Generalizing Positional Numeral Systems

Luā Tîng-Giān

December 19, 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.

Chapter 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. Take the number “2016” for example:

Numeral system	**notation**
Chinese numerals	兩千零一十六
Roman numerals	MMXVI
Egyptian numerals	𓆼𓆼𓎆𓏺𓏺𓏺𓏺𓏺𓏺

Even so, 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**.

1.1.1 Digits

Any set of symbols can be used as digits as long as we know how to *assign* each digit to the value it represents.

Numeral system	**Digits**															
decimal	0	1	2	3	4	5	6	7	8	9						
binary	0	1														
hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Assigned value	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

We place a bar above a digit to indicate its assignment. Below is the assignments of hexadecimal digits.

$$\begin{array}{cccc}
\bar{0} \mapsto 0 & \bar{1} \mapsto 1 & \bar{2} \mapsto 2 & \bar{3} \mapsto 3 \\
\bar{4} \mapsto 4 & \bar{5} \mapsto 5 & \bar{6} \mapsto 6 & \bar{7} \mapsto 7 \\
\bar{8} \mapsto 8 & \bar{9} \mapsto 9 & \bar{A} \mapsto 10 & \bar{B} \mapsto 11 \\
\bar{C} \mapsto 12 & \bar{D} \mapsto 13 & \bar{E} \mapsto 14 & \bar{F} \mapsto 15
\end{array}$$

Positional numeral systems represent a number by lining up a series of digits:

$$\xrightarrow{2016}$$

In this case, 6 is called the *least significant digit*, and 2 is known as the *most significant digit*. Except when writing decimal numbers, we will write down numbers in reverse order, from the least significant digit to the most significant digit like this

$$\xleftarrow{6102}$$

1.1.2 Syntax and Semantics

Syntax bears no meaning; its semantics can only be expressed through the process of *converting* to some other syntax. Numeral systems are merely syntax. The same notation can represent different numbers in different context.

Take the notation ”11” for example; it could have several meanings.

Numeral system	**number in decimal**
decimal	11
binary	3
hexadecimal	17

To make things clear, we call a sequence of digits a **numeral**, or **notation**; the number it expresses a **value**, or simply a **number**; the process that converts notations to values an **evaluation**. From now on, **numeral systems** only refer to the positional ones. We will not concern ourselves with other kinds of numeral systems.

1.1.3 Evaluating Numerals

What we mean by a *context* in the previous section is the **base** of a numeral system. 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.

Numeral system	**Base**	**Digits**															
decimal	10	0	1	2	3	4	5	6	7	8	9						
binary	2	0	1														
hexadecimal	16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Assigned value		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

A numeral system of base n has exactly n digits, which are assigned values from 0 to $n-1$.

Conventionally, the base of a system is annotated by subscripting it to the right of a numeral, like $(2016)_{10}$. We replace the parenthesis with a fancy pair of semantics brackets, like $[\![2016]\!]_{10}$ to emphasize its role as the evaluation function.

To evaluate a notation of a certain base:

$$[\![d_0 d_1 d_2 ... d_n]\!]_{base} = \bar{d_0} \times base^0 + \bar{d_1} \times base^1 + \bar{d_2} \times base^2 + ... + \bar{d_n} \times base^n$$

Where d_n is a digit for all n.

1.2 Unary Numbers and Peano Numbers

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, if we are to evaluate unary numerals with the model we have just settled, the only digit of the unary system would have to be assigned as 0 and every numeral would evaluate to zero as a result.

problem The definition of digit assignments can be modified to allow unary digits to start counting from 1, but that would lead to inconsistency among systems of other bases. We will resolve this inconsistency by generalizing digit assignments in the next chapter.

Numeral system	**Base**	**Digits**															
decimal	10	0	1	2	3	4	5	6	7	8	9						
binary	2	0	1														
hexadecimal	16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
unary	1		1														
Assigned value		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

1.3 Binary Numerals in Digital Circults

Recall how arithmetics such as long addition are performed by hand.

$$\begin{array}{r} 1\,2\,3 \\ +\ \ 3\,4 \\ \hline 1\,5\,7 \end{array}$$

The greater a number is, the longer its notation will be, which in terms determines the time it takes to perform operations. Since a system can only have **finitely many** digits, operations such as addition on these digits must be **constant time**. Consequently, the time complexity of operations such as long addition on a numeral would be $O(lgn)$ at best. The choice of the base is immaterial as long as it is not unary (which would degenerate to $O(n)$).

