

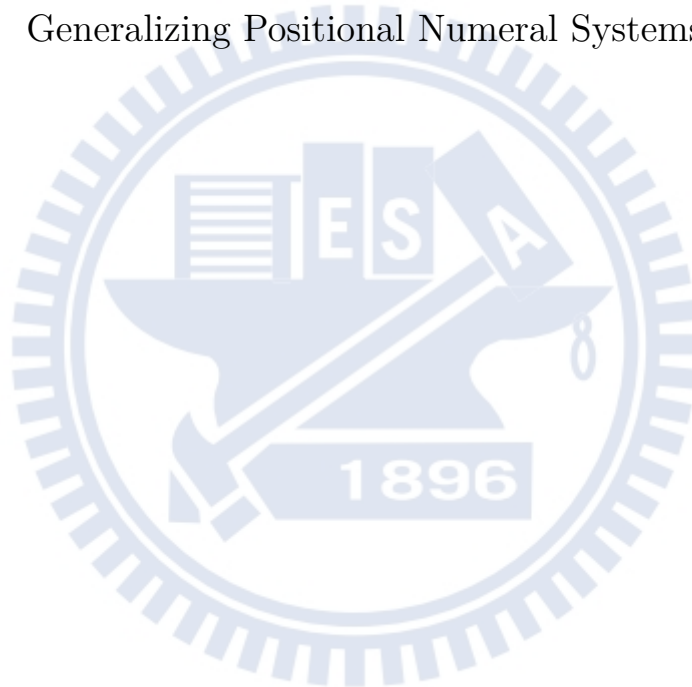
國立交通大學

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 Tīng-Giān

指導教授：穆信成、楊武

Advisor：Shin-Cheng Mu, Wu 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 Tīng-Giān, Taiwan

中華民國 105 年 7 月

一般化位值計數系統

學生：賴廷彥

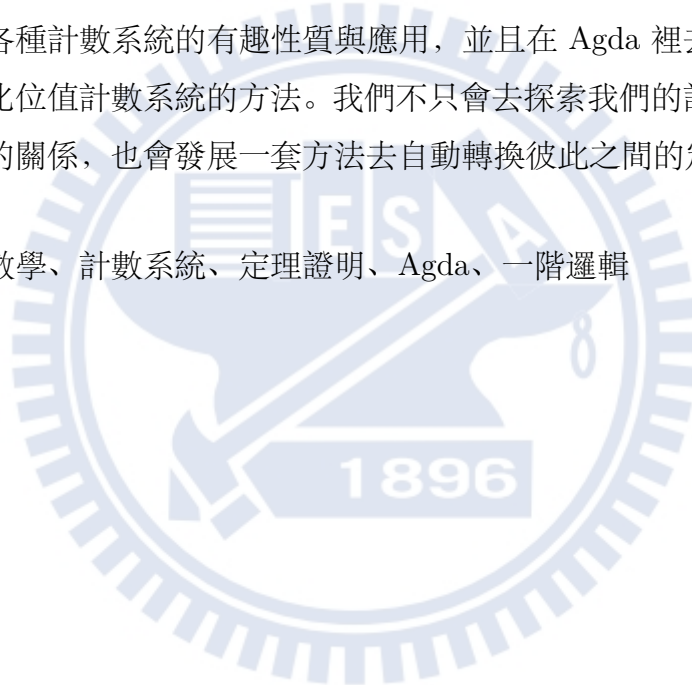
指導教授：穆信成、楊武 教授

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：構造型數學、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, Wu 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	Positional Numeral Systems	
1.1.1	Digits	
1.1.2	Syntax and Semantics	
1.1.3	Evaluating Numerals	
1.2	Unary Numbers and Peano Numbers	
1.3	Binary Numerals in Digital Circuits	
1.4	Numerical representation	
1.4.1	The correspondence	
1.5	Motivation and Outline	

