Generalizing Positional Numeral Systems

January 12, 2017

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	𓆼𓆼 𓎆 𓏿

Even so, most of the systems we are using today are positional notations[4] 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. For instance, these are 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.

Problem 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 to 0 and every numeral would evaluate to zero as a result.

The definition of digit assignments can be modified to allow unary digits to start counting from 1.

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**

However, the representation for numeral systems would have to be generalized to manage the inconsistency of digit assignment among systems of different bases. This generalization will be introduced in Chapter 2.

Moreover, theorems developed on natural numbers à la Peano cannot be migrated to numbers of other representations, although the proofs are mostly identical. Because the relation and similarities between these two representations we have observed and described are uttered with a *metalanguage* (English, in this case), while those proofs are often written in some *object language*. We will demonstrate how to *encode* propositions expressed in the metalanguage to an object language that we have constructed and how to manipulate them

1.3 Binary Numerals in Digital Circults

Suppose we are asked to anwser the questions below.

$$\begin{array}{r} 2\,5\,3\,2\,9\,8\,1\,2\,3 \\ +\,3\,4\,7\,8\,4\,4\,2\,3\,5 \\ \hline ? \end{array} \qquad\qquad \begin{array}{r} 1\,2\,3 \\ +\quad 3\,4 \\ \hline ? \end{array}$$

It would take much more effort to perform long addition and anwser the question on the left, because it has greater numbers. 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 can be implemented in **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.

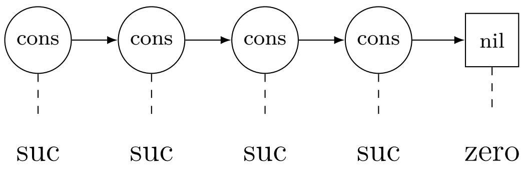
1.4 Numerical representation

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



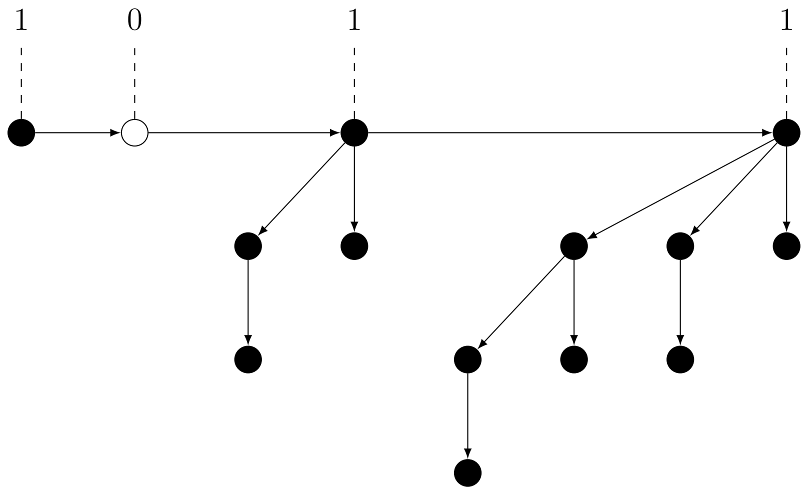
Now let's compare addition on unary numbers and merge (append) on lists:

```
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)
```

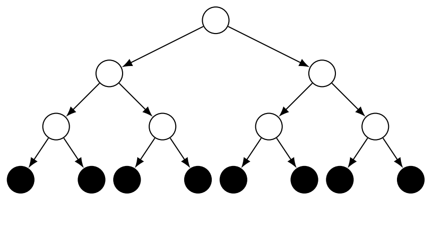
Aside from having virtually identical implementations, operations on unary numbers and lists both have the same time complexity. Incrementing a unary number takes $O(1)$, inserting an element into a list also takes $O(1)$; adding two unary numbers takes $O(n)$, appending a list to another also takes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

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



The figure above is a binomial heap containing 13 elements. [1] From left to right, there are *binomial trees* of different *ranks* attached to the path that we call *"the spine"*. A binomial heap is composed of binomial trees just as a numeral is composed of digits. If we read the nodes with binomial trees as 1 and those without as 0, then we get the numeral of 13 in binary.

building blocks Single cells in lists and binomial trees in binomial heaps are all different kind of simple data structures that are called *building blocks*. There are also other kinds of building blocks, such as perfect leaf trees.

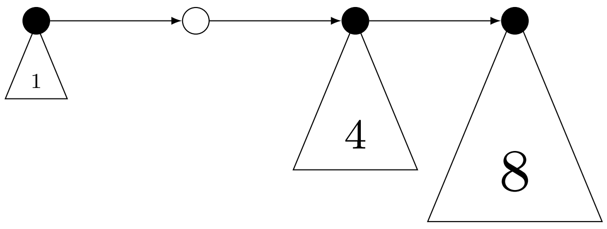


These building blocks can have different ranks. A binary leaf tree of rank n, for instance, would contain 2^n elements. The data structures we have addressed so far are all composed of a series of building blocks that are ordered by their ranks.

However, these building blocks do not necessarily have to be binary based, as long as multiple building blocks of the same rank can be merged into a building block of a higher rank or vice versa.

[1]Nodes that contain elements are painted black.

random access lists and binary numbers Accessing an element on lists typically takes $O(n)$. Instead of using single cells, *random access lists* adopts perfect leaf trees as building blocks. This improves the time complexity of random access from $O(n)$ to $O(lgn)$ as a random access list would have at most $O(lgn)$ building blocks, and the tallest perfect leaf tree takes at most $O(lglgn)$ to traverse.



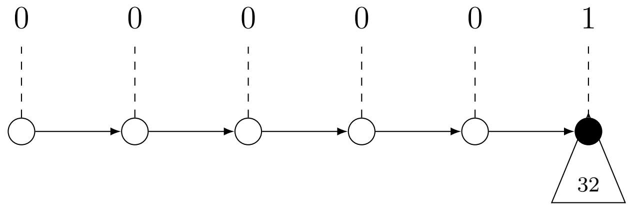
Similar to that of binomial heaps, random access lists also have spines. Also, treating building blocks as digits also yields binary numerals of the container's size.

1.4.1 The correspondence

The strong analogy between data structures and positional numeral systems suggests that numeral systems can serve as templates for designing containers. Such data structures are called **Numerical Representations**[10] [3].

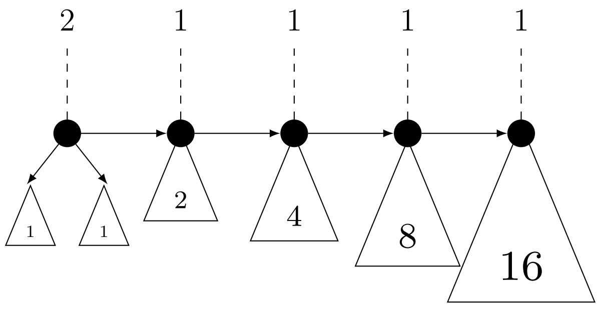
a container of size n	corresponds to	a numeral of n
a building block of rank n	corresponds to	a digit of weight $base^n$
inserting an element	corresponds to	incrementing a numeral
merging two containers	corresponds to	adding two numerals

Problem Retrieving the first element (`head`) of a list typically takes only constant time. On the other hand, it takes $O(lgn)$ on random access lists. To illustrate the problem, consider a random access list with 32 elements:



To access any elements from the list above, we have to skip through four empty nodes before reaching the first building block. Nodes that correspond to the digit “0” contain no elements. They are not only ineffective but also hinders traversal.

However, if we replace the digits “0” and “1” with “1” and “2”, then the number 32 can be represented as 21111 instead of 000001. By doing so, we eliminate empty nodes and shorten the spine, thus improving the performance of list traversal.



The data structure introduced above is called the *1-2 random access list*. Its presence suggests that a binary numeral system with digits “1” and “2” should be admissible.

Hinze argues[3] that if we add the digit “0” back to the *1-2 random access list*, then the resulting numerical representation, so-called *0-1-2 random access list*, would even have a better performance of insertion and deletion in certain edge cases.

To accommodate these numerical representations, we need a more versatile representation for numeral systems.

1.5 Motivation and Outline

We started off with numerical representations, but then we found that their corresponding numeral systems alone are interesting enough.

- How to capture the numeral systems behind these data structures?
- What are these numeral systems capable of?
- Which kinds of numeral systems are suitable for modelling numerical representations? Do they support increment (insertion) or addition (merge)?
- What are the relations between these numeral systems and the natural numbers?

- How to translate propositions and proofs from the natural numbers to our representation?

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 an introduction to *Agda*, the language we use to construct and reason about the representation.

Chapter 4 introduces *equational reasoning* and relevant properties of natural numbers exercised in the coming chapters.

Chapter 5 constructs the representation for these numeral systems, searches for suitable systems for modeling data structures by defining operations such as increment and addition, and investigates the relationships between these systems and the natural numbers.

Chapter 6 demonstrates how to translate propositions and proofs from the natural numbers à la Peano to our presentation of numbers with universe constructions.

Chapter ?? concludes everything.

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 start and we will continue to abstract more things away.

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

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

to

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

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.

1-2 binary system Recall *1-2 random access lists* from the previous chapter, which is the numerical representation of a binary numeral system with an offset of 1. Let us see how to count to ten in the 1-2 binary system. [1]

Number	**Numeral**	**Number**	**Numeral**
1	1	6	22
2	2	7	111
3	11	8	211
4	21	9	121
5	12	10	221

There are no restrictions on the symbols of digits. But nonetheless, it is reasonable to choose symbols that match their assigned values, as we choose the symbol "1" and "2" as digits for the 1-2 binary system.

bijective numerations Systems with an offset of 1 are also known as *bijective numerations* because every number can be represented by exactly one numeral. In other words, the evaluation function is bijective. The 1-2 binary system is one such numeration.

zeroless representations 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. The reason is that 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, as we have seen in random access lists from the previous chapter.

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

[1]As a reminder, the order of non-decimal numerals are reversed.

this, a 32-bit binary number (*Int32*) 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:

$$offset...offset + base - 1$$

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

$$offset...offset + \#digit - 1$$

Redundancy Numeral systems like *Int32* are said to be **redundant** because there is more than one way to represent a number. In fact, systems that admit 0 as one of the digits must be redundant, since we can always take a numeral and add leading zeros without changing it's value.

Incrementing a decimal numeral such as 999999 takes much more time than 999998 because carries or borrows can propagate. Redundancy provides "buffer" against operations these carries and borrows. In this case, incrementing 999999 and 999998 of *Int32* both results in a cost of constant time.

Numerical representations modeled after redundant numeral systems also enjoy similar properties. When designed properly, redundancy can improve the performance of operations of data structures.

2.4 Relations with Natural Numbers

The following table contains all of the numeral systems we have addressed so far, with *base*, *offset*, and *#digit* taken into account. [2]

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

[2] *Int32* and *Int64* are respectively 32-bit and 64-bit machine integers.

Although we are now capable of expressing those numeral systems with just a few indices, there are also some unexpected inhabitants included in this representation. There is always a trade-off between expressiveness and properties.

We will explore the various types of numeral systems and define operations on them in the chapter 5.

Chapter 3

A gentle introduction to dependently typed programming in Agda

There are already plenty of tutorials and introductions of Agda [9][8][5]. We will nonetheless compile a simple and self-contained tutorial from the materials cited above, covering the parts (and only the parts) 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. Readers 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*[6], hence the Curry-Howard correspondence[11], 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 having them predefined?

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.

```
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 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. 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
```

`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 even as well.

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

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

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

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

`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 [12]. 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)
```

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 "broken", or "split" into two steps:

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

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

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 can see that proofs are composable just like programs.

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

However, 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[2].

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 to another, step by step, justified by theorems. The first and the last steps correspond 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 is 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 that 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 Decidable Equality on Natural Numbers

Equality is decidable on natural numbers, which means that we can always tell whether two numbers are equal, and explain the reason with a proof.

```
_≟_ : Decidable {A = ℕ} _≡_
zero  ≟ zero   = yes refl
suc m ≟ suc n  with m ≟ n
suc m ≟ suc .m | yes refl = yes refl
suc m ≟ suc n  | no prf   =
    no (prf ∘ (λ p → subst (λ x → m ≡ pred x) p refl))
zero  ≟ suc n  = no λ()
suc m ≟ zero   = no λ()
```

Decidable functions are often used together with *with-abstractions.*

```
answer : ℕ → Bool
answer n with n ≟ 42
answer n | yes p = true
answer n | no ¬p = false
```

where `p : n ≡ 42` and `¬p : n ≢ 42`.

4.4 Preorder

Aside from stating that two expressions are equal, a proposition can also assert 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*. One of the instances of preorder on natural numbers is the binary relation _≤_ (less than or equal to.)

```
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 that 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
```

4.5 Preorder reasoning

Combinators for equational reasoning can be further generalized to support *preorder reasoning*. Preorders are *reflexive* and *transitive*, which means that 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. Given `x ≤ y`, we can substitute `f x` by `f y` using a proof that `f` is monotonic. [3].

4.6 Decidable Preorder on Natural Numbers

Preorder is also decidable on natural numbers, which means that we can always tell whether one number is less than or equal to another.

```
_≤?_ : Decidable _≤_
zero  ≤? _     = yes z≤n
suc m ≤? zero  = no λ()
suc m ≤? suc n with m ≤? n
...            | yes m≤n = yes (s≤s m≤n)
...            | no  m≰n = no  (m≰n ∘ ≤-pred)
```

With with-abstractions we could define some function like this:

```
threshold : ℕ → ℕ
threshold n with n ≤? 87
threshold n | yes p = n
```

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

[3]Such a proof or theorem often goes by the name of `f-mono`

```
threshold n | no ¬p = 87
```

where `p : n ≤ 42` and `¬p : n ≰ 42`.

4.7 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.

4.8 Relevant Properties of Natural Numbers

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

Aside from some basic properties taken from the standard library, we have also added some useful theorems, lemmata, and corollaries. [4]

4.8.1 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

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

[4] *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.

- `+-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.

4.8.2 Relational Propositions [work in process]

natural number

- `≤-pred : ∀ {m n} → suc m ≤ suc n → m ≤ n`
  inverse of `s≤s`.
- `<⇒≤ : _<_ ⇒ _≤_`
- `>⇒≰ : _>_ ⇒ _≰_`
- `≤⇒≯ : _≤_ ⇒ _≯_`
- `<⇒≱ : _<_ ⇒ _≱_`
- `>⇒≢ : _>_ ⇒ _≢_`
- `≥⇒≮ : _≥_ ⇒ _≮_`
- `<⇒≢ : _<_ ⇒ _≢_`
- `≤∧≢⇒< : ∀ {m n} → m ≤ n → m ≢ n → m < n`
- `≥∧≢⇒> : ∀ {m n} → m ≥ n → m ≢ 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`

multiplication

monus

min and max

Chapter 5

Constructions

The representation for positional numeral systems will be constructed and formalized with Agda in this section, along with the generalizations introduced in section 2.

- **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`.

5.1 Digit: the basic building block

As the fundamental building block of numerals, we will devise a suitable representation for digits in this section.

5.1.1 Fin

To represent a digit, we use a datatype conventionally called *Fin* which can be indexed to have an 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.

5.1.2 Definition of `Digit`

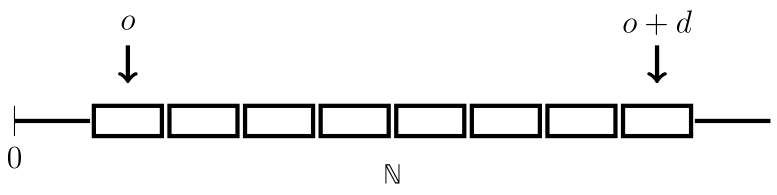
`Digit` is simply just a synonym for `Fin`, indexed by the number of digits `d` of a system. Since the same digit may represent different values in different numeral systems, it is essential to make the context clear.

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

Ordinary 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 Digit Assignment



Digits are assigned to ℕ together with the offset `o` of a system, ranging from o to $d + o$.

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

1

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 increased to o. Values greater than $d + o$ are prohibited by the supplied upper-bound.

[1] `toℕ : ∀ {n} → Fin n → ℕ`
converts from `Fin n` to ℕ.

