Kalpa Publications in Computing

Volume 18, 2024, Pages 92–106

LPAR 2024 Complementary Volume

Numeric Base Conversion with Rewriting

Olivier Hermant[1] and Wojciech Loboda[2]

[1] CRI, Mines Paris, PSL University, Paris, France
[2] Institute of Computer Science, AGH University of Science and Technology, Poland

Abstract

We introduce and discuss a term-rewriting technique to convert numbers from any numeric base to any other one without explicitly appealing to division. The rewrite system is purely local and conversion has a quadratic complexity, matching the one of the standard base-conversion algorithm. We prove confluence and termination, and give an implementation and benchmarks in the Dedukti type checker. This work extends and generalizes a previous work by Delahaye, published in a vulgarization scientific review.

1 Introduction

Conversion between numeral positional systems in different bases is a well known concept. In his article "Changer de numération avec le système esperluette" [4], Jean-Paul Delahaye presents an original way to convert a number to any base without using division or multiplication explicitly. Indeed, the algorithm works by locally rewriting a representation with few simple rules, that allow to switch from base 1 to any other base back and forth.

This yields an indirect conversion algorithm between two bases, as we must use base 1 as an intermediate step. In the rewrite system of [4], & stands for the only digit in base 1. The rules for conversion of a representation from base 1 to base 2 are the following, where x and y stand for any sequence of $0, 1$, and &, including the empty sequence:

$$\begin{aligned} \&x &\longrightarrow 0\&x && (r1)\\ x0\&\&y &\longrightarrow x\&0y && (r2)\\ x0\&y &\longrightarrow x1y && (r3) \end{aligned}$$

The rules for the converse base 2 to base 1 transformation are the reversed rules:

$$\begin{aligned} 0\&x &\longrightarrow \&x && (r1')\\ x\&0y &\longrightarrow x0\&\&y && (r2')\\ x1y &\longrightarrow x0\&y && (r3') \end{aligned}$$

The systems presented above are string rewriting systems (x and y are words) and not term rewriting systems. Moreover, systems for conversion form any base to base 1 are not confluent. As noticed by the author, premature application of $(r3)$ leads to dead-end situations, which is a symptom of a non-confluent rewrite system. We car be stuck in an hybrid representation

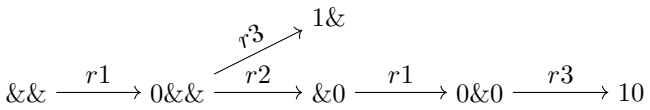

Figure 1: Confluence issue with the original rewrite system of [4].

instead of reaching the correct binary representation. Fig. 1 displays an example on the natural number 2, that can lead to 1& if the rewrite system is not used properly.

Term rewriting is a natural way of formalizing the original ideas [4], and we first design a term rewrite system that represents it, slightly improved to correct the confluence issue mentioned above. We then generalize the approach by defining a direct translation between any two numeric bases which is sound, confluent and terminating.

We begin our work by formalizing terms representing a number, we introduce the term rewrite systems for conversion from any base to base 1, which directly corresponds to original system, and systems for conversion from base 1 to any base, by modifying the original idea to make the systems confluent. We then introduce a class of systems for direct conversion between any two bases and prove soundness, confluence and termination for each system.

We implemented our systems in Dedukti [1], a logical framework based on the $\lambda\Pi$-calculus modulo [3]. Dedukti contains a rewrite engine, capable of rewriting terms using rules given by the user [5]. We provide a tool for generating a Dedukti file with systems for direct conversion and benchmark base conversion with an extensive testing. We also compute and measure the time complexity of the conversion with Dedukti and our systems and compare it to the time of conversion using the usual algorithm.

2 Base Conversion and Rewriting

Term rewriting is a branch of computer science and logic which formalizes methods of replacing terms with other terms. By term, we mean an expression consisting of variables, and symbols with fixed arity, that are fully applied to the relevant number of terms. We can further divide symbols into constants (0-ary symbols) and functions. For a term t, $Var(t)$ denotes set of variables in t. Below, we recall the basic definitions of rewriting [2].

Definition 1 (Rewriting)**.** *A rewrite rule is an identity $l \longrightarrow r$, such that l, r are terms, l is not a variable and $Var(l) \supseteq Var(r)$. A term rewrite system R is a set of rewrite rules.*

A term t rewrites to a term u, denoted $t \to_R t'$ if there is a position ω in t, a substitution σ, and a rule $l \to r \in R$, such that $t_\omega = l\sigma$, and $t' = t_\omega[u\sigma]$.

In other words, rewriting t into t' with the rewrite rule $l \to r$ means finding an instance of l in t (by pattern matching) and replacing this instance by the corresponding instance of r. We will use $\to$ instead of $\to_R$ if R is clear from the context. $\to^*$ denotes the reflexive and transitive closure of $\to$ [2].

Definition 2 (Normal Form)**.** *We call t' a normal form of t, if $t \to^* t'$ and there is no u such that $t' \to u$.*

Definition 3 (Termination)**.** *A term rewrite system is terminating, if, and only if, there is no infinite rewriting chain $t_0 \to t_1 \to \dots$.*

Definition 4 (Confluence)**.** *A term rewrite system is confluent, if, and only if,* $t \to^* u_1$ *and* $t \to^* u_2$ *implies the existence of some* t'*, such that* $u_1 \to^* t'$ *and* $u_2 \to^* t'$*.*

When the rewrite system is terminating, any term has at least one normal form, and if it is, in addition, confluent, the normal form is unique.

We can treat $\to$ as a directed computation and use term rewrite system for numeric base conversion. For this, we will need to define numeric base representations and design a rewrite system that, given a representation in an initial numeric base b, yields a normal form that is a representation in the target numeric base b'.

