Generalizing Positional Numeral Systems

Luā Tîng-Giān

November 30, 2016

Abstract

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

1 Introduction

1.1 Positional numeral systems

A numeral system is a writing system for expressing numbers, and humans have invented various kinds of numeral systems throughout history. Most of the systems we are using today are positional notations[2] because they can express infinite numbers with just a finite set of symbols called **digits**.

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

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

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

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

1.1.1 Symtems of different bases

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

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

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

1.1.2 Digits of different ranges

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

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

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

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

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

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

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

1.1.3 Numerical representation

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

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

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

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

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

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

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

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

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

[insert some images here]

The strong analogy between positional numeral systems and certain data structures suggests that, numeral systems can serve as templates for designing containers. Such data structures are called **Numerical Representations**[7] [1].

1.1.4 Redundancy

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

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

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

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

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

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

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

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

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

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

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

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

There are more than one way to represent a number in this system.

Number	Notation
2	0

Such a numeral system is said to be redundant. In fact, systems that allow 0 as one of the digits must be redundant, since we can always take a number and add leading zeros without changing it's value.

Data structures modeled after redundant numeral systems have some interesting properties. Self-balancing containers

1.1.5 From numeral systems to numbers

Outline The remainder of the thesis is organized as follows.

2 A gental introduction to dependently typed programming in Agda

There are already plenty of tutorials and introductions of Agda[6][5][3]. We will nonetheless compile a simple and self-contained tutorial from the materials cited above, covering the part (and only the part) we need in this work.

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

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

2.1 Some basics

Agda is a dependently typed functional programming language based on **Martin-Löf type theory** [4]. The first version of Agda was originally developed by Catarina Coquand at Chalmers University of Technology, the current version (Agda2) is a completely rewrite by Ulf Norell during his PhD at Chalmers.

Agda is also an interactive theorem prover. Because proving theorems involves a lot of conversations between the programmer and the type checker. Just like programming, the process is incremental. It is difficult, if not impossible, to develop and prove a theorom at one stroke. Agda allows us to leave some "holes" in a program, refine them gradually, and complete your proofs "hole by hole".

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

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

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

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

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

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

This is basically what pragramming and proving things looks like in Agda.

2.2 Simply typed programming in Agda

In the beginning there was nothing 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?

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

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

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

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

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

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

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

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

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

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

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

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

Addition on ℕ can be defined as a recursive function.

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

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

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

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

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

id-ℕ : ℕ → ℕ
id-ℕ x = x
```

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

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

In Agda, function types are denoted as:

```
A → B
```

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

```
(x : A) → B x
```

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".

In this instance, if `A` happens to be `Set`, the type of all small types, and the result type happens to be solely `x`:

```
(x : Set) → x
```

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

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

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

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

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 can 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 when being applied.

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

Any arguments can be made implicit, but it doesn't imply that values of implicit arguments can always be inferred or derived from context. We can always make them implicit arguments explicit when being applying:

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

Or when they are relevant to the definition:

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

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

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

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

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

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

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

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

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

```
[] : {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`.

Indexed Datatypes Let's look at something more interesting, a datatype that is similar to `List`, but more powerful, in that it can tell you not only the type of its element, but also its length.

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

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

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

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

We say that `Vec` is *parameterized* by a type of `Set` and is *indexed* by values of ℕ. But it's not obvious how indices are different from parameters.

Parameters are *parametric* (no pun intended), 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 won't reveal anything useful about their whereabouts. Because they are *uniform* across all constructors, you can always replace the value of an 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 index. In that case, pattern matching on indices may yield important information about their constructors.

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

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

```
head : ∀ {A n} → Vec A (suc n) → A
```

```
head (x :: xs) = x
```

Note that parameters could be thought as a degenerate case of indices whose distribution of values are uniform across all constructors.

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

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

We are stuck here, because the result of `p x` is only available in runtime. Fortunately, with abstraction allows us to pattern match on 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 There are two special types, the *unit type* and the *bottom type*, denoted as `⊤` and `⊥`, which as 1 and 0 values respectively.

```
data ⊤ : Set where
    tt : ⊤

data ⊥ : Set where
```

Where `⊤` has only a value, `tt`, and `⊥` has none.

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

Say, we want to define a `head` on `List` that is statically checked and won't crash on any inputs. So we need to make sure that all inputs are not empty. But unlike on `Vec`, without indices, we need other means of guaranteeing that the list is not empty. Naturally we would come up with a predicate like this:

```
non-empty : ∀ {A} → List A → Bool
non-empty []       = false
```

```
non-empty (x ∷ xs) = true
```

The problem is that we can only know if a list is null in runtime. However, with `⊤` and `⊥`, we could instead define a predicate like this:

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

Notice that now this function is returning a type, and thus `head` can be defined as:

```
head : ∀ {A} → (xs : List A) → non-empty xs → A
head []       proof = ?0
head (x ∷ xs) proof = ?1
```

3 Num : a representation for positional numeral systems

3.1 Digits

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

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

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

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

Our digits are simply `Fin`.

```
Digit : ℕ → Set
Digit = Fin
```

Binary digits for example can be defined as:

```
Binary : Set
Binary = Digit 2

零 : Binary
零 = zero

一 : Binary
一 = suc zero
```

But these digits bears no meaning without converting to ℕ. Here `d`

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

3.2 Bases

3.3 Offsets

3.4 Number of digits

4 Properties of Num

4.1 Maximum

4.2 Bounded

4.3 Bounded

4.4 Views

5 Conclusions

References

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

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

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

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

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

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

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