Generalizing Positional Numeral Systems

Luā Tîng-Giān

December 7, 2016

Abstract

Numbers are everywhere in our daily lives, and positional numeral systems are arguably the most important and common representation of numbers. In this work we have constructed a generalized positional numeral system in Agda to model many of these representations, and investigate some of their properties and relationship with the classical unary representation of the natural numbers.

1 Introduction

1.1 Positional numeral systems

A numeral system is a writing system for expressing numbers, and humans have invented various kinds of numeral systems throughout history. Most of the systems we are using today are positional notations[3] because they can express infinite numbers with just a finite set of symbols called **digits**.

Positional numeral systems represent a number by adding up a sequence of digits of different orders of magnitude. Take a decimal number for example:

$$(2016)_{10} = 2 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 6 \times 10^0$$

6 is called *least significant digit* and 2 is called the *most significant digit* in this example. From now on, except when writing decimal numbers, we will write down numbers in reverse order, from the least significant digit to the most significant digit, like 6102.

To make things clear, we call a sequence of digits a **numeral**, or a **notation**; and the number it expresses a **value**, or simply a **number**. That is, we distinguish syntax from semantics. Syntax bears no meaning; its semantics can only be carried out by converting to some other syntax. We call a function that converts notations to values an **evaluator**, and the process an **evaluation**.

1.1.1 Symtems of different bases

These numeral systems can take on different *bases*. The ubiquitous decimal numeral system as we know has the base of 10. While the binaries that can be found in our machines nowadays has the base of 2. To evaluate a notation of certain system of base:

$$(d_0 d_1 d_2 d_3 ...)_{base} = d_0 \times base^0 + d_1 \times base^1 + d_2 \times base^2 + d_3 \times base^3 ...$$

Where d_n is a digit that ranges from 0 to $base - 1$ for all n.

1.1.2 Digits of different ranges

Some computer scientists and mathematicians seem to be more comfortable with unary (base-1) numbers because they are isomorphic to the natural numbers à la Peano.

$$(1111)_1 \cong \overbrace{\text{suc (suc (suc (suc}}^{4} + \text{zero)))}$$

Statements established on such construction can be proven using mathematical induction. Moreover, people have implemented and proven a great deal of functions and properties on these unary numbers because they are easy to work with.

However, the formula we have just put down for evaluating positional numeral systems doesn't work for unary numbers, as it requires the digits to range from 0 to $base - 1$. That is, the only digit has to be 0, yet the only digit unary numbers have is 1.

To cooperate unary numbers, we relax the constraint on the range of digits by introducing a new variable, *offset*:

$$(d_0 d_1 d_2 d_3 ...)_{base} = d_0 \times base^0 + d_1 \times base^1 + d_2 \times base^2 + d_3 \times base^3 ...$$

Where d_n ranges from *offset* to *offset + base - 1* for all n. Now that unary numbers would have an offset of 1, and systems of other bases would have offsets of 0.

1.1.3 Redundancy

With the generalization of *base* and *offset*, so far we have been able to cover some different kinds of numeral systems, but the binary numeral system that is implemented in virtually all arithmetic logic unit (ALU) hardware is not among them.

In our representation, operations such as addition would take $O(logn)$ for some number n in a system where $base{>}1$. Since the number n would have length $log_{base}n$ and operations on each digit should be constant. However, these operations only takes constant time in machines!

That seems to be a big performance issue, but there's a catch! Because our representation is capable of what is called *arbitrary-precision arithmetic*, i.e., it could perform calculations on numbers of arbitrary size while the binary numbers that reside in machines are bounded by the hardware, which could only perform *fixed-precision arithmetic*.

Surprisingly, we could fit these binary numbers into our representation with just a tweak. If we allow a system to have more digits, then a fixed-precision binary number can be regarded as a single digit! To illustrate this, a 32-bit binary number would become a single digit that ranges from 0 to 2^{32}, while everything else including the base remains the same.

Formerly in our representation, there are exactly *base* number of digits that range from:

$$\text{offset}...\text{offset} + \text{base} - 1$$

We introduce a new index *#digit* to generalize the number of digits. Now they range from:

$$\text{offset}...\text{offset} + \text{\#digit} - 1$$

Here's a table of the configurations about the systems that we've addressed:

Numeral system	base	#digit	offset
Decimal	10	10	0
Binary	2	2	0
Unary	1	1	1
Int32	2	2^{32}	0

Consider this numeral system, the oridinary binary numbers with an extra digit: 2.