```
Digit-fromℕ : ∀ {d}
    → (n o : ℕ)
    → (upper-bound : 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
                ≈⟨ m+n∸n≡m d o ⟩
                    d
                □
```

[2]

Properties of **`Digit`**

$$\begin{tikzcd}\mathbb{N} \arrow[r, shift left, "\texttt{Digit-fromℕ}"] & \texttt{Digit d} \arrow[l, shift left, "\texttt{Digit-toℕ}"]\end{tikzcd}$$

`Digit-fromℕ-toℕ` states that the value of a natural number should remain the same, after converting 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 {d} {o} n lb ub with n ∸ o ≤? d
Digit-fromℕ-toℕ {d} {o} n lb ub | yes q =
    begin
        toℕ (fromℕ≤ (s≤s q)) + o
    ≡⟨ cong (λ x → x + o) (toℕ-fromℕ≤ (s≤s q)) ⟩
        n ∸ o + o
    ≡⟨ m∸n+n≡m lb ⟩
        n
    ∎
Digit-fromℕ-toℕ {d} {o} n lb ub | no ¬q = contradiction q ¬q
    where   q : n ∸ o ≤ d
```

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

```
            q = +n-mono-inverse o (
                start
                    n ∸ o + o
                ≈⟨ m∸n+n≡m lb ⟩
                    n
                ≤⟨ ub ⟩
                    d + o
                □)
```

[3]

Digits have a upper-bound and a lower-bound after evaluated to ℕ.

```
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)
```

[4]

5.1.4 Functions on `Digit`

Increment

To increment a digit, the digit must not be *the greatest*.

```
digit+1 : ∀ {d}
    → (x : Digit d)
    → (¬greatest : ¬ (Greatest x))
    → Digit d
digit+1 x ¬greatest =
    fromℕ≤ {suc (toℕ x)} (≤∧≢⇒< (bounded x) ¬greatest)
```

where `≤∧≢⇒< (bounded x) ¬greatest : suc (toℕ x) < d`.

[3] `toℕ-fromℕ≤ : ∀ {m n} (m<n : m < n) → toℕ (fromℕ≤ m<n) ≡ m`
states that a number should remain the same after converting back and forth.

[4] `bounded : ∀ {n} (i : Fin n) → toℕ i < n`
a property about the upper-bound of a `Fin n`.

$$\begin{tikzcd}
\texttt{Digit d} \arrow[r, "\texttt{digit+1}"] \arrow[d, "\texttt{Digit-to}\mathbb{N}"'] & \texttt{Digit d} \arrow[d, "\texttt{Digit-to}\mathbb{N}"] \\
\mathbb{N} \arrow[r, "\texttt{suc}"'] & \mathbb{N}
\end{tikzcd}$$

A digit taking these two routes should result in the same ℕ.

```
digit+1-toℕ : ∀ {d o}
    → (x : Digit d)
    → (¬greatest : ¬ (Greatest x))
    → Digit-toℕ (digit+1 x ¬greatest) o ≡ suc (Digit-toℕ x o)
digit+1-toℕ {d} {o} x ¬greatest =
    begin
        Digit-toℕ (digit+1 x ¬greatest) o
    ≡⟨ cong (λ w → w + o) (toℕ-fromℕ≤ (≤∧≢⇒< (bounded x) ¬greatest)) ⟩
        suc (Digit-toℕ x o)
    ∎
```

Increase and then Subtract

Increases a digit and then subtract it by n. This function is useful for implementing *carrying*. When the digit to increase is already the greatest, it has to be subtracted from an amount (usually the base) after the increment.

```
digit+1-n : ∀ {d}
    → (x : Digit d)
    → Greatest x
    → (n : ℕ)
    → n > 0
    → Digit d
digit+1-n x greatest n n>0 =
    fromℕ≤ (digit+1-n-lemma x greatest n n>0)
```

$$\begin{tikzcd}
\texttt{Digit d} \arrow[rr, "\texttt{digit+1-n}"] \arrow[d, "\texttt{Digit-to}\mathbb{N}"'] & & \texttt{Digit d} \arrow[d, "\texttt{Digit-to}\mathbb{N}"] \\
\mathbb{N} \arrow[r, "\texttt{suc}"'] & \mathbb{N} \arrow[r, "\texttt{∸n}"'] & \mathbb{N}
\end{tikzcd}$$

A digit taking these two routes should result in the same ℕ.

```
digit+1-n-toℕ : ∀ {d o}
    → (x : Digit d)
    → (greatest : Greatest x)
    → (n : ℕ)
    → (n>0 : n > 0)
    → n ≤ d
    → Digit-toℕ (digit+1-n x greatest n n>0) o ≡ suc (Digit-toℕ x o) ∸ n
digit+1-n-toℕ {zero}  {o} () greatest n n>0 n≤d
digit+1-n-toℕ {suc d} {o} x greatest n n>0  n≤d =
    begin
        toℕ (digit+1-n x greatest n n>0) + o
    ≡⟨ cong (λ w → w + o) (toℕ-fromℕ≤ (digit+1-n-lemma x greatest n n>0)) ⟩
        suc (toℕ x) ∸ n + o
    ≡⟨ +-comm (suc (toℕ x) ∸ n) o ⟩
        o + (suc (toℕ x) ∸ n)
    ≡⟨ sym (+-∸-assoc o {suc (toℕ x)} {n} (
        start
            n
        ≤⟨ n≤d ⟩
            suc d
        ≈⟨ sym greatest ⟩
            suc (toℕ x)
        □)
    ⟩
        (o + suc (toℕ x)) ∸ n
    ≡⟨ cong (λ w → w ∸ n) (+-comm o (suc (toℕ x))) ⟩
        suc (toℕ x) + o ∸ n
    ∎)
```

5.1.5 Special Digits

The Greatest Digit

The greatest digit of a system is constructed by converting the index `d` to `Fin`.

```
greatest-digit : ∀ d → Digit (suc d)
greatest-digit d = fromℕ d
```

5

predicates We can see whether a digit is the greatest by converting it to ℕ. This predicate also comes with a decidable version.

```
Greatest : ∀ {d} (x : Digit d) → Set
Greatest {d} x = suc (toℕ x) ≡ d
```

[5] `fromℕ : ∀ {n} → Fin n → ℕ`
construct the greatest possible `Fin n` when given an index `n`.

```
Greatest? : ∀ {d} (x : Digit d) → Dec (Greatest x)
Greatest? {d} x = suc (toℕ x) ≟ d
```

properties Converting from the greatest digit to ℕ should result in `d + o`.

```
greatest-digit-toℕ : ∀ {d o}
    → (x : Digit (suc d))
    → Greatest x
    → Digit-toℕ x o ≡ d + o
greatest-digit-toℕ {d} {o} x greatest = cancel-suc (
    begin
        suc (Digit-toℕ x o)
    ≡⟨ refl ⟩
        suc (toℕ x) + o
    ≡⟨ cong (λ w → w + o) greatest ⟩
        suc d + o
    ∎)
```

A digit is the greatest if and only if it is greater than or equal to all other digits. This proposition is proven by induction on both of the compared digits.

```
greatest-of-all : ∀ {d} (o : ℕ) → (x y : Digit d)
    → Greatest x
    → Digit-toℕ x o ≥ Digit-toℕ y o
greatest-of-all o zero    zero     refl     = ≤-refl
greatest-of-all o zero    (suc ()) refl
greatest-of-all o (suc x) zero     greatest
    = +n-mono o {zero} {suc (toℕ x)} z≤n
greatest-of-all o (suc x) (suc y)  greatest
    = s≤s (greatest-of-all o x y (cancel-suc greatest))
```

The Carry

A carry is a digit that is transferred to a more significant digit to compensate the "loss" of the original digit.

The carry is defined as the greater of these two values:

- the least digit of a system
- the digit that is assigned to 1

We define the carry as the greater number of 1 and *o*. In case that the least digit (which is determined by *o*) is assigned to 0, rendering the carry useless.

```
carry : ℕ → ℕ
carry o = 1 ⊔ o
```

The corresponding numeral of the carry is constructed by converting `carry o` to `Digit`.

```
carry-digit : ∀ d o → 2 ≤ suc d + o → Digit (suc d)
carry-digit d o proper =
    Digit-fromℕ
        (carry o)
        o
        (carry-upper-bound {d} proper)
```

properties The value of the carry should remain the same after convertions.

```
carry-digit-toℕ : ∀ d o
    → (proper : 2 ≤ suc (d + o))
    → Digit-toℕ (carry-digit d o proper) o ≡ carry o
carry-digit-toℕ d o proper
    = Digit-fromℕ-toℕ
        (carry o)
        (m≤n⊔m o 1)
        (carry-upper-bound {d} proper)
```

The carry also have an upper-bound and a lower-bound, similar to that of `Digit`.

```
carry-lower-bound : ∀ {o} → carry o ≥ o
carry-lower-bound {o} = m≤n⊔m o 1

carry-upper-bound : ∀ {d o} → 2 ≤ suc d + o → carry o ≤ d + o
carry-upper-bound {d} {zero}  proper = ≤-pred proper
carry-upper-bound {d} {suc o} proper = n≤m+n d (suc o)
```

5.2 Num: a representation for positional numeral systems

5.2.1 Definition

Numerals of positional numeral systems are composed of sequences of digits. Consequently the definition of `Numeral` will be 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 near `_•` while the least significant digit is placed at the end of the sequence.

`Numeral` has three indices, which corresponds to the three generalizations we have introduced.

```
infixr 5 _∷_

data Numeral : (b d o : 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) •
```

where `#_ : ∀ m {n} {m<n : True (suc m ≤? n)} → Fin n` converts from ℕ to `Fin n` provided that the number is small enough.

`lsd` extracts the least significant digit of a numeral.

```
lsd : ∀ {b d o} → (xs : Numeral b d o) → Digit d
lsd (x •    ) = x
lsd (x ∷ xs) = x
```

5.2.2 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
```

5.3 Dissecting Numeral Systems with Views

There are many kinds of numeral systems inhabit in `Numeral`. These systems have different interesting properties that should be treated differently, so we sort them into **four categories** accordingly.

Systems with no digits at all The number of digits of a system is determined by the index `d`. If `d` happens to be 0, then there will be no digits in any of these systems. Although they seem useless, these systems have plenty of properties. Since there are not digits at all, any property that is related to digits would hold vacuously.

Systems with base 0 If `b`, the base of a system, happens to be 0, then only the least significant digit would have effects on the evaluation, because the rest of the digits would diminish into nothing.

```
⟦ x •    ⟧ = Digit-toℕ x o
⟦ x ∷ xs ⟧ = Digit-toℕ x o + ⟦ xs ⟧ * 0
```

Systems with only zeros Consider when `d` is set to 1 and `o` set to 0. There will be one digit. however, the only digit can only be assigned to 0.

```
0, 00, 000, 0000, ...
```

As a result, every numeral would evaluate to 0 regardless of the base.

"Proper" systems The rest of systems that does not fall into any of the categories above are considered *proper*.

5.3.1 Categorizing Systems

These "categories" are represented with a datatype called `NumView` that is indexed by the three indices: `b`, `d`, and `o`.

```
data NumView : (b d o : 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
```

By pattern matching on indices, different configurations of indices are sorted into different `NumViews`.

```
numView : ∀ b d o → NumView b d o
numView b        zero           o       = NoDigits b o
numView zero     (suc d)        o       = NullBase d o
numView (suc b)  (suc zero)     zero    = AllZeros b
numView (suc b)  (suc zero)     (suc o) = Proper b zero (suc o) _
numView (suc b)  (suc (suc d))  o       = Proper b (suc d) o _
```

Together with *with-abstractions*, we can, for example, define a function to determine whether a numeral system is interesting or not:

```
interesting : ∀ b d o → Bool
interesting b d o with numView b d o
interesting _ _ _ | NullBase d o        = false
interesting _ _ _ | NoDigits b o        = false
interesting _ _ _ | AllZeros b          = false
interesting _ _ _ | Proper b d o proper  = true
```

As we can see, the function `numView` does more than sorting indices into different categories. It also reveals relevant information and properties about these categories. For instance, if a system `Numeral b d o` is classified as *Proper*, then we know that:

- `b` is greater than 0.
- `d` is also greater than 0.
- `o` can be any value as long as `d + o ≥ 2`; we name this requirement *proper*.

5.3.2 Views

The sole purpose of `NumView` is to sort out and expose some interesting properties about its indices. Such datatypes are called *views*[13] as they present different aspects of the same object. Functions like `numView` are called *view functions* or *eliminators*[7] because they provide different ways of eliminating a datatype.

Views are **reusable** as they free us from having to pattern match on the same indices or data again and again. On the other hand, they can be customized to our needs since they are just *ordinary functions*. We will define more views and use them extensively in the coming sections.

5.4 Properties of each Category

NoDigits Although systems with no digits have no practical use, they are pretty easy to deal with because all properties related to digits would hold unconditionally for systems of `NoDigits`. This is proven by deploying *the principle of explosion*.

```
NoDigits-explode : ∀ {b o a} {Whatever : Set a}
    → (xs : Numeral b 0 o)
    → Whatever
NoDigits-explode (() •   )
NoDigits-explode (() ∷ xs)
```

NullBase The theorem below states that, evaluating a numeral of `NullBase` would results the same value as evaluating its least significant digit.

```
toℕ-NullBase : ∀ {d o}
    → (x : Digit d)
    → (xs : Numeral 0 d o)
    → ⟦ x ∷ xs ⟧ ≡ Digit-toℕ x o
toℕ-NullBase {d} {o} x xs =
    begin
        Digit-toℕ x o + ⟦ xs ⟧ * 0
    ≡⟨ cong (λ w → Digit-toℕ x o + w) (*-right-zero ⟦ xs ⟧) ⟩
        Digit-toℕ x o + 0
    ≡⟨ +-right-identity (Digit-toℕ x o) ⟩
        Digit-toℕ x o
    ∎
```

AllZeros The theorem below states every numeral of `AllZeros` would evaluate to 0 regardless of the base. We pattern match on the digit to eliminate other possible cases to exploit the fact that there is only one digit in such numerals.

```
toℕ-AllZeros : ∀ {b} → (xs : Numeral b 1 0) → ⟦ xs ⟧ ≡ 0
toℕ-AllZeros     (z    •   ) = refl
toℕ-AllZeros     (s () •   )
toℕ-AllZeros {b} (z    ∷ xs)
    = cong (λ w → w * b) (toℕ-AllZeros xs)
toℕ-AllZeros     (s () ∷ xs)
```

5.5 The Maximum Numeral

A number is said to be the *maximum* if there it is greater than or equal to all the other numbers.

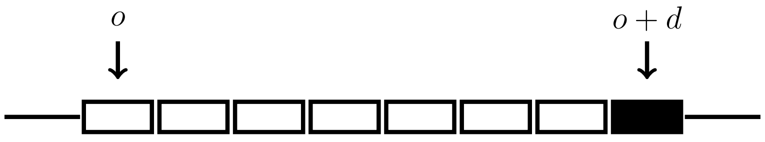
```
Maximum : ∀ {b d o} → (xs : Numeral b d o) → Set
Maximum {b} {d} {o} xs = (ys : Numeral b d o) → ⟦ xs ⟧ ≥ ⟦ ys ⟧
```

If a numeral is a maximum [6], then its least significant digit (LSD) must be the greatest.

```
Maximum⇒Greatest-LSD : ∀ {b} {d} {o}
    → (xs : Numeral b d o)
    → Maximum xs
    → Greatest (lsd xs)
```

5.5.1 Properties of each Category

NullBase



It is obvious that systems of `NullBase` have maximum. If a numeral's LSD happens to be the greatest, then the numeral must be the maximum.

```
Maximum-NullBase-Greatest : ∀ {d} {o}
    → (xs : Numeral 0 (suc d) o)
    → Greatest (lsd xs)
    → Maximum xs
```

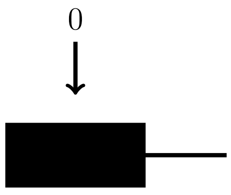
With this lemma, we can tell whether a numeral is a maximum by looking at its LSD. In case that the LSD is not the greatest, we could disprove the proposition by contraposition.

```
    Maximum-NullBase : ∀ {d} {o}
        → (xs : Numeral 0 (suc d) o)
```

[6] More precisely, "If a value of a numeral is a maximum ..."

```
        → Dec (Maximum xs)
    Maximum-NullBase xs with Greatest? (lsd xs)
    Maximum-NullBase xs | yes greatest =
        yes (Maximum-NullBase-Greatest xs greatest)
    Maximum-NullBase | no ¬greatest =
        no (contraposition (Maximum⇒Greatest-LSD xs) ¬greatest)
```

AllZeros



All numerals of systems of `AllZeros` are maxima since they are all mapped to 0.