2.1 Numeric Base Conversion Basics

2.1.1 Representation in a Single Base

A numeric base b is a set of b pairwise distinct digits used to represent a number in a positional numeral system. Along with the base, the position of a digit in a numeric representation w determines the value this digit weighs in the value associated with w.

Formally speaking, in a system with base $b = 1, 2, ...$ and digits $D = \{d_0, d_1, ..., d_{b-1}\}$, the representation of a number is a finite sequence $(a_i)_{i=0}^n$ with $a_i \in D$. Sequences are typically denoted as $a_n a_{n-1} ... a_0$. The value associated with the representation, $\nu(a_n a_{n-1} ... a_0)$, can be calculated iteratively or recursively:

$$\nu(a_n a_{n-1} ... a_0) = \sum_{i=0}^{n} \nu(a_i) * b^i = \nu(a_0) + b * \nu(a_n a_{n-1} ... a_1),$$

where $\nu(a_i)$ denotes the value associated with the digit a_i. For example, in base 2, with $D = \{\mathbf{0}, \mathbf{1}\}$ and $\nu(\mathbf{0}) = 0, \nu(\mathbf{1}) = 1$:

$$\nu(\mathbf{1010}) = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0.$$

The system with base 1 is called the unary system and the only digit in this system corresponds to 1. Base 1 is very peculiar and may not fully deserve the name "base". It is not really a positional base, unlike all the other bases. It is also not possible to use the standard base-conversion algorithm to express a number in unary. Nevertheless, it fits in the general framework describe in this article.

Representations of a number are not unique, save for base 1, because of the potential leading zeroes. This is harmless.

In the rest of the article we will use the standard set of digits. For values from 0 to 9 we will use the digits: $\mathbf{0}, \mathbf{1}, ..., \mathbf{9}$, for the higher values we use the uppercase letters: $\mathbf{A}, \mathbf{B}, \mathbf{C}, ...$

Table 1: Values of the 16 First Standard Digits.

$\nu(\mathbf{0}) = 0$	$\nu(\mathbf{1}) = 1$	$\nu(\mathbf{2}) = 2$	$\nu(\mathbf{3}) = 3$
$\nu(\mathbf{4}) = 4$	$\nu(\mathbf{5}) = 5$	$\nu(\mathbf{6}) = 6$	$\nu(\mathbf{7}) = 7$
$\nu(\mathbf{8}) = 8$	$\nu(\mathbf{9}) = 9$	$\nu(\mathbf{A}) = 10$	$\nu(\mathbf{B}) = 11$
$\nu(\mathbf{C}) = 12$	$\nu(\mathbf{D}) = 13$	$\nu(\mathbf{E}) = 14$	$\nu(\mathbf{F}) = 15$

2.1.2 Distinguishing Digits in Different Bases

For each digit, we must be able to identify the base it comes from. Therefore, we assume that different bases contain disjoint sets of digits. To enforce this distinction, while maintaining the standard digit notation, we annotate each digit by the base itself. E.g. $\mathbf{3}_4 \in D_4$, while $\mathbf{3}_{16} \in D_{16}$. We omit the annotation when the base is clear from the context. Of course, all the digits, independently of their base, have the same standard value, as shown in Tab. 1.

Therefore each base b is composed of the digits $D_b = \{d_b \mid \nu(d) = 0, 1, ..., b-1\}$.

2.1.3 Basic Conversion Algorithm

The usual algorithm for expressing the representation of a value $v \in \mathbb{N}$ in base $b > 1$ works by performing the integral, or Euclidean, division of v by b : the remainder becomes the rightmost digit of the representation, v becomes the quotient of the integral division and we repeat the process until $v = 0$. In other words, the i-th rightmost digit of the representation of v is the remainder of the division of v/b^{i-1} by b.

2.2 Mixed-Base Representation

In order to describe the rewrite system inspired by the original article [4], the definition of a new numeral system is needed. We must allow the presence of digits associated with different bases in a single representation. This new relaxed system can be considered a generalization of standard positional system presented in Sec. 2.1.

Definition 5 (Mixed-Base Representation)**.** *A mixed-base representation is a sequence of digits* $(a_i)_{i=0}^n$*, denoted* $a_n a_{n-1} ... a_0$*, such that* $a_i \in D_{b_i}$ *for a sequence of bases* $(b_i)_{i=0}^n$*.*

The base factor *of the sequence, denoted* $\mathbf{bf}(a_n a_{n-1} ... a_0)$ *is the product of the respective bases of the digits,* $\prod_{i=0}^n b_i$*.*

The value of a mixed-base representation is defined as

$$\nu(a_n a_{n-1} ... a_0) = \sum_{i=0}^{n} \nu(a_i)\mathbf{bf}(a_{i-1} ... a_0) = \sum_{i=0}^{n} \left(\nu(a_i) \prod_{j=0}^{i-1} b_j \right) = \nu(a_0) + \nu(a_n a_{n-1} ... a_1) * b_0.$$

Example calculation of the value corresponding to a mixed-base representation:

$$\nu(\mathbf{3}_4\mathbf{1}_2\mathbf{0}_2\mathbf{A}_{16}\mathbf{1}_2) = 3 * 2^3 * 16 + 1 * 2^2 * 16 + 0 * 2 * 16 + 10 * 2 + 1 = 469$$

Proposition 6. *The iterative and recursive definitions of evaluation in Def. 5 are consistent with each other.*

Proof. By induction on the length of the sequence. □

We define the following set of terms, that we call T in the rest of the paper. They are lists of digits that correspond to numbers in the newly introduced mixed-base representation:

$$< \texttt{list of digits} > := \texttt{empty} \parallel < \texttt{member} > \cdot < \texttt{list of digits} >$$
$$< \texttt{member} > := \texttt{begin} \parallel \mathbf{1}_1 \parallel \mathbf{0}_2 \parallel \mathbf{1}_2 \parallel \mathbf{0}_3 \parallel \mathbf{1}_3 \parallel \mathbf{2}_3 \parallel ...$$

We further restrict this grammar by imposing the conditions that, in any list of digit, there must be an explicit `begin` symbol at the beginning, and that `begin` cannot appear anywhere except the beginning of a list.

