (susu s)        = susu (sul s)

sur : ∀ {l m n} → Add l m n → Add l (su m) (su n)
sur zel             = zel
sur (zer {ze})      = zel
sur (zer {su n})    = susu zer
sur (susu s)        = susu (sur s)
Δ (times p r (susu a)) = times (Δ p) (↑r) (sur a) ⊕ times p (Δ r) (sul a)
```

Of course, we should like the convenience of our ⊗ operator, at least for *constructing* polynomials. We need simply prove that Add respects +.

```
add : (l m : Nat) → Add l m (l + m)
add ze m       = zel
add (su l) m = sul (add l m)
```

```
_⊗_ : ∀ {l m} → Poly l → Poly m → Poly (l + m)
_⊗_ {l} {m} p r = times p r (add l m)
infixr 6 _⊗_
```

Similarly, let us grant ourselves versions of κ and ι which fix their degree, and define the exponential.

```
ι₁ : Poly 1
ι₁ = ι
κ₀ : Nat → Poly 0
κ₀ n = κ n
_⊘_ : ∀ {m} → Poly m → (n : Nat) → Poly (n × m)
p ⊘ ze = κ 1
p ⊘ su n = (p ⊘ n) ⊗ p
```

We may now write polynomials, taking care to use ι_1 in the highest degree term,

```
(ι₁ ⊘ 2) ⊕ κ₀ 2 ⊗ ι ⊕ κ 1 : Poly 2
```

thus fixing the bound and leaving some slack in the other terms.

We now have a definition of Polynomials indexed by a bound on their degree which readily admits a candidate for the forward-difference operator Δ.

4 Stating the testing principle

Let us be careful to define the conditions under which we expect two polynomials to be pointwise equal. We shall require a conjunction of individual equations, hence let us define the relevant logical apparatus: the unit type and the cartesian product.

```
record 1 : Set where constructor ⟨⟩
record _×_ (S T : Set) : Set where constructor _,_; field fst : S; snd : T
open _×_ public
```

To state the condition that polynomials coincide on $\{i|i < k\}$, we may define:

```
TestEq : (k : Nat) {n : Nat} (p r : Poly n) → Set
TestEq ze      p r = 1
TestEq (su k) p r = (⟦ p ⟧ ze ≡ ⟦ r ⟧ ze) × TestEq k (↑p) (↑r)
```

We can now state the key lemma that we hope to prove.

```
testLem : (n : Nat) (p r : Poly n) → TestEq (su n) p r → (x : Nat) → ⟦ p ⟧ x ≡ ⟦ r ⟧ x
```

Of course, in order to *prove* the testing principle, we shall need the machinery for constructing proofs of equations.

5 Some kit for equational reasoning

It is standard to make use of Agda's mixfix notation for presenting equational explanations in readable form. I give two ways to take a step, combining symmetry with transitivity, and one way to finish.

```
_=[_⟩=_ : forall {X : Set} (x : X) {y z} → x ≡ y → y ≡ z → x ≡ z
x =[ refl ⟩= q = q
_=⟨_]=_ : forall {X : Set} (x : X) {y z} → y ≡ x → y ≡ z → x ≡ z
x =⟨ refl ]= q = q
_□ : forall {X : Set} (x : X) → x ≡ x
x □ = refl
infixr 2 _□ _=[_⟩=_ _=⟨_]=_
```

Meanwhile, the following apparatus is useful for building *structural* explanations of equality between applicative forms.

```
⌈_⌉ : forall {X : Set} (x : X) → x ≡ x
⌈ x ⌉ = refl
_=$=_ : {S T : Set} {f g : S → T} {x y : S} → f ≡ g → x ≡ y → f x ≡ g y
refl =$= refl = refl
infixl 9 _=$=_
```

For example, let us prove the lemma that any polynomial of degree at most *0* is constant.