However, this is not the case for the binary numeral system implemented in arithmetic logic units (ALU). These digital circuits are designed to perform fast arithmetics. Regarding addition, it takes only *constant time*.

It seems that either we have been doing long addition wrong since primary school, or the chip manufacturers have been cheating all the time. But there's a catch! Because we are capable of what is called *arbitrary-precision* *arithmetic*, i.e., we 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*.

problem Judging from the time complexity of operations, the binary numerals running in digital circuits is certainly different from the ordinary binary numerals we have known. In the next chapter, we will show that these special characteristics can be captured by ordinary numeral systems with just a little tweak.

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 precisely what the `length` function, a homomorphism from lists to unary numbers, does.

Now let's compare addition on unary numbers 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].

«««< HEAD A container with n elements can be modeled after the representation of the number n. These containers are composed of smaller building blocks that house elements as numerals are composed of digits.

problem However, conventional numeral systems are incapable of modelling these numerical representations.

[explain why] [insert a table of numerical representation and their correspondense numeral systems]

We need a more versatile numeral system to accommodate these numerical representations.

1.5 Outline

The remainder of the thesis is organized as follows. Chapter 2 resolves the problems we have addressed in this chapter by proposing some generalizations to the conventional positional numeral systems. Chapter 3 gives a introduction to *Agda*, the language we use to construct and formalize the representation. Chapter 4 introduces *equational reasoning* and relevent properties of natural numbers used in the rest of the thesis. Chapter **??** constructs the representation for numeral systems and develops properties of such representation.

Chapter 2

Generalizations

2.1 Base

Recall the evaluation function.

$$[\![d_0 d_1 d_2 ... d_n]\!]_{base} = \bar{d_0} \times base^0 + \bar{d_1} \times base^1 + \bar{d_2} \times base^2 + ... + \bar{d_n} \times base^n$$

Where $\bar{d_n}$ ranges from 0 to $base - 1$ for all n.

As we can see the base of numeral systems has already been generalized. But nonetheless, it is a good basis for further generalizations.

2.2 Offset

To cooperate unary numerals, we relax the constraint on the range of digit assignment by introducing a new variable, *offset*:

$$[\![d_0 d_1 d_2 ... d_n]\!]_{base} = \bar{d_0} \times base^0 + \bar{d_1} \times base^1 + \bar{d_2} \times base^2 + ... + \bar{d_n} \times base^n$$

The evaluation of numerals remains the same but the assignment of digits has changed from

$$0, 1, ..., \textit{offset - 1}$$

to

$$\textit{offset}, \textit{offset + 1}, ..., \textit{offset + base - 1}$$

The codomain of the digit assignment function is *shifted* by *offset*. Now that unary numerals would have an offset of 1 and systems of other bases would have offsets of 0.

Systems with an offset of 1 are known as *bijective numerations* because every number can be represented by exactly one numeral. In other words, the evaluation function is bijective.

Let us see how to count to ten in a binary numeral system with an offset of 1. [1]

Number	**Numeral**
1	1
2	2
3	11
4	21
5	12

Number	**Numeral**
6	22
7	111
8	211
9	121
10	221

Such a numeral system is also named *1-2 binary system* because its digits are assigned 1 and 2. Notice that how the symbol of digits are deliberately chosen to match their assigned value.

[1] As a reminder, the order of digits are reversed.

Numeral system	**Base**	**Offset**
decimal	10	0
binary	2	0
hexadecimal	16	0
unary	1	1
1-2 binary	2	1

A numeral system is said to be *zeroless* if no digits are assigned 0, i.e., offset > 0. Data structures modeled after zeroless systems are called *zeroless representations*. These containers are preferable to their ”zeroful” counterparts. Because a digit of value 0 corresponds to a building block with 0 elements, and a building block that contains no element is not only useless, but also hinders traversal as it takes time to skip over these empty nodes.

2.3 Number of Digits

The binary numeral system running in circuits looks different from what we have in hand. Surprisingly, these binary numbers can fit 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 and their assignments range from:

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

By introducing a new index *#digit* to generalize the number of digits, their assignments range from:

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

Taking *#digit* into account, here is a table of all the numeral systems we have addressed along with machine integer types such as *Int32*.

Numeral system	**Base**	**#Digit**	**Offset**
decimal	10	10	0
binary	2	2	0
hexadecimal	16	16	0
unary	1	1	1
1-2 binary	2	2	1
Int32	2	2^{32}	0
Int64	2	2^{64}	0

Chapter 3

A gentle 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 thesis.

