Gray Code Representation of Exact Real Numbers

Hideki Tsuiki
Department of Fundamental Sciences
Kyoto University

Revised 1999 September

Abstract

We propose a new representation of exact real numbers using Gray code. This representation is injective and with a modified Type 2 machine induces the same computable structure on real numbers as admissible representations. This means that real numbers with the standard computable structure can be embedded into the space of infinite words by considering a different sort of Type 2 machine. We also show that basic algorithms can be expressed naturally with respect to this representation.

1 Introduction

One of the ways of defining computability of real functions is by representing a real number as an infinite sequence and defining the computability of a function by the existence of a machine which inputs and outputs the representations one-way from left to right. This kind of machine with infinite input and output is called a Type 2 machine. We can also define the notion of a computable real number as a special case with no input. This notion of computability dates back to Turing[Turing 1936], and is the basis of the effective analysis [Weihrauch 1985, Weihrauch 1995]. It looks intuitively natural in that, for example, it says π is computable because there is a program which outputs the sequence 3.14159... infinitely. The wide use of stream based programming also shows the naturality of this kind of computation.

Though this notion of computability depends on the choice of the representation we use, there is a "natural" one which realizes the view that a machine inputs and outputs based on approximation information of real numbers, and which is equivalent to the computability notion defined through other approaches such as domain theoretic [Gianantonio 1996a] and Pour-El and Richards[Pour-El and Richards 1989]. Those representations which induce this computability are called admissible, and signed binary representation, shrinking interval representation, and Cauchy representation are examples of admissible representations. One of the fundamental properties of admissible representations is that they are not injective. That is, some numbers have more than one representations. This redundancy is considered as essential in many approaches to exact real arithmetics [Boehm *et al.* 1986, Edalat and Potts 1997, Gianantonio 1996a, Gianantonio 1996b, Vuillemin 1990]. In this way, the computability of real functions is defined in two steps: first the computability of infinite sequences is defined using Type 2 machines, and then it is connected with the computability of real functions using representations. This redundancy of admissible representations means that we cannot define the computability of real functions more directly. That is, we cannot consider an embedding of real numbers into the set of infinite sequences so that a Type 2 machine can be considered as operating directly on real numbers.

In this paper, we make such a direct definition possible by modifying computation on infinite sequences. That is, we show that by defining computability on infinite sequences via a slightly modified machine, we can obtain an injective representation without modifying the notion of computability notion on real numbers. This representation is based on the Gray code expansion, which is another binary expansion of real numbers. It has the alphabet $\{0, 1, \perp\}$, and the character $\perp$, which appears at most once in each representation, means that the digit is not defined. Because of the existence of $\perp$, a machine cannot have sequential access to inputs and outputs like a Type 2 machine. However, because $\perp$ appears only at most once, we can resolve it by adding two heads on a tape and by allowing nondeterministic behavior to a machine. We call such a machine Nondeterministic Multihead Type 2 machine (NM2-machine in short).

Though a machine becomes a bit complicated, being injective means that the space of real numbers is embedded into the space of infinite sequences of $\{0, 1, \perp\}$, and thus an NM2-machine can be considered as operating directly on real numbers. It also means that each digit of one real number is uniquely determined, which is impossible with, e.g., the signed binary representation.

One remarkable thing about this representation is that it has three recursive structures though it is characterized by two recursive equations. This fact is used in composing basic recursive algorithms like addition. Another thing is that it has symmetricity in many levels, which enables us to calculate, for example, the operation $x \mapsto 1 - x$ on $(0 < x < 1)$ by only inverting one digit, where this operation needs to invert infinitely many digits with the signed binary operation.

In this paper, we only consider the unit open interval $(0, 1)$ instead of $\mathcal{R}$. However, it can be extended to $\mathcal{R}$ as we explain in conclusion.

After reviewing the first steps of Type 2 computability in Section 2, we introduce Gray code representation in Section 3 and our machine in Section 4. Then, we study the computability notion and topological structure induced by this representation in Section 5 and 6, respectively. In Section 7 and 8, we consider basic algorithms with respect to this representation.

2 Type 2 machine and computable real numbers

Let Σ be a sufficiently large alphabet which includes all the characters we need except for the blank character B. We write Σ^* for the set of finite sequences of Σ (ranged over by α and β), and Σ^ω for the set of infinite sequences of Σ (ranged over by p and q). We write $f :\subseteq X \to Y$ when f is a partial function from X to Y. We call a number of the form $m \times 2^{-n}$ for integers m and n a dyadic number.