```
Maximum-AllZeros : ∀ {b}
    → (xs : Numeral b 1 0)
    → Maximum xs
Maximum-AllZeros xs ys = reflexive (
    begin
        ⟦ ys ⟧
    ≡⟨ toℕ-AllZeros ys ⟩
        zero
    ≡⟨ sym (toℕ-AllZeros xs) ⟩
        ⟦ xs ⟧
    ∎)
```

Proper On the contrary, there are no maxima in the systems of `Proper`. In fact, that is the reason why they are categorized as *proper* in the first place. The theorem below is proven by contradicting two propositions:

- Given `claim : Maximum xs`, we claim that `xs` is greater than or equal to `greatest-digit d ∷ xs`, a numeral we composed by prefixing it with the greatest digit.
- On the other hand, we prove that `xs` is less than `greatest-digit d ∷ xs`.

```
Maximum-Proper : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → ¬ (Maximum xs)
Maximum-Proper {b} {d} {o} xs proper claim = contradiction p ¬p
    where
        p : ⟦ xs ⟧ ≥ ⟦ greatest-digit d ∷ xs ⟧
        p = claim (greatest-digit d ∷ xs)
        ¬p : ⟦ xs ⟧ ≱ ⟦ greatest-digit d ∷ xs ⟧
        ¬p = <⇒≱ (
            start
                suc ⟦ xs ⟧
            ≈⟨ cong suc (sym (*-right-identity ⟦ xs ⟧)) ⟩
                suc (⟦ xs ⟧ * 1)
            ≤⟨ s≤s (n*-mono ⟦ xs ⟧ (s≤s z≤n)) ⟩
                suc (⟦ xs ⟧ * suc b)
            ≤⟨ +n-mono (⟦ xs ⟧ * suc b) (≤-pred proper) ⟩
                d + o + ⟦ xs ⟧ * suc b
            ≈⟨ cong
                (λ w → w + ⟦ xs ⟧ * suc b)
                (sym (greatest-digit-toℕ (Fin.fromℕ d)
                (greatest-digit-is-the-Greatest d)))
            ⟩
                ⟦ greatest-digit d ∷ xs ⟧
            □)
```

5.5.2 Determine the Maximum

We can *decide* whether a numeral is a maximum by applying them to lemmata of each category.

```
Maximum? : ∀ {b d o}
    → (xs : Numeral b d o)
    → Dec (Maximum xs)
Maximum? {b} {d} {o} xs with numView b d o
Maximum? xs | NullBase d o = Maximum-NullBase xs
Maximum? xs | NoDigits b o = no (NoDigits-explode xs)
Maximum? xs | AllZeros b   = yes (Maximum-AllZeros xs)
Maximum? xs | Proper b d o proper = no (Maximum-Proper xs proper)
```

Summary

Properties	NullBase	NoDigits	AllZeros	Proper
has an maximum	yes	no	yes	no

5.6 Systems with an Upper Bound

We say that a system is *bounded* if there *exists* a maximum in such system. In Agda, an existential proposition like this is expressed with a *dependent sum type*, which is essentially a *pair* where the type of the second term is dependent on the first.

```
data Σ (A : Set) (B : A → Set) : Set where
    _,_ : (x : A) → B x → Σ A B

proj₁ : ∀ {A B} → Σ A B → A
proj₂ : ∀ {A B} → (pair : Σ A B) → B (proj₁ pair)
```

For example, to prove that there exists a right identity for addition on natural numbers; we place the "evidence" on the left and its justification on the right.

```
prop : ∀ n → Σ ℕ (λ e → n + e ≡ n)
prop n = 0 , +-right-identity n
```

With a little syntax sugar, we can rewrite the proposition as `Σ[ x ∈ A ] B` instead of `Σ A (λ x → B)`.

```
Bounded : ∀ b d o → Set
Bounded b d o = Σ[ xs ∈ Numeral b d o ] Maximum xs
```

The proposition above can be read as follows: *There exists a numeral `xs` of type `Numeral b d o` such that `Maximum xs` holds.* To prove that a system is bounded, we have to place the numeral that we consider to be a maximum on the left of the pair, and a proof to justify it on the right.

5.6.1 Properties of each Category

NullBase We have proven that any numeral with the greatest digit as its LSD is a maximum.

```
Bounded-NullBase : ∀ d o → Bounded 0 (suc d) o
Bounded-NullBase d o =
    (greatest-digit d •) ,
    (Maximum-NullBase-Greatest
        (greatest-digit d •)
        (greatest-digit-is-the-Greatest d))
```

NoDigits Since `¬ (Bounded b 0 o)` reduces to `Bounded b 0 o → ⊥`, we are given an addition argument, which is a proof claiming that the system is bounded. Pattern match on this argument yields a pair with a numeral of `Numeral b 0 o` on the left. At this point, we do not actually care whether the numeral is a maximum because it should not have existed in the first place. Hand the numeral to `NoDigits-explode` and we are done.

```
Bounded-NoDigits : ∀ b o → ¬ (Bounded b 0 o)
Bounded-NoDigits b o (xs , claim) = NoDigits-explode xs
```

AllZeros Similar to that of `Bounded-NullBase`:

```
Bounded-AllZeros : ∀ b → Bounded (suc b) 1 0
Bounded-AllZeros b = (z •) , Maximum-AllZeros (z •)
```

Proper This proposition is proven by contradicting the fact that a proper numeral system has no maximum.

```
Bounded-Proper : ∀ b d o → (proper : 2 ≤ suc (d + o))
    → ¬ (Bounded (suc b) (suc d) o)
Bounded-Proper b d o proper (xs , claim) =
    contradiction claim (Maximum-Proper xs proper)
```

5.6.2 The Decidable Predicate

We can determine whether a system is bounded by delegating the job to the helper functions we have defined above.

```
Bounded? : ∀ b d o → Dec (Bounded b d o)
Bounded? b d o with numView b d o
Bounded? _ _ _ | NullBase d o
    = yes (Bounded-NullBase d o)
Bounded? _ _ _ | NoDigits b o
    = no (Bounded-NoDigits b o)
Bounded? _ _ _ | AllZeros b
    = yes (Bounded-AllZeros b)
Bounded? _ _ _ | Proper b d o proper
    = no (Bounded-Proper b d o proper)
```

Summary

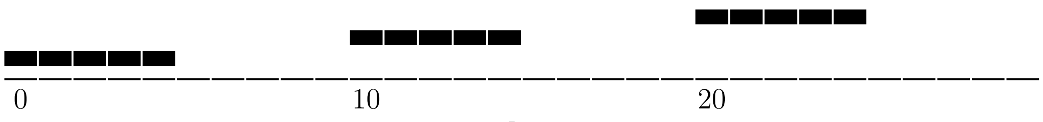
Properties	NullBase	NoDigits	AllZeros	Proper
has an maximum	yes	no	yes	no
has an upper bound	yes	no	yes	no

5.7 The Next Numeral

Paul Benacerraf once argued[1] that there are two kinds of *counting* which correspond to **transitive** and **intransitive** uses of the verb "to count." Transitive counting, in his sense, is to assign one of the numbers to the cardinality of a set, by establishing a one-to-one correspondence between the numbers and the objects one is counting, all the way from none to all. Intransitive counting, on the other hand, is to generate a sequence of notation, that could go as far as we need. It seems that one can only learn how to count intransitively first, before knowing how to count transitively, but not vice versa. So it is important to know how to generate the next numeral we need.

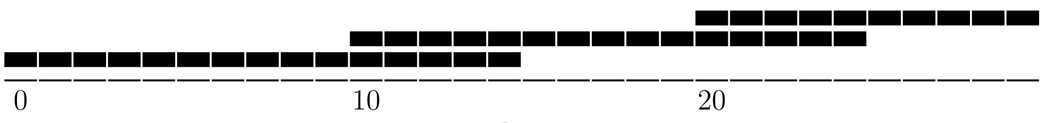
5.7.1 What does it mean by "The Next"?

When mapping some systems such as `Numeral 10 5 0` onto the natural numbers, their number lines appear to be "gapped" because the evaluation is not *surjective*. The number "5", for example, has no correspondence on the numeral side.



`Numeral 10 5 0`

While systems such as `Numeral 10 15 0` map more than one numeral onto the same number, we can see immediately that the evaluation is not *injective* because those number lines "overlap" with each other.



`Numeral 10 15 0`

Apparently numerals do not align with numbers perfectly. Finding "the next number" thus becomes a non-trivial problem. Therefore, we define the next numeral as **the least numeral that is greater than itself**.

5.7.2 Implementation

Given a numeral, say `xs`, to find the next numeral of `xs`, it reasonable to expect that `xs` must not be a maximum. Otherwise, there would exist no such a numeral that is *greater than itself*.

Again, the indices are classified by `numView` into four categories. The case of `NoDigits` and `AllZeros` are rather trivial. We focus on the other two.

```
next-numeral : ∀ {b d o}
    → (xs : Numeral b d o)
    → ¬ (Maximum xs)
    → Numeral b d o
next-numeral {b} {d} {o} xs ¬max with numView b d o
next-numeral xs ¬max | NullBase d o = ?
next-numeral xs ¬max | NoDigits b o = NoDigits-explode xs
next-numeral xs ¬max | AllZeros b   =
    contradiction (Maximum-AllZeros xs) ¬max
next-numeral xs ¬max | Proper b d o proper = ?
```

5.7.3 Validating the Implementation

To validate the correctness of the implementation of `next-numeral`, i.e., to make sure that the function does return a numeral that is:

1. greater than the given numeral
2. the least of all such numerals

We need to prove these two propositions:

```
next-numeral-is-greater : ∀ {b d o}
    → (xs : Numeral b d o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ next-numeral xs ¬max ⟧ > ⟦ xs ⟧
```

```
next-numeral-is-immediate : ∀ {b d o}
    → (xs : Numeral b d o)
    → (ys : Numeral b d o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ ys ⟧ > ⟦ xs ⟧
    → ⟦ ys ⟧ ≥ ⟦ next-numeral xs ¬max ⟧
```

We come up with the name “immediate” because “the least numeral that is greater than itself” is a bit too wordy.

5.7.4 Outline

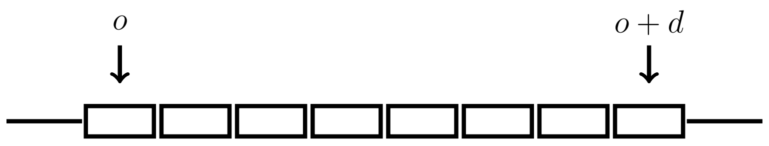
We are going to finish programs and proofs of systems of `NullBase` and `Proper` respectively.

- `NullBase`
  1. `next-numeral-NullBase`
  2. `next-numeral-is-greater-NullBase`
  3. `next-numeral-is-immediate-NullBase`
- `Proper`
  - `next-numeral-Proper`
  - `next-numeral-is-greater-Proper`
  - `next-numeral-is-immediate-Proper`

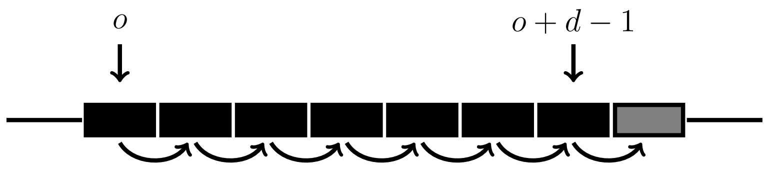
However, programs and proofs of systems of `Proper` will be defined *simultaneously* with mutually recursive definitions. The reason why we need the property `next-numeral-is-greater-Proper` in the first place is that they are necessary for implementing `next-numeral-Proper`.

5.7.5 The Next Numeral of NullBase

To find the next numeral of systems of `NullBase`, all we have to do is to manipulate the LSD. Since only the value of the LSD has effect on the evaluation. All numerals are mapped onto a single and continuous number line as shown below.



Finding the next numeral is as simple as incrementing the LSD. To make room for the increment, numerals that are mapped onto the rightmost number of the line have to be excluded.



`next-numeral-NullBase`

Numerals with the greatest LSD are first ruled out because they also happen to be maxima. The rest of the numerals are then pattern matched to have their LSDs replaced with an incremented one.

```
next-numeral-NullBase : ∀ {d o}
    → (xs : Numeral 0 (suc d) o)
    → ¬ (Maximum xs)
    → Numeral 0 (suc d) o
next-numeral-NullBase xs       ¬max with Greatest? (lsd xs)
next-numeral-NullBase xs       ¬max | yes greatest
    = contradiction (Maximum-NullBase-Greatest xs greatest) ¬max
next-numeral-NullBase (x •)    ¬max | no ¬greatest
    = digit+1 x ¬greatest •
next-numeral-NullBase (x ∷ xs) ¬max | no ¬greatest
    = digit+1 x ¬greatest ∷ xs
```

Lemma

Instead of proving that `next-numeral-NullBase` does compute a numeral that is:

1. greater than the given numeral
2. the least of all such numerals

We justify a stronger proposition; that `next-numeral-NullBase` would compute the *successor* of the given number.

```
next-numeral-NullBase-lemma : ∀ {d o}
    → (xs : Numeral 0 (suc d) o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ next-numeral-NullBase xs ¬max ⟧ ≡ suc ⟦ xs ⟧
next-numeral-NullBase-lemma {d} {o} xs    ¬max with Greatest? (lsd xs)
next-numeral-NullBase-lemma {d} {o} xs    ¬max | yes greatest =
    contradiction (Maximum-NullBase-Greatest xs greatest) ¬max
next-numeral-NullBase-lemma {d} {o} (x •) ¬max | no ¬greatest =
    begin
```

```
        ⟦ digit+1 x ¬greatest • ⟧
    ≡⟨ refl ⟩
        Digit-toℕ (digit+1 x ¬greatest) o
    ≡⟨ digit+1-toℕ x ¬greatest ⟩
        suc (Fin.toℕ x + o)
    ≡⟨ refl ⟩
        suc ⟦ x • ⟧
    ∎
next-numeral-NullBase-lemma {d} {o} (x ∷ xs) ¬max | no ¬greatest =
    begin
        ⟦ digit+1 x ¬greatest ∷ xs ⟧
    ≡⟨ refl ⟩
        Digit-toℕ (digit+1 x ¬greatest) o + ⟦ xs ⟧ * zero
    ≡⟨ cong (λ w → w + ⟦ xs ⟧ * zero) (digit+1-toℕ x ¬greatest) ⟩
        suc (Fin.toℕ x + o + ⟦ xs ⟧ * zero)
    ≡⟨ refl ⟩
        suc ⟦ x ∷ xs ⟧
    ∎
```

`next-numeral-is-greater-NullBase`

Recall that `x < y` is essentially a synonym of `suc x ≤ y`. We complete this proof by applying the lemma we have proven.

