Generalizing Positional Numeral Systems

Tîng-Giān Luā

November 20, 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 What are numbers?

1.2 Positional numeral systems

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[4][3][1]. 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** [2]. 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.

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 saw 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
```

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:

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

∀-Quantification There's another syntax sugar for type expressions. When the type of some value can be inferred, we could replace `(A : _)` with `∀ A` and `{A : _}` with `∀ {A}`.

With abstraction

Absurd pattern

2.3 Dependently typed programming in Agda

3 Num : a representation for positional numeral systems

3.1 Bases

3.2 Offsets

3.3 Number of digits

4 Properties of Num

4.1 Maximum

4.2 Bounded

4.3 Bounded

4.4 Views

5 Conclusions

References

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

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

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

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