In this paper, we define a (simplified) Type 2 machine as follows:

Definition 1 A *Type 2 machine* with k inputs is composed of the followings:

1. k input tapes named $I_1, I_2, \ldots, I_k$ and one output tape named O, each with one head.
2. several work tapes with one head and with an alphabet Γ which includes B,
3. a finite set Q of states with one initial state $q_0 \in Q$.
4. computational rules of the following form:

$$q, h_1(c_1), \ldots, h_j(c_j), w_1(d_1), \ldots, w_l(d_l) \Rightarrow$$
$$q', [o(c),] w'_1(d'_1), \ldots, w'_m(d'_m), M_1(w''_1), \ldots, M_n(w''_n).$$

   Here, q and q' are states in Q, h_i are heads of different input tapes, o is the head of the output tape, and w_i, w'_i, and w''_i are heads of work tapes, c_i $(i = 1, \ldots, j)$ and c are characters from Σ, and d_i and d'_i are characters from Γ. M_i is '+' or '$-$'. Each part of the rule is optional; there may be a rule without $o(c)$, for example. The meaning of this rule is that if the state is q and the characters under the heads h_i (i = 1,. . .,j) and w_e (e = 1,. . .,l) are c_i and d_e, respectively, then change the state to q', write the characters c and d'_i $(i = 1, \ldots, m)$ under the heads o and w'_i, respectively, move the heads w''_i $(i = 1, \ldots, n)$ forward or backward depending on whether M_i = '+' or '$-$', and move h_i $(i = 1, \ldots, j)$ and o forward. The conditions of the rules need to be non-overlapping.

In order to ensure that each cell of the input/output tape is read/filled once from left to right, the heads of input/output tapes are defined to move automatically to the right after a read/write operation. Note that the blank character B appears only on work tapes in each rule. We can simplify the machine by omitting the states because we can simulate it with a work tape without head movement. We do not do this to ease the presentation of Section 8.

A Type 2 machine starts with the state set to q_0, the output tape and the work tapes filled with B, and input tapes filled with infinite sequences. It continues executing an applicable rule, if exists, infinitely. A Turing machine has its meaning when it terminates. On the other hand, a Type 2 machine is meaningful when it does not terminate and it outputs infinitely.

Definition 2 *A Type 2 machine M with k inputs defines a partial function f_M from $(\Sigma^\omega)^k$ to Σ^ω which maps $(p_1, \ldots, p_k)$ to p when M with the input $(p_1, \ldots, p_k)$ produces p infinitely.*

We call a partial function $f :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ computable when $f = f_M$ for some Type 2 machine M.

Let $\alpha\Sigma^\omega$ be the set of infinite sequences with the prefix α. Cantor topology is the topology with an open base $\{\alpha\Sigma^\omega \mid \alpha \in \Sigma^*\}$. Suppose that a Type 2 machine M writes, at some point, a finite prefix α to an input p and let β be the prefix of p read by M at that time. Then, one can see that M outputs α to all the inputs in $\beta\Sigma^\omega$. This shows that a computable function is continuous.

Proposition 1 *A computable function from $(\Sigma^\omega)^n$ to Σ^ω is continuous.*

Definition 3 *A representation of a set X is a surjective partial function from Σ^ω to X.*

Here, surjectivity means that all the elements have representations, and partiality means that not all the sequences need to denote an element. The unit interval $\mathcal{I} = (0, 1)$ has the decimal representation and the binary representation. Here, we adopt a modified injective version so that we can easily compare it with our representation.

Binary representation δ_{bin} uses the alphabet $\Gamma = \{0, 1\}$ and it is defined to be the partial function $\delta_{bin} : \Gamma^\omega \to \mathcal{I}$ which returns $\Sigma_{i=1}^{\infty}\{a_i \cdot 2^{-i}\}$ to $a_1a_2\ldots$ when $a_i = 1$ for some i and $\forall k \exists j \geq k\ a_j \neq 1$. Decimal representation $\delta_{dec} : \Sigma \to \mathcal{I}$ is also defined similarly. We introduce here two more representations.