Numeral system	base	#digit	offset
0-1-2 Binary	2	3	0

Number (in decimal)	Notation
0	0
1	1
2	01, 2
3	11
4	001, 21
5	101, 12

Such a numeral system is said to be **redundant**, because there are more than one way to represent a number. In fact, systems that allow 0 as one of the digits must be redundant, since we can always take a number and add leading zeros without changing it's value. Systems that does not have zeros are said to be **zeroless**.

We will see that there's a deep connection between data structures and numeral systems. Data structures modeled after redundant numeral systems have some interesting properties.

1.1.4 Numerical representation

One may notice that the structure of unary numbers looks suspiciously similar to that of lists'. Let's compare their definition in Haskell.

```
data Nat = Zero
         | Suc Nat
```

```
data List a = Nil
            | Cons a (List a)
```

If we replace every `Cons _` with `Suc` and `Nil` with `Zero`, then a list becomes an unary number. And that is exactly what the `length` function, a homomorphism from lists to unary numbers, does.

Now let's compare addition on unary number and merge (append) on lists:

```
add : Nat → Nat → Nat
add Zero    y = y
add (Suc x) y =
    Suc (add x y)
```

```
append : List a → List a → List a
append Nil         ys = ys
append (Cons x xs) ys =
    Cons x (append xs ys)
```

Aside from having virtually identical implementations, operations on unary numbers and lists both have the same time complexity. Incrementing a unary number takes $O(1)$, inserting an element into a list also takes $O(1)$; adding two unary numbers takes $O(n)$, appending a list to another also takes $O(n)$.

If we look at implementations and operations of binary numbers and binomial heaps, the resemblances are also uncanny.

[insert some images here]

The strong analogy between positional numeral systems and certain data structures suggests that, numeral systems can serve as templates for designing containers. Such data structures are called **Numerical Representations**[8] [2].

[say something about redundant data structures]

Outline The remainder of the thesis is organized as follows.

2 A gental introduction to dependently typed programming in Agda

There are already plenty of tutorials and introductions of Agda[7][6][4]. We will nonetheless compile a simple and self-contained tutorial from the materials cited above, covering the part (and only the part) we need in this work.

Some of the more advenced constructions (such as views and universes) used in the following sections will be introduced along the way.

We assume that all readers have some basic understanding of Haskell, and those who are familiar with Agda and dependently typed programming may skip this chapter.

2.1 Some basics

Agda is a *dependently typed functional programming language* and also an *interactive proof assistant*. It can be both because it's based on *Martin-Löf type theory*[5], hence the Curry-Howard correspondence[9], which states that: "propositions are types" and "proofs are programs". In other words, proving theorems and writing programs are essentially the same. In Agda we are free to interchange between these two interpretations. The current version (Agda2) is a completely rewrite by Ulf Norell during his PhD at Chalmers University of Technology.

We say that Agda is interactive because proving theorems involves a lot of conversations between the programmer and the type checker. And it is often difficult, if not impossible, to develop and prove a theorom at one stroke. Just like programming, the process is incremental. So Agda allows us to leave some "holes" in a program, refine them gradually, and complete the proofs "hole by hole".

Take this half-finished function definition for instance, we could leave out the right-hand side.

```
is-zero : Int → Bool
is-zero x = ?
```

In practice, we would ask, for example, ”what's the type of the goal?”, ”what's the context of this case?”, etc. And Agda would reply us with:

```
GOAL : Bool
x : Int
```

Then we may ask Agda to pattern match on `x` and rewrite the program for us:

```
is-zero : Int → Bool
is-zero zero    = ?
is-zero (suc x) = ?
```

We could fulfill these goals by giving an answer, we may even ask Agda to solve the problem for us, if it is not too difficult.

```
is-zero : Int → Bool
is-zero zero    = true
is-zero (suc x) = false
```

After all of the goals have been accomplished and type-checked, we consider the program to be finished. Often, there's not much point in running a Agda program, because it's mostly about static constructions that is checked in compile-time. This is basically what pragramming and proving things looks like in Agda.