These conditions make it possible to introduce rules that add or remove leading digits. This unique `begin` symbol should be then followed by a sequence of proper digits.

We can calculate the value of a term, $\nu(t), t \in T$ by ignoring `begin` and `empty` and calculating the value of sequence of digits as before:

$$\nu(\texttt{begin} \cdot a_n \cdot a_{n-1} \cdot ... \cdot a_0 \cdot \texttt{empty}) = \nu(a_n a_{n-1} ... a_0),$$
$$\nu(a_m \cdot a_{m-1} \cdot ... \cdot a_0 \cdot \texttt{empty}) = \nu(a_m a_{m-1} ... a_0)$$

2.3 Conversion to Base 1

In this section we introduce rewrite systems that can be used to convert a number represented in any base $b > 1$ to unary. The input term will be expressed as in Sec. 2.2, with the proviso that all digits in list should be from base b. Then we rewrite t using the suitable system. The normal form of t will correspond to its representation in unary. In this section we write & instead of $\mathbf{1}_1$, with $\nu(\&) = 1$, to stay as close as possible as to the original work [4].

Systems for converting numbers from base $b > 1$ to base 1 are all based on the same principle, and do not differ significantly from [4], except that they are now expressed on lists, as term rewrite rules, instead of string rewrite rules. They consist of 3 types of rules:

$$\texttt{begin} \cdot \mathbf{0} \cdot tl \longrightarrow \texttt{begin} \cdot tl \qquad \text{(type I)}$$

$$\& \cdot \mathbf{0} \cdot tl \longrightarrow \mathbf{0} \cdot (\&\cdot)^b tl \qquad \text{(type II)}$$

The rationale behind the rule of type II it that it rewrites the sequence & and $\mathbf{0}$, by the percolation of $\mathbf{0}$ leftwards, thereby "multiplying" & by "ten in base b" (i.e., b). It therefore yields the sequence composed of $\mathbf{0}$ and b consecutive ampersands.

$$d \cdot tl \longrightarrow \mathbf{0} \cdot (\&\cdot)^{\nu(d)} tl \qquad \text{(type III)}$$

The rules of type III rewrite every digit $d \in D_b, \nu(d) > 0$ to $\mathbf{0}$ and $\nu(d)$ ampersands. The $\mathbf{0}$ keeps track of the positional nature of the representation. It will be handled later by the rules of type I and II.

For every system that converts a number representation in base b to a representation in base 1 there is one rule of type I, one rule of type II and $b - 1$ rules of type III, which means that the total number of rules is $b + 1$ in those systems. For example, here is the system that converts numbers in base 2 to base 1:

$$\begin{aligned} \texttt{begin} \cdot \mathbf{0} \cdot tl &\longrightarrow \texttt{begin} \cdot tl && \text{(type I)} \\ \& \cdot \mathbf{0} \cdot tl &\longrightarrow \mathbf{0} \cdot \& \cdot \& \cdot tl && \text{(type II)} \\ \mathbf{1} \cdot tl &\longrightarrow \mathbf{0} \cdot \& \cdot tl && \text{(type III)} \end{aligned}$$

A rewrite system is sound with respect to the value associated with the representation, if, and only if, the value of a term does not change when we apply rules.

Definition 7 (Soundness of a Rewrite System)**.** *Let R a rewrite system, t and t' be two terms, such that $t \to_R^* t'$. R is sound if, and only if, $\nu(t) = \nu(t')$.*

Although this definition is generic, we will apply it only in the sub-case where t and t' mix only two bases b_1 and b_2, and R is the conversion system from b_1 to b_2.

2.3.1 Proof of Soundness

We show that any instance of the rules of type I, II and III preserves the value. We denote the term before applying a rule as t and after rule application as t', b is the initial base, $k \in \mathbb{N}$ is the base factor of the tail, $\mathbf{bf}(tl)$.

The rule of type I removes a leading digit that has value 0 and does not contribute to the value associated with the whole representation, thus the value does not change.

In case of application of the rule of type II:

$$\nu(t) = \nu(tl) + \nu(\mathbf{0}) * k + \nu(\&) * (b * k) = \nu(tl) + \nu(\&) * (b * k),$$
$$\nu(t') = \nu(tl) + (b * \nu(\&)) * k + \nu(\mathbf{0}) * (1^b * k) = \nu(tl) + \nu(\&) * (b * k),$$
$$\text{thus } \nu(t) = \nu(t').$$

In case of application of a rule of type III:

$$\nu(t) = \nu(tl) + \nu(d) * k,$$
$$\nu(t') = \nu(tl) + \Sigma_{i=0}^{\nu(d)-1} \left(\nu(\&) * (1^i * k)\right) + \nu(\mathbf{0}) * (1^{\nu(d)} * k) = \nu(tl) + \nu(d) * k,$$
$$\text{thus } \nu(t) = \nu(t').$$

The associated value effectively remains the same for any instance of any rule. To obtain soundness, we still have to check that rewriting at a position ω in a context (following Def. 1) does not change the final value. For this, notice that the context itself does not change, that the rewritten sub-terms do not change value, and, critically, that :

- type II and type III rules do not change the base factor of the rewritten term, because the base factor of & is 1;
- the type I rule does trigger the factor of the rewritten term, but that it only applies at top level, so that, in this particular case, the context has to be empty.

In all cases, the value is preserved.