Definition 4 *(Modified) signed binary representation δ_{sbin} of $\mathcal{I}$ uses the alphabet $\Gamma = \{0, 1, \bar{1}\}$ with $\bar{1}$ denoting -1, and it is a partial function $\delta_{sbin} :\subseteq \Gamma\omega \to \mathcal{I}$ which returns $\Sigma_{i=1}^{\infty}\{a_i \cdot 2^{-i}\}$ to $a_1a_2\ldots$ when $a_1 = 1$ and $\forall k \exists j \geq k$ and $j \geq l$ such that $a_j \neq 1$ and $a_l \neq \bar{1}$.*

We denote by $\delta(\alpha)$ the set $\delta(\alpha\Sigma^\omega)$ for δ a representation and α in Σ^*. Then one can easily check that $\delta_{sbin}(1) = \mathcal{I}$; it is surjective. Because $3/8 = 1/2 - 1/8 = 1/2 - 1/4 + 1/8$, we have two representations $10\bar{1}000...$ and $1\bar{1}1000...$ to $3/8$. In general, it has finite number of representations for a dyadic number and infinite number of representations for a non-dyadic number. We have $\delta_{sbin}(1\underline{1}) = (0, 1/2)$, $\delta_{sbin}(10) = (1/4, 3/4)$, and $\delta_{sbin}(11) = (1/2, 1)$. In this way, the intervals overlap on an interval.

As another representation, we consider the shrinking interval representation. We fix one way of expressing an infinite sequence of pairs of rational numbers as an infinite sequence of characters. This representation is defined through this kind of encoding.

Definition 5 *Shrinking interval representation ρ_I is a partial function which maps to x a sequence of pairs of rational numbers $(a_1, b_1), (a_2, b_2), \ldots$ such that $a_1 \leq a_2 \leq \ldots$, $b_1 \geq b_2 \geq \ldots$, and a_i and b_i converge to the same number x.*

Now, we relate via representations the computability of functions on sequences with the computability of functions on real numbers.

Definition 6 *Let $\delta_0, \ldots, \delta_n$ be representations of $X_0, \ldots, X_n$.*
1) A partial function $F : X_1 \times \ldots \times X_n \to X_0$ is $(\delta_1, \ldots, \delta_n, \delta_0)$-computable if there is a computable function $f :\subseteq (\Sigma^\omega)^n \to \Sigma^\omega$ such that $F(\delta_1(p_1), \ldots, \delta_1(p_n)) = \delta_0(f(p_1, \ldots, p_n))$ when $F(\delta_1(p_1), \ldots, \delta_1(p_n))$ exists.
2) A relation $Q \subseteq X_1 \times \ldots \times X_n \times X_0$ is $(\delta_1, \ldots, \delta_n, \delta_0)$-computable if there is a computable function $f :\subseteq (\Sigma^\omega)^n \to \Sigma^\omega$ such that $(\delta_1(p_1), \ldots, \delta_1(p_n), \delta_0(f(p_1, \ldots, p_n)) \in Q$ when there is an $x \in X_0$ which satisfies $(\delta_1(p_1), \ldots, \delta_1(p_n), x) \in Q$.

This notion of computability depends on the choice of the representation we use, and thus we have many different notions of computability. For example, the function to multiply by three is not $(\delta_{dec}, \delta_{dec})$-computable; if there exists a machine which inputs a decimal representation of a real number x and outputs that of $3 \times x$, then it must output 10000... to 033333.... However, a machine cannot output the first digit as 1 in a finite time because it can not determine that the result is in the interval $[0.1, 0.2)$ from a finite prefix of the input. On the other hand, this function is trivially computable with respect to the ternary representation. It is also $(\delta_{sbin}, \delta_{sbin})$-computable; it can output the second digit when it knows that the result is in one of the three ranges $(0, 1/2)$,$(1/4, 3/4)$, and $(1/2, 1)$.

Through these examples, one can see that a Type 2 machine can manipulate real numbers only through finite prefixes of the representation. Therefore, it is natural to compare representations based on the amount of information available from finite prefixes. For example, finite prefixes of x in δ_{bin} has the information whether x is in $(0, 1/2)$ or in $[1/2, 1)$, though those in δ_{sbin} do not have it. The following relation is such an ordering based on the convertibility by a Type 2 machine.