2.2 Simply typed programming in Agda

Since Agda was heavily influenced by Haskell, simply typed programming in Agda is similar to that in Haskell.

Datatypes Unlike in other programming languages, there are no ”built-in” datatypes such as *Int*, *String*, or *Bool*. The reason is that they can all be created out of thin air, so why bother?

Datatypes are introduced with `data` declarations. Here is a classical example, the type of booleans.

```
data Bool : Set where
    true  : Bool
```

```
    false : Bool
```

The name of the datatype (`Bool`) and its constructors (`true` and `false`) are brought into scope in this declaration. This notation also allow us to explicitly specify the types of these newly introduced entities.

1. `Bool` has the type of `Set`[1]
2. `true` has the type of `Bool`
3. `false` has the type of `Bool`

Pattern matching Similar to Haskell, datatypes are eliminated with pattern matching.

Here's a function that pattern matches on `Bool`.

```
not : Bool → Bool
not true  = false
not false = true
```

Agda is a *total* language, that means partial functions are not valid constructions. Functions are guarantee to terminate and will not crash on all possible inputs. The following example won't be accecpted by the type checker, because the case `false` is missing.

```
not : Bool → Bool
not true  = false
```

In practice, Agda would automatically expand all of the cases for us on demand.

Inductive datatype Let's move on to a more interesting datatype with inductive definition. Here's the type of natural numbers.

```
data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ
```

[1] `Set` is the type of small types, and `Set₁` is the type of `Set`, and so on. They form a hierarchy of types.

The decimal number "4" is represented as `suc (suc (suc (suc zero)))`. Agda also accepts arabic literals if the datatype ℕ complies with certain language pragma.

Addition on ℕ can be defined as a recursive function.

```
_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc (x + y)
```

We define `_+_` by pattern matching on the first argument, which results in two cases: the base case, and the inductive step. We are allowed to make recursive calls, as long as the type checker is convinced that the function would terminate.

The underlines surrounding `_+_` act as placeholders for arguments, making it an infix function in this instance.

Dependent functions and type arguments Up till now everything looks much the same as in Haskell, but a problem arises as we move on to defining something that needs more power of abstraction. Take identity functions for example:

```
id-Bool : Bool → Bool
id-Bool x = x

id-ℕ : ℕ → ℕ
id-ℕ x = x
```

In order to define a more general identity function, those concrete types have to be abstracted away. That is, we need parametric polymorphism, and this is where dependent types come into play.

A dependent type is a type whose definition may depend on a value. A dependent function is a function whose result type may depend on the value of an argument.

In Agda, function types are denoted as:

```
A → B
```

Where `A` is the type of domain and `B` is the type of codomain. To let `B` depends on the value of `A`, the value has to *named*, in Agda we write:

```
(x : A) → B x
```

The value of `A` is named `x` and then fed to `B`. As a matter of fact, `A → B` is just a syntax sugar for `(_ : A) → B` with the name of the value being irrelevant. The underline _ here means "I don't bother naming it".

Back to our identity function, if `A` happens to be `Set`, the type of all small types, and the result type happens to be solely `x`:

```
(x : Set) → x
```

Voila, we have polymorphism, and thus the identity function can now be defined as:

```
id : (A : Set) → A → A
id A x = x
```

`id` now takes an extra argument, the type of the second arguement. `id Bool true` evaluates to `true`

Implicit arguments We have implemented an identity function and seen how polymorphism can be modeled with dependent types. However, the additional argument that the identity function takes is rather unnecessary, since its value can always be determined by looking at the type of the second argument.

Fortunately, Agda supports *implicit arguments*, a syntax sugar that could save us the trouble of having to spell them out. Implicit arguments are enclosed in curly brackets in the type expression. We are free to dispense with these arguments when their values are irrelevant to the definition.

```
id : {A : Set} → A → A
id x = x
```

Or when the type checker can figure them out on function application.

```
val : Bool
val = id true
```

Any arguments can be made implicit, but it does not imply that values of implicit arguments can always be inferred or derived from context. We can always make them implicit arguments explicit on application:

```
val : Bool
val = id {Bool} true
```

Or when they are relevant to the definition:

```
silly-not : {_ : Bool} → Bool
silly-not {true}  = false
silly-not {false} = true
```

More syntax sugars We could skip arrows between arguments in parentheses or braces:

```
id : {A : Set} (a : A) → A
id {A} x = x
```

And there is a shorthand for merging names of arguments of the same type, implicit or not:

```
const : {A B : Set} → A → B → A
const a _ = a
```

Sometimes when the type of some value can be inferred, we could either replace the type with an underscore, say `(A : _)`, or we could write it as `∀ A`. For the implicit counterpart, `{A : _}` can be written as `∀ {A}`.