2.3.2 Proof of Termination

In order to prove termination we introduce a reduction order on T by mapping each term $t \in T$ to the pair $(\pi(t), \pi'(t))$, with $\pi, \pi' : T \to \mathbb{N}$. Pairs are compared lexicographically. π gathers the values of the source base digits, while π' identifies the first encountered digit and yields 0 if it is a D_b digit, and 1 if it is $\& \in D_1$, as follows:

$\pi(d \cdot tl) = \nu(d) + 1 + \pi(tl)$	$\pi'(d \cdot tl) = 0$
$\pi(\& \cdot tl) = \pi(tl)$	$\pi'(\& \cdot tl) = 1$
$\pi(\texttt{begin} \cdot tl) = \pi(tl)$	$\pi'(\texttt{begin} \cdot tl) = \pi(tl)$
$\pi(\texttt{empty}) = 0$	$\pi'(\texttt{empty}) = 0$

For the rule of type I, $l = \texttt{begin} \cdot 0 \cdot tl$, $r = \texttt{begin} \cdot tl$, and we have :

$$(\pi(l), \pi'(l)) = (1 + \pi(tl), 0) > (\pi(tl), 0) = (\pi(r), \pi'(r))$$

For the rule of type II, $l = \& \cdot 0 \cdot tl$, $r = 0 \cdot (\&\cdot)^b tl$, and we have

$$(\pi(l), \pi'(l)) = (1 + \pi(tl), 1) > (1 + \pi(tl), 0) = (\pi(r), \pi'(r))$$

For the rules of type III, $l = d \cdot tl, r = (\&\cdot)^{\nu(d)} tl$, and we have

$$(\pi(l), \pi'(l)) = (\nu(d) + 1 + \pi(tl), 0) > (\pi(tl), 1) = (\pi(r), \pi'(r))$$

For all 3 types of rules $l \longrightarrow r$, $(\pi(l), \pi'(l)) > (\pi(r), \pi'(r))$, thus by Thm. 5.2.3 from [2], the conversion systems to base 1 are terminating.

2.3.3 Proof of Confluence

Rules do not produce critical pairs and systems are terminating, thus, by Cor. 6.2.5 from [2], systems are confluent.

2.4 From Base 1

Systems for converting numbers from base 1 to any base $b > 1$ also consist of 3 types of rules:

$$\texttt{begin} \cdot \& \cdot tl \longrightarrow \texttt{begin} \cdot \mathbf{0} \cdot \& \cdot tl \qquad \text{(type I)}$$

The rule of type II rewrites the largest digit of D_b (such that $\nu(d) = b - 1$) followed by an ampersand to ampersand and 0.

$$d \cdot \& \cdot tl \longrightarrow \& \cdot \mathbf{0} \cdot tl \qquad \text{(type II)}$$

Rules of type III accumulate the ampersands in the digit d, so that d takes values in $[0...b-2]$ and d' is such, that $\nu(d') = \nu(d) + 1$.

$$d \cdot \& \cdot tl \longrightarrow d' \cdot tl \qquad \text{(type III)}$$

Every system for conversion from base 1 to any base b consists of 1 rule of type I, 1 rule of type II and $b - 1$ rules of type III, which means that every system consists of $b + 1$ rules. The concrete systems that convert numbers in base 1 respectively to binary and to base 10 are displayed in Tab. 2. Compared to the base 2 conversion system of [4], type I and III rules are identical, while type II rule has changed.

Table 2: Conversion Rules from Base 1 to : Base 2 (left) and Base 10 (right)

$\texttt{begin} \cdot \& \cdot tl \longrightarrow \texttt{begin} \cdot \mathbf{0} \cdot \& \cdot tl$	$\texttt{begin} \cdot \& \cdot tl \longrightarrow \texttt{begin} \cdot \mathbf{0} \cdot \& \cdot tl$ (type I)
$\mathbf{1} \cdot \& \cdot tl \longrightarrow \& \cdot \mathbf{0} \cdot tl$	$\mathbf{9} \cdot \& \cdot tl \longrightarrow \& \cdot \mathbf{0} \cdot tl$ (type II)
$\mathbf{0} \cdot \& \cdot tl \longrightarrow \mathbf{1} \cdot tl$	$\mathbf{0} \cdot \& \cdot tl \longrightarrow \mathbf{1} \cdot tl$
	$\mathbf{1} \cdot \& \cdot tl \longrightarrow \mathbf{2} \cdot tl$
	$\mathbf{2} \cdot \& \cdot tl \longrightarrow \mathbf{3} \cdot tl$
	$\mathbf{3} \cdot \& \cdot tl \longrightarrow \mathbf{4} \cdot tl$
	$\mathbf{4} \cdot \& \cdot tl \longrightarrow \mathbf{5} \cdot tl$ (type III)
	$\mathbf{5} \cdot \& \cdot tl \longrightarrow \mathbf{6} \cdot tl$
	$\mathbf{6} \cdot \& \cdot tl \longrightarrow \mathbf{7} \cdot tl$
	$\mathbf{7} \cdot \& \cdot tl \longrightarrow \mathbf{8} \cdot tl$
	$\mathbf{8} \cdot \& \cdot tl \longrightarrow \mathbf{9} \cdot tl$

2.4.1 Proof of Soundness

We use the same notations as in Sec. 2.3. Systems introduced above are also sound with respect to value associated with representation.

Rules of type I and II are the exact reverse of the rules of type I and II of Sec. 2.3, so their instances are sound by virtue of Sec. 2.3.1. For rules of type III,

$$\nu(t) = \nu(tl) + \nu(\&) * k + \nu(d) * (1 * k) = \nu(tl) + (\nu(d) + 1) * k = \nu(tl) + \nu(d') * k = \nu(t').$$

The value associated with the representation does not change when we instantiate the rules, and the base factor does not change either, save in the case of type I rule. But in this latter case, rewriting can only happen at top level of a representation. Thus the systems from base 1 are sound, for the same reasons as in Sec. 2.3.1.