Some of the more advanced constructions (such as views and universes) will not be introduced in this chapter, but in other places where we need them.

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

3.1 Some basics

Agda is a *dependently typed functional programming language* and also an *interactive proof assistant*. This language can serve both purposes because it is 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 Ph.D. at Chalmers University of Technology.

We say that Agda is interactive because theorem proving involves a lot of conversations between the programmer and the type checker. Moreover, it is often difficult, if not impossible, to develop and prove a theorem 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 example.

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

We can leave out the right-hand side and ask: "what's the type of the goal?", "what's the context of this case?", etc. Agda would reply us with:

```
GOAL : Bool
x : ℕ
```

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

```
is-zero : ℕ → Bool
```

```
is-zero zero    = ?
is-zero (suc x) = ?
```

We could fulfill these goals by giving an answer, or even ask Agda to solve the problem (by pure guessing) 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 is not much point in running an Agda program, because it is mostly about compile-time static constructions. This is what programming and proving things looks like in Agda.

3.2 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
```

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

1. `Bool` has type `Set`[1]

2. `true` has type `Bool`

3. `false` has type `Bool`

Pattern matching Similar to Haskell, datatypes are eliminated by pattern matching. Here is a function that pattern matches on `Bool`.

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

Agda is a *total* language, which means that partial functions are not valid constructions. Programmers are obliged to convince Agda that a program terminates and does not crash on all possible inputs. The following example will not be accepted by the termination checker because the case `false` is missing.

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

Inductive datatype Let us move on to a more interesting datatype with an inductive definition. Here is the type of natural numbers.

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

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

The decimal number ”4” is represented as `suc (suc (suc (suc zero)))`. Agda also accepts decimal 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.

Those 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 need 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 type may depend on a value of its arguments.

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 argument. `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
```

Also, 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 spreads over the entire declaration so that 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 encodes 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`. We distinguish 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 into 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, given a 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 is 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 at 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 argument `proof` would have type `⊤`, and the right-hand side is simply `x`; in the `[]` case, the argument `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 argument with `()`.

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

Whenever `head` is applied to some list `xs`, the programmer is obliged to convince Agda that `non-empty xs` reduces to `⊤`, which is only possible when `xs` is not an empty list. On the other hand, applying an empty list to `head` would result in a function of type `head [] : ⊥ → A` which is impossible to be fulfilled.

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.

`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 `⊥`, which 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. «««< HEAD Contrary to classical logic, where every propositions are assigned one of two truth values. We can see that `⊤` and `⊥` corresponds to *true* and *false* in this sense.

Negation is 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
```

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

`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 number 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 arguments instead of one because the return type `¬ (Even (suc (suc n)))` is actually a synonym of `Even (suc (suc n)) → ⊥`. 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 be even by induction on its predecessor's predecessor. `step` and `two-steps-back` can be viewed as functions that transform 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.

However, expressions such as `a + b` and `b + a` are not considered equal by Agda, neither *definitionally* nor *computationally*, because there are simply no rules in Agda saying so.

`a + b` and `b + a` are only *extensionally equal* in the sense that, given **any** pair of numbers, say `1` and `2`, Agda can see that `1 + 2` and `2 + 1` are computationally equal. But when it comes to **every** pair of numbers, Agda fails to justify that.

We could convince Agda about the fact that `a + b` and `b + a` are equal for every pair of `a` and `b` by encoding this theorem in a *proposition* and then prove that the proposition holds. This kind of proposition can be expressed with *identity types*.

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

This inductive datatype says that: 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`, and we say that `a` and `b` are *propositionally equal*!

`_≡_` 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 refl = ?
```

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] Some primitives that require special treatments, such as IO, are taken care of by language pragmas provided by Agda.

Chapter 4

Properties of Natural Numbers and Equational Reasoning

Properties of natural numbers play a big role in the development of proofs in this thesis. With propositional equality at our disposal, we will demonstrate how to prove properties such as the commutative property of addition. As proofs get more complicated, we will make proving easier by introducing a powerful tool: *equational reasoning*.