Parameterized Datatypes Just as functions can be polymorphic, datatypes can be parameterized by other types, too. The datatype of lists is defined as follows:

```
data List (A : Set) : Set where
    []  : List A
    _∷_ : A → List A → List A
```

The scope of the parameters extends over the entire declaration, so they can appear in the constructors. Here are the types of the datatype and its constructors.

```
infixr 5 _∷_

[] : {A : Set} → List A
_∷_ : {A : Set} → A → List A → List A
List : Set → Set
```

Where `A` can be anything, even `List (List (List Bool))`, as long as it is of type `Set`. `infixr` specifies the precedence of the operator `_∷_`.

Indexed Datatypes `Vec` is a datatype that is similar to `List`, but more powerful, in that it can tell you not only the type of its element, but also its length.

```
data Vec (A : Set) : ℕ → Set where
    []  : Vec A zero
    _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

`Vec A n` is a vector of values of type `A` and has the length of `n`. Here are some of its inhabitants:

```
nil : Vec Bool zero
nil = []

vec : Vec Bool (suc (suc zero))
vec = true ∷ false ∷ []
```

We say that `Vec` is *parameterized* by a type of `Set` and is *indexed* by values of ℕ. And we distinct indices from parameters. However, it is not obvious how they are different by looking at the declaration.

Parameters are *parametric*, in the sense that, they have no effect on the ”shape” of a datatype. The choice of parameters only effects which kind of values are placed there. Pattern matching on parameters does not reveal any insights about their whereabouts. Because they are *uniform* across all constructors, one can always replace the value of a parameter with another one of the same type.

On the other hand, indices may affect which inhabitants are allowed in the datatype. Different constructors may have different indices. In that case, pattern matching on indices may yield relevant information about their constructors.

For example, if there's term whose type is `Vec Bool zero`, then we are certain that the constructor must be `[]`, and if the type is `Vec Bool (suc n)` for some `n`, then the constructor must be `_∷_`.

We could for instance define a `head` function that cannot crash.

```
head : ∀ {A n} → Vec A (suc n) → A
head (x ∷ xs) = x
```

As a side note, parameters can be thought as a degenerate case of indices whose distribution of values are uniform across all constructors.

With abstraction Say, we want to define `filter` on `List`:

```
filter : ∀ {A} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) = ?
```

We are stuck here, because the result of `p x` is only available in runtime. Fortunately, with abstraction allows us to pattern match on the result of an intermediate computation by adding the result as an extra argument on the left-hand side:

```
filter : ∀ {A} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) with f x
filter p (x ∷ xs) | true  = x ∷ filter p xs
filter p (x ∷ xs) | false = filter p xs
```

Absurd patterns The *unit type*, or *top*, is a datatype inhabited by exactly one value, denoted `tt`.

```
data ⊤ : Set where
    tt : ⊤
```

The *empty type*, or *bottom*, on the other hand, is a datatype that is inhabited by nothing at all.

```
data ⊥ : Set where
```

These types seem useless, and without constructors, it is impossible to construct an instance of `⊥`. What is an type that cannot be constructed good for?

Say, we want to define a safe `head` on `List` that does not crash on any inputs. Naturally, in a language like Haskell, we would come up with a predicate like this to filter out empty lists `[]` before passing them to `head`.

```
non-empty : ∀ {A} → List A → Bool
non-empty []       = false
non-empty (x ∷ xs) = true
```

The predicate only works at runtime. It is impossible for the type checker to determine whether the input is empty or not at compile time.

However, things are quite different quite in Agda. With *top* and *bottom*, we could do some tricks on the predicate, making it returns a *Set*, rather than a *Bool*!