2.4.2 Proof of Termination

As in Sec. 2.3, we prove termination by introducing a mapping from mixed-based terms to pairs of natural numbers $(\pi(t), \pi'(t))$, where π and π' are defined as follows:

$$\begin{array}{ll} \pi(d \cdot tl) = \nu(d) + \pi(tl) & \pi'(d \cdot tl) = 0 \\ \pi(\& \cdot tl) = 2 + \pi(tl) & \pi'(\& \cdot tl) = 1 \\ \pi(\texttt{begin} \cdot tl) = \pi(tl) & \pi'(\texttt{begin} \cdot tl) = \pi'(tl) \\ \pi(\texttt{empty}) = 0 & \pi'(\texttt{empty}) = 0 \end{array}$$

For the rule of type I, $l = \texttt{begin} \cdot \& \cdot tl$, $r = \texttt{begin} \cdot 0 \cdot \& \cdot tl$, and

$$(\pi(l), \pi'(l)) = (1 + \pi(tl), 1) > (1 + \pi(tl), 0) = (\pi(r), \pi'(r)).$$

For the rule of type II, $l = d \cdot \& \cdot tl$, $r = \& \cdot 0 \cdot tl$, and we have

$$(\pi(l), \pi'(l)) = ((b-1) + 2 + \pi(tl), 0) > (2 + \pi(tl), 1) = (\pi(r), \pi'(r)).$$

For the rules of type III, $l = d \cdot \& \cdot tl$, $r = d' \cdot tl$, and we have

$$(\pi(l), \pi'(l)) = (\nu(d) + 2 + \pi(tl), 0) > (\nu(d) + 1 + \pi(tl), 0) = (\nu(d') + \pi(tl), 0) = (\pi(r), \pi'(r)).$$

For all 3 types of rules $l \longrightarrow r$, $(\pi(l), \pi'(l)) > (\pi(r), \pi'(r))$, thus by Thm. 5.2.3 of [2] systems are terminating.

2.4.3 Proof of Confluence

Rules do not produce critical pairs and systems are terminating, thus by Cor. 6.2.5 from [2] systems are confluent.

2.5 From and to any Base

The drawback of base b_1 to b_2 conversion using the previous rewrite systems is that we need a detour through base 1, which is rather inefficient.

In this section we introduce a rewrite system for direct conversion from any base $b_1 > 1$ to any other base $b_2 > 1$. As with the previous systems, firstly we need to construct a term $t \in T$ corresponding to a pure representation of some number in base b_1, then rewrite t using the suitable system. The normal form of t will corresponds to the pure representation of this

number in base b_2. In between, we will manipulate a mixed-base representation.

The systems for direct conversion from b_1 to b_2 consist of 3 types of rules, that make b_1 digits percolate leftwards and, eventually, vanish. The rule of type I removes initial zeroes :

$$\texttt{begin} \cdot \mathbf{0}_{b_1} \cdot tl \longrightarrow \texttt{begin} \cdot tl \qquad \text{(type I)}$$

Rules of type II rewrite non-zero b_1 digits at the beginning of a representation. d'_{b_1} is replaced by the unique d_{b_1} and d_{b_2} that verify $\nu(d_{b_1}) = \nu(d'_{b_1})/b_2$ and $\nu(d_{b_2}) = \nu(d'_{b_1})\%b_2$. Here and below, division is the integral division, and % is the modulo operator.

So, rules of type II replace d'_{b_1} by the quotient and remainder of its Euclidean division by b_2. The quotient is itself expressed in base b_1, therefore guaranteeing that it can be represented with exactly one digit. Further improvements could be made, but this presentation has the advantage to be uniform :

$$\texttt{begin} \cdot d'_{b_1} \cdot tl \longrightarrow \texttt{begin} \cdot d_{b_1} \cdot d_{b_2} \cdot tl \qquad \text{(type II)}$$

Rules of type III rewrite the sequence $d_{b_2} d_{b_1}$ to the sequence $d'_{b_1} d'_{b_2}$, where $\nu(d'_{b_1}) = \nu(d_{b_2} d_{b_1})/b_2 = (\nu(d_{b_1}) + \nu(d_{b_2}) * b_1)/b_2$ and $\nu(d'_{b_2}) = \nu(d_{b_2} d_{b_1})\%b_2 = (\nu(d_{b_1}) + \nu(d_{b_2}) * b_1)\%b_2$. Note that $\nu(d_{b_2} d_{b_1})/b_2 < b_1$, so the existence of d'_{b_1} is ensured :

$$d_{b_2} \cdot d_{b_1} \cdot tl \longrightarrow d'_{b_1} \cdot d'_{b_2} \cdot tl \qquad \text{(type III)}$$

Every system for direct conversion from base b_1 to base b_2 has 1 rule of type I, $b_1 - 1$ rules of type II and $b_1 * b_2$ rules of type III. Tab. 3 displays such a system for the conversion from base 2 to base 10.

Table 3: Direct Conversion System from base 2 to base 10.