Definition 7 *Let $\delta :\subseteq \Gamma^\omega \to X$ and $\delta' :\subseteq \Gamma^\omega \to X'$ be representations. We call that δ is reducible to δ' ($\delta \leq \delta'$) when there is a computable function $f :\subseteq \Gamma^\omega \to \Gamma'^\omega$ such that $\delta(p) = \delta'(f(p))$ for all $p \in dom(\delta)$. We call that δ and δ' are equivalent ($\delta \equiv \delta'$) when $\delta \leq \delta'$ and $\delta' \leq \delta$.*

Equivalent representations induce the same computability on real numbers.

Proposition 2 *$\delta_{dec} \leq \delta_{sbin}$, $\delta_{sbin} \not\leq \delta_{dec}$, $\delta_{bin} \leq \delta_{sbin}$, $\delta_{sbin} \not\leq \delta_{bin}$. $\rho_I \equiv \delta_{sbin}$.*

In order to consider approximation-based computation, it is natural to restrict our interest to representations whose finite prefixes have the approximation information. That is, when $\delta(p) = x$ and α_i is the prefix of p with length i $(i = 1, 2, \ldots)$, there are two sequences x_i $(i = 1, 2, \ldots)$ and y_i $(i = 1, 2, \ldots)$ of rational numbers computable from p such that $\delta(\alpha_i) \subset (x_i, y_i)$ and $\lim_i x_i = \lim_i y_i = x$. In other words, δ satisfies $\delta \leq \rho_I$. All the representations we have defined, ρ_I, $\delta_{dec}, \delta_{bin}$, and δ_{sbin} satisfy this property. Among such representations, there are maximal ones, which are equivalent to ρ_I and δ_{sbin}. Such representations are called admissible representations

number	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Figure 1: Binary code and Gray code of integers

[1] and one may say that finite prefixes of admissible representations only have the approximation information. Thus, admissible representations induce a natural notion of computability. One of the properties of admissible representations is that they are not injective. More precisely, we have infinite level of redundancy almost everywhere on $\mathcal{I}$.

Proposition 3 *An admissible representation is injective.*

3 Gray Code Representation

Gray code is another binary encoding of natural numbers. Figure 1 shows the usual binary code and the Gray code of integers from 0 to 15. In this way, n-bit Gray code is composed by putting the n-th bit on and reversing the order of the coding up to (n-1)-bits, instead of repeating the coding up to (n-1)-bits as we do in the usual binary code. The importance of this code lies in the fact that only one bit differs between the encoding of a number and that of its successor. This code is used in many areas of computer science such as image compression [Amalraj *et al.* 1990] and finding minimal digital circuits [Dewney 1993], and also appears in many areas of mathematics.

[1] There are two notions of admissibility; c-admissibility and t-admissibility [Weihrauch 1985]. In this paper, admissibility means c-admissibility. which is stronger than t-admissibility.

The conversion between these two encodings is easy. Gray code is obtained from the usual binary code by taking the bitwise xor of the sequence and its one-bit shift. Therefore, the function to convert from binary code to Gray code is written using the notation of a functional language Haskell as follows:

```
conv s = map xor (zip s (0:s)).
```

Here, `xor` is the "exclusive or" defined as

```
xor 0 0 = 0
xor 0 1 = 1
xor 1 0 = 1
xor 1 1 = 0
```

and `zip` is a function taking two lists (of length l and m) and returning a list of pairs (of length $min(l, m)$). This conversion is injective and the inverse is written as

```
rconv x = rconv1(x,0)
rconv1 (a:s,x) = xor(a,x):rconv1(s,xor(a,x))
rconv1 ([],x) = [].
```

We will extend this coding to real numbers. Since the function `conv` is applicable also to infinite lists, we can obtain the Gray code expansion of a real number x by applying `conv` to the binary expansion of x.

The Gray code expansions of real numbers in the unit interval $(0, 1)$ can be visualized in Figure 3. Here, a horizontal line means that the corresponding bit has value 1 on the line and has value 0 on other part. This figure has a fine fractal structure and shows symmetricity of bits bigger than n at every dyadic number $m \times 2^{-n}$.