```
next-numeral-is-greater-NullBase : ∀ {d o}
    → (xs : Numeral 0 (suc d) o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ next-numeral-NullBase xs ¬max ⟧ > ⟦ xs ⟧
next-numeral-is-greater-NullBase xs ¬max =
    start
        suc ⟦ xs ⟧
    ≈⟨ sym (next-numeral-NullBase-lemma xs ¬max) ⟩
        ⟦ next-numeral-NullBase xs ¬max ⟧
    □
```

`next-numeral-is-immediate-NullBase`

Here, the argument `ys` stands for *any* numeral of `Numeral 0 (suc d) o`.

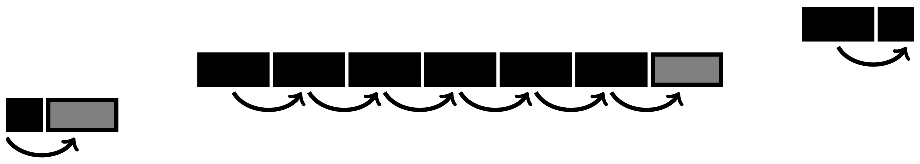
```
next-numeral-is-immediate-NullBase : ∀ {d o}
    → (xs : Numeral 0 (suc d) o)
    → (ys : Numeral 0 (suc d) o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ ys ⟧ > ⟦ xs ⟧
    → ⟦ ys ⟧ ≥ ⟦ next-numeral-NullBase xs ¬max ⟧
next-numeral-is-immediate-NullBase xs ys ¬max prop =
    start
        ⟦ next-numeral-NullBase xs ¬max ⟧
    ≈⟨ next-numeral-NullBase-lemma xs ¬max ⟩
```

```
        suc ⟦ xs ⟧
    ≤⟨ prop ⟩
        ⟦ ys ⟧
    □
```

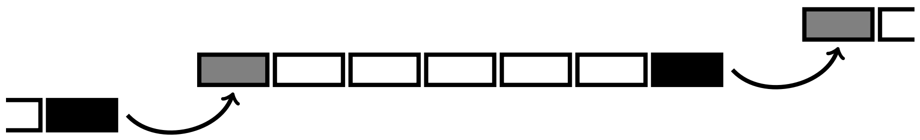
5.7.6 The Next Numeral of Proper

Unlike in systems of `NullBase` where every numeral gets mapped onto a single and continuous number line. There are *three different cases* where the given numeral might be located in systems of `Proper`.

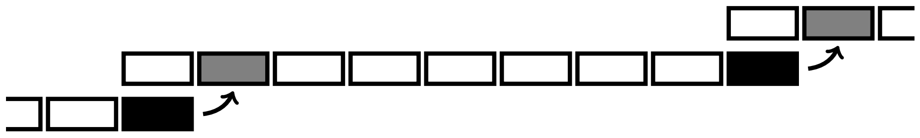
Intervals This is the simplest case like in `NullBase`. The given numeral's LSD is not located at the end of the number line.



Gapped Endpoints When the given numeral's LSD happens to be the greatest and there is a *gap* before the next numeral. There are no other numerals in between that could bridge the gap.



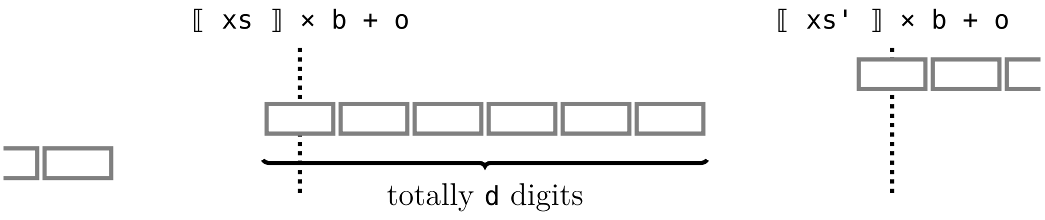
Ungapped Endpoints When the given numeral's LSD happens to be the greatest but we can find another numeral right ahead of it.



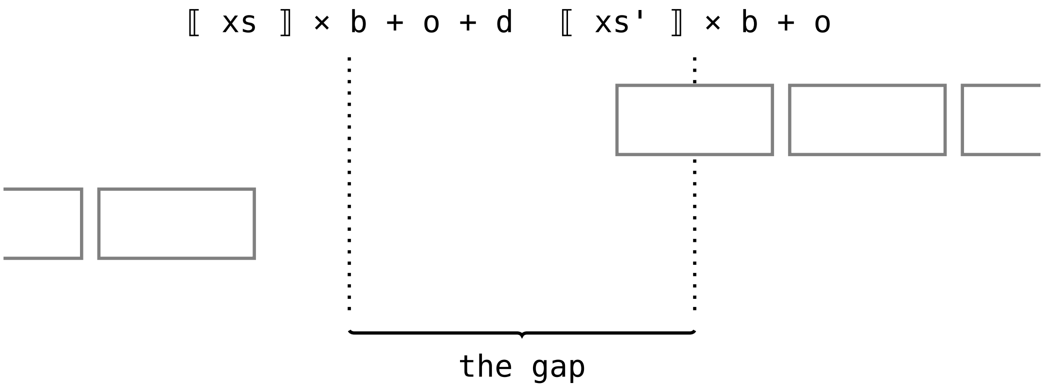
Gaps

It is easy to tell if a given numeral is located in intervals or at endpoints just by looking at its LSD. However, to tell apart from *gapped* and *ungapped* endpoints is an entirely different story. First things first, we need a proposition for expressing this matter.

Given a numeral `x ∷ xs` of `Numeral b d o` and suppose that the next numeral of `xs` is `xs'`. The value of `x ∷ xs` ranges from `⟦ xs ⟧ × b + o` to `⟦ xs ⟧ × b + o + d - 1`.



If we look closely, we can see that the gap comes from the difference between `⟦ xs ⟧ × b + o + d` and `⟦ xs' ⟧ × b + o`.



If there are enough digits to bridge the gap between `⟦ xs ⟧ × b + o` and `⟦ xs' ⟧ × b + o`, then there will be no gaps. Thus the predicate for telling if there is a gap ahead of `x ∷ xs` can then be expressed as follows. Notice that `next-numeral-Proper` is used as part of the definition.

```
Gapped : ∀ b d o
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Set
Gapped b d o xs proper
    = suc d < (⟦ next-numeral-Proper xs proper ⟧ ∸ ⟦ xs ⟧) * suc b
```

However, the predicate above does not take numerals like `x •` into account, i.e., numerals that are composed of only a single digit. In that case, the gap ahead will be *the first gap* among the others. The predicate of the first gap is as simple as comparing the number of digits `d` with `carry o × b`, where `carry o = 1 ⊔ o` computes the least (be greater than zero) digit of higher significance.

```
Gapped : ∀ b d o → Set
Gapped b d o = suc d < carry o * suc b
```

Combining the two predicates above, we devise a predicate for predicting the gap ahead of a numeral.

```
Gapped#0 : ∀ b d o → Set
Gapped#0 b d o = suc d < carry o * suc b

Gapped#N : ∀ b d o
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Set
Gapped#N b d o xs proper
    = suc d < (⟦ next-numeral-Proper xs proper ⟧ ∸ ⟦ xs ⟧) * suc b

Gapped : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Set
Gapped {b} {d} {o} (x •)    proper = Gapped#0 b d o
Gapped {b} {d} {o} (x ∷ xs) proper = Gapped#N b d o xs proper
```

along with decidable versions.

```
Gapped#0? :  ∀ b d o → Dec (Gapped#0 b d o)
Gapped#N? :  ∀ b d o
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Dec (Gapped#N b d o xs proper)
Gapped? : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Dec (Gapped {b} {d} {o} xs proper)
```

There is an important theorem about the intriguing relation between `Gapped#0` and `Gapped#N` which will be proven and used in later chapters.

A View within another

With the predicate `Gapped` and its friends, we can further classify systems of `Proper` into finer categories.

```
data NextView : (b d o : ℕ) (xs : Numeral b d o) (proper : 2 ≤ d + o) → Set where
    Interval : ∀ b d o
        → {xs : Numeral (suc b) (suc d) o}
        → {proper : 2 ≤ suc (d + o)}
        → (¬greatest : ¬ (Greatest (lsd xs)))
        → NextView (suc b) (suc d) o xs proper
    GappedEndpoint : ∀ b d o
        → {xs : Numeral (suc b) (suc d) o}
        → {proper : 2 ≤ suc (d + o)}
        → (greatest : Greatest (lsd xs))
        → (gapped : Gapped xs proper)
        → NextView (suc b) (suc d) o xs proper
    UngappedEndpoint : ∀ b d o
        → {xs : Numeral (suc b) (suc d) o}
        → {proper : 2 ≤ suc (d + o)}
        → (greatest : Greatest (lsd xs))
        → (¬gapped : ¬ (Gapped xs proper))
        → NextView (suc b) (suc d) o xs proper
```

```
nextView : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → NextView (suc b) (suc d) o xs proper
nextView {b} {d} {o} xs proper with Greatest? (lsd xs)
nextView {b} {d} {o} xs proper | yes greatest with Gapped? xs proper
nextView {b} {d} {o} xs proper | yes greatest | yes gapped
    = GappedEndpoint  b d o greatest gapped
nextView {b} {d} {o} xs proper | yes greatest | no ¬gapped
    = UngappedEndpoint b d o greatest ¬gapped
nextView {b} {d} {o} xs proper | no ¬greatest
    = Interval b d o ¬greatest
```

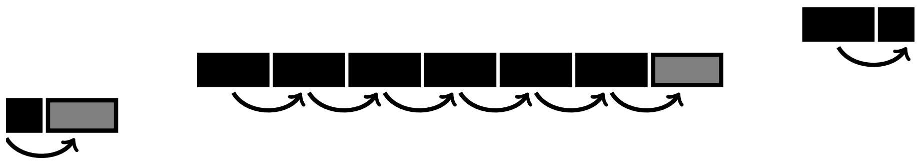
`next-numeral-Proper`

Finally, to compute the next numeral of systems of `Proper`, we categorize the given numeral with `NextView` and delegate the task to corresponding helper functions.

```
next-numeral-Proper : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Numeral (suc b) (suc d) o
next-numeral-Proper xs proper with nextView xs proper
```

```
next-numeral-Proper xs proper | Interval b d o ¬greatest
    = next-numeral-Proper-Interval xs ¬greatest proper
next-numeral-Proper xs proper | GappedEndpoint b d o greatest gapped
    = next-numeral-Proper-GappedEndpoint xs proper gapped
next-numeral-Proper xs proper | UngappedEndpoint b d o greatest ¬gapped
    = next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped
```

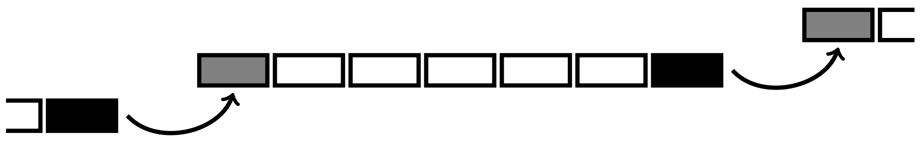
Interval



To reach the next numeral, all we have to do is to increment the LSD. There is no need of performing carries or messing with other digits.

```
next-numeral-Proper-Interval : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (¬greatest : ¬ (Greatest (lsd xs)))
    → (proper : 2 ≤ suc (d + o))
    → Numeral (suc b) (suc d) o
next-numeral-Proper-Interval (x •)    ¬greatest proper
    = digit+1 x ¬greatest •
next-numeral-Proper-Interval (x ∷ xs) ¬greatest proper
    = digit+1 x ¬greatest ∷ xs
```

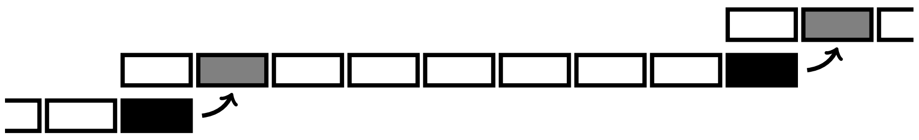
Gapped Endpoints



To reach the next numeral, the LSD is reset to the least digit; the rest of the numeral (digits of higher significance) are replaced accordingly to compensate for the loss of the LSD.

```
next-numeral-Proper-GappedEndpoint : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → (gapped : Gapped xs proper)
    → Numeral (suc b) (suc d) o
next-numeral-Proper-GappedEndpoint (x •)    proper gapped
    = z ∷ carry-digit d o proper •
next-numeral-Proper-GappedEndpoint (x ∷ xs) proper gapped
    = z ∷ next-numeral-Proper xs proper
```

Ungapped Endpoints



Just like in the case of *gapped endpoints*, the more significant part of the numeral is replaced with something greater (either a carry-digit or the next numeral). The LSD is incremented and then subtracted from *the exact size of the coming gap.* To compute the size of the gap, it is necessary to compute `next-xs`, the next numeral of the more significant part beforehand. Moreover, we also need to assure that `next-xs` is greater than `xs`. Hence these constructions have to be written as mutually recursive definitions.

```
next-numeral-Proper-UngappedEndpoint : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (greatest : Greatest (lsd xs))
    → (proper : 2 ≤ suc (d + o))
    → (¬gapped : ¬ (Gapped xs proper))
    → Numeral (suc b) (suc d) o
next-numeral-Proper-UngappedEndpoint {b} {d} {o} (x •) greatest proper gapped
    = digit+1-n x greatest gap lower-bound ∷ carry-digit d o proper •
    where
        gap : ℕ
        gap = carry o * suc b

        lower-bound : gap > 0
        lower-bound = ...
next-numeral-Proper-UngappedEndpoint {b} {d} {o} (x ∷ xs) greatest proper gapped
    = digit+1-n x greatest gap lower-bound ∷ next-xs
    where
        gap : ℕ
        gap = (⟦ next-numeral-Proper xs proper ⟧ ∸ ⟦ xs ⟧) * suc b

        lower-bound : gap > 0
        lower-bound =  ... next-numeral-is-greater-Proper xs proper ...
```

Lemmata

These lemmata describe the relations between a given numeral and its next of each category.

Remark Here we show only the propositions because their proofs contain some whopping ~200 lines of equational and preorder reasoning.

```
next-numeral-Proper-Interval-lemma : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (¬greatest : ¬ (Greatest (lsd xs)))
    → (proper : 2 ≤ suc (d + o))
    → ⟦ next-numeral-Proper-Interval xs ¬greatest proper ⟧ ≡ suc ⟦ xs ⟧
```

```
next-numeral-Proper-GappedEndpoint-lemma : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (greatest : Greatest (lsd xs))
    → (proper : 2 ≤ suc (d + o))
    → (gapped : Gapped xs proper)
    → ⟦ next-numeral-Proper-GappedEndpoint xs proper gapped ⟧ > suc ⟦ xs ⟧
```

```
next-numeral-Proper-UngappedEndpoint-lemma : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (greatest : Greatest (lsd xs))
    → (proper : 2 ≤ suc (d + o))
    → (¬gapped : ¬ (Gapped xs proper))
    → ⟦ next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped ⟧
        ≡ suc ⟦ xs ⟧
```

`next-numeral-is-greater-Proper`

With lemmata above, proving `next-numeral-is-greater-Proper` becomes an easy task.

```
next-numeral-is-greater-Proper : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → ⟦ next-numeral-Proper xs proper ⟧ > ⟦ xs ⟧
next-numeral-is-greater-Proper xs proper with nextView xs proper
next-numeral-is-greater-Proper xs proper | Interval b d o ¬greatest =
    start
        suc ⟦ xs ⟧
    ≈⟨ sym (next-numeral-Proper-Interval-lemma xs ¬greatest proper) ⟩
        ⟦ next-numeral-Proper-Interval xs ¬greatest proper ⟧
    □
next-numeral-is-greater-Proper xs proper | GappedEndpoint b d o greatest gapped =
    start
        suc ⟦ xs ⟧
    ≤⟨ n≤1+n (suc ⟦ xs ⟧) ⟩
```

```
        suc (suc ⟦ xs ⟧)
    ≤⟨ next-numeral-Proper-GappedEndpoint-lemma xs greatest proper gapped ⟩
        ⟦ next-numeral-Proper-GappedEndpoint xs proper gapped ⟧
    □
next-numeral-is-greater-Proper xs proper | UngappedEndpoint b d o greatest ¬gapped =
    start
        suc ⟦ xs ⟧
    ≈⟨ sym (next-numeral-Proper-UngappedEndpoint-lemma xs greatest proper ¬gapped) ⟩
        ⟦ next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped ⟧
    □
```

next-numeral-is-immediate-Proper

This proof also follows the same pattern, except that the case of `GappedEndpoint` becomes more complicated as its corresponding lemma establishes only a pre-order relation.

