國立交通大學

資訊科學與工程研究所

碩士論文

一般化位值計數系統

Generalizing Positional Numeral Systems

研　　究　　生 ： 賴廷彥

指　導　教　授 ： 穆信成、楊武　教授

中華民國 106 年 1 月

一般化位值計數系統

Generalizing Positional Numeral Systems

研究生	：	賴廷彥	Student	:	Luā Tîng-Giān
指導教授	：	穆信成、楊武	Advisor	:	Shin-Cheng Mu, Wuu Yang

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis Submitted to Institute of Computer Science and Engineering College of Computer Science National Chiao Tung University in Partial Fulfillment of the Requirements for the Degree of Master in Computer and Information Science

July 2016

Luā Tîng-Giān, Taiwan

中華民國 105 年 7 月

一般化位值計數系統

學生 ： 賴廷彥
指導教授 ： 穆信成、楊武 教授

國立交通大學資訊科學與工程研究所碩士班

摘　　要

日常生活中充滿了數值，而位值計數系統是最常見的數值表示方式。在這篇論文中我們探討並研究各種計數系統的有趣性質與應用，並且在 Agda 裡去建構並且驗證我們發展的一種一般化位值計數系統的方法。我們不只會去探索我們的計數系統與皮亞諾公理的自然數之間的關係，也會發展一套方法去自動轉換彼此之間的定理與證明。

關鍵字：構造性數學、計數系統、定理證明、Agda、一階邏輯

Generalizing Positional Numeral Systems

Student : Luā Tîng-Giān
Advisor : Prof. Shin-Cheng Mu, Wuu Yang

Institute of Computer Science and Engineering
National Chiao Tung University

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 will investigate interesting properties and applications of some variations of positional numeral systems and construct a generalized positional numeral system to model and formalize them in Agda. We will not only examine properties of our representation and its relationship with the classical unary representation of natural numbers but also demonstrate how to translate propositions and proofs between them.

Keywords: Constructive mathematics, Numeral systems, Theorem proving, Agda, First-order logic

Contents

1 Introduction **1**

1.1 Positional Numeral Systems . . . 1

1.1.1 Digits . . . 1

1.1.2 Syntax and Semantics . . . 2

1.1.3 Evaluating Numerals . . . 3

1.2 Unary Numbers and Peano Numbers . . . 3

1.3 Binary Numerals in Digital Circults . . . 4

1.4 Numerical representation . . . 5

1.4.1 The correspondence . . . 7

1.5 Motivation and Outline . . . 9

2 Generalizations **10**

2.1 Base . . . 10

2.2 Offset . . . 10

2.3 Number of Digits . . . 12

2.4 Relations with Natural Numbers . . . 13

3 A gentle introduction to dependently typed programming in Agda **14**

3.1 Some basics . . . 14

3.2 Simply typed programming in Agda . . . 15

List of Figures

1 Example of long additions 4

2 Structure of lists and unary numerals 5

3 Structure of binomial heaps and binary numerals 6

4 A Perfect leaf tree of rank 3 7

5 A random access list with 13 elements 7

6 A random access list with 32 elements 8

7 A 1-2 random access list with 32 elements 8

List of Tables

1 Various kinds of numeral systems 1

2 Assignements of digits of different numeral systems 1

3 Assignements of hexadecimal digits 2

4 Semantics of the notation “11” in different contexts 2

5 Numeral systems of different bases 3

6 Numeral systems of different bases (with unary numerals) 4

7 Correspondences of Numerical Representations 8

8 1-2 binary numeral system 11

9 Summary of indices of common numeral systems 13

Chapter 1

Introduction

1.1 Positional Numeral Systems

A numeral system is a writing system for expressing numbers, and humans have invented various kinds of numeral systems throughout history. Take the number “2016” for example:

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

Table 1: Various kinds of numeral systems

Even so, 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**.

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**

Table 2: Assignements of digits of different numeral systems

We place a bar above a digit to indicate its assignment. For instance, these are the assignments of hexadecimal digits.

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

Table 3: Assignements of hexadecimal digits

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 contexts.

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

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

Table 4: Semantics of the notation “11” in different contexts

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**

Table 5: Numeral systems of different bases

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**

Table 6: Numeral systems of different bases (with unary numerals)

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.

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

Figure 1: Example of long additions

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.