```
non-empty : ∀ {A} → List A → Set
non-empty []       = ⊥
non-empty (x ∷ xs) = ⊤
```

Notice that now this predicate is returning a type. So we can use it in the type expression. `head` can thus be defined as:

```
head : ∀ {A} → (xs : List A) → non-empty xs → A
head []       proof = ?
head (x ∷ xs) proof = x
```

In the `(x ∷ xs)` case, the arguement `proof` would have type `⊤`, and the right-hand side is simply `x`; in the `[]` case, the arguement `proof` would have type `⊥`, but what should be returned at the right-hand side?

It turns out that, the right-hand side of the `[]` case would be the least thing to worry about, because it is completely impossible to have such a case. Recall that `⊥` has no inhabitants, so if a case has an argument of that type, it is too good to be true.

Type inhabitance is in general an undecidable problem. However, when pattern matching on a type that is obviously empty (such as `⊥`), Agda allows us to drop the right-hand side and eliminate the arguement with `()`.

```
head : ∀ {A} → (xs : List A) → non-empty xs → A
head []       ()
head (x ∷ xs) proof = x
```

Whenever an empty list is applied to `head`, the resulting function would have type `head [] : ⊥ → A`, which is impossible to fulfill unless one could find a value of type `⊥`.

Propositions as types, proofs as programs The previous paragraphs are mostly about the *programming* aspect of the language, but there is another aspect to it. Recall the Curry–Howard correspondence, propositions are types and proofs are programs. A proof exists for a proposition the way that a value inhabits a type.

So `non-empty xs` is a type, but it can also be thought of as a proposition stating that `xs` is not empty. When `non-empty xs` evaluates to `⊥`, no value inhabits `⊥`, that means no proof exists for the proposition `⊥`; when `non-empty xs` evaluates to `⊥`, `tt` inhabits `⊥`, a trivial proof exists for the proposition `⊤`.

In intuitionistic logic, a proposition is considered to be ”true” when it is inhabited by a proof, and considered to be ”false” when there exists no proof.

Contrary to classical logic, where propositions evaluates to truth values. We can see that `⊤` and `⊥` correspondes to *true* and *false* in this sense.

Negation can be defined as a function from a proposition to `⊥`.

```
¬ : Set → Set
¬ P = P → ⊥
```

We could exploit `⊥` further to deploy the principle of explosion of intuitionistic logic, which states that: "from falsehood, anything (follows)" (Latin: *ex falso (sequitur) quodlibet*).

```
⊥-elim : ∀ {Whatever : Set} → ⊥ → Whatever
⊥-elim ()
```

Decidable propositions A proposition is decidable when it can be proved *or* disapproved. [2]

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

`Dec` is very similar to its two-valued cousin `Bool`, but way more powerful, because it also explains (with a proof) why a proposition holds or why it does not.

Suppose we want to know if a natural is even or odd. We know that `zero` is an even number, and if a number is even then its successor's successor is also even.

```
    data Even : ℕ → Set where
        base : Even zero
        step : ∀ {n} → Even n → Even (suc (suc n))
```

We also need the opposite of `step` as a lemma.

```
    2-steps-back : ∀ {n} → ¬ (Even n) → ¬ (Even (suc (suc n)))
    2-steps-back ¬p q = ?
```

`2-steps-back` takes two argument instead of one because the return type `¬ (Even (suc (suc n)))` is actually a synonym of `Even (suc (suc n)) → ⊥`.

[2] The connective *or* here is not a disjunction in the classical sense. Either way, a proof or a disproval has to be given.

Pattern matching on the second argument of type `Even (suc (suc n))` further reveals that it could only be constructed by `step`. By contradicting `¬p : ¬ (Even n)` and `p : Even n`, we complete the proof of this lemma.

```
    contradiction : ∀ {P Whatever : Set} → P → ¬ P → Whatever
    contradiction p ¬p = ⊥-elim (¬p p)

    two-steps-back : ∀ {n} → ¬ (Even n) → ¬ (Even (suc (suc n)))
    two-steps-back ¬p (step p) = contradiction p ¬p
```

Finally, `Even?` determines a number is even by induction on its predecessor's predecessor. `step` and `two-steps-back` can be viewed as functions that transforms proofs.