In the usual binary expansion, we have two expansions for dyadic numbers. For example, $3/4$ can be expressed as 0.1100000... and also as 0.10111111.... This is also the case for the Gray code expansion. For example, by applying `conv` to these sequences, we have the two sequences 0.1110000... and 0.101000... of $3/4$. However, one can find that the two sequences differ only at one bit (in this case, the 2nd). This means that the information that this number is $3/4$ is given only by the remaining bits and the 2nd bit does not contribute to this fact. Therefore, it would be natural to introduce the character $\perp$ denoting undefinedness and consider the sequence $0.1\perp10000...$ as the representation of $3/4$. Note also that the sequence after the bit where they differ is always 10000.... Thus, we define Gray code representation of $\mathcal{I} = (0, 1)$ as a modification of the Gray code expansion in that a dyadic number is represented as $s\perp10000...$ with $s \in \{0, 1\}^*$.

Definition 8 The Gray code representation $G(x)$ of a real number x $(0 < x < 1)$ is an infinite list $a_0a_1\ldots$ of the alphabet $\{\perp, 0, 1\}$ composed as follows: $a_i = 1$ if $m \times 2^{-i} - 2^{-(i+1)} < x < m \times 2^{-i} + 2^{-(i+1)}$ for an odd number m, and $a_i = 0$ if the same holds for an even number m. If $x = m \times 2^{-i} - 2^{-(i+1)}$ for some integer m, then a_i is $\perp$.

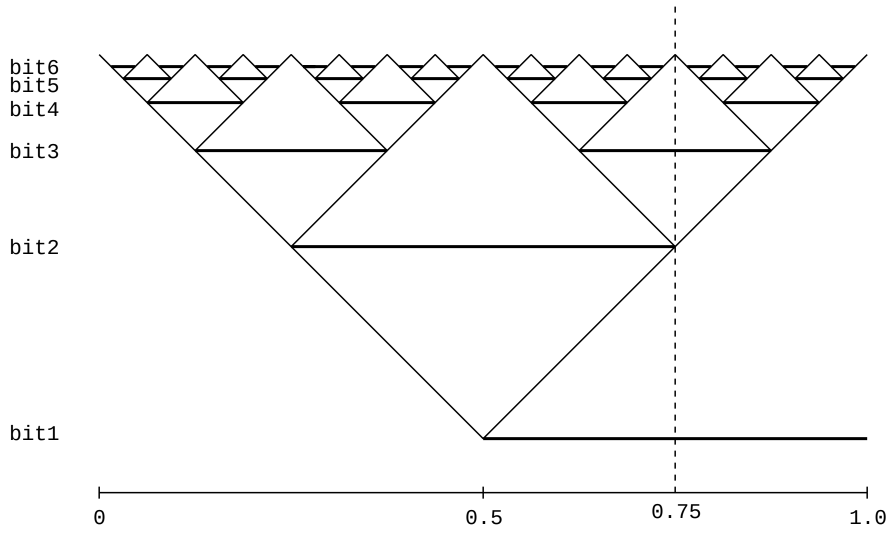

Figure 2: Gray code of real numbers

In Definition 3, a representation of a set M is defined as a surjective partial function from Σ^ω to M. Here, we defined the inverse function G as the representation since it is injective and therefore the code of a real number is uniquely defined. The codomain of G, which is the domain of the representation in the usual sense, is $\{0,1\}^\omega - \{0,1\}^*0^\omega + \{0,1\}^*\perp 10^\omega$.

Note that this expansion coincides with the itinerary by the so-called tent map

$$f(x) = \begin{array}{ll} 2x & (x <= 1/2) \\ 1-2x & (x >= 1/2) \end{array}$$

which is essential in symbolic dynamical systems [Hata and Yamaguti 1984].

4 Nondeterministic Multihead Type 2 machine

Consider that we calculate a real number x $(0 < x < 1)$ as a limit of approximations and output the result as a Gray code representation. That is, we consider a calculation which produces shrinking intervals (r_n, s_n) $(n = 0, 1, \ldots)$ successively so that $\lim s_n = \lim r_n = x$. As we have seen in Section 2, this corresponds to the output of a Type 2 machine with respect to an admissible representation.