4.1 Proving Equational Propositions

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*. However, this is not the case for the *right identity*. 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`. As for `?1`, we see that its type looks a lot like the proposition we are proving, except that both sides of the equation are "coated" with a `suc`. With `cong suc : ∀ {x y} → x ≡ y → suc x ≡ suc y`, we could substitute a term in `suc` with another if they are equal, and finish the proof by recursively calling itself with a *smaller* argument.

```
+-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 more advanced theorems. The proof also follows a similar pattern as that of `+-right-identity`. [1]

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

[1] In fact, all of these proofs (hence programs) can be generalized with a *fold*, but that is not the point here.

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` enables 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))
```

4.2 Equational Reasoning

We see that proofs are composable just like programs. However, 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 an expression equates with another, step by step, justified with theorems. The first and the last step corresponds to two sides of the equation of a proposition. `begin_` marks the beginning of a reasoning; `_≡⟨_⟩_` chains two expressions with the justification placed in between; `_∎` marks the end of a reasoning (*QED*).

4.2.1 Anatomy of Equational Reasoning

A typical equational reasoning can often be broken down into **three** parts.

1. Starting from the left-hand side of the equation, through a series of steps , the expression will be ”arranged” into a form that allows the induction hypothesis to be applied. In the following example of `+-comm`, nothing needs to be arranged because these two expressions are computationally equal (the `refl` can be omitted).

```
begin
    suc m + n
≡⟨ refl ⟩
    suc (m + n)
```

2. `m + n` emerged as part of the proposition which enables us to apply the induction hypothesis.

```
    suc (m + n)
≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
```

3. After applying the induction hypothesis, the expression are then ”rearranged” into the right-hand side of the equation, hence completes the proof.

```
    suc (n + m)
≡⟨ sym (+-suc n m) ⟩
    n + suc m
∎
```

arranging expressions To arrange an expression into the shape we desire as in part 1 and part 3, while remaining equal. We need properties such as commutativity or associativity of some operator, or distributive properties when there is more than one operator.

The operators we will be dealing with often comes with these properties. Take addition and multiplication, for example; together they form a nice semiring structure.

substituting equals for equals As what we have seen in 2, sometimes there is only a part of an expression needs to be substituted. Say, we have a proof `eq : X ≡ Y`, and we want to substitute `X` for `Y` in a more complex expression `a b (c X) d`. We could ask `cong` to "target" the part to substitute by supplying a function like this:

```
λ w → a b (c w) d
```

Which abstracts the part we want to substitute away, such that:

```
cong (λ w → a b (c w) d) eq : a b (c X) d ≡ a b (c Y) d
```

4.3 Preorder

Aside from stating that two expressions are equal, a proposition can also state that one expression is "less than or equal to" than another, given a preorder.

A preorder is a binary relation that is `reflexive` and `transitive`. Often denoted as `_≤_`, such a binary relation on natural numbers is defined as:

```
data _≤_ : ℕ → ℕ → Set where
    z≤n : ∀ {n}                  → zero  ≤ n
```

```
    s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
```

The following is a proof of $3 \leq 5$:

```
3≤5 : 3 ≤ 5
3≤5 = s≤s (s≤s (s≤s z≤n))
```

To prove $3 \leq 5$, we need a proof of $2 \leq 4$ for `s≤s`, and so on, until it reaches zero where it ends with a `z≤n`.

Here are some other binary relations than can be defined with `_≤_`.

```
_<_ : Rel ℕ Level.zero
m < n = suc m ≤ n

_≰_ : Rel ℕ Level.zero
a ≰ b = ¬ a ≤ b

_≥_ : Rel ℕ Level.zero
m ≥ n = n ≤ m
```

4.4 Preorder reasoning

Combinators for equational reasoning can be further generalized to support *preorder reasoning*. Preorders are `reflexive` and `transitive`, that means expressions can be chained with a series of relations just as that of equational reasoning.

Suppose we already have `m≤m+n : ∀ m n → m ≤ m + n` and we want to prove a slightly different theorem.

```
m≤n+m : ∀ m n → m ≤ n + m
m≤n+m m n =
    start
```

```
    m
≤⟨ m≤m+n m n ⟩
    m + n
≈⟨ +-comm m n ⟩
    n + m
□
```

Where `_≤⟨_⟩_` and `_≈⟨_⟩_` are respectively transitive and reflexive combinators.[2] Step by step, starting from the left-hand side of the relation, expressions get greater and greater as it reaches the right-hand side the relation.

monotonicity of operators In equational reasoning, we could substitute part of an expression with something equal with `cong` because `_≡_` is congruent. However, we cannot substitute part of an expression with something *greater* in general.