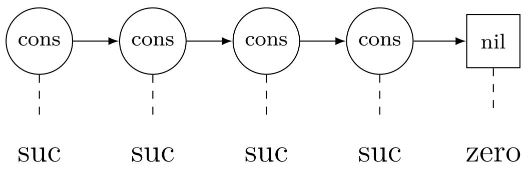

Figure 2: Structure of lists and unary numerals

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

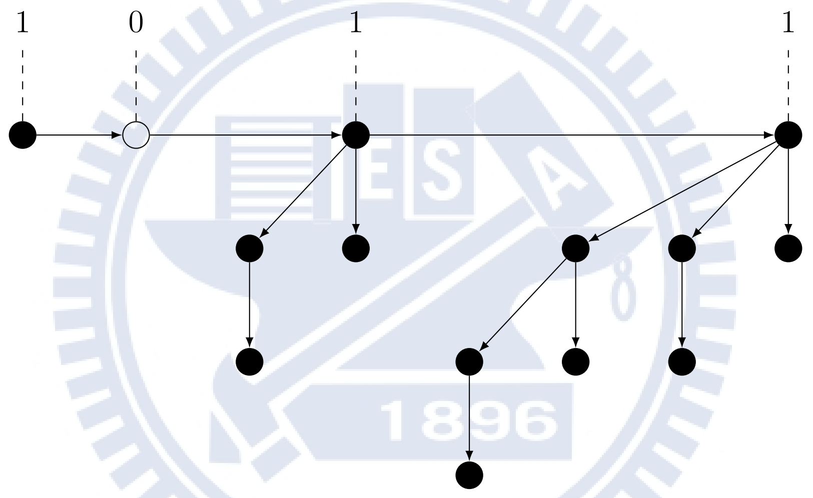

Figure 3: Structure of binomial heaps and binary numerals

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.

[1]Nodes that contain elements are painted black.

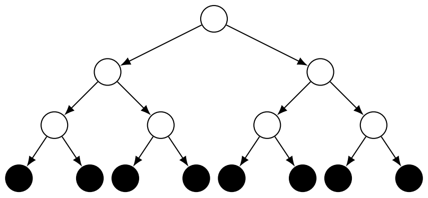

Figure 4: A Perfect leaf tree of rank 3

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.

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.

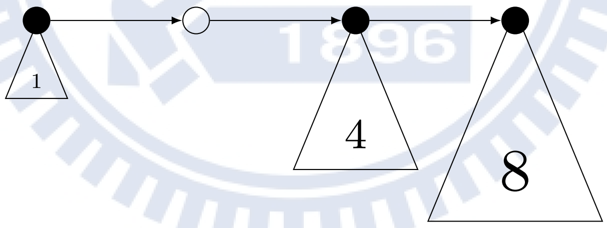

Figure 5: A random access list with 13 elements

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**[7] [1].

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

Table 7: Correspondences of Numerical Representations

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:

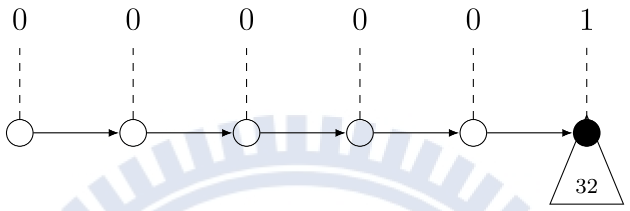

Figure 6: A random access list with 32 elements

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.

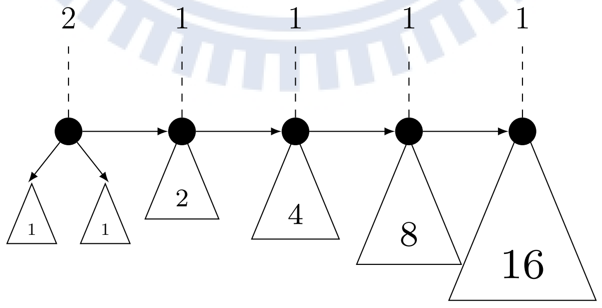

Figure 7: A 1-2 random access list with 32 elements

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[1] 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 ?? introduces *equational reasoning* and relevant properties of natural numbers exercised in the coming chapters.

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

Chapter ?? discusses some related topics and 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

$$0, 1, ..., base - 1$$

to

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

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

Table 8: 1-2 binary numeral system

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., $\mathit{offset} > 0$. 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.

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

2.3 Number of Digits

The binary numeral system running in circuits looks different from what we have in hand. Surprisingly, these binary numbers can fit into our representation with just a tweak. If we allow a system to have more digits, then a fixed-precision binary number can be regarded as a single digit! To illustrate this, a 32-bit binary number (*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:

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

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

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

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 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

Table 9: Summary of indices of common numeral systems

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 **??**.

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

Chapter 3

A gentle 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 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.

Remark 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*[4], hence the Curry-Howard correspondence[8], 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.

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

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

The decimal number “4” is represented as `suc (suc (suc (suc zero)))`. Agda also accepts decimal literals if the datatype ℕ complies with certain language pragma.

Addition on ℕ can be defined as a recursive function.

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

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

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

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

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

id-ℕ : ℕ → ℕ
id-ℕ x = x
```

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

A dependent type is a type whose definition may depend on a value. A dependent function is a function whose type may depend on a value of its arguments.

In Agda, function types are denoted as:

```
A → B
```

where `A` is the type of domain and `B` is the type of codomain. To let `B` depends on the value of `A`, the value has to *named*. In Agda we write:

```
(x : A) → B x
```

The value of `A` is named `x` and then fed to `B`. As a matter of fact, `A → B` is just a syntax sugar for `(_ : A) → B` with the name of the value being irrelevant. The underline _ here means "I don't bother naming it".

Back to our identity function, if `A` happens to be `Set`, the type of all small types, and the result type happens to be solely `x`:

```
(x : Set) → x
```

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

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

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

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

Fortunately, Agda supports *implicit arguments*, a syntax sugar that could save us the trouble of having to spell them out. Implicit arguments are enclosed in curly brackets in the type expression. We are free to dispense with these arguments when their values are irrelevant to the definition.