```
next-numeral-is-immediate-Proper : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (ys : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → ⟦ ys ⟧ > ⟦ xs ⟧
    → ⟦ ys ⟧ ≥ ⟦ next-numeral-Proper xs proper ⟧
next-numeral-is-immediate-Proper xs ys proper prop with nextView xs proper
next-numeral-is-immediate-Proper xs ys proper prop | Interval b d o ¬greatest =
    start
        ⟦ next-numeral-Proper-Interval xs ¬greatest proper ⟧
    ≈⟨ next-numeral-Proper-Interval-lemma xs ¬greatest proper ⟩
        suc ⟦ xs ⟧
    ≤⟨ prop ⟩
        ⟦ ys ⟧
    □
next-numeral-is-immediate-Proper xs (y •) proper prop
    | GappedEndpoint b d o greatest gapped
    = contradiction prop $ >⇒≰ $
        start
            suc ⟦ y • ⟧
        ≤⟨ s≤s (greatest-of-all o (lsd xs) y greatest) ⟩
            suc (Digit-toℕ (lsd xs) o)
        ≤⟨ s≤s (lsd-toℕ xs) ⟩
            suc ⟦ xs ⟧
        □
next-numeral-is-immediate-Proper (x •) (y ∷ ys) proper prop
    | GappedEndpoint b d o greatest gapped =
    start
        ⟦ z ∷ carry-digit d o proper • ⟧
    ≤⟨ ... ⟩
        o + carry o * suc b
    ≤⟨ n+-mono o (*n-mono (suc b) ys-lower-bound) ⟩
        o + ⟦ ys ⟧ * suc b
    ≤⟨ ... ⟩
        ⟦ y ∷ ys ⟧
    □
    where
        ys-lower-bound : ⟦ ys ⟧ ≥ carry o
        ys-lower-bound = ...
next-numeral-is-immediate-Proper (x ∷ xs) (y ∷ ys) proper prop
```

```
    | GappedEndpoint b d o greatest gapped =
    start
        ⟦ z ∷ next-numeral-Proper xs proper ⟧
    ≤⟨ ... ⟩
        o + ⟦ next-numeral-Proper xs proper ⟧ * suc b
    ≤⟨ n+-mono o (*n-mono (suc b) ⟦next-xs⟧≤⟦ys⟧) ⟩
        o + ⟦ ys ⟧ * suc b
    ≤⟨ ... ⟩
        ⟦ y ∷ ys ⟧
    □
    where
        ⟦xs⟧<⟦ys⟧ : ⟦ xs ⟧ < ⟦ ys ⟧
        ⟦xs⟧<⟦ys⟧ = ...
        ⟦next-xs⟧≤⟦ys⟧ : ⟦ next-numeral-Proper xs proper ⟧ ≤ ⟦ ys ⟧
        ⟦next-xs⟧≤⟦ys⟧ = next-numeral-is-immediate-Proper xs ys proper ⟦xs⟧<⟦ys⟧

next-numeral-is-immediate-Proper xs ys proper prop
    | UngappedEndpoint b d o greatest ¬gapped =
    start
        ⟦ next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped ⟧
    ≈⟨ next-numeral-Proper-UngappedEndpoint-lemma xs greatest proper ¬gapped ⟩
        suc ⟦ xs ⟧
    ≤⟨ prop ⟩
        ⟦ ys ⟧
    □
```

5.7.7 Putting Everything Together

Finally, we can assemble everything we have constructed and finish these functions and proofs.

next-numeral

```
next-numeral : ∀ {b d o}
    → (xs : Numeral b d o)
    → ¬ (Maximum xs)
    → Numeral b d o
next-numeral {b} {d} {o} xs ¬max with numView b d o
next-numeral xs ¬max | NullBase d o = next-numeral-NullBase xs ¬max
next-numeral xs ¬max | NoDigits b o = NoDigits-explode xs
next-numeral xs ¬max | AllZeros b   = contradiction (Maximum-AllZeros xs) ¬max
next-numeral xs ¬max | Proper b d o proper = next-numeral-Proper xs proper
```

next-numeral-is-greater

```
next-numeral-is-greater : ∀ {b d o}
    → (xs : Numeral b d o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ next-numeral xs ¬max ⟧ > ⟦ xs ⟧
next-numeral-is-greater {b} {d} {o} xs ¬max with numView b d o
next-numeral-is-greater xs ¬max | NullBase d o
```

```
    = next-numeral-is-greater-NullBase xs ¬max
next-numeral-is-greater xs ¬max | NoDigits b o
    = NoDigits-explode xs
next-numeral-is-greater xs ¬max | AllZeros b
    = contradiction (Maximum-AllZeros xs) ¬max
next-numeral-is-greater xs ¬max | Proper b d o proper
    = next-numeral-is-greater-Proper xs proper
```

`next-numeral-is-immediate`

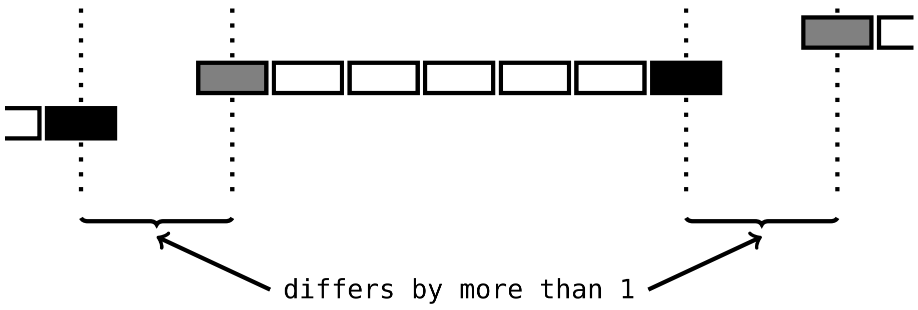
```
next-numeral-is-immediate : ∀ {b d o}
    → (xs : Numeral b d o)
    → (ys : Numeral b d o)
    → (¬max : ¬ (Maximum xs))
    → ⟦ ys ⟧ > ⟦ xs ⟧
    → ⟦ ys ⟧ ≥ ⟦ next-numeral xs ¬max ⟧
next-numeral-is-immediate {b} {d} {o} xs ys ¬max prop with numView b d o
next-numeral-is-immediate xs ys ¬max prop | NullBase d o
    = next-numeral-is-immediate-NullBase xs ys ¬max prop
next-numeral-is-immediate xs ys ¬max prop | NoDigits b o
    = NoDigits-explode xs
next-numeral-is-immediate xs ys ¬max prop | AllZeros b
    = contradiction (Maximum-AllZeros xs) ¬max
next-numeral-is-immediate xs ys ¬max prop | Proper b d o proper
    = next-numeral-is-immediate-Proper xs ys proper prop
```

5.8 Incrementing a Numeral

Given a numeral `xs` of `Numeral b d o` and a proof of `¬ (Maximum xs)`, with functions and theorems constructed in the last section, we can:

- Find the next numeral of `xs` using `next-numeral`.
- Know that the next numeral will be greater than `xs` by `next-numeral-is-greater`.
- Know that the next numeral will be the least numeral that is greater than `xs` by `next-numeral-is-immediate`.

However, none of the theorems above guarantees that the next numeral will be the **successor** of `xs`, i.e., the next numeral and `xs` differs by only 1. For example, we have seen from the previous section that numerals of `GappedEndpoint` of systems of `Proper` have no successors.



Suppose we are to define a function called `increment` that returns the successor of a given numeral. For those numerals that are eligible for increment, we can simply compute them with `next-numeral`; and for those that are not eligible, we need a predicate for discriminating them.

5.8.1 `Incrementable`

Similar to the definition of `Bounded`, we define the predicate `Incrementable` as an existential proposition. To prove that a numeral `xs` is `Incrementable`, one is obliged to present the successor *and* a proof to justify it.

```
Incrementable : ∀ {b d o} → (xs : Numeral b d o) → Set
Incrementable {b} {d} {o} xs = Σ[ xs' ∈ Numeral b d o ] ⟦ xs' ⟧ ≡ suc ⟦ xs ⟧
```

We can then develop lemmata and theorems about `Incrementable`.

Maximum Numerals

If a numeral is a maximum, then there is no such a thing as the next numeral, much less a successor.

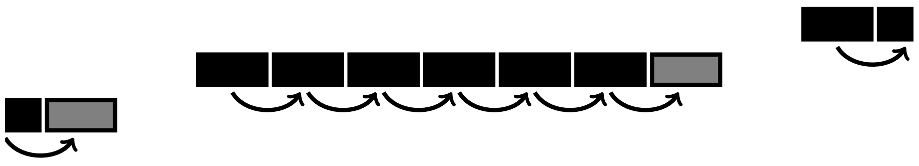
```
Maximum⇒¬Incrementable : ∀ {b d o}
    → (xs : Numeral b d o)
    → (max : Maximum xs)
    → ¬ (Incrementable xs)
Maximum⇒¬Incrementable xs max (incremented , claim)
    = contradiction
        (max incremented)
        (>⇒≰ (m≡1+n⇒m>n claim))
%
```

where `incremented` is the claimed successor and `claim : ⟦ incremented ⟧ ≡ suc ⟦ xs ⟧`. This is proven by contradicting these two propositions:

- `max incremented : ⟦ xs ⟧ ≥ ⟦ incremented ⟧`
- `>⇒≰ (m≡1+n⇒m>n claim) : ⟦ xs ⟧ ≰ ⟦ incremented ⟧`

Numerals in Intervals

Numerals that are located in intervals of digit lines have successors.

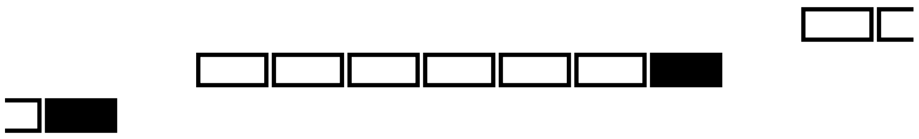


We invoke `next-numeral-Proper-Interval-lemma` to establish the successor relation between the given numeral and the next numeral.

```
Interval⇒Incrementable : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (¬greatest : ¬ (Greatest (lsd xs)))
    → (proper : 2 ≤ suc (d + o))
    → Incrementable xs
Interval⇒Incrementable {b} {d} {o} xs ¬greatest proper
    = (next-numeral-Proper xs proper) , (begin
            ⟦ next-numeral-Proper xs proper ⟧
        ≡⟨ cong ⟦_⟧ (next-numeral-Proper-refine xs proper
            (Interval b d o ¬greatest))
        ⟩
            ⟦ next-numeral-Proper-Interval xs ¬greatest proper ⟧
        ≡⟨ next-numeral-Proper-Interval-lemma xs ¬greatest proper ⟩
            suc ⟦ xs ⟧
        ∎)
```

Numerals at Gapped Endpoints

Numerals that are located at gapped endpoints do not have successors.



```
GappedEndpoint⇒¬Incrementable : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (greatest : Greatest (lsd xs))
    → (proper : 2 ≤ suc (d + o))
    → (gapped : Gapped xs proper)
    → ¬ (Incrementable xs)
GappedEndpoint⇒¬Incrementable xs greatest proper gapped (incremented , claim)
    = contradiction ⟦next⟧>⟦incremented⟧ ⟦next⟧≯⟦incremented⟧
```

This is proven by contradicting two propositions as well.

First, we show that the next numeral is greater than the claimed successor by rephrasing the lemma `next-numeral-Proper-GappedEndpoint-lemma`. As a side note, `next-numeral-Proper` delegates tasks to other helper functions. `next-numeral-Proper-refine` is here to narrow the term computed by `next-numeral-Proper` down to `next-numeral-Proper-GappedEndpoint` by providing evidence that the computation is delegated that way.

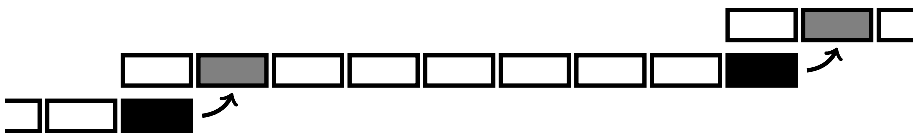
```
⟦next⟧>⟦incremented⟧ : ⟦ next-numeral-Proper xs proper ⟧ > ⟦ incremented ⟧
⟦next⟧>⟦incremented⟧ =
    start
        suc ⟦ incremented ⟧
    ≈⟨ cong suc claim ⟩
        suc (suc ⟦ xs ⟧)
    ≤⟨ next-numeral-Proper-GappedEndpoint-lemma xs greatest proper gapped ⟩
        ⟦ next-numeral-Proper-GappedEndpoint xs proper gapped ⟧
    ≈⟨ cong ⟦_⟧ (sym (next-numeral-Proper-refine
        xs proper (GappedEndpoint b d o greatest gapped)))
    ⟩
        ⟦ next-numeral-Proper xs proper ⟧
    □
```

However, the next numeral should only be greater than the given numeral `xs` and nothing else.

```
⟦next⟧≯⟦incremented⟧ : ⟦ next-numeral-Proper xs proper ⟧ ≯ ⟦ incremented ⟧
⟦next⟧≯⟦incremented⟧ = ≤⇒≯ (next-numeral-is-immediate-Proper
    xs incremented proper (m≡1+n⇒m>n claim))
```

Numerals at Ungapped Endpoints

Numerals that are located at ungapped endpoints also possess successors.



Similar to the case of `Interval`, the key to the proof lies in the help of `next-numeral-Proper-UngappedEndpoint-lemma`.

```
UngappedEndpoint⇒Incrementable : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (greatest : Greatest (lsd xs))
    → (proper : 2 ≤ suc (d + o))
    → (¬gapped : ¬ (Gapped xs proper))
    → Incrementable xs
UngappedEndpoint⇒Incrementable {b} {d} {o} xs greatest proper ¬gapped
    = (next-numeral-Proper xs proper) , (begin
            ⟦ next-numeral-Proper xs proper ⟧
      ≡⟨ cong ⟦_⟧ (next-numeral-Proper-refine xs proper
            (UngappedEndpoint b d o greatest ¬gapped)) ⟩
            ⟦ next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped ⟧
      ≡⟨ next-numeral-Proper-UngappedEndpoint-lemma xs greatest proper ¬gapped ⟩
            suc ⟦ xs ⟧
      ∎)
```

5.8.2 Deciding Incrementable Numerals

With all of the lemmata above, we can finish the most difficult part of the decider.

```
Incrementable?-Proper : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Dec (Incrementable xs)
Incrementable?-Proper xs proper with nextView xs proper
Incrementable?-Proper xs proper | Interval b d o ¬greatest
    = yes (Interval⇒Incrementable xs ¬greatest proper)
Incrementable?-Proper xs proper | GappedEndpoint b d o greatest gapped
    = no (GappedEndpoint⇒¬Incrementable xs greatest proper gapped)
Incrementable?-Proper xs proper | UngappedEndpoint b d o greatest ¬gapped
    = yes (UngappedEndpoint⇒Incrementable xs greatest proper ¬gapped)
```

We can determine if numerals of other categories are incrementable as well.

```
Incrementable? : ∀ {b d o}
    → (xs : Numeral b d o)
    → Dec (Incrementable xs)
Incrementable? xs with Maximum? xs
Incrementable? xs | yes max = no (Maximum⇒¬Incrementable xs max)
Incrementable? {b} {d} {o} xs | no ¬max with numView b d o
Incrementable? xs | no ¬max | NullBase d o
    = yes ((next-numeral-NullBase xs ¬max) ,
        (next-numeral-NullBase-lemma xs ¬max))
Incrementable? xs | no ¬max | NoDigits b o
    = yes (NoDigits-explode xs)
Incrementable? xs | no ¬max | AllZeros b
```

```
    = no (contradiction (Maximum-AllZeros xs) ¬max)
Incrementable? xs | no ¬max | Proper b d o proper
    = Incrementable?-Proper xs proper
```

The table below sums up whether a non-maximum numeral can be incremented in systems of each category.

NullBase	**NoDigits**	**AllZeros**	**Proper**		
			Inteval	**Gapped**	**Ungapped**
yes	yes	no	yes	no	yes

Note that the decision of making numerals of `NoDigits` incrementable is completely arbitrary as it will always hold vacuously.

5.8.3 `increment`

We can actually do without the previous section about `Incrementable?` and still be able to define a function that increments numerals. All we have to do is to ask the user to prove that the numeral he or she has given is incrementable. By doing so, the user is obliged to provide the actual successor, and then we can steal it and pretend that we have found the successor. How outrageous!

```
increment : ∀ {b d o}
    → (xs : Numeral b d o)
    → (incr : Incrementable xs)
    → Numeral b d o
increment xs incr = proj₁ incr
```

Obviously this is not the desired implementation of `increment`. The reason why we construct `Incrementable?` is that it not only does the real work of computing successors for us (the credit also goes to `next-numeral`) but also explains why a numeral is incrementable (or not).

The decidability of `Incrementable?` enables us to embed it in types. To filter out numerals that are not eligible for increment, we use the same trick as when implementing safe `head` on lists.

```
True : {P : Set} → Dec P → Set
True (yes _) = ⊤
True (no _) = ⊥
```

`True` translates positive results of a decidable predicate to `⊤` and negative results to `⊥`, where as `toWitness` reclaims the proof of the given proposition for us.

```
toWitness : {P : Set} {Q : Dec P} → True Q → P
toWitness {Q = yes p} _  = p
toWitness {Q = no  _} ()
```

Finally, the true implementation of `increment` is as follows:

```
increment : ∀ {b d o}
    → (xs : Numeral b d o)
    → (incr : True (Incrementable? xs))
    → Numeral b d o
increment xs incr = proj₁ (toWitness incr)
```

5.8.4 Properties of `increment`

```
increment-next-numeral : ∀ {b d o}
    → (xs : Numeral b d o)
    → (¬max : ¬ (Maximum xs))
    → (incr : True (Incrementable? xs))
    → increment xs incr ≡ next-numeral xs ¬max
```

This property relates `increment` with `next-numeral`. It may look trivial, however, the underlying implementation of `increment` does not actually involve `next-numeral`, but helper functions like `next-numeral-Proper` and `next-numeral-NullBase`. We dispense with the proof of this property as it is comprised mostly of pattern matching and seemly meaningless `refl`s.

5.9 Continuous Systems

We say a numeral system is *continuous* if there are no gaps in the number line and we can always find a successor after each numeral. In other words, a system is continuous if every numeral is *incrementable*.

```
Continuous : ∀ b d o → Set
Continuous b d o = (xs : Numeral b d o) → Incrementable xs
```

Bounded systems are deemed to be incontinuous because there are no successor after the maximum numeral.