```
kLem : (p : Poly 0) (x y : Nat) → ⟦ p ⟧ x ≡ ⟦ p ⟧ y
kLem (κ n)          x y = refl
kLem (↑p)           x y =
  ⟦ ↑p ⟧ x                =[ refl ⟩=
  ⟦ p ⟧ (su x)            =[ kLem p (su x) (su y) ⟩=
  ⟦ p ⟧ (su y)            =⟨ refl ]=
  ⟦ ↑p ⟧ y                □
kLem (p ⊕ r)        x y =
  ⟦ (p ⊕ r) ⟧ x           =[ refl ⟩=
  ⟦ p ⟧ x + ⟦ r ⟧ x       =[ ⌈_+_⌉ =$= kLem p x y =$= kLem r x y ⟩=
  ⟦ p ⟧ y + ⟦ r ⟧ y       =⟨ refl ]=
  ⟦ (p ⊕ r) ⟧ y           □
kLem (times p r zel) x y =
  ⟦ (p ⊗ r) ⟧ x           =[ refl ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ x       =[ ⌈_×_⌉ =$= kLem p x y =$= kLem r x y ⟩=
  ⟦ p ⟧ y × ⟦ r ⟧ y       =⟨ refl ]=
  ⟦ (p ⊗ r) ⟧ y           □
kLem (times p r zer) x y =
  ⟦ (p ⊗ r) ⟧ x           =[ refl ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ x       =[ ⌈_×_⌉ =$= kLem p x y =$= kLem r x y ⟩=
  ⟦ p ⟧ y × ⟦ r ⟧ y       =⟨ refl ]=
  ⟦ (p ⊗ r) ⟧ y           □
```

The steps where the equality proof is refl can be omitted, as they follow just by symbolic evaluation. It may, however, add clarity to retain them.

6 Proving the testing principle

Let us now show that the testing principle follows from the as yet unproven hypothesis that Δ satisfies its specification:

```
ΔLem : ∀ {n} (p : Poly (su n)) x → ⟦ p ⟧ (su x) ≡ ⟦ p ⟧ x + ⟦ Δ p ⟧ x
```

As identified earlier, the key fact is that the test conditions for some p and r imply the test conditions for Δ p and Δ r.

```
testΔLem : (k : Nat) {n : Nat} (p r : Poly (su n)) →
  TestEq (su k) p r → TestEq k (Δ p) (Δ r)
```

Assuming this, we may deliver the main induction.

```
testLem ze      p r (q , ⟨⟩) x        =
  ⟦ p ⟧ x                            =[ kLem p x ze ⟩=
  ⟦ p ⟧ 0                            =[ q ⟩=
  ⟦ r ⟧ 0                            =⟨ kLem r x ze ]=
  ⟦ r ⟧ x                            □
testLem (su n) p r (q0 , qs) ze      =
  ⟦ p ⟧ ze                           =[ q0 ⟩=
  ⟦ r ⟧ ze                           □
testLem (su n) p r qs         (su x)  =
  ⟦ p ⟧ (su x)                       =[ ΔLem p x ⟩=
  ⟦ p ⟧ x + ⟦ Δ p ⟧ x
    =[ ((⌈_+_⌉ =$= testLem (su n) p r qs x =$= testLem n (Δ p) (Δ r) (testΔLem (su n) p r qs) x) ⟩=
  ⟦ r ⟧ x + ⟦ Δ r ⟧ x                =⟨ ΔLem r x ]=
  ⟦ r ⟧ (su x)                       □
```

To establish testΔLem, we need to establish equality of *differences* by cancelling what the differences are added to. We shall need the *left-cancellation* property of addition.