```
id : {A : Set} → A → A
id x = x
```

Or when the type checker can figure them out on function application.

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

Any arguments can be made implicit, but it does not imply that values of implicit arguments can always be inferred or derived from context. We can always make them implicit arguments explicit on application:

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

Or when they are relevant to the definition:

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

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

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

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

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

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

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

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

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

```
infixr 5 _∷_

[] : {A : Set} → List A
_∷_ : {A : Set} → A → List A → List A
List : Set → Set
```

where `A` can be anything, even `List (List (List Bool))`, as long as it is of type `Set`. `infixr` specifies the precedence of the operator `_∷_`.

Indexed Datatypes `Vec` is a datatype that is similar to `List`, but more powerful, in that it encodes not only the type of its element but also its length.

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

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

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

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

We say that `Vec` is *parameterized* by a type of `Set` and is *indexed* by values of `ℕ`. We distinguish indices from parameters. However, it is not obvious how they are different by looking at the declaration.

Parameters are *parametric*, in the sense that, they have no effect on the "shape" of a datatype. The choice of parameters only effects which kind of values are placed there. Pattern matching on parameters does not reveal any insights into their whereabouts. Because they are *uniform* across all constructors, one can always replace the value of a parameter with another one of the same type.

On the other hand, indices may affect which inhabitants are allowed in the datatype. Different constructors may have different indices. In that case, pattern matching on indices may yield relevant information about their constructors.

For example, given a term whose type is `Vec Bool zero`, then we are certain that the constructor must be `[]`, and if the type is `Vec Bool (suc n)` for some `n`, then the constructor must be `_∷_`.

We could, for instance, define a `head` function that cannot crash.

```
head : ∀ {A n} → Vec A (suc n) → A
head (x ∷ xs) = x
```

As a side note, parameters can be thought as a degenerate case of indices whose distribution of values is uniform across all constructors.

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

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

We are stuck here because the result of `p x` is only available at runtime. Fortunately, with abstraction allows us to pattern match on the result of an intermediate computation by adding the result as an extra argument on the left-hand side:

```
filter : ∀ {A} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) with f x
filter p (x ∷ xs) | true  = x ∷ filter p xs
filter p (x ∷ xs) | false = filter p xs
```

Absurd patterns The *unit type*, or *top*, is a datatype inhabited by exactly one value, denoted `tt`.

```
data ⊤ : Set where
    tt : ⊤
```

The *empty type*, or *bottom*, on the other hand, is a datatype that is inhabited by nothing at all.

```
data ⊥ : Set where
```

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

Say, we want to define a safe `head` on `List` that does not crash on any inputs. Naturally, in a language like Haskell, we would come up with a predicate like this to filter out empty lists `[]` before passing them to `head`.

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

The predicate only works at runtime. It is impossible for the type checker to determine whether the input is empty or not at compile time.

However, things are quite different quite in Agda. With *top* and *bottom*, we could do some tricks on the predicate, making it returns a *Set*, rather than a *Bool*!

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

Notice that now this predicate is returning a type. So we can use it in the type expression. `head` can thus be defined as:

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

In the `(x ∷ xs)` case, the argument `proof` would have type `⊤`, and the right-hand side is simply `x`; in the `[]` case, the argument `proof` would have type `⊥`, but what should be returned at the right-hand side?

It turns out that, the right-hand side of the `[]` case would be the least thing to worry about because it is completely impossible to have such a case. Recall that `⊥` has no inhabitants, so if a case has an argument of that type, it is too good to be true.

Type inhabitance is, in general, an undecidable problem. However, when pattern matching on a type that is obviously empty (such as `⊥`), Agda allows us to drop the right-hand side and eliminate the argument with `()`.

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

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

Propositions as types, proofs as programs The previous paragraphs are mostly about the *programming* aspect of the language, but there is another aspect to it. Recall the Curry−Howard correspondence, propositions are types and proofs are programs. A proof exists for a proposition the way that a value inhabits a type.

`non-empty xs` is a type, but it can also be thought of as a proposition stating that `xs` is not empty. When `non-empty xs` evaluates to `⊥`, no value inhabits `⊥`, which means no proof exists for the proposition `⊥`; when `non-empty xs` evaluates to `⊤`, `tt` inhabits `⊥`, a trivial proof exists for the proposition `⊤`.

In intuitionistic logic, a proposition is considered to be "true" when it is inhabited by a proof, and considered to be "false" when there exists no proof. 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]

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

```
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-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 [9]. 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.

Bibliography

[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.

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

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