```
    Even? : (n : ℕ) → Dec (Even n)
    Even? zero          = yes base
    Even? (suc zero)    = no (λ ())
    Even? (suc (suc n)) with Even? n
    Even? (suc (suc n)) | yes p = yes (step p)
    Even? (suc (suc n)) | no ¬p = no  (two-steps-back ¬p)
```

The syntax of `λ ()` looks weird, as the result of contracting an argument of type `⊥` of a lambda expression `λ x → ?`. It is a convention to suffix a decidable function's name with `?`.

Propositional equality Saying that two things are "equal" is a notoriously intricate topic in type theory. There are many different notions of equality [10]. We will not go into each kind of equalities in depth but only skim through those exist in Agda.

Definitional equality, or *intensional equality* is simply a synonym, a relation between linguistic expressions. It is a primitive judgement of the system, stating that two things are the same to the type checker **by definition**.

Computational equality is a slightly more powerful notion. Two programs are consider equal if they compute (beta-reduce) to the same value. For example, `1 + 1` and `2` are equal in Agda in this notion.

But we cannot say that `a + b` and `b + a` are equal with definitional or computational equality, because this kind of equality is *extensional*. However, it could be expressed as a *proposition* with *identity types*.

```
data _≡_ {A : Set} (x : A) : A → Set where
    refl : x ≡ x
```

For all `a b : A`, if `a` and `b` are *computationally equal*, that is, both computes to the same value, then `refl` is a proof of `a ≡ b`, the *propositional equality* of `a` and `b`.

`_≡_` is an equivalence relation. It means that `_≡_` is *reflexive* (by definition), *symmetric* and *transitive*.

```
sym : {A : Set} {a b : A} → a ≡ b → b ≡ a
sym refl = refl

trans : {A : Set} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl
```

`_≡_` is congruent, meaning that we could **substitute equals for equals**.

```
cong : {A B : Set} {a b : A} → (f : A → B) → a ≡ b → f a ≡ f b
cong f refl = refl
```

Although these `refl`s look all the same at term level, they are proofs of different propositional equalities.

Dotted patterns Consider an alternative version of `sym` on ℕ.

```
sym' : (a b : ℕ) → a ≡ b → b ≡ a
sym' a b eq = ?
```

Where `eq` has type `a ≡ b`. If we pattern match on `eq` then Agda would rewrite `b` as `.a` and the goal type becomes `a ≡ a`.

```
sym' : (a .a : ℕ) → a ≡ a → a ≡ a
sym' a .a eq = ?
```

What happened under the hood is that `a` and `b` are *unified* as the same thing. The second argument is dotted to signify that it is *constrained* by the first argument `a`. `a` becomes the only argument available for further binding or pattern matching.

Standard library It would be inconvenient if we have to construct everything we need from scratch. Luckily, the community has maintained a standard library that comes with many useful and common constructions.

The standard library is not ”chartered” by the compiler or the type checker, there's simply nothing special about it. We may as well as roll our own library. [3]

3 Proving Properties of Numbers with Equational Reasoning

Because this thesis is mostly about numbers, we will put more focus on proving numerical properties and equational reasoning.

With propositional equality at our disposal, we will show how to prove properties such as the commutative property of addition. And as proofs get more complicated, we will introduce tools for equational reasoning to make life easier.

Right identity of addition Recap the definition of addition on ℕ.

```
_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc (x + y)
```

`_+_` is defined by induction on the first argument. That means we get the *left identity* of addition for free, as `zero + y` and `y` are *computationally equal*. But this is not the case for the *right identity* of addition. It has to be proven explicitly.

```
+-right-identity : (n : ℕ) → n + 0 ≡ n
+-right-identity zero    = ?0
+-right-identity (suc n) = ?1
```

By induction on the only argument, we get two sub-goals:

```
?0 : 0 ≡ 0
?1 : suc (n + 0) ≡ suc n
```

`?0` can be trivially proven with `refl`. The type of `?1` looks a lot like the proposition we are proving, except that both side of the equation are "coated" in a `suc`. With `cong suc : ∀ {x y} → x ≡ y → suc x ≡ suc y`, we could substitute something in `suc` with another if they are also equal, and finish the proof by recursively calling itself with a *smaller* argument.

[3]Some primitives that require special treatments, such as IO, are take cared with language pragmas exposed by Agda.