```
+cancel : ∀ w y {x z} → w ≡ y → w + x ≡ y + z → x ≡ z

testΔLem ze     p r q          = ⟨⟩
testΔLem (su k) p r (q0 , qs)  =
  +cancel (⟦ p ⟧ 0) (⟦ r ⟧ 0) q0 (
    ⟦ p ⟧ 0 + ⟦ Δ p ⟧ 0 =⟨ ΔLem p 0 ]=
    ⟦ p ⟧ 1             =[ fst qs ⟩=
    ⟦ r ⟧ 1             =[ ΔLem r 0 ⟩=
    ⟦ r ⟧ 0 + ⟦ Δ r ⟧ 0 □) ,
  testΔLem k (↑p) (↑r) qs
```

7 No confusion, cancellation, decision

The left-cancellation property ultimately boils down to iterating the observation that su is injective. Likewise, to show that we can actually decide the test condition, we shall need to show that numeric equality is decidable, which also relies on the *'no confusion'* property of the datatype Nat—constructors are injective and disjoint. Back when we worked for Rod, Healfdene Goguen and James McKinna taught me a reflective method to state and prove a bunch of constructor properties simultaneously: we define the intended consequences of an equation between constructor forms, and then show that those consequences do indeed hold on the diagonal [6].

```
data 0 : Set where

NoConf : Nat → Nat → Set
NoConf ze     ze     = 1
NoConf (su x) (su y) = x ≡ y
NoConf _      _      = 0

noConf : ∀ {x y} → x ≡ y → NoConf x y
noConf {ze}   refl = ⟨⟩
noConf {su x} refl = refl
```

The cancellation property we need follows by an easy induction. Again, we work along the diagonal, using noConf to strip a su at each step.

```
+cancel ze     .ze     refl refl = refl
+cancel (su w) ∘ (su w) refl q    = +cancel w w refl (noConf q)
```

In the step case, q : su $(w + x) \equiv$ su $(w + z)$, so noConf $q : (w + x) \equiv (w + z)$. Meanwhile, we can frame the decision problem for the numeric equation $x \equiv y$ as the question of whether the type has an inhabitant or is provably empty.

```
data Dec (P : Set) : Set where
  yes : P →         Dec P
  no  : (P → 0) → Dec P
```

The method for deciding equality is just like the usual method for testing it, except that we generate evidence via noConf.

```
decEq : (x y : Nat) → Dec (x ≡ y)
decEq ze     ze                    = yes refl
decEq ze     (su y)                = no noConf
decEq (su x) ze                    = no noConf
decEq (su x) (su y)  with decEq x y
decEq (su x) (su .x) |  yes refl  = yes refl
decEq (su x) (su y)  |  no nq     = no λ q → nq (noConf q)
```

While we are in a decisive mode, let us show that the test condition is decidable, just by iterating numerical equality tests.

```
testEq : (k : Nat) {n : Nat} (p r : Poly n) → Dec (TestEq k p r)
testEq ze     p r = yes ⟨⟩
testEq (su k) p r with decEq (⟦ p ⟧ 0) (⟦ r ⟧ 0) | testEq k (↑p) (↑r)
... | yes y  | yes z  = yes (y , z)
... | yes y  | no np  = no λ xy → np (snd xy)
... | no np  | _      = no λ xy → np (fst xy)
```

We can now give our testing principle a friendly user interface, incorporating decision.

8 The testing deliverable

Much as with NoConf, we can compute for any pair of polynomials a statement which might be of interest—if the test condition holds, we state that the polynomials are pointwise equal—and we can prove that statement, because deciding the test condition delivers the evidence we need.

```
TestStmt : (n : Nat) (p q : Poly n) → Set
TestStmt n p r with testEq (su n) p r
... | yes qs = (x : Nat) → ⟦ p ⟧ x ≡ ⟦ r ⟧ x
... | no _ = 1

testStmt : {n : Nat} (p r : Poly n) → TestStmt n p r
testStmt {n} p r with testEq (su n) p r
... | yes qs  = testLem n p r qs
... | no _    = ⟨⟩
```

9 Associativity, Commutativity, Distributivity

Before we can tackle the correctness of Δ, we shall need some basic facts about + and ×. There is little surprising about these proofs, but I include them for completeness.

Associativity of addition follows by an induction which triggers computation.

```
+assoc : (l m n : Nat) → (l + m) + n ≡ l + (m + n)
+assoc ze      m n = refl
+assoc (su l) m n = ⌈su⌉ =$= +assoc l m n
```

Commutativity of addition follows from lemmas which simulate the behaviour of addition by recursion on the second argument, oriented right-to-left.