When we know that $x < 1/2$ (i.e. $s_n < 1/2$ for some n), we can write 0 as the first digit. And when we know that $1/2 < x$ (i.e. $r_n > 1/2$ for some n), we can write 1. However, when $x = 1/2$, neither will happen and we cannot ever write the first digit. Even so, when we know $1/4 < x < 3/4$, we can skip the first and write 1 as the second digit, and when we know $3/8 < x < 5/8$, we can write 0 as the third digit. Thus, when $x = 1/2$, we can continue producing the digits skipping the first one and we can write the sequence $1000\ldots$ from the second digit. In order to produce the Gray code representation as the result, we need to fill the first cell with $\perp$, which is impossible because we cannot obtain the information $x = 1/2$ in a finite time. To solve this, we consider that the output tape is filled with $\perp$ at the beginning. That is, we define $\perp$ as the "blank character" of the output tape. usual Turing machine. Thus, when a cell is skipped and is not filled eternally, it is left as $\perp$.

Suppose that we know $1/4 < x < 3/4$ and we have written the second digit as 1 skipping the first one. As the next output, we have two possibilities: to write the third digit as 0 because we know that $3/8 < x < 5/8$; or to write the first digit because we obtain the information that $x < 1/2$ or $x > 1/2$. Therefore, when we consider a machine with Gray code output, the output tape is not written one-way from left to right. To present this behavior in a simple way, we consider two one-way heads $O1$ and $O2$ on an output tape O. At the beginning, $O1$ is located above the first and $O2$ is located above the second cell. After an output from $O2$, $O2$ is moved to the next cell, and after an output from $O1$, $O1$ is moved to the position of $O2$ and $O2$ is moved to the next cell. Thus, in order to fill the output tape as $\perp\perp\perp\perp\perp\ldots \to 0\perp\perp\perp\perp\ldots \to 0\perp 1\perp\perp\ldots \to 0\perp 10\perp\ldots \to 0110\perp\ldots \to 01101\ldots$, we output $O1(0), O2(1), O2(0), O1(1), O1(1)$. Here, $Oi(j)$ means to output j from Oi $(i = 1, 2,\ j = 0, 1)$. With this head movement rule, each cell is filled at most once and a cell is not filled eternally when $O1$ is located on that cell and output is made only from $O2$. When $O1$ and $O2$ are on the s-th and t-th cell of an output tape, the i-th cells $(i < s, s < i < t)$ are already output and no longer accessible. Therefore, $O1$ and $O2$ are always located at the first and the second unfilled cells and the machine treats the tape as if it were $[O[s], O[t], O[t+1], \ldots]$.

Next, we consider how to input a Gray code representation. In order to make our notion of computation equivalent to that induced by admissible representations, we need to define our input mechanism so that finite input contains only approximation information. On the other hand, the character $\perp$ with its preceding prefix specifies the number exactly. Therefore, our machine should not recognize that the cell under the head is $\perp$. This requirement is also supported by the way an input tape is filled when it is produced as an output of another machine; the character $\perp$ may be overwritten by 0 or 1 in the future and it is impossible to recognize that a particular cell is left eternally as $\perp$. It also means that a machine cannot wait for the cell under a head to be filled because it may not be filled eternally. Therefore, our machine needs to have something other than the usual sequential access.

To solve this, we consider multiple heads and consider that the machine waits for multiple cells to be filled. Since at most one cell is left unfilled in Gray code representation, two heads are sufficient for our purpose. Therefore, we consider two heads $I1$ and $I2$ on an input tape $\mathcal{I}$, which move in the same way as output heads when they input characters. Note that the character $\perp$ cannot be recognized by our machine, which is the difference from the blank character B on a work tape and $\perp$ on input/output tape.

In order that the machine can continue working even when the cell under $I1$ or

$I2$ is $\perp$, we need, at each time, a rule applicable without reading from $I1$ and a rule applicable without reading from $I2$. This means that, if both head positions of an input tape are filled, we have two applicable rules. Thus, our machine comes to have a nondeterministic behavior.

To summarize, we have the following definition.

Definition 9 A *nondeterministic multihead Type 2 machine* (NM2-machine in short) with k inputs is a modification of a Type 2 machine in that:

1. Each input/output tape T has two heads $T1$ and $T2$.

2. When it executes a computational rule of the form

$$q, h_1(c_1), \ldots, h_j(c_j), w_1(d_1), \ldots, w_l(d_l) \Rightarrow$$
$$q', [o(c),] w'_1(d'_1), \ldots, w'_m(d'_m), M_1(w''_1), \ldots, M_n(w''_n)$$

   with h_i heads of different input tapes and o a head of the output tape, these heads are moved as follows. When h_i $(i = 1, \ldots, j)$ or o is a head $T1$ of a tape T, $T1$ is moved to the position of $T2$ and $T2$ is moved to the next cell, and when it is $T2$, the position of $T1$ is left unchanged and $T2$ is moved to the next cell.