$$\begin{array}{rcl}
\texttt{begin} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \texttt{begin} \cdot tl \\
\texttt{begin} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \texttt{begin} \cdot \mathbf{0}_2 \cdot \mathbf{1}_{10} \cdot tl
\end{array}$$

$$\begin{array}{rcl|rcl}
\mathbf{0}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{0}_{10} \cdot tl & \mathbf{0}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{1}_{10} \cdot tl \\
\mathbf{1}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{2}_{10} \cdot tl & \mathbf{1}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{3}_{10} \cdot tl \\
\mathbf{2}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{4}_{10} \cdot tl & \mathbf{2}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{5}_{10} \cdot tl \\
\mathbf{3}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{6}_{10} \cdot tl & \mathbf{3}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{7}_{10} \cdot tl \\
\mathbf{4}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{8}_{10} \cdot tl & \mathbf{4}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{0}_2 \cdot \mathbf{9}_{10} \cdot tl \\
\mathbf{5}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{0}_{10} \cdot tl & \mathbf{5}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{1}_{10} \cdot tl \\
\mathbf{6}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{2}_{10} \cdot tl & \mathbf{6}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{3}_{10} \cdot tl \\
\mathbf{7}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{4}_{10} \cdot tl & \mathbf{7}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{5}_{10} \cdot tl \\
\mathbf{8}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{6}_{10} \cdot tl & \mathbf{8}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{7}_{10} \cdot tl \\
\mathbf{9}_{10} \cdot \mathbf{0}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{8}_{10} \cdot tl & \mathbf{9}_{10} \cdot \mathbf{1}_2 \cdot tl & \longrightarrow & \mathbf{1}_2 \cdot \mathbf{9}_{10} \cdot tl
\end{array}$$

2.5.1 Proof of Soundness

Systems for direct conversion are sound with respect to value associated with the representation. As before we start by showing that any instance of each type of rule preserves the associated

value. We denote the term before applying a rule as t and after as t', b_1 is the initial base, b_2 is the target base we are converting, and $k \in \mathbb{N}$ is the base factor of the tail.

When applying the rule of type I we remove the leftmost digit corresponding to the value 0. This digit does not contribute by any value to the value of t.

In the case of the rules of type II, we have

$$\begin{aligned}
\nu(t') &= \nu(tl) + \nu(d_{b_2}) * k + \nu(d_{b_1}) * b_2 * k \\
&= \nu(tl) + (\nu(d'_{b_1})\%b_2) * k + (\nu(d'_{b_1})/b_2) * b_2 * k && \text{(by definition of } d_{b_2} \text{ and } d_{b_1}\text{)} \\
&= \nu(tl) + \left[\nu(d'_{b_1})\%b_2 + (\nu(d'_{b_1})/b_2) * b_2\right] * k \\
&= \nu(tl) + \nu(d'_{b_1}) * k && \text{(by definition of integral division)} \\
&= \nu(t). && \text{(by definition of } \nu\text{)}
\end{aligned}$$

In the case of the rules of type III, we have

$$\begin{aligned}
\nu(t') &= \nu(tl) + \nu(d'_{b_2}) * k + \nu(d'_{b_1}) * b_2 * k \\
&= \nu(tl) + [\nu(d_{b_2}d_{b_1})\%b_2 + (\nu(d_{b_2}d_{b_1})/b_2) * b_2] * k && \text{(by definition of } d'_{b_2} \text{ and } d'_{b_1}\text{)} \\
&= \nu(tl) + \nu(d_{b_2}d_{b_1}) * k && \text{(by definition of integral division)} \\
&= \nu(t). && \text{(by definition of } \nu\text{)}
\end{aligned}$$

The value associated with the representation does not change in the instances of all types of rules. Rules of type III do not change the base factor of t and t'. Rules of type I and II do change the base factor, but they apply only at top level. The system is therefore sound.

2.5.2 Proof of Termination

We prove termination of the rewrite systems for direct conversion by introducing a reduction order on T. We map each $t \in T$ to a triplet of natural numbers, $(\pi(t), \pi'(t), \pi''(t))$, that we order lexicographically. π counts the number of digits in base b_1, π' is the position of the first occurrence of a b_1 digit in a representation, and π'' is the value of this first occurrence, as follows :

$$\begin{array}{ccc}
\pi(d_{b_1} \cdot tl) = 1 + \pi(tl) & \pi'(d_{b_1} \cdot tl) = 0 & \pi''(d_{b_1} \cdot tl) = \nu(d_{b_1}) \\
\pi(d_{b_2} \cdot tl) = \pi(tl) & \pi'(d_{b_2} \cdot tl) = 1 + \pi'(tl) & \pi''(d_{b_2} \cdot tl) = \pi''(tl) \\
\pi(\texttt{begin} \cdot tl) = \pi(tl) & \pi'(\texttt{begin} \cdot tl) = \pi'(tl) & \pi''(\texttt{begin} \cdot tl) = \pi''(tl) \\
\pi(\texttt{empty}) = 0 & \pi'(\texttt{empty}) = 0 & \pi''(\texttt{empty}) = 0
\end{array}$$

The lexicographic order decreases strictly, as follows :

1. π decreases strictly for any instance of the rule of type I, and is not affected by instances of any rule of type II and III.

2. π' decreases strictly for any instance of a rule of type III, and is not affected (it remains equal to 0) by instances of any rule of type II.

3. Finally, π'' decreases strictly for any instance of a rule of type II, since $\nu(d_{b_1}) = \nu(d'_{b_1})/b_2$, and $b_2 > 1$.

Each rule $l \longrightarrow r$ of every rewrite system for direct conversion is such that the measure decreases. By Thm. 5.2.3 of [2] all these systems are terminating.

2.5.3 Proof of Completeness

Here, we show that every normal form contains only base b_2 digits. Assume that t is in normal form and contains a b_1 digit. It cannot be the leftmost digit of t, otherwise a type I or type II rule would apply, as together they cover every possible b_1 digit in first position. Therefore, the first b_1 digit in t cannot be the leftmost, and there must be a b_2 digit on its left. But this triggers a type III rule, which is a contradiction.

The completeness proofs for the base 1 conversion systems are similar and left to the reader.

2.5.4 Proof of Confluence

Rules do not produce critical pairs and systems are terminating, thus by Cor. 6.2.5 from [2] systems are confluent.

2.5.5 Complexity Analysis

Type I and II rules act on a b_1 digit at the initial position of a representation, while type III rules act on a b_1 digit that is *not* in initial position. So, type I and II rules commute with type III rules. Note as well, that type II and type III rules preserve the number of b_1 digits, and that all these digits are to disappear in the normal form.