```
Bounded⇒¬Continuous : ∀ {b d o}
    → Bounded b d o
    → ¬ (Continuous b d o)
```

```
Bounded⇒¬Continuous (xs , max) claim
    = contradiction (claim xs) (Maximum⇒¬Incrementable xs max)
```

Since systems of `NullBase` and `AllZeros` are all bounded, they cannot be continuous. We will only be concerning ourselves with systems of `Proper` in the rest of the section.

Suppose we want to determine if a system of `Proper` is continuous, we can examine *every* numeral of that system with the view function `nextView`. If *all* numerals are sorted as `Interval` and `UngappedEndpoint`, then the system is continuous; if *any* numeral is sorted as `GappedEndpoint`, then the system must not be continuous.

But it is impossible to go through every numeral because we know that systems of `Proper` are non-exhaustive. Therefore we need another way of deciding the continuity of a system.

5.9.1 Look no further than the Gaps

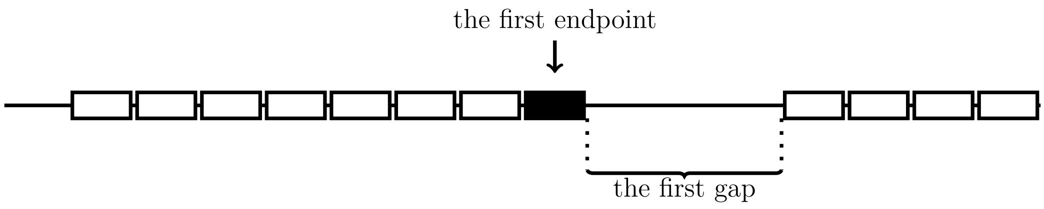
In the section 5.7, we have propositions for describing gaps.

```
Gapped#0 : ∀ b d o → Set
Gapped#0 b d o = suc d < carry o * suc b

Gapped#N : ∀ b d o
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Set
Gapped#N b d o xs proper
    = suc d < (⟦ next-numeral-Proper xs proper ⟧ ∸ ⟦ xs ⟧) * suc b

Gapped : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Set
Gapped {b} {d} {o} (x •)    proper = Gapped#0 b d o
Gapped {b} {d} {o} (x ∷ xs) proper = Gapped#N b d o xs proper
```

`Gapped#0` describes the first gap of a system, whereas `Gapped#N` covers the rest. Compared to `Gapped#N`, the definition of `Gapped#0` is much less demanding because it only depends on the three indices, making it really easy to be determined.

5.9.2 When the First Gap is Open

Let's start with the easier case, we can know for sure that when the first gap `Gapped#0` is open, the system must not be continuous.



We construct the first endpoint with the greatest digit.

```
first-endpoint : ∀ b d o → Numeral (suc b) (suc d) o
first-endpoint b d o = greatest-digit d •
```

The first endpoint is proven to be not incrementable.

```
first-endpoint-¬Incrementable : ∀ {b d o}
    → (proper : 2 ≤ suc (d + o))
    → (gapped : Gapped#0 b d o)
    → ¬ (Incrementable (first-endpoint b d o))
first-endpoint-¬Incrementable {b} {d} {o} proper gapped
    with nextView (first-endpoint b d o) proper
first-endpoint-¬Incrementable proper gapped
    | Interval b d o ¬greatest
    = contradiction (greatest-digit-is-the-Greatest d) ¬greatest
first-endpoint-¬Incrementable proper gapped
    | GappedEndpoint b d o greatest _
    = GappedEndpoint⇒¬Incrementable
        (first-endpoint b d o) greatest proper gapped
first-endpoint-¬Incrementable proper gapped
    | UngappedEndpoint b d o greatest ¬gapped
    = contradiction gapped ¬gapped
```

The first endpoint can then be used as an counter example of the continuity of the system.

```
Gapped#0⇒¬Continuous : ∀ {b d o}
    → (proper : 2 ≤ suc (d + o))
    → (gapped : Gapped#0 b d o)
    → ¬ (Continuous (suc b) (suc d) o)
Gapped#0⇒¬Continuous {b} {d} {o} proper gapped cont
```

```
    = contradiction
        (cont (first-endpoint b d o))
        (first-endpoint-¬Incrementable proper gapped)
```

Now we can see if a system is incontiuous just by checking the first gap.

```
Continuous-Proper : ∀ b d o
    → (proper : 2 ≤ suc (d + o))
    → Dec (Continuous (suc b) (suc d) o)
Continuous-Proper b d o proper with Gapped#0? b d o
Continuous-Proper b d o proper | yes gapped#0
    = no (Gapped#0⇒¬Continuous proper gapped#0)
Continuous-Proper b d o proper | no ¬gapped#0
    = ?
```

However, when the first gap is closed, it is uncertain whether the given system will be continuous.

5.9.3 When the First Gap is Closed

If we can prove that `Gapped#N` **implies** `Gapped#0`, then by contraposition, we can know that all gap of a system are closed if the first gap is also closed.

```
Gapped#N⇒Gapped#0 : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → Gapped#N b d o xs proper
    → Gapped#0 b d o
Gapped#N⇒Gapped#0 xs proper gapped#N with nextView xs proper
Gapped#N⇒Gapped#0 xs proper gapped#N | Interval b d o ¬greatest =
    start
        suc (suc d)
    ≤⟨ gapped#N ⟩
        (⟦ next-numeral-Proper-Interval xs ¬greatest proper ⟧ ∸ ⟦ xs ⟧) * suc b
    ≤⟨ *n-mono (suc b) (...next-numeral-Proper-Interval-lemma xs ¬greatest...) ⟩
        (suc zero ⊔ o) * suc b
    □)
Gapped#N⇒Gapped#0 (x •)    proper _ | GappedEndpoint b d o greatest gapped#0
    = gapped#0
Gapped#N⇒Gapped#0 (x ∷ xs) proper _ | GappedEndpoint b d o greatest gapped#N
    = Gapped#N⇒Gapped#0 xs proper gapped#N
Gapped#N⇒Gapped#0 xs proper gapped#N | UngappedEndpoint b d o greatest ¬gapped =
    start
        suc (suc d)
    ≤⟨ gapped#N ⟩
        (⟦ next-numeral-Proper-UngappedEndpoint xs greatest proper ¬gapped ⟧
            ∸ ⟦ xs ⟧) * suc b
    ≤⟨ *n-mono (suc b) (... next-numeral-Proper-UngappedEndpoint-lemma ...) ⟩
        (suc zero ⊔ o) * suc b
    □)
```

We can prove that, if the first gap is closed, then so do the rest of the gaps, by contraposition.

```
¬Gapped#0⇒¬Gapped : ∀ {b d o}
    → (xs : Numeral (suc b) (suc d) o)
    → (proper : 2 ≤ suc (d + o))
    → ¬ (Gapped#0 b d o)
    → ¬ (Gapped xs proper)
¬Gapped#0⇒¬Gapped (x •)    proper ¬Gapped#0 = ¬Gapped#0
¬Gapped#0⇒¬Gapped (x ∷ xs) proper ¬Gapped#0 = contraposition
    (Gapped#N⇒Gapped#0 xs proper)
    ¬Gapped#0
```

We can conclude that, if all gaps are closed, then all numerals should be incrementable, therefore the system is continuous.

```
¬Gapped#0⇒Continuous : ∀ {b d o}
    → (proper : 2 ≤ suc (d + o))
    → (¬gapped : ¬ (Gapped#0 b d o))
    → Continuous (suc b) (suc d) o
¬Gapped#0⇒Continuous proper ¬gapped#0 xs with Incrementable? xs
¬Gapped#0⇒Continuous proper ¬gapped#0 xs | yes incr = incr
¬Gapped#0⇒Continuous proper ¬gapped#0 xs | no ¬incr = contradiction
    (¬Gapped⇒Incrementable xs proper (¬Gapped#0⇒¬Gapped xs proper ¬gapped#0))
    ¬incr
```

5.9.4 Determining Continuity

Finally, we can decide if a system is continuous.

```
Continuous-Proper : ∀ b d o
    → (proper : 2 ≤ suc (d + o))
    → Dec (Continuous (suc b) (suc d) o)
Continuous-Proper b d o proper with Gapped#0? b d o
Continuous-Proper b d o proper | yes gapped#0
    = no (Gapped#0⇒¬Continuous proper gapped#0)
Continuous-Proper b d o proper | no ¬gapped#0
    = yes (¬Gapped#0⇒Continuous proper ¬gapped#0)

Continuous? : ∀ b d o → Dec (Continuous b d o)
Continuous? b d o with numView b d o
Continuous? _ _ _ | NullBase d o = no (Bounded⇒¬Continuous (Bounded-NullBase d o))
Continuous? _ _ _ | NoDigits b o = yes (λ xs → NoDigits-explode xs)
Continuous? _ _ _ | AllZeros b = no (Bounded⇒¬Continuous (Bounded-AllZeros b))
Continuous? _ _ _ | Proper b d o proper = Continuous-Proper b d o proper
```

5.9.5 Successor Function

Since the proof of `Continuous` is essentially a successor function, we can use it to increment a numeral.

```
1+ : ∀ {b d o}
    → {cont : True (Continuous? b d o)}
    → (xs : Numeral b d o)
    → Numeral b d o
1+ {cont = cont} xs = proj₁ (toWitness cont xs)
```

A numeral taking these two routes should result in the same ℕ.

$$\begin{tikzcd}
\texttt{Numeral b d o} \arrow[r, "\texttt{+1}"] \arrow[d, "⟦\_⟧"'] & \texttt{Numeral b d o} \arrow[d, "⟦\_⟧"] \\
ℕ \arrow[r, "\texttt{suc}"'] & ℕ
\end{tikzcd}$$

The proof also comes for free from `Continuous` as part of its definition.

```
1+-toℕ : ∀ {b d o}
    → {cont : True (Continuous? b d o)}
    → (xs : Numeral b d o)
    → ⟦ 1+ {cont = cont} xs ⟧ ≡ suc ⟦ xs ⟧
1+-toℕ {cont = cont} xs = proj₂ (toWitness cont xs)
```

5.10 Converting from Natural Numbers

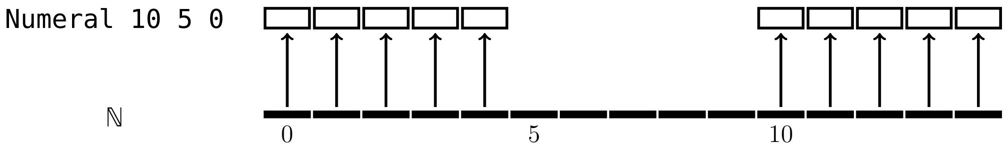
`Numeral` is a family of types, so are functions that are defined on `Numeral`. It is important to regard functions such as ⟦_⟧ as a collective; each has a different configuration of indices.

Here, we name the function that converts natural numbers to corresponding numerals `fromℕ`.

$$\begin{tikzcd}
\texttt{Numeral b d o} \arrow[r, shift left, "⟦\_⟧"] & ℕ \arrow[l, shift left, "\texttt{fromℕ}"]
\end{tikzcd}$$

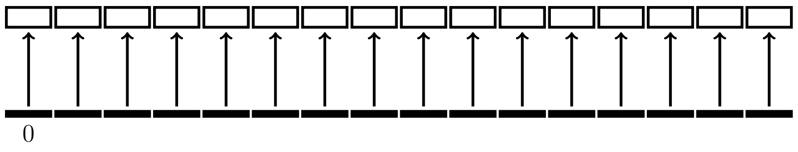
Because `fromℕ` should behave like an inverse function of ⟦_⟧, the codomain of ⟦_⟧ should be the domain of `fromℕ`. For `fromℕ` to be well-defined, its corresponding ⟦_⟧ must be surjective. However, not all instances of ⟦_⟧ are surjective.

Take the numeral system `Numeral 10 5 0` for example. Natural numbers such as 5 or 6 cannot be converted to `Numeral 10 5 0` because its evaluation function ⟦_⟧ is not surjective.



5.10.1 Domains of `fromℕ`

We can define `fromℕ` only on a subset of `Numeral` that possess surjective evaluation functions. For a system to be surjective, in addition of being continuous, its numerals would also have to be able to cover every natural number, including "0".

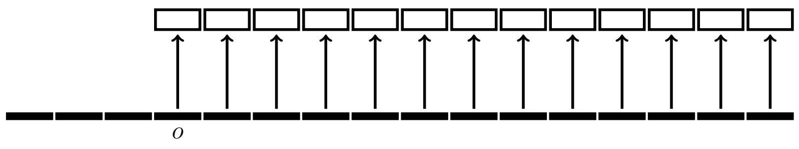


Therefore, the index *o* can only be 0 because that is where the least numeral starts counting from. We can convert all natural numbers to such systems by induction and recursively replacing the successor of ℕ with `1+`.

```
fromℕ : ∀ {b d}
    → {cont : True (Continuous? b (suc d) 0)}
    → ℕ
    → Numeral b (suc d) 0
fromℕ zero                   = z •
fromℕ {cont = cont} (suc n) = 1+ {cont = cont} (fromℕ n)
```

However, it would be a shame if only a few of `Numeral` can be converted from natural numbers, and all that effort we have put into generalizing *o* will be gone to waste.

If we make `fromℕ` a partial function, allowing it to convert only a subset of natural numbers that are greater than o instead of 0, then much more numeral systems will be applicable for `fromℕ`.



Besides, functions and predicates such as `+1` and `Continuous` will still be useful; we did not work our head off constructing these things for nothing.

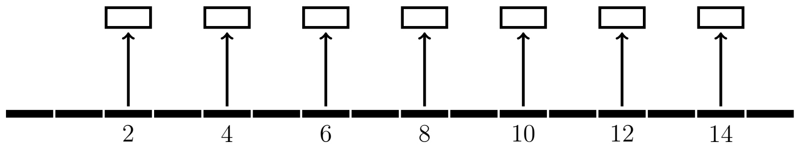
We generalize the previous definition of `fromℕ` and make o the lower bound of the given ℕ. If the given number n happens to be equal to o then we return the least numeral `z •`, else we replace the successor with `+1` recursively by induction on n.

```
fromℕ : ∀ {b d o}
    → {cont : True (Continuous? b (suc d) o)}
    → (n : ℕ)
    → n ≥ o
    → Numeral b (suc d) o
fromℕ {o = o}       n       p   with o ≟ n
fromℕ               n       n≥o | yes eq = z •
fromℕ {o = o}       zero    n≥o | no ¬eq
    = contradiction (≤0⇒≡0 o n≥o) ¬eq
fromℕ {cont = cont} (suc n) n≥o | no ¬eq
    = 1+
      {cont = cont}
      (fromℕ {cont = cont} n (≤-pred (≤∧≢⇒< n≥o ¬eq)))
```

The domain we have chosen for `fromℕ` seems reasonable. However, there are other possible choices of the domain.

5.10.2 Other options of the domain of `fromℕ`

Consider `Numeral 1 1 2`, a unary system with only one digit that is “2”.



Such a numeral system can represent all even numbers except for zero. To convert natural numbers to `Numeral 1 1 2`, we can devise a predicate like `Even-but-not-zero` such that only these numbers fit.

```
fromℕ : ∀ {b d o}
    → (n : ℕ)
    → (Even-but-not-zero n)
    → Numeral b (suc d) o
```

In fact, we can even generalize from `Numeral 1 1 1` (ordinary unary numerals) and `Numeral 1 1 2` to `Numeral 1 1 n` to describe all cyclic groups of order n.

What about other kinds of predicates? Can we choose any subset of natural numbers to convert from as long as it fits the targeting numeral system? The point is, the reason why we choose a subset of natural numbers as the domain of `fromℕ` is **arbitrary** and **empirical**.

Nonetheless, we settled for this specific kind of `fromℕ` because we think that it can best describe the numerals that are closed under the successor function and even the addition function, which may be suitable for indexing numerical representations that supports operations such as insertion and merge.

```
fromℕ : ∀ {b d o}
    → {cont : True (Continuous? b (suc d) o)}
    → (n : ℕ)
    → n ≥ o
    → Numeral b (suc d) o
```

5.10.3 Properties of `fromℕ`

The property below states that `fromℕ` is a *right inverse* of ⟦_⟧. Having a right inverse means that ⟦_⟧ is *surjective*.