2 Background

3 Design

4 Implementation

5 Evaluation

6 Related Work

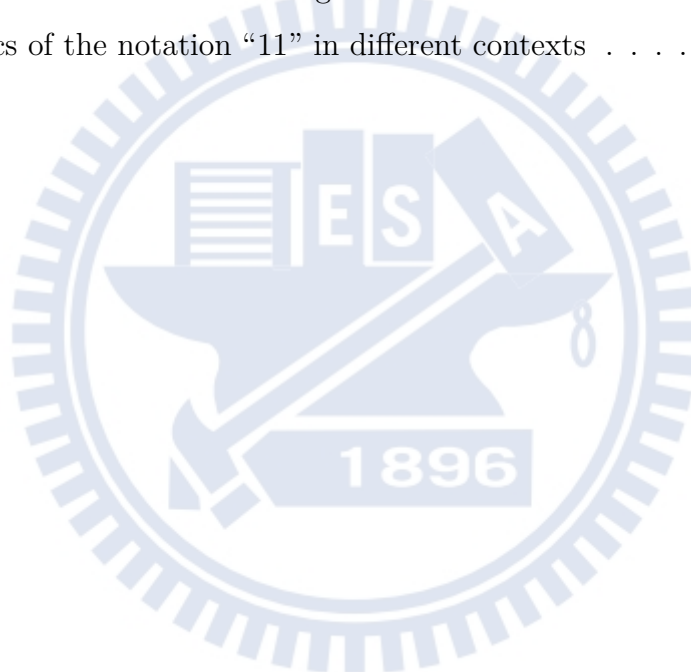
7 Conclusion

List of Figures



List of Tables

1	Various kinds of numeral systems
2	Assignments of digits of different numeral systems
3	Assignments of hexadecimal digits
4	Semantics of the notation “11” in different contexts



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	

Table 1: Various kinds of numeral systems

Even so, most of the systems we are using today are positional notations[[knuth1998art](#)] 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: Assignments 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

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 $\llbracket 2016 \rrbracket_{10}$ to emphasize its role as the evaluation function.

To evaluate a notation of a certain base:

$$\llbracket d_0 d_1 d_2 \dots d_n \rrbracket_{base} = \bar{d}_0 \times base^0 + \bar{d}_1 \times base^1 + \bar{d}_2 \times base^2 + \dots + \bar{d}_n \times base^n$$

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.

$$\llbracket 1111 \rrbracket_1 \cong \overbrace{\text{suc} (\text{suc} (\text{suc} (\text{suc} + \text{zero})))}^4$$

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 ??.

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 Circuits

Suppose we are asked to anwser the questions below.

$$\begin{array}{r}
 253298123 \\
 + 347844235 \\
 \hline
 ?
 \end{array}$$

$$\begin{array}{r}
 123 \\
 + 34 \\
 \hline
 ?
 \end{array}$$

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 $O(\lg n)$ 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*.

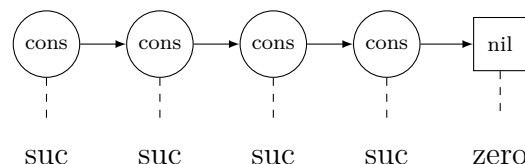
Problem Judging from the time complexity of operations, the binary numerals running in digital circuits is certainly different from the ordinary binary numerals we have known.

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.

1	data Nat = Zero	1	data List a = Nil
2	Suc Nat	2	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.

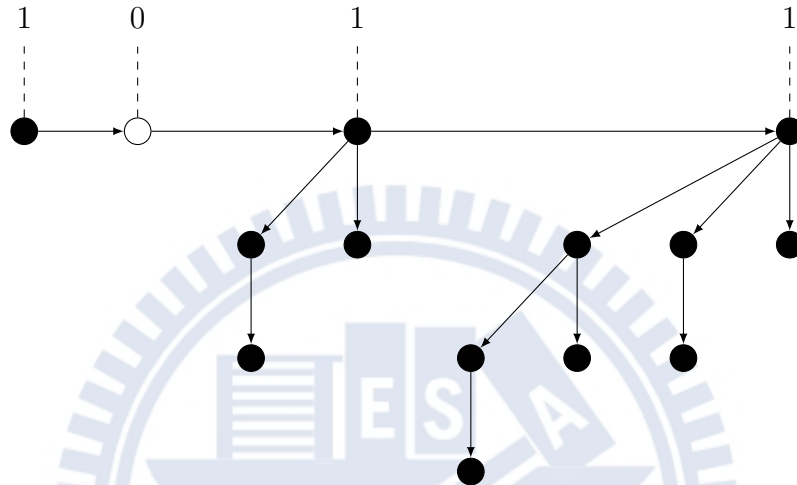


Now let's compare addition on unary numbers and merge (append) on lists:

1	add : Nat → Nat → Nat	1	append : List a → List a → List a
2	add Zero y = y	2	append Nil ys = ys
3	add (Suc x) y =	3	append (Cons x xs) ys =
4	Suc (add x y)	4	Cons x (append xs ys)

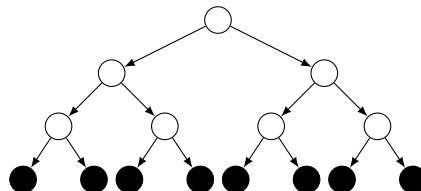
Aside from having virtually identical implementations, operations on unary numbers and lists both have the same time complexity. Incrementing a unary number takes $O(1)$, inserting an element into a list also takes $O(1)$; adding two unary numbers takes $O(n)$, appending a list to another also takes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

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 *rank*s 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.

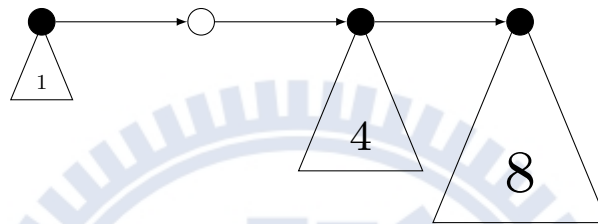


These building blocks can have different ranks. A binary leaf tree of rank n , for instance, would contain 2^n elements. The data structures we have addressed so far are all composed of a series of building blocks that are ordered by their ranks.

¹Nodes that contain elements are painted black.

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 $O(\lg n)$ as a random access list would have at most $O(\lg n)$ building blocks, and the tallest perfect leaf tree takes at most $O(\lg \lg n)$ to traverse.



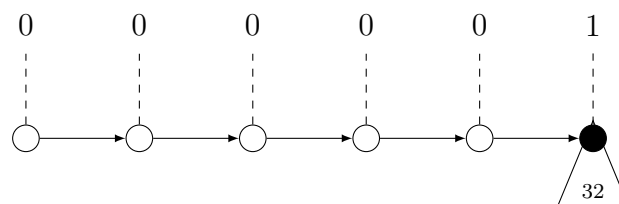
Similar to that of binomial heaps, random access lists also have spines. Also, treating building blocks as digits also yields binary numerals of the container's size.

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**[okasaki1996purely] [hinze1998numerical].

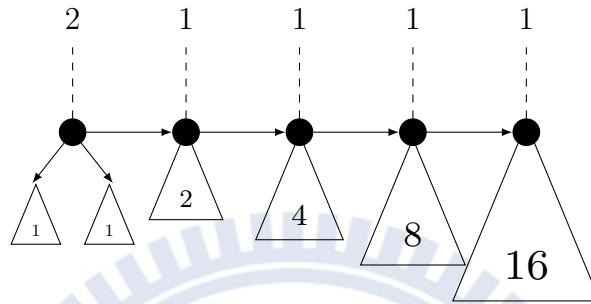
a container of size n	corresponds to	a numeral of n
a building block of rank n	corresponds to	a digit of weight $base^n$
inserting an element	corresponds to	incrementing a numeral
merging two containers	corresponds to	adding two numerals

Problem Retrieving the first element (head) of a list typically takes only constant time. On the other hand, it takes $O(\lg n)$ 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[hinze1998numerical] 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 ?? resolves the problems we have addressed in this chapter by proposing some generalizations to the conventional positional numeral systems.

Chapter ?? 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

Background



Chapter 3

Design



Chapter 4

Implementation



Chapter 5

Evaluation



Chapter 6

Related Work



Chapter 7

Conclusion