The machine starts with the output tape filled with $\perp$, work tapes filled with B, the state set to q_0, the heads of work tapes located on the first cell, and $T1$ and $T2$ of an input/output tape T are located above the first and the second cell, respectively. At each step, the machine chooses one applicable rule and applies it. When more than one rules is applicable, only one is selected in a nondeterministic way.

Note 1. We can define a nondeterministic multihead Type 2 machine more generally so that it may have more than two heads. The computational power is changed because we can leave $n-1$ cells undefined when we have n heads.

Note 2. Here, we defined as if the full contents of the input tapes are given at the beginning. However, an input is usually generated as an output of another machine, and given incrementally.

Note 3. Though we have defined a machine only with Gray code input/output, it can be mixed with the input/output of an usual Type 2 machine in the following way. First, we extend an NM2-machine with a sequence $(Y_1, \ldots, Y_k, Y_0)$ indicating the input/output type. Here, Y_i is the character G indicating that it is a Gray code representation or Σ^ω for some alphabet Σ indicating an infinite sequence of Σ. Here, we suppose that Σ does not include $\perp$. When an NM2-machine has type $(Y_1, \ldots, Y_k, Y_0)$, we suppose that it has k input tapes with type Y_i $(i = 1, \ldots, k)$ and one output tape of type Y_0. When Y_i is not G, the corresponding tape has one head which moves to the right when it reads/writes a character. It can also be modified to allow input/output of finite sequences. In this case, we place Σ^* for the corresponding Y_i in the type. We call such an extension a generalized NM2-machine.

5 Gray code representation and computability of real numbers

As we have seen, an NM2-machine yields many different results to the same input. However, for the purpose of implementing a real-valued function, the output should be unique. Thus, we define as follows:

Definition 10 *Let M be an NM2-machine with k inputs. We say that M strongly produces τ to σ_i $(i = 1, \ldots, k)$ if, when M is started with the input tapes filled with σ_i, the output tape is filled with τ after an infinite run not depending on the way the rules are applied. More precisely, if $\tau[j] = c$ and c is not $\perp$, then $O[j]$, the j-th cell of the output tape O becomes c at some time, and if $\tau[j] = \perp$, then $O[j]$ will never be written a character. When M strongly produces the result for any input, we call that M is a strong NM2-machine.*

Thus, the meaning of non-deterministic computation here is that every computational path will produce the same result. This use of the word non-determinism is different from the usual one like non-deterministic Turing machine that at least one of the computational paths will produce a result. This property is called indeterminism in [Brattka 1998].

Definition 11 A partial function $f :\subseteq \mathcal{I}^k \to \mathcal{I}$ is Gray code computable if there is a NM2-machine M with k inputs which strongly produces $G(f(x_1, \ldots, x_k))$ to $G(x_1), \ldots, G(x_k)$ when $(x_1, \ldots, x_k) \in dom(f)$. In this case, we say that M computes f.

Suppose that ρ is some admissible representation. Then, since ρ is not injective, we have $\rho(p) = \rho(q)$ for some $p, q \in \Sigma^\omega$. Therefore, in order that a Type 2 machine M defines a real-valued function, it should satisfy $\rho(f_M(p)) = \rho(f_M(q))$ when $\rho(p) = \rho(q)$ for f_M the function on infinite sequences induced by M. One can find that this condition corresponds to the strictness condition of an NM2-machine.

From the multi-valuedness of an NM2-machine, one can naturally define the notion of a computable relation.

Definition 12 A relation $Q \subset \mathcal{I}^k \times \mathcal{I}$ is Gray code computable if there is a NM2-machine M with k inputs whose outputs to $G(x_1), \ldots, G(x_k)$ form a subset of $Q(x_1, \ldots, x_k)$ if $Q(x_1, \ldots, x_k)$ is not empty. Here, $Q(x_1, \ldots, x_k)$ is the set $\{z \in I | (x_1, \ldots, x_k, z) \in Q\}$.