```
fromℕ-toℕ : ∀ {b d o}
    → (cont : True (Continuous? b (suc d) o))
    → (n : ℕ)
    → (n≥o : n ≥ o)
    → ⟦ fromℕ {cont = cont} n n≥o ⟧ ≡ n
fromℕ-toℕ {o = o} cont n       n≥o with o ≟ n
fromℕ-toℕ         cont n       n≥o | yes eq = eq
fromℕ-toℕ {o = o} cont zero    n≥o | no ¬eq
    = contradiction (≤0⇒≡0 o n≥o) ¬eq
fromℕ-toℕ         cont (suc n) n≥o | no ¬eq =
    let
        n≥o' = ≤-pred (≤∧≢⇒< n≥o ¬eq)
    in
    begin
        ⟦ 1+ {cont = cont} (fromℕ {cont = cont} n n≥o') ⟧
    ≡⟨ 1+-toℕ {cont = cont} (fromℕ {cont = cont} n n≥o') ⟩
        suc ⟦ fromℕ {cont = cont} n n≥o' ⟧
    ≡⟨ cong suc (fromℕ-toℕ cont n n≥o') ⟩
        suc n
    ∎
```

Conversely, ⟦_⟧ would also have a *left inverse* if it is injective. However, ⟦_⟧ would never be injective because some numeral systems such as the decimals are *redundant* as they map more than one numeral onto the same number.

5.11 The Addition Function

The addition function on ℕ can consequently be defined by recursively "moving" the successor function `suc` from one number to the another.

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

We may attempt to define addition on `Numeral` with the same strategy.

```
_+_ : ∀ {b d o}
    → Numeral b d o
    → Numeral b d o
    → Numeral b d o
x •      + ys = ? x ys
(x ∷ xs) + ys = ? x (xs + ys)
```

However, there is not a corresponding successor function on `Numeral` that allows us to recursively move a digit from one numeral to the another with. What we need is some sort of "one-sided" addition function that takes *a digit* and *a numeral* instead of two numerals.

```
n+ : ∀ {b d o}
    → Digit d
    → Numeral b d o
    → Numeral b d o
```

Moreover, to ensure that a system is closed under these operations, we require these systems to be `Continuous`.

```
∀ {b d o} → True (Continuous? b d o)
```

5.11.1 Sum of Two Digits

Before implementing `n+`, we need to take a step back and figure out how to add *two digits* together. Adding two digits should result in **a digit** and possibly **a carry** when the sum overflows.

The carry is often a *fixed digit* in most of the systems that we are familiar with. However, when it comes to redundant systems, the carry is often *indefinite*, because there is more than one way of representing a number.

View for the Sum

We capture these cases with a view called `Sum`.

```
data Sum : (b d o : ℕ) (x y : Digit (suc d)) → Set where
```

When the sum of two digits can still be represented by a single digit, we compute the sum as `leftover` and support it with a proof.

```
    NoCarry : ∀ {b d o x y}
        → (leftover : Digit (suc d))
        → (property : Digit-toℕ leftover o ≡ sum o x y)
        → Sum b d o x y
```

When the sum of two digits exceeds the upper bound of a digit, we move the exceeding part to `carry` and leave the remainder to `leftover`. `property` ensures the integrity of this computation.

We opt for the constructor `Fixed` when the exceeding part that goes to `carry` is fixed (equals to `1 ⊔ o`) and `Floating` when it is indefinite.

```
    Fixed Floating : ∀ {b d o x y}
        → (leftover carry : Digit (suc d))
        → (property : Digit-toℕ leftover o
                    + (Digit-toℕ carry o) * suc b ≡ sum o x y)
        → Sum b d o x y
```

We dispense with the implementation of the corresponding view function `sumView` for brevity. [7]

[7]about 300 lines of code and reasoning.

```
sumView : ∀ b d o
    → (¬gapped : ¬ (Gapped#0 b d o))
    → (proper : 2 ≤ suc d + o)
    → (x y : Digit (suc d))
    → Sum b d o x y
```

n+ on Proper Systems

Implementing `n+` on proper systems are relatively simple since most of the hard work has been done by `sumView`. Carries are added back by recursively calling `n+-Proper`.

```
n+-Proper : ∀ {b d o}
    → (¬gapped : ¬ (Gapped#0 b d o))
    → (proper : suc d + o ≥ 2)
    → (x : Digit (suc d))
    → (xs : Numeral (suc b) (suc d) o)
    → Numeral (suc b) (suc d) o
n+-Proper {b} {d} {o} ¬gapped proper x xs
    with sumView b d o ¬gapped proper x (lsd xs)
n+-Proper ¬gapped proper x (_ •)    | NoCarry leftover property
    = leftover •
n+-Proper ¬gapped proper x (_ ∷ xs) | NoCarry leftover property
    = leftover ∷ xs
n+-Proper ¬gapped proper x (_ •)    | Fixed leftover carry property
    = leftover ∷ carry •
n+-Proper ¬gapped proper x (_ ∷ xs) | Fixed leftover carry property
    = leftover ∷ n+-Proper ¬gapped proper carry xs
n+-Proper ¬gapped proper x (_ •)    | Floating leftover carry property
    = leftover ∷ carry •
n+-Proper ¬gapped proper x (_ ∷ xs) | Floating leftover carry property
    = leftover ∷ n+-Proper ¬gapped proper carry xs
```

Properties of n+-Proper

The property below verifies the correctness of `n+-Proper`.

```
n+-Proper-toℕ : ∀ {b d o}
    → (¬gapped : ¬ (Gapped#0 b d o))
    → (proper : suc d + o ≥ 2)
    → (x : Digit (suc d))
    → (xs : Numeral (suc b) (suc d) o)
    → ⟦ n+-Proper ¬gapped proper x xs ⟧ ≡ Digit-toℕ x o + ⟦ xs ⟧
n+-Proper-toℕ ¬gapped proper x xs with sumView b d o ¬gapped proper x (lsd xs)
n+-Proper-toℕ ¬gapped proper x (_ •)     | NoCarry _ property = property
n+-Proper-toℕ ¬gapped proper x (x' ∷ xs) | NoCarry leftover property =
    begin
        ⟦ leftover ∷ xs ⟧
    ≡⟨ ... property ... ⟩
        Digit-toℕ x o + ⟦ x' ∷ xs ⟧
    ∎
```

```
n+-Proper-toℕ ¬gapped proper x (_ •)      | Fixed _ _ property = property
n+-Proper-toℕ ¬gapped proper x (x' ∷ xs) | Fixed leftover carry property =
    begin
        ⟦ leftover ∷ n+-Proper ¬gapped proper carry xs ⟧
    ≡⟨ ... n+-Proper-toℕ ... ⟩
        Digit-toℕ leftover o + (Digit-toℕ carry o + ⟦ xs ⟧) * suc b
    ≡⟨ ... property ... ⟩
        Digit-toℕ x o + ⟦ x' ∷ xs ⟧
    ∎
n+-Proper-toℕ ¬gapped proper x (_ •)      | Floating _ _ property = property
n+-Proper-toℕ ¬gapped proper x (x' ∷ xs) | Floating leftover carry property =
    begin
        ⟦ leftover ∷ n+-Proper ¬gapped proper carry xs ⟧
    ≡⟨ ... n+-Proper-toℕ ... ⟩
        Digit-toℕ leftover o + (Digit-toℕ carry o + ⟦ xs ⟧) * suc b
    ≡⟨ ... property ... ⟩
        Digit-toℕ x o + ⟦ x' ∷ xs ⟧
    ∎
```

Generalizing `n+-Proper` to all Systems

By imposing the condition of continuity, we exclude systems that are not acceptable for `n+-Proper`.

```
n+ : ∀ {b d o}
    → {cont : True (Continuous? b d o)}
    → (n : Digit d)
    → (xs : Numeral b d o)
    → Numeral b d o
n+ {b} {d} {o}      n xs with numView b d o
n+ {_} {_} {_} {()} n xs | NullBase d o
n+ {_} {_} {_}      n xs | NoDigits b o = NoDigits-explode xs
n+ {_} {_} {_} {()} n xs | AllZeros b
n+ {_} {_} {_}      n xs | Proper b d o proper with Gapped#0? b d o
n+ {_} {_} {_} {()} n xs | Proper b d o proper | yes gapped#0
n+ {_} {_} {_}      n xs | Proper b d o proper | no ¬gapped#0
    = n+-Proper ¬gapped#0 proper n xs
```

The proof of the correctness of `n+-toℕ` also follows the same pattern.

```
n+-toℕ : ∀ {b d o}
    → {cont : True (Continuous? b d o)}
    → (n : Digit d)
    → (xs : Numeral b d o)
    → ⟦ n+ {cont = cont} n xs ⟧ ≡ Digit-toℕ n o + ⟦ xs ⟧
```

Implmentation of the Addition Function

First, we define the addition function on systems of `Proper` with the one-sided addition function `n+-Proper`.

```
+-Proper : ∀ {b d o}
    → (¬gapped : ¬ (Gapped#0 b d o))
    → (proper : suc d + o ≥ 2)
    → (xs ys : Numeral (suc b) (suc d) o)
    → Numeral (suc b) (suc d) o
+-Proper ¬gapped proper (x •)    ys       = n+-Proper ¬gapped proper x ys
+-Proper ¬gapped proper (x ∷ xs) (y •)    = n+-Proper ¬gapped proper y (x ∷ xs)
+-Proper {b} {d} {o} ¬gapped proper (x ∷ xs) (y ∷ ys)
    with sumView b d o ¬gapped proper x y
+-Proper ¬gapped proper (x ∷ xs) (y ∷ ys) | NoCarry leftover property
    = leftover ∷ +-Proper ¬gapped proper xs ys
+-Proper ¬gapped proper (x ∷ xs) (y ∷ ys) | Fixed leftover carry property
    = leftover ∷ n+-Proper ¬gapped proper carry (+-Proper ¬gapped proper xs ys)
+-Proper ¬gapped proper (x ∷ xs) (y ∷ ys) | Floating leftover carry property
    = leftover ∷ n+-Proper ¬gapped proper carry (+-Proper ¬gapped proper xs ys)
```

By generalizing `+-Proper` to all continuous numeral systems, the addition function we have been long for can be implemented as follows.

```
_+_ : ∀ {b d o}
    → {cont : True (Continuous? b d o)}
    → (xs ys : Numeral b d o)
    → Numeral b d o
_+_ {b} {d} {o}  xs ys with numView b d o
_+_ {cont = ()}   xs ys | NullBase d o
_+_ {cont = cont} xs ys | NoDigits b o = NoDigits-explode xs
_+_ {cont = ()}   xs ys | AllZeros b
_+_ {cont = cont} xs ys | Proper b d o proper with Gapped#0? b d o
_+_ {cont = ()}   xs ys | Proper b d o proper | yes ¬gapped#0
_+_ {cont = cont} xs ys | Proper b d o proper | no ¬gapped#0
    = +-Proper ¬gapped#0 proper xs ys
```

Properties of the Addition Function

We can prove that the evaluation function ⟦_⟧ is a homomorphism between `Numeral` and ℕ that preserves their addition functions.

```
toℕ-+-homo : ∀ {b d o}
    → (cont : True (Continuous? b d o))
    → (xs ys : Numeral b d o)
    → ⟦ _+_ {cont = cont} xs ys ⟧ ≡ ⟦ xs ⟧ + ⟦ ys ⟧
```

The proofs are skipped for brevity. [8]

[8]about 100 lines of code and reasoning.

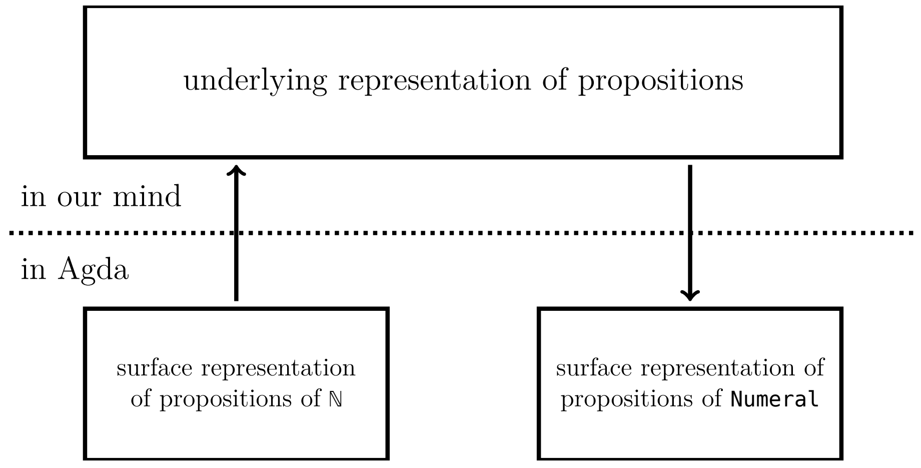
Chapter 6

Translating Propositions and Proofs

In this chapter, we demonstrate how to translate propositions and proofs between ℕ and a subset of `Numeral`.

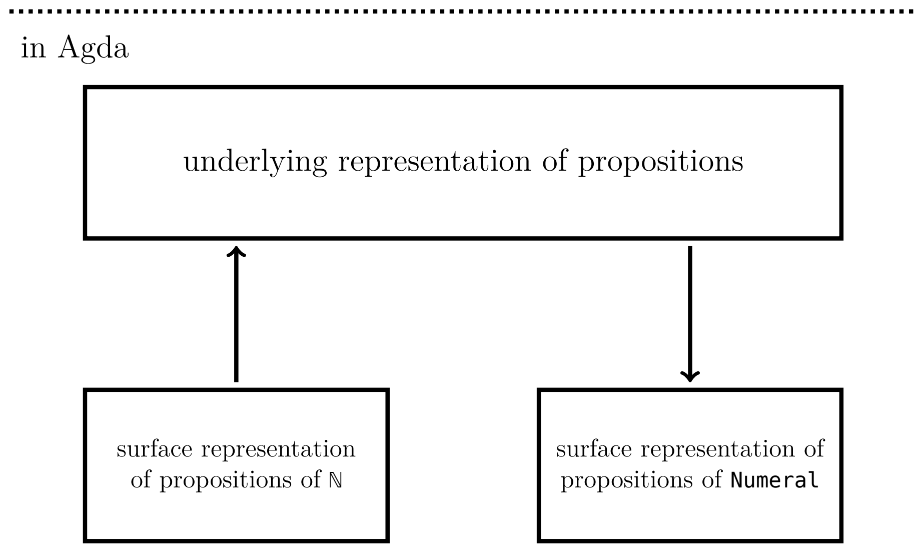
People have put a lot of effort into proving theorems and properties on ℕ. However, all these results are unusable for `Numeral`, although ℕ and `Numeral` both have a similar structure and share the same purpose of representing natural numbers. We cannot simply apply a property of ℕ, say, the commutative property of addition, on instances of `Numeral`.

The reason behind this misery comes from the difference between so-called **object languages** and **metalanguages**. The relation and similarities between ℕ and `Numeral` we have observed are expressed using some *metalanguage* in our mind; while the propositions and proofs we want to reuse are formalized with Agda, the *object language* we have been dwelling on.


underlying representation of propositions
in our mind
in Agda
surface representation of propositions of ℕ
surface representation of propositions of Numeral

When we are proving propositions of `Numeral` base on the ones that already existed on ℕ, we are actually translating propositions via some meta-language in our mind.

To free ourselves from these brainworks, we can encode the underlying representation of propositions in Agda, so that we can manipulate and translate them with the help of constructions such as functions available from the object language.


in Agda
underlying representation of propositions
surface representation of propositions of ℕ
surface representation of propositions of Numeral

The underlying representation of propositions can be encoded in Agda using a generic programming technique called *universe construction* [9].

6.1 Universe Constructions

A universe is a set of types; a universe construction consists of:

- A universe
- A datatype of "codes"
- A decoding function that maps "codes" to the types in the universe

6.1.1 A Simple Universe

There is an universe construction we are familiar with: the universe of `⊤` and `⊥`.

Types

```
data ⊤ : Set where
    tt : ⊤
data ⊥ : Set where
```

Codes

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

Decoder

```
isTrue : Bool → Set
isTrue true  = ⊤
isTrue false = ⊥
```

As we have shown, we can use this universe to implement, for instance, a safe `head` function that cannot be type-checked on empty inputs.

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

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

head : ∀ {A} → (xs : List A) → isTrue (not (null xs)) → A
head []       ()
head (x ∷ xs) proof = x
```

We can generate code from `null`, manipulate the code with `not`, and finally, realize them to `⊤` or `⊥` with `isTrue`.

6.1.2 Summary

Below is a table that sums up the correspondences of universe construction.

universe	corresponds to	surface representation	corresponds to	semantics
codes	corresponds to	underlying representation	corresponds to	syntax
decoder	corresponds to	realization	corresponds to	interpretation

6.2 First-order Logic on the Representations of Natural Numbers

To translate propositions (and predicates) from $\mathbb{N}$ to `Numeral` (or vice versa), we need an underlying syntax for expressing these propositions. For that matter, we will demonstrate how to build a minimalistic first-order logic with universe constructions.