Take the following function `f` as example.

```
f : ℕ → ℕ
f 0 = 1
f 1 = 0
f _ = 1
```

`f` returns 1 on all inputs except for 1. `0 ≤ 1` holds, but it does not imply that `f 0 ≤ f 1` also holds. As a result, a generic mechanism like `cong` does not exist in preorder reasoning. We can only substitute part of an expression when the function is *monotonic*.

[2]Combinators for preorder reasoning are renamed to prevent conflictions with equational reasoning.

4.5 Skipping trivial proofs

From now on, we will dispense with most of the steps and justifications in equational and preorder reasonings, because it is often obvious to see what happened in the process.

In fact, there are is no formal distinction between the proofs we disregard and those we feel important. They are all equally indispensable to Agda.

4.6 Relevant Properties of Natural Numbers

Relevant properties of ℕ used in the remainder of the thesis will be introduced in this section.

Some of the properties listed here are taken from the stantard library. [3]

4.6.1 Equational Propositions

natural number

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

- `cancel-suc : ∀ {x y} → suc x ≡ suc y → x ≡ y`
  suc is injective.

addition

[3] *Theorem*, *lemma*, *corollary* and *property* are all synonyms for *established proposition*. There are no formal distinction between these terms and they are used exchangeably in the thesis.

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

- `+-right-identity : ∀ n → n + 0 ≡ n`

  the right identity of addition.

- `+-suc : ∀ m n → m + suc n ≡ suc (m + n)`

  moving `suc` from one term to another.

- `+-assoc : ∀ m n o → (m + n) + o ≡ m + (n + o)`

  the associative property of addition.

- `+-comm : ∀ m n → m + n ≡ n + m`

  the commutative property of addition.

- `[a+b]+c≡[a+c]+b : ∀ a b c → a + b + c ≡ a + c + b`

  a convenient corollary for swapping terms.

- `a+[b+c]≡b+[a+c] : ∀ a b c → a + (b + c) ≡ b + (a + c)`

  a convenient corollary for swapping terms.

- `cancel-+-left : ∀ i {j k} → i + j ≡ i + k → j ≡ k`

  the left cancellation property of addition.

- `cancel-+-right : ∀ k {i j} → i + k ≡ j + k → i ≡ j`

  the right cancellation property of addition.

multiplication

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

- `*-right-zero : ∀ n → n * 0 ≡ 0`

  the right absorbing element of multiplication.

- `*-left-identity : ∀ n → 1 * n ≡ n`

  the left identity of addition multiplication.

- `*-right-identity : ∀ n → n * 1 ≡ n`

  the right identity of addition multiplication.

- `+-*-suc : ∀ m n → m * suc n ≡ m + m * n`

  multiplication over `suc`.

- `*-assoc : ∀ m n o → (m * n) * o ≡ m * (n * o)`

  the associative property of multiplication.

- `*-comm : ∀ m n → m * n ≡ n * m`

  the commutative property of multiplication.

- `distribʳ-*-+ : ∀ m n o → (n + o) * m ≡ n * m + o * m`

  the right distributive property of multiplication over addition.

- `distrib-left-*-+ : ∀ m n o → m * (n + o) ≡ m * n + m * o`

  the left distributive property of multiplication over addition.

monus

Monus, or *truncated subtraction*, is a kind of subtraction that never goes negative when the subtrahend is greater than the minued.

```
_∸_ : Nat → Nat → Nat
n     ∸ zero = n
zero  ∸ suc m = zero
suc n ∸ suc m = n ∸ m
```

- `0∸n≡0 : ∀ n → 0 ∸ n ≡ 0`
- `n∸n≡0 : ∀ n → n ∸ n ≡ 0`
- `m+n∸n≡m : ∀ m n → (m + n) ∸ n ≡ m`
- `m+n∸m≡n : ∀ {m n} → m ≤ n → m + (n ∸ m) ≡ n`
- `m∸n+n≡m : ∀ {m n} → n ≤ m → m ∸ n + n ≡ m`
- `∸-+-assoc : ∀ m n o → (m ∸ n) ∸ o ≡ m ∸ (n + o)`

  the associative property of monus and addition.
- `+-∸-assoc : ∀ m {n o} → o ≤ n → (m + n) ∸ o ≡ m + (n ∸ o)`

  the associative property of monus and addition.
- `*-distrib-∸ʳ : ∀ m n o → (n ∸ o) * m ≡ n * m ∸ o * m`

  the right distributive property of monus over multiplication.

min and max

So called `min` and `max` in Haskell. Min `_⊓_` computes the lesser of two numbers.

```
_⊓_ : ℕ → ℕ → ℕ
zero  ⊓ n     = zero
suc m ⊓ zero  = zero
suc m ⊓ suc n = suc (m ⊓ n)
```

Max `_⊔_` computes the greater of two numbers.

```
_⊔_ : ℕ → ℕ → ℕ
zero  ⊔ n     = n
suc m ⊔ zero  = suc m
suc m ⊔ suc n = suc (m ⊔ n)
```

- `⊓-comm : ∀ m n → m ⊓ n ≡ n ⊓ m`

  the commutative property of min.