```
+ze : {n : Nat} → n ≡ n + ze
+ze {ze}          = refl
+ze {su n}        = ⌈su⌉ =$= +ze

+su : (m n : Nat) → su (m + n) ≡ m + su n
+su ze       n    = refl
+su (su m) n      = ⌈su⌉ =$= +su m n

+comm : (m n : Nat) → m + n ≡ n + m
+comm ze       n = +ze
+comm (su m) n = su (m + n) =[ ⌈su⌉ =$= +comm m n ⟩=
                 su (n + m) =[ +su n m ⟩=
                 n + su m   □
```

We shall also need the distributivity of multiplication over addition. James McKinna taught me to prepare by proving a lemma that transposes the middle of a sum of sums.

```
mid4 : (w x y z : Nat) → (w + x) + (y + z) ≡ (w + y) + (x + z)
mid4 w x y z =
  (w + x) + (y + z) =[ +assoc w x (y + z) ⟩=
  w + (x + (y + z)) =[ ⌈_+_ w⌉ =$= (x + (y + z) =⟨ +assoc x y z ]=
                                    (x + y) + z =[ ⌈_+_⌉ =$= +comm x y =$= ⌈z⌉ ⟩=
                                    (y + x) + z =[ +assoc y x z ⟩=
                                    y + (x + z) □) ⟩=
  w + (y + (x + z)) =⟨ +assoc w y (x + z) ]=
  (w + y) + (x + z) □
```

The step case of left-distributivity, is just such a transposition.

```
*dist+ : (a b c : Nat) → a × (b + c) ≡ a × b + a × c
*dist+ ze     b c = refl
*dist+ (su a) b c =
  a × (b + c) + (b + c)        =[ ⌈_+_⌉ =$= *dist+ a b c =$= ⌈(b + c)⌉ ⟩=
  (a × b + a × c) + b + c      =[ mid4 (a × b) (a × c) b c ⟩=
  (a × b + b) + (a × c + c) □
```

Meanwhile, the strategic insertion of a ze will bring the step case of right-distributivity also into this form.

```
+dist* : (a b c : Nat) → (a + b) × c ≡ a × c + b × c
+dist* ze b c = refl
+dist* (su a) b c =
  (a + b) × c + c                =[ ⌈_+_⌉ =$= +dist* a b c =$= +ze ⟩=
  (a × c + b × c) + (c + ze) =[ mid4 (a × c) (b × c) c ze ⟩=
  (a × c + c) + (b × c + ze) =⟨ ⌈_+_ (su a × c)⌉ =$= +ze ]=
  su a × c + b × c □
```

We need neither associativity nor commutativity of multiplication, as our products are binary, and the definition of Δ preserves left-to-right order. Thus equipped with rather less than the ring theory of the integers, we are none the less ready to proceed.

10 Correctness of Δ

It is high time we sealed the argument with a proof that Δ satisfies its specification, really computing the difference between ↑p and p.

```
ΔLem : ∀ {n} (p : Poly (su n)) x → ⟦ p ⟧ (su x) ≡ ⟦ p ⟧ x + ⟦ Δ p ⟧ x
```

The basic plan is to do induction over the computation of Δ, then basic algebraic rearrangement using the above facts. It is thoroughly unremarkable, if a little gory in places.

Constants, identity and shifting are very simple. Addition yet again requires us to swap the middle of a sum of sums.

```
ΔLem (κ n)    x = n =[ +ze ⟩= n + 0 □
ΔLem ι        x = 1 + x =[ +comm 1 x ⟩= x + 1 □
ΔLem (↑p)     x = ΔLem p (su x)
ΔLem (p ⊕ r) x =
  ⟦ p ⟧ (su x) + ⟦ r ⟧ (su x)                        =[ ⌈_+_⌉ =$= ΔLem p x =$= ΔLem r x ⟩=
  (⟦ p ⟧ x + ⟦ Δ p ⟧ x) + (⟦ r ⟧ x + ⟦ Δ r ⟧ x) =[ mid4 (⟦ p ⟧ x) (⟦ Δ p ⟧ x) (⟦ r ⟧ x) (⟦ Δ r ⟧ x) ⟩=
  ⟦ p ⊕ r ⟧ x + ⟦ Δ (p ⊕ r) ⟧ x                     □
```

The edge cases of multiplication require the constancy of zero-degree polynomials. The two subproblems are mirror images.