6.2.1 Syntax

Syntax and semantics are two key parts of first-order logic.

What we called "proposition" are actually *predicates* that have quantifiers ranging over. Predicates are composed of smaller predicates or basic units that we called *terms*.

There are many kinds of predicates we can make on natural numbers. However, we will target only at the predicates that can be expressed in both $\mathbb{N}$ and `Numeral`.

Therefore, the set of terms will be inductively defined by only two rules:

- **Variable**: Any variable of the *domain of discourse* ($\mathbb{N}$ or `Numeral`) is a term.
- **Addition**: Any two terms put together with the addition function of that domain is term as well.

We will only have terms that look like this:

```
x + (y + z)
```

The set of predicates is inductively defined by three rules:

- **Equality**: If ϕ and ψ are two terms, then $\phi \equiv \psi$ is a predicate.
- **Implication**: If ϕ and ψ are two predicates, then $\phi \implies \psi$ is also a predicate.

- **Universal Quantification**: If ϕ is predicate that may contain $1 + n$ free variables, then $\forall\phi$ is a predicate that may only contain n free variables.

With these connectives and quantifiers, we can make predicates such as:

```
(x + y) + z ≈ x + (y + z)
∀ x . x ≈ y → y ≈ y
```

6.2.2 Codes for Syntax

To describe the syntax, we have two datatypes for codes, one for *terms* and another for *predicates*.

Terms

```
data Term : ℕ → Set where
    var : ∀ {n} → Fin n → Term n
    _+T_ : ∀ {n} → Term n → Term n → Term n
```

The code for terms is indexed by a ℕ which denotes the number of *free variables* that may occur in a predicate. The `Fin n` that `var` takes designates its binder.

Predicates

```
data Predicate : ℕ → Set where
    _≈P_ : ∀ {n} → (t₁ : Term n) → (t₂ : Term n) → Predicate n
    _→P_ : ∀ {n} → (p₁ : Predicate n) → (p₂ : Predicate n)
         → Predicate n
    ∀P   : ∀ {n} → (p : Predicate (suc n)) → Predicate n
```

The code for predicates indexed by the number of *free variables* as well. `_≈P_` and `_→P_` correspond to equality and implication connectives respectively. It is worth noting that `∀P` returns a predicate with a smaller index because one of the free variable of the given predicate has been bound.

Examples

The transitivity of identity of the equality connective can be described as follows:

```
≈-trans : Predicate zero
≈-trans = let   x = var (# 0)
                y = var (# 1)
                z = var (# 2)
    in ∀P (∀P (∀P (((x ≈P y) →P (y ≈P z)) →P (x ≈P z))))
```

6.2.3 Semantics

The semantics determine meanings behind these propositions, and by meanings, we mean the selection of types of the universe we are constructing.

Signatures

A syntax can be interpreted as many different semantics. Each semantics has its own choice of interpreting a piece of syntax, we call these choices a *signature*. Therefore each semantics has its own signature.

The datatype of signature can defined as follows:

```
record Signature : Set₁ where
    constructor sig
    field
        carrier : Set
        _⊕_ : carrier → carrier → carrier
        _≈_ : carrier → carrier → Set
```

Different semantics have different signatures.

```
ℕ-sig : Signature
ℕ-sig = sig ℕ _+_ _≡_

Numeral-sig : (b d : ℕ) → True (Continuous? b d 0) → Signature
Numeral-sig b d cont = sig (Numeral b d 0) (_+_ {cont = cont}) _≈_
```

Note that `ℕ-sig` and `Numeral-sig` are equipped with different equality connectives. The one equipped by `Numeral-sig` is defined as follows:

```
_≈_ : ∀ {b d o}
    → (xs ys : Numeral b d o)
    → Set
xs ≈ ys = ⟦ xs ⟧ ≡ ⟦ ys ⟧
```

Two numerals considered equal by `_≈_` whenever they evaluate to the same value. On the other hand, two numerals are considered equal by `_≡_`

only when they are in the same *canonical form*. However, unlike ℕ, canonical forms for representing numbers may not exist for systems of `Numeral` that are *redundant*. In other words, `_≋_` is extensional while `_≡_` is intensional.

Also, the index o of `Numeral` is fixed to 0. We will explain why the domain of discourse is limited that way in the later section.

Meaning of Variables

Terms and predicates can have free variables, but variables are given meanings only when they are substituted.

```
var : ∀ {n} → (i : Fin n) → Term n
```

`var` have to pick a number i of `Fin n` as its name, because values of all variables are stored in a `Vec n` (a list of length n.) Upon substitution, we pick the ith element of the list as its value.

Interpreting Terms

The interpreter of the syntax for terms is defined as follows:

```
⟦_⟧T : ∀ {n}
    → Term n
    → (sig : Signature)
    → Vec (carrier sig) n
    → carrier sig
⟦ var i ⟧T _ env = lookup i env
⟦ term₁ +T term₂ ⟧T (sig A _⊕_ _≈_) env
    = ⟦ term₁ ⟧T (sig A _⊕_ _≈_) env ⊕ ⟦ term₂ ⟧T (sig A _⊕_ _≈_) env
```

`carrier sig` evaluates to the carrier of the signature, the domain of discours. For example, `carrier ℕ-sig` evaluates to ℕ. In addition to the term, `⟦_⟧T` also takes a signature, meanings of all variables, and returns an element of the carrier.

`var` `lookup` retrieves the ith value for the variable from `env`. This is where the variable substitution happens.

`+T` The semantics of both terms are interpreted recursively, and then computed with `_⊕_`, the addition function supplied by the signature.

Interpreting Predicates

The interpreter of the syntax for predicates is defined as follows:

```
⟦_⟧P : ∀ {n}
    → Predicate n
    → (sig : Signature)
    → Env (carrier sig) n
    → Set
⟦ t₁ ≋P t₂ ⟧P (sig carrier _⊕_ _≈_) env
    = ⟦ t₁ ⟧T (sig carrier _⊕_ _≈_) env ≈ ⟦ t₂ ⟧T (sig carrier _⊕_ _≈_) env
⟦ p →P q   ⟧P signature env = ⟦ p ⟧P signature env → ⟦ q ⟧P signature env
⟦ ∀P pred  ⟧P signature env = ∀ x → ⟦ pred ⟧P signature (x ∷ env)
```

Similar to that of interpreting terms, except that `⟦_⟧P` returns a *type*.

`_≋P_` The semantics of both terms are interpreted using `⟦_⟧T` whereas `_≈_` is the equality connective supplied by the signature.

`_→P_` The semantics of both predicates are interpreted recursively and connected with the native implication `_→_` provided by Agda.

`∀P` A new variable is introduced and then added to the list of meanings of variables for the interpreted predicate to consume.

Examples

We can have both semantics of `≋-trans` by supplying different signatures.

```
≋-trans-ℕ : Set
≋-trans-ℕ = ⟦ ≋-trans ⟧P ℕ-sig []

≋-trans-Numeral : (b d : ℕ) → True (Continuous? b d 0) → Set
≋-trans-Numeral b d prop
    = ⟦ ≋-trans ⟧P (Numeral-sig b d prop) []
```

`≋-trans-ℕ` evaluates to:

```
(x y z : ℕ) → (z ≡ y → y ≡ x) → z ≡ x
```

`≋-trans-Numeral 10 10 _` evaluates to:

```
(x y z : Numeral 10 10 0)
    → (⟦ z ≡ ⟦ y ⟧ → ⟦ y ⟧ ≡ ⟦ x ⟧)
    → ⟦ z ⟧ ≡ ⟦ x ⟧
```

6.3 Converting between Semantics of ℕ and `Numeral`

6.3.1 Terms

We start from proving that both semantics of terms are equal after evaluation.

```
toℕ-term-homo : ∀ {b d n}
    → (cont : True (Continuous? b d 0))
    → (t : Term n)
    → (env : Vec (Numeral b d 0) n)
    → ⟦ t ⟧T ℕ-sig (map ⟦_⟧ env) ≡ ⟦ ⟦ t ⟧T (Numeral-sig b d cont) env ⟧
toℕ-term-homo         cont (var i)   env = lookup-map ⟦_⟧ env i
toℕ-term-homo {b} {d} cont (t₁ +T t₂) env
    rewrite toℕ-term-homo cont t₁ env
          | toℕ-term-homo cont t₂ env
    = sym (toℕ-+-homo cont
        (⟦ t₁ ⟧T (Numeral-sig b d cont) env)
        (⟦ t₂ ⟧T (Numeral-sig b d cont) env))
```

We can see how ⟦_⟧ *preserves* the interpretation of syntax. `lookup-map` is a lemma which states that:

```
lookup-map ⟦_⟧ env i : lookup i (map ⟦_⟧ xs) ≡ ⟦ lookup i xs ⟧
```

The `rewrite` construction takes a proof of type `lhs ≡ rhs` and pattern match on the proof so that `lhs` and `rhs` may be unified as one. If ⟦_⟧ preserves addition functions of both semantics then it will preserve the interpretation of `+T`, too.

6.3.2 Predicates

Our wish is to translate semantics of predicates from ℕ to `Numeral`. However, because of the contravariant nature of the input type of function types `_→_` that are introduced when interpreting implication connectives, we also need to know how to translate semantics the other way around, from `Numeral` to ℕ.

These two interpreters are defined simultaneously with the `mutual` construction.

From ℕ to `Numeral`

```
toℕ-pred-ℕ⇒Numeral : ∀ {b d n}
    → (cont : True (Continuous? b (suc d) 0))
    → (pred : Predicate n)
    → (env : Vec (Numeral b (suc d) 0) n)
    → ⟦ pred ⟧P ℕ-sig (map ⟦_⟧ env)
    → ⟦ pred ⟧P (Numeral-sig b (suc d) cont) env
toℕ-pred-ℕ⇒Numeral {b} {d} cont (t₁ ≈P t₂) env sem-ℕ =
    begin
        ⟦ ⟦ t₁ ⟧T (Numeral-sig b (suc d) cont) env ⟧
    ≡⟨ sym (toℕ-term-homo cont t₁ env) ⟩
        ⟦ t₁ ⟧T ℕ-sig (map ⟦_⟧ env)
    ≡⟨ sem-ℕ ⟩
        ⟦ t₂ ⟧T ℕ-sig (map ⟦_⟧ env)
    ≡⟨ toℕ-term-homo cont t₂ env ⟩
        ⟦ ⟦ t₂ ⟧T (Numeral-sig b (suc d) cont) env ⟧
    ∎
toℕ-pred-ℕ⇒Numeral cont (p →P q) env sem-ℕ ⟦p⟧P-Numeral =
    toℕ-pred-ℕ⇒Numeral cont q env
        (sem-ℕ (toℕ-pred-Numeral⇒ℕ cont p env ⟦p⟧P-Numeral))
toℕ-pred-ℕ⇒Numeral cont (∀P pred) env sem-ℕ x =
    toℕ-pred-ℕ⇒Numeral cont pred (x ∷ env) (sem-ℕ ⟦ x ⟧)
```

`_≈P_` The reasoning can be replaced with the `rewrite` construction. However, we are keeping it for conciseness. `sem-ℕ` is the original semantics of ℕ stating that the semantics of `t₁` and `t₂` are equal.

`_→P_` Since `_→P_` is interpreted as an implication (hence a function type) `_→_` on `Numeral`, we are given `⟦p⟧P-Numeral` as the antecedent of the implication. However, what we have is `sem-ℕ`, an implication on the semantics of ℕ. This where we need the result of translation from the opposite direction.

$$\begin{tikzcd}
\mathtt{Numeral} \arrow[r, "\mathtt{to\mathbb{N}\text{-}pred\text{-}Numeral{\Rightarrow}\mathbb{N}}"] & \mathbb{N} \arrow[d, "\mathtt{sem\text{-}\mathbb{N}}"] \\
\mathtt{Numeral} & \mathbb{N} \arrow[l, "\mathtt{to\mathbb{N}\text{-}pred\text{-}\mathbb{N}{\Rightarrow}Numeral}"']
\end{tikzcd}$$

`∀P` `∀P` is interpreted as a function with an argument `x` of type `Numeral`. We can get a semantics of `Numeral` by recursively calling itself with the list of meanings of variables extended by the value of `x`.

$$\begin{tikzcd}
\texttt{Numeral} \arrow[r, "\llbracket\_\rrbracket"] & \mathbb{N} \arrow[d, "\texttt{sem-}\mathbb{N}"] \\
\texttt{Numeral} & \mathbb{N} \arrow[l, "\texttt{to}\mathbb{N}\texttt{-pred-}\mathbb{N}\Rightarrow\texttt{Numeral}"']
\end{tikzcd}$$

From `Numeral` to ℕ

```
toℕ-pred-Numeral⇒ℕ : ∀ {b d n}
    → (cont : True (Continuous? b (suc d) 0))
    → (pred : Predicate n)
    → (env : Vec (Numeral b (suc d) 0) n)
    → ⟦ pred ⟧P (Numeral-sig b (suc d) cont) env
    → ⟦ pred ⟧P ℕ-sig (map ⟦_⟧ env)
toℕ-pred-Numeral⇒ℕ {b} {d} cont (t₁ ≈P t₂) env sem-Num =
    begin
        ⟦ t₁ ⟧T ℕ-sig (map ⟦_⟧ env)
    ≡⟨ toℕ-term-homo cont t₁ env ⟩
        ⟦ ⟦ t₁ ⟧T (Numeral-sig b (suc d) cont) env ⟧
    ≡⟨ sem-Num ⟩
        ⟦ ⟦ t₂ ⟧T (Numeral-sig b (suc d) cont) env ⟧
    ≡⟨ sym (toℕ-term-homo cont t₂ env) ⟩
        ⟦ t₂ ⟧T ℕ-sig (map ⟦_⟧ env)
    ∎
toℕ-pred-Numeral⇒ℕ cont (p →P q) env sem-Num ⟦p⟧P
    = toℕ-pred-Numeral⇒ℕ cont q env
        (sem-Num
            (toℕ-pred-ℕ⇒Numeral cont p env ⟦p⟧P))
toℕ-pred-Numeral⇒ℕ {b} {d} cont (∀P pred) env sem-Num n
    with n ≟ ⟦ fromℕ {cont = cont} n z≤n ⟧
toℕ-pred-Numeral⇒ℕ {b} {d} cont (∀P pred) env sem-Num n
    | yes eq
    rewrite eq
    = toℕ-pred-Numeral⇒ℕ
        cont pred
        (fromℕ {cont = cont} n _ ∷ env)
        (sem-Num (fromℕ {cont = cont} n _))
toℕ-pred-Numeral⇒ℕ {b} {d} cont (∀P pred) env sem-Num n
    | no ¬eq
    = contradiction (sym (fromℕ-toℕ cont n _)) ¬eq
```

The translation of semantics from `Numeral` to ℕ is mostly the same as in `toℕ-pred-ℕ⇒Numeral`, except for the case of `∀P`.

∀P Suppose there exists a numeral `xs` such that `n ≡ ⟦ xs ⟧`. Because `⟦_⟧` may not be injective, we do not have a left inverse of `⟦_⟧`. Therefore we cannot assert that `xs ≡ fromℕ ⟦ xs ⟧ _`. However, since we do not care about the canonical forms of numerals, we can convince Agda that `xs ≋ fromℕ ⟦ xs ⟧ _` is good enough. To do this, we force evaluate the decidable equality between `fromℕ {cont = cont} n z≤n` and `n` and unify them using the `rewrite` construction.

Examples

Suppose we already have a proof for the commutative property of addition on ℕ: `+-comm-ℕ : ∀ a b → a + b ≡ b + a`. To convert `+-comm-ℕ` to a proof that works on `Numeral` as well, the first step is to describe the predicate with our syntax.

```
+-comm-Predicate : Predicate 0
+-comm-Predicate = ∀P (∀P ((var (# 1) +T var (# 0)) ≋P (var (# 0) +T var (# 1))))
```

Hand it to `toℕ-pred-ℕ⇒Numeral` and we are done.

```
+-comm-Num : ∀ {b d}
    → {cont : True (Continuous? b (suc d) 0)}
    → (xs ys : Numeral b (suc d) 0)
    → (_+_ {cont = cont} xs ys) ≋ (_+_ {cont = cont} ys xs)
+-comm-Num {cont = cont} = toℕ-pred-ℕ⇒Numeral cont +-comm-P [] +-comm-ℕ
```

Bibliography

[1] P. Benacerraf. What numbers could not be. *The Philosophical Review*, 74(1):47–73, 1965.

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

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

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

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

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

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

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

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

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

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

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

[13] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM, 1987.