Generalizing Positional Numeral Systems

Tîng-Giān Luā

November 23, 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 to model many of these representations, and investigate some of their properties and relationship with the classical unary representation of natural number.

1 Introduction

1.1 Numbers

The *numbers* we will be constructing in this work are the *natural numbers*. And before

The *numbers* we are constructing in this work are the *natural numbers*. We learned how to count with these numbers when we are little. But what are numbers, really?

Recursive Paul Benacerraf once argued[1] that, there are two kinds of *counting* which corresponds 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 establish 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 hands, is to generate a sequence of notation, that could go as far as we need. And it seems that one can only learn how to count intransitively first, before knowning how to count transitively, but not vice versa.

He further discussed that the numbers must be at least recursive

Numbers are recursive

Internal definition is immaterial

1.2 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 because they can express **infinite** numbers with just a **finite** set of symbols.

While the most ubiquitous positional notation used in our daily lives is the base-10 (decimal) system, and binary (base-2) numbers can be found in almost all digital circuits. Some computer scientists and mathematicians seem to be more comfortable with unary (base-1) numbers, which is no harder than counting pebbles.

Natural numbers (à la Peano) are defined in a way that is isomorphic to unary numbers. Statements established on such numbers can be proven using mathematical induction, an essential proving technique.

People have implemented and proven a great deal of functions and properties on these unary numbers because they are easy to work with. But the simplicity comes at a cost, the time complexity of functions defined on unary numbers is significantly higher than those defined on systems of other bases, say, binary numbers. Hence, it's not practical in doing heavy calculations.

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[5][4][2]. 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** [3]. 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.

```
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.

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

```
_+_ : 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 $\mathbb{N}$. 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. Now `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.

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

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

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

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

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