```
ΔLem (times p r zel) x =
  ⟦ p ⟧ (su x) × ⟦ r ⟧ (su x)                  =[ ⌈_×_⌉ =$= kLem p (su x) 0 =$= ΔLem r x ⟩=
  ⟦ p ⟧ 0 × (⟦ r ⟧ x + ⟦ Δ r ⟧ x)              =[ *dist+ (⟦ p ⟧ 0) (⟦ r ⟧ x) (⟦ Δ r ⟧ x) ⟩=
  ⟦ p ⟧ 0 × ⟦ r ⟧ x + ⟦ p ⟧ 0 × ⟦ Δ r ⟧ x =⟨ ⌈_+_⌉ =$= (⌈_×_⌉ =$= kLem p x 0 =$= refl) =$= refl ]=
  ⟦ p ⟧ x × ⟦ r ⟧ x + ⟦ p ⟧ 0 × ⟦ Δ r ⟧ x □
```

```
ΔLem (times p r zer) x =
  ⟦ p ⟧ (su x) × ⟦ r ⟧ (su x)              =[ ⌈_×_⌉ =$= ΔLem p x =$= kLem r (su x) 0 ⟩=
  (⟦ p ⟧ x + ⟦ Δ p ⟧ x) × ⟦ r ⟧ 0           =[ +dist∗ (⟦ p ⟧ x) (⟦ Δ p ⟧ x) (⟦ r ⟧ ze) ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ 0 + ⟦ Δ p ⟧ x × ⟦ r ⟧ 0 =⟨ ⌈_+_⌉ =$= (⌈_×_ (⟦ p ⟧ x)⌉ =$= kLem r x 0) =$= refl ]=
  ⟦ p ⟧ x × ⟦ r ⟧ x + ⟦ Δ p ⟧ x × ⟦ r ⟧ 0 □
```

The only tricky case is the nondegenerate multiplication. Here, I am careful about when to apply the induction hypothesis for r, as ⟦ r ⟧ (su x) occurs on both sides: I wait until after it has been duplicated by distribution, then rewrite only one of the copies.

```
ΔLem (times p r (susu a)) x =
  ⟦ p ⟧ (su x) × ⟦ r ⟧ (su x)              =[ ⌈_×_⌉ =$= ΔLem p x =$= refl ⟩=
  (⟦ p ⟧ x + ⟦ Δ p ⟧ x) × ⟦ r ⟧ (su x) =[ +dist∗ (⟦ p ⟧ x) (⟦ Δ p ⟧ x) (⟦ r ⟧ (su x)) ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ (su x) + ⟦ Δ p ⟧ x × ⟦ r ⟧ (su x)
    =[ ⌈_+_⌉ =$= (⌈_×_ (⟦ p ⟧ x)⌉ =$= ΔLem r x) =$= refl ⟩=
  ⟦ p ⟧ x × (⟦ r ⟧ x + ⟦ Δ r ⟧ x) + ⟦ Δ p ⟧ x × ⟦ r ⟧ (su x)
    =[ ⌈_+_⌉ =$= ∗dist+ (⟦ p ⟧ x) (⟦ r ⟧ x) (⟦ Δ r ⟧ x) =$= refl ⟩=
  (⟦ p ⟧ x × ⟦ r ⟧ x + ⟦ p ⟧ x × ⟦ Δ r ⟧ x) + ⟦ Δ p ⟧ x × ⟦ r ⟧ (su x)
    =[ +assoc (⟦ p ⟧ x × ⟦ r ⟧ x) _ _ ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ x + ⟦ p ⟧ x × ⟦ Δ r ⟧ x + ⟦ Δ p ⟧ x × ⟦ r ⟧ (su x)
    =[ ⌈_+_ (⟦ p ⟧ x × ⟦ r ⟧ x)⌉ =$= +comm (⟦ p ⟧ x × ⟦ Δ r ⟧ x) (⟦ Δ p ⟧ x × ⟦ r ⟧ (su x)) ⟩=
  ⟦ p ⟧ x × ⟦ r ⟧ x + ⟦ Δ p ⟧ x × ⟦ r ⟧ (su x) + ⟦ p ⟧ x × ⟦ Δ r ⟧ x □
```

The proof boils down to rewriting by kLem and inductive hypotheses, then elementary ring-solving: the latter could readily be disposed of by a reflective tactic in the style of Boutin [1].

11 Summing up

We have proven our polynomial testing principle, but we have rather lost sight of the problem which led us to it—proving results about *summation*.