- `⊔-comm : ∀ m n → m ⊔ n ≡ n ⊔ m`

  the commutative property of max.

4.6.2 Relational Propositions

natural number

- `≤-pred : ∀ {m n} → suc m ≤ suc n → m ≤ n`

addition

- `≤-step : ∀ {m n} → m ≤ n → m ≤ 1 + n`
- `≤-steps : ∀ {m n} k → m ≤ n → m ≤ k + n`
- `m≤m+n : ∀ m n → m ≤ m + n`
- `n≤m+n : ∀ m n → n ≤ m + n`
- `_+-mono_ : ∀ {m₁ m₂ n₁ n₂} → m₁ ≤ m₂ → n₁ ≤ n₂ → m₁ + n₁ ≤ m₂ + n₂`

  the monotonicity of addition

- `n+-mono : ∀ {i j} n → i ≤ j → n + i ≤ n + j`

  `_+-mono_` with the first argument fixed.

- `+n-mono : ∀ {i j} n → i ≤ j → n + i ≤ n + j`

  `_+-mono_` with the second argument fixed.

- `n+-mono-inverse : ∀ n → ∀ {a b} → n + a ≤ n + b → a ≤ b`

  the inverse of `n+-mono`

- `+n-mono-inverse : ∀ n → ∀ {a b} → a + n ≤ b + n → a ≤ b`

  the inverse of `+n-mono`

- `+-mono-contra : ∀ {a b c d} → a ≥ b → a + c < b + d → c < d`

-

-

Chapter 5

Constructions

5.1 Digit: the basic building block

Numerals are composed of sequences of **digits**. We will demonstrate how to choose a suitable representation for digits in this section.

The same digit may represent different values in different numeral systems, so it is essential to make the context clear. Here are the generalizations introduced in section 1 that may effect the evaluation of a digit.

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

5.1.1 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` would have 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`.

5.1.2 Definition

`Digit` is simply just a synonym for `Fin`, indexed by the number of digits `d` of a system.

```
Digit : ℕ → Set
Digit d = Fin d
```

Binary digits for example can thus be represented as:

```
Binary : Set
Binary = Digit 2
```

```
零 : Binary
零 = zero

一 : Binary
一 = suc zero
```

5.1.3 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
```

However, not all natural numbers can be converted to digits. The value has to be in a certain range, between o and $d + o$. Values less than o are [synonym of truncated] to o. Values greater than $d + o$ are prohibited by `upper-bound : d + o ≥ n`.

```
Digit-fromℕ : ∀ {d}
    → (n o : ℕ)
    → (upper-bound : d + o ≥ n)
    → Digit (suc d)
Digit-fromℕ = ...
```

Properties `Digit-fromℕ-toℕ` states that the value of a natural number should remain the same, after converted back and forth between `Digit` and ℕ.

```
Digit-fromℕ-toℕ : ∀ {d o}
    → (n : ℕ)
    → (lower-bound :     o ≤ n)
```

```
    → (upper-bound : d + o ≥ n)
    → Digit-toℕ (Digit-fromℕ {d} n o upper-bound) o ≡ n
Digit-fromℕ-toℕ = ...
```

Digits have a upper-bound and a lower-bound after evaluation.

```
Digit-upper-bound : ∀ {d} → (o : ℕ) → (x : Digit d) → Digit-toℕ x o < d + o
Digit-upper-bound {d} o x = +n-mono o (bounded x)

Digit-lower-bound : ∀ {d} → (o : ℕ) → (x : Digit d) → Digit-toℕ x o ≥ o
Digit-lower-bound {d} o x = m≤n+m o (toℕ x)
```

5.1.4 Constants

These ”constants” are special digits that inhabited in each system.

The greatest digit

The carry

5.2 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`.

Properties

5.2.1 Num

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

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) •
```

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
```

5.3 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.

5.3.1 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 ⟧
```

5.4 Conclusions

Bibliography

[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.