Any rewrite sequence can therefore be organized, without changing its length, as starting with a pure b_1 representation, to which we apply rewriting in a series of sub-sequences of the following form :

- a series of type I and type II rules that make the leading b_1 digit disappear,
- followed by a series of type III rules that push all the b_2 digits rightmost, and make a leading b_1 digit appear again.

Sub-sequences enjoy the following properties :

- each sub-sequence makes exactly one b_1 digit disappear (the leading one), so there are exactly n sub-sequences, if the input representation has n digits.
- at the end of a sub-sequence, all b_1 digits are on the left, and all b_2 digits are on the right.
- type III rules commute with each other, as they do not overlap, so we can push the b_2 digits on the right with the strategy of our choice.

We now evaluate the maximal length of a sub-sequence. For every leading d_{b_1}, there is at most a constant k number of steps, after which a type I rule is applied[1]. This introduces a sequence of k b_2 digits.

In a sub-sequence, there is, therefore, at most $(k+1)$ rewrite steps of type I and type II. Then, type III rules will push all the new b_2 digits to the right. Each new b_2 digit will swap its position with each remaining b_1 digits : at most $k * n$ steps.

In total, a sub-sequence involves at most $(k+1) + k * n = O(n)$ rewrite steps, and the total rewrite sequence is of length $O(n^2)$. Lastly, to assess correctly the complexity, we have to take into account the complexity of pattern matching.

Naive pattern matching on a term of length n is of complexity $O(n)$, so our conversion algorithm is of complexity $O(n^3)$. But, at least theoretically, we can improve this bound and have an $O(1)$ pattern matching for type III rules:

[1] To be precise, $k = \lceil \log_{b_2}(b_1) \rceil$, as each type II rule divides by b_2 the value of the leading digit, which is at most $b_1 - 1$.

- in a sub-sequence, after the application of a type III rule, that swaps a b_1 and b_2 digit, we are at the exact position of the next potential type III rule application.
- instead of inspecting the term again from its beginning, it is therefore much more efficient to keep inspecting the current position, which is $O(1)$, given the purely local nature of the rewrite system.
- this will make each b_2 digit swap rightmost in $O(n)$. As there are at most k such digits to process, each sub-sequence can be run in time $O(n)$.

In total, this rewriting strategy yields an $O(n^2)$ conversion algorithm.

3 Implementation and Benchmark

We created a python and a Haskell program for generating all the systems for direct conversion in Dedukti and tested conversion for any pair of bases between 2 and 19. We also measured the conversion times using our systems.

3.1 Implementation in Dedukti

The scripts are available in a repository[2] and can be used to generate systems for direct conversion between bases in Dedukti. To generate a Dedukti module with the system for base conversion, we use the python script `genrules.py`. We must provide the source base and target base as command-line arguments.

```
$python gen_rules.py base1 base2
```

Here is the example of the conversion from base 2 to base 10, exactly corresponding to Tab. 3, that the script generates in Dedukti. `b` represents `begin`, `Nil` represents `empty`, `cons` is the list constructor (denoted as $\cdot$ in Sec. 2), and the digits can be recognized effortlessly :

```
Digit : Type.
Term : Type.
Nil : Term.
b : Digit.
def cons : Digit -> Term -> Term.
0t : Digit.
1t : Digit.
2t : Digit.
3t : Digit.
4t : Digit.
5t : Digit.
6t : Digit.
7t : Digit.
8t : Digit.
9t : Digit.
0 : Digit.
1 : Digit.
```

[2] https://github.com/wojciechloboda/base-conversion-with-term-rewriting

```
[tail] cons b (cons 0 (tail)) --> cons b (tail).
[tail] cons b (cons 1 (tail)) --> cons b (cons 0 (cons 1t (tail))).
[tail] cons 0t (cons 0 (tail)) --> cons 0 (cons 0t (tail)).
[tail] cons 0t (cons 1 (tail)) --> cons 0 (cons 1t (tail)).
[tail] cons 1t (cons 0 (tail)) --> cons 0 (cons 2t (tail)).
[tail] cons 1t (cons 1 (tail)) --> cons 0 (cons 3t (tail)).
[tail] cons 2t (cons 0 (tail)) --> cons 0 (cons 4t (tail)).
[tail] cons 2t (cons 1 (tail)) --> cons 0 (cons 5t (tail)).
[tail] cons 3t (cons 0 (tail)) --> cons 0 (cons 6t (tail)).
[tail] cons 3t (cons 1 (tail)) --> cons 0 (cons 7t (tail)).
[tail] cons 4t (cons 0 (tail)) --> cons 0 (cons 8t (tail)).
[tail] cons 4t (cons 1 (tail)) --> cons 0 (cons 9t (tail)).
[tail] cons 5t (cons 0 (tail)) --> cons 1 (cons 0t (tail)).
[tail] cons 5t (cons 1 (tail)) --> cons 1 (cons 1t (tail)).
[tail] cons 6t (cons 0 (tail)) --> cons 1 (cons 2t (tail)).
[tail] cons 6t (cons 1 (tail)) --> cons 1 (cons 3t (tail)).
[tail] cons 7t (cons 0 (tail)) --> cons 1 (cons 4t (tail)).
[tail] cons 7t (cons 1 (tail)) --> cons 1 (cons 5t (tail)).
[tail] cons 8t (cons 0 (tail)) --> cons 1 (cons 6t (tail)).
[tail] cons 8t (cons 1 (tail)) --> cons 1 (cons 7t (tail)).
[tail] cons 9t (cons 0 (tail)) --> cons 1 (cons 8t (tail)).
[tail] cons 9t (cons 1 (tail)) --> cons 1 (cons 9t (tail)).
```

These systems are instantly declared terminating and confluent by the tools Wanda and CSI.