```
Σ : (f : Nat → Nat) → Nat → Nat
Σ f ze      = 0
Σ f (su n)  = Σ f n + f n
```

Now, our language of Polynomials with constants in Nat does not admit summation: that would require us to work with rationals at considerable additional effort. However, we can just extend the *syntax* with summation, *incrementing degree*, and give Σ as its semantics.

```
σ : ∀ {n} → Poly n → Poly (su n)
⟦ σ p ⟧ x = Σ ⟦ p ⟧ x
```

Of course, we are then obliged to augment the rest of the development. The non-zero index of σ means that no obligation arises for kLem. Meanwhile, by careful alignment, Σ is exactly the notion of ‘integration’ corresponding to forward-difference Δ, hence the extension of Δ is direct and the corresponding case of ΔLem trivial.

```
Δ      (σ p)   = p
ΔLem (σ p) x  = refl
```

And with those seven extra lines, we are ready to prove classic results by finitary testing.

```
triangle :  (x : Nat) → 2 × Σ (λ i → i) (su x) ≡ x × (x + 1)
triangle =  testStmt (κ₀ 2 ⊗ ↑(σ ι₁)) (ι₁ ⊗ (ι₁ ⊕ κ 1))

square   :  (x : Nat) → Σ (λ i → 2 × i + 1) x ≡ x ^ 2
square   =  testStmt (σ (κ₀ 2 ⊗ ι₁ ⊕ κ 1)) (ι₁ ⊛ 2)

cubes    :  (x : Nat) → Σ (λ i → i ^ 3) x ≡ (Σ (λ i → i) x) ^ 2
cubes    =  testStmt (σ (ι₁ ⊛ 3)) (σ ι₁ ⊛ 2)
```

In effect, it is enough for us to say so, and the machine will see for itself. The essence of proof is to explain how infinitary statements can be reduced to a finitary collection of tests. By exposing an intrinsic notion of degree for polynomials, their very form tells us how to test them. What is unusual is that these problems do not require any algebraic calculation or symbolic evaluation, just arithmetic. In some sense, we have found a basis from which testing can reveal the absence of bugs, not just, as Dijkstra observed, their presence [3]. Rod Burstall's casual remark about trying three examples has exposed a remarkably simple way to see truths which are often presented as subtle. Indeed, Rod's guidance it is that has me striving for vantage points from which the view is clear. This paper is for him.

References

1 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.

2 Rod M. Burstall. ProveEasy: helping people learn to do proofs. *Electronic Notes in Theoretical Computer Science*, 31:16–32, 2000.

3 J.N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 October 1969*, chapter 3.1. NATO, 1970. Transcription of a discussion involving Dijkstra.

4 Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS, University of Edinburgh, 1992.

5 Per Martin-Löf. A theory of types. *Unpublished manuscript*, 1971.

6 Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

7 Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

8 Frederick McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, The Queen's University of Belfast, May 1970. `http://personal.strath.ac.uk/conor.mcbride/FVMcB-PhD.pdf`.

9 Ulf Norell. Dependently Typed Programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

10 Marko Petkovšek, Herbert Wilf, and Doron Zeilberger. *A=B*. A K Peters/CRC Press, 1996.