```
+-right-identity : ∀ n → n + 0 ≡ n
+-right-identity zero    = refl
+-right-identity (suc n) = cong suc (+-right-identity n)
```

Moving suc to the other side This is an essential lemma for proving other theorems. The proof also follows a similar pattern as that of `+-right-identity`. [4]

```
+-suc : ∀ m n → m + suc n ≡ suc (m + n)
+-suc zero    n = refl
+-suc (suc m) n = cong suc (+-suc m n)
```

Commutative property of addition Similarly, by induction on the first argument, we get two sub-goals:

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm zero    n = ?0
+-comm (suc m) n = ?1

?0 : n           ≡ m + zero
?1 : suc (m + n) ≡ m + suc n
```

`?0` can be solved with `+-right-identity` with a ”twist”. The symmetry of equality `sym` enable us to swap both sides of an equation.

```
+-comm zero    n = sym (+-right-identity n)
```

However, it is not obvious how to solve `?1` straight out. The proof has to be break into two steps:

1. Apply `+-suc` with `sym` to the right-hand side of the equation to get `suc (m + n) ≡ suc (n + m)`.
2. Apply the induction hypothesis to `cong suc`.

These small pieces of proofs are glued back together with the transitivity of equality `trans`.

```
+-comm (suc m) n = trans (cong suc (+-comm m n)) (sym (+-suc n m))
```

[4] In fact, all of these proofs (hense programs) can be generalized with some kinds of *fold*, but that is not the point here.

Equational Reasoning We see that proofs are composable just like programs. But look at the line we have just proven above:

```
trans (cong suc (+-comm m n)) (sym (+-suc n m))
```

It is difficult to see what is going on in between these clauses, and it could get only worse as propositions get more complicated. Imagine having dozens of `trans`, `sym` and `cong` spreading everywhere.

Fortunately, these complex proofs can be written in a concise and modular manner with a simple yet powerful technique called *equational reasoning*. Agda's flexible mixfix syntax allows the technique to be implemented with just a few combinators[1].

This is best illustrated by an example:

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm zero    n = sym (+-right-identity n)
+-comm (suc m) n =
    begin
        suc m + n
    ≡⟨ refl ⟩
        suc (m + n)
    ≡⟨ cong suc (+-comm m n) ⟩
        suc (n + m)
    ≡⟨ sym (+-suc n m) ⟩
        n + suc m
    ∎
```

With equational reasoning, we can see how a proposition equates with another, step by step, justified with theorems. `begin_` marks the beginning of a reasoning; `_≡⟨_⟩_` chains two propositions with the justication placed in between; `_∎` marks the end of a reasoning.

4 Num : a representation for positional numeral systems

In this section, we will demonstrate how to construct the representation for positional numeral systems in Agda. The representation is constructed as a datatype, indexed by the generalizations introduced in section 1.

- **base**: the base of a numeral system, denoted `b`.
- **#digit**: the number of digits, denoted `d`.
- **offset**: the number where the digits starts from, denoted `o`.

4.1 Digits

A system can only have **finitely many** digits. Operations on these digits, such as addition, must be **constant time**. Notice that the problem size of time complexity we are discussing here refers only to the value of a numeral. And since the value of digits is independent of the value of a numeral, time complexity of functions on digits should be trivially constant.

4.1.1 Fin

To represent a digit, we use a datatype that is conventionally called *Fin* which can be indexed to have some exact number of inhabitants.

```
data Fin : ℕ → Set where
    zero : {n : ℕ} → Fin (suc n)
    suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

The definition of `Fin` looks the same as `ℕ` on the term level, but different on the type level. The index of a `Fin` increases with every `suc`, and there can only be at most `n` of them before reaching `Fin (suc n)`. In other words, `Fin n` has exactly n inhabitants.

Fin is available in the stardard library, along with other auxiliary functions:

- `toℕ : ∀ {n} → Fin n → ℕ`
  converts from `Fin n` to `ℕ`.

- `fromℕ≤ : ∀ {m n} → m < n → Fin n`
  converts from `ℕ` to `Fin n` given the number is small enough.

- `#_ : ∀ m {n} {m<n : True (suc m N≤? n)} → Fin n`
  similar to `fromℕ≤`, but more convenient, since the proof of `m<n` is decidable thus can be inferred and made implicit.

- `inject≤ : ∀ {m n} → Fin m → m ≤ n → Fin n`
  converts a smaller `Fin` to a larger `Fin`.