3.2 Benchmark

We measured the time of base conversion in Dedukti for a range of combination of source and target bases, and for representations with various numbers of digits n. Digits have been randomly chosen. Conversion is done by running the command

```
dk check --stdin module.dk
```

where `module.dk` is the name of the Dedukti module that contains the definition of the rewrite system. We provide a term representing a number in b_1 as an input.

Table 4: Conversion Time in Seconds.

Digits\System	2 to 10	10 to 2	5 to 10	10 to 5	9 to 10	10 to 9	5 to 16	16 to 5
100	0.0056	0.1101	0.0155	0.0282	0.0180	0.0193	0.0093	0.0387
200	0.0122	0.4578	0.0363	0.1037	0.0611	0.0698	0.0320	0.1652
300	0.0263	1.1028	0.0806	0.2432	0.1371	0.1533	0.0703	0.3549
400	0.0433	2.0179	0.1441	0.4428	0.2426	0.2836	0.1242	0.6551
500	0.0660	3.3339	0.2327	0.7098	0.3968	0.4493	0.1931	1.0819
600	0.0959	4.9116	0.3281	1.1245	0.6712	0.6609	0.2801	1.5495
700	0.1287	6.7356	0.5456	1.5785	0.9968	1.2142	0.5097	2.5296
800	0.1796	10.751	0.6798	2.0055	1.0964	1.2706	0.5081	3.3080
900	0.2209	12.081	0.8194	2.5501	1.3622	1.6939	0.6673	3.8998

As we can notice in Tab. 4, there is a significant difference in conversion time depending on the bases combination. It is not symmetric : time increases proportionally to the quotient of

the initial and target bases. The number of rules also affects the conversion time, conversion with systems with more rules is slower.

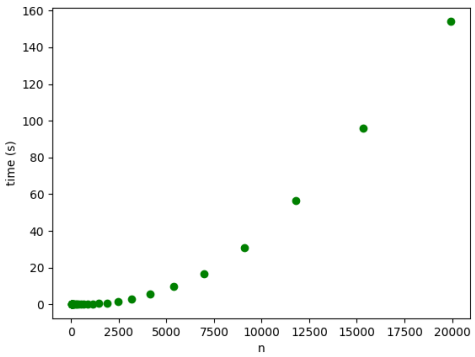

Figure 2: Base 2 to Base 10, Dedukti.

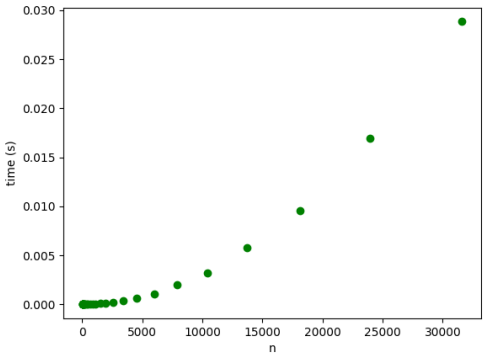

Figure 3: Base 2 to Base 10, Numpy.

We compared the conversion time using our systems in Dedukti with the conversion time using the usual algorithm implemented in Python. As we can see in Fig. 2 and Fig. 3, execution times differ drastically, in favor of Python. It is not very clear whether Dedukti demonstrates a quadratic or a cubic time behavior, while Numpy results are more clearly quadratic.

4 Conclusion

In this article, we have developed the idea of numeric base conversion using term rewriting systems. We first presented a generalization of the standard representation of natural numbers, the mixed-base representation. With this new representation, we presented a method for constructing term rewrite systems that can be used for conversion between any two bases in a indirect way, using intermediate unary base, and in a direct way. We have shown that they enjoy all the necessary properties : soundness, completeness, confluence and termination.

The algorithm that we obtained makes no call to division, unlike standard base conversion. Euclidean division is only needed to *precompute* the finite term rewriting system. Actually, our approach can even be used to implement Euclidean division by b : given a number n, it suffices to convert n into base b to get the quotient and the remainder.

The rewrite system is linear, and purely local, flipping two digits at a time. Moreover, the (type-III) rules to convert from base b_1 to b_2 are the reversal of the (type-III) rules to convert from base b_2 to b_1. Proof of this result is left to the reader.

On the practical side, we provide a simple script for generating these systems in Dedukti and extensively tested our conversion algorithm for all the combination of bases between 2 and 19 (we displayed only the most relevant ones in the benchmark section). We conclude that numeric base conversion introduced in this article works properly for the tested bases, with very decent performances.

Regarding the efficiency of the algorithm, we have analyzed the complexity of conversion using our systems in terms of number of rule applications and of time complexity. We have also

compared the actual time of conversion with our systems and Dedukti to the time of conversion using the usual algorithm for expressing number in a certain base. Conversion with the usual algorithm implemented in python turned out to be significantly faster, which is certainly due to the efficiency of the Python library that we used.

However, the asymptotic complexity of our algorithm seems of interest, and it as least can sustain comparison with the usual base conversion algorithm, both from the practical and the theoretical sides. This analysis still needs to be refined.

As further work, we will try to generate more optimized rewrite systems. In particular rules of type II can be improved to directly generate a pure sequence of b_2 digits. Factually, this would merge type I and type II rules, without changing the complexity of the algorithm. We may as well try to implement a fast pattern-matching algorithm that would speed-up (in our specific case) the complexity bound given by naive pattern matching. We also intend to use more heavily optimized rewrite engines, like Maude.

5 Acknowledgments

Thanks to Gilles Dowek for the remark, that base conversion can be used to perform division, and to the Deducteam project-team and CRI colleagues for the numerous discussions.

References

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the λΠ-calculus modulo theory. *CoRR*, abs/2311.07185, 2023.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[3] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[4] Jean-Paul Delahaye. Changer de numération avec le système &. *Pour la Science*, 519:80–85, January 2021.

[5] Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description.