4.1.2 Definition

`Digit` is simply just a synonym for `Fin`.

```
Digit : ℕ → Set
Digit = Fin
```

Binary digits for example can be represented as:

```
Binary : Set
Binary = Digit 2

零 : Binary
零 = zero

一 : Binary
一 = suc zero
```

4.1.3 Converting from and to natural numbers

Digit are evaluated together with the offset `o` of a system.

```
Digit-toℕ : ∀ {d} → Digit d → ℕ → ℕ
Digit-toℕ x o = toℕ x + o
```

Not all natural numbers can be converted to digits. The value has to be in certain range, between o and $d + o$.

```
Digit-fromℕ : ∀ {d} → (n o : ℕ) → d + o ≥ n → Digit (suc d)
Digit-fromℕ {d} n o upper-bound with n ∸ o ≤? d
Digit-fromℕ {d} n o upper-bound | yes p = fromℕ≤ (s≤s p)
Digit-fromℕ {d} n o upper-bound | no ¬p = contradiction p ¬p
    where   p : n ∸ o ≤ d
            p = start
                    n ∸ o
                ≤⟨ ∸n-mono o upper-bound ⟩
                    (d + o) ∸ o
                ≤⟨ reflexive (m+n∸n≡m d o) ⟩
                    d
                □
```

4.1.4 Properties

4.2 Num

Numerals in positional numeral systems are composed of sequences of **digits**.

4.2.1 Definition

The definition of `Numeral` is similar to that of `List`, except that a `Numeral` must contain at least one digit while a list may contain no elements at all. The most significant digit is placed in `_•` while the least significant digit is placed at the end of the sequence. `Numeral` is indexed by all three generalizations.

```
infixr 5 _∷_

data Numeral : ℕ → ℕ → ℕ → Set where
    _•  : ∀ {b d o} → Digit d → Numeral b d o
    _∷_ : ∀ {b d o} → Digit d → Numeral b d o → Numeral b d o
```

The decimal number ”2016” for example can be represented as:

```
MMXVI : Numeral 10 10 0
MMXVI = # 6 ∷ # 1 ∷ # 0 ∷ (# 2) •
```

4.2.2 Converting to natural numbers

Converting to natural numbers is fairly trivial:

```
⟦_⟧ : ∀ {b d o} → (xs : Numeral b d o) → ℕ
⟦_⟧ {_} {_} {o} (x •)    = Digit-toℕ x o
⟦_⟧ {b} {_} {o} (x ∷ xs) = Digit-toℕ x o + ⟦ xs ⟧ * b
```

5 Dissecting Num: Properties of different kinds of numeral systems

There are many kinds of numeral systems inhabit in `Num`. Some have infinitely many numerals and some have none.

We sort the systems in `Num` into four groups, each of them have different interesting properties.

5.1 Views

```
data NumView : ℕ → ℕ → ℕ → Set where
    NullBase    : ∀   d o                          → NumView 0
(suc d) o
    NoDigits    : ∀ b o                            → NumView b
0       o
    AllZeros    : ∀ b                              → NumView (suc b) 1
0
    Proper      : ∀ b d o → (proper : suc d + o ≥ 2) → NumView (suc b) (suc d) o
\begin{lstlisting}
```

```
\subsection{Maximum}

A number is said to be \textit{maximum} if there are no other number greater tha
itself.

\begin{lstlisting}
Maximum : ∀ {b d o} → (xs : Numeral b d o) → Set
Maximum {b} {d} {o} xs = ∀ (ys : Numeral b d o) → ⟦ xs ⟧ ≥ ⟦ ys ⟧
```

6 Conclusions

References

[1] P. V. Erik Hesselink. Equational reasoning in agda. Presentation slides, sep 2008.

[2] R. Hinze et al. Numerical representations as higher order nested datatypes. Technical report, Citeseer, 1998.

[3] D. E. Knuth. The art of computer programming. volume 2, seminumerical algorithms. 1998.

[4] J. Malakhovski. Brutal [meta]introduction to dependent types in agda, mar 2013.

[5] P. Martin-Lef. Intuitionistic type theory. *Naples: Bibliopolis*, 76, 1984.

[6] S.-C. Mu. Dependently typed programming. Lecture handouts, jul 2016.

[7] U. Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[8] C. Okasaki. *Purely functional data structures.* PhD thesis, Citeseer, 1996.

[9] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[10] T. B. Urs Schreiber. equality, dec 2016.