6 Equivalence of the two computability notions

Now, we prove that Gray code computability is equivalent to the one induced by admissible representations with Type 2 machines. Before that, we show that the behavior of an NM2-machine can be simulated by a Type 2 machine. Since a Type

2 machine cannot skip a cell of an input/output tape, we consider another representation based on the Gray code.

Definition 13 An enumeration of $p \in \{0, 1, \perp\}^\omega$ is an infinite sequence r of pairs (n, s) such that $n \in \omega$, $s \in \{0, 1\}$, and $p[n] = s$ iff (n, s) is a member of r.

An enumeration can be expressed as an infinite sequence of characters, and is a representation of the unit open interval if we combine it with the Gray code representation.

Proposition 4 *Let M be an NM2-machine with k inputs which strongly produces τ to σ_i $(i = 1, \ldots, k)$. Then, there is a Type 2 Machine M' which, given enumerations of σ_i, outputs the enumeration of τ.*

Proof:
M' has, corresponding to the input tapes, work tapes T_i $(i = 1, \ldots, k)$ on which the Gray code is reconstructed. It also has a work tape T_0 corresponding to the output tape. We need work tapes to make a note of the head positions of the input/output tapes of M. M' repeats finding and executing an applicable rule of M using T_i $(i = 0, \ldots, k)$ instead of input/output tapes and outputs (n', s') when n'-th cell of T_0 is filled with s'. When there is no more applicable rule, it reads a pair (n, s) from one of the input tapes, fill the n-th cell of T_i with s, and repeat this process. ∎

Proposition 5 *If M is a NM2-machine with k inputs and N_i $(i = 1, \ldots, k)$ are NM2-machines with n inputs, then there is a NM2-machine $M \circ \langle N_1, \ldots, N_k \rangle$ with n inputs which satisfies the following condition: when N_i strongly produces τ_i to σ $(i = 1, \ldots, k)$ and M strongly produces τ to τ_i $(i = 1, \ldots, k)$, then $M \circ \langle N_1, \ldots, N_k \rangle$ strongly produces τ to σ.*

Proof: First, we consider the case $k = 1$. We write N for N_1. We use the input tapes of N as those of $M \circ N$ and the output tape of M as that of $M \circ N$. We use a work tape T with the alphabet $\{0, 1, B\}$ which connects the parts representing N and M, and work tapes to simulate the head movements of the input tape of M and that of the output tape of N. It is easy to change the rules of M and N so that M reads from T and N writes on T. We define the rules so that if both the parts representing M and N have applicable rules, then execute a rule coming from M.

When $k > 1$, we need to copy the input tapes onto work tapes so that it is shared by the parts representing $N_1, \ldots, N_k$. ∎

This proposition can easily be extended to generalized NM2-machines in the sense of Note 3 of Definition 9.

Theorem 6 *A function $f :\subseteq \mathcal{I}^k \to \mathcal{I}$ is Gray code computable iff it is $((\delta_{sbin})^k, \delta_{sbin})$-computable.*

Proof: As we will see in Section 8, the conversions between the Gray code representation and the signed binary representation can be expressed using NM2-machines. Suppose that M is a NM2-machine which computes f. By composing it with the NM2-machines we will define in Examples 1 and 2, we can form, by Proposition 5, a generalized NM2-machine which strongly produces a δ_{sbin}-representation of $f(x)$ when a δ_{sbin}-representation of x is given. Though this machine may still have a non-deterministic behavior, the type of this machine is $((\{-1,0,1\}^\omega)^k, \{-1,0,1\}^\omega)$, and therefore, one can easily simulate it in a deterministic way using an NM2-machine.

On the other hand, suppose that there is a Type 2 machine which $((\delta_{sbin})^k, \delta_{sbin})$-computes f. Since a Type 2 machine is a special case of a (generalized) NM2-machine, again, by composing the NM2-machines we well define in Example 1 and 2, we can form a NM2-machine which computes f. ∎

We can give another proof by showing the equivalence of the representations δ_{sbin} and the representation in Definition 13. Similarly, we can prove the following:

Theorem 7 *A relation $Q \subset \mathcal{I}^k \times \mathcal{I}$ is Gray code computable iff it is $((\delta_{sbin})^k, \delta_{sbin})$-computable.*

7 Topological Properties

In this section, we show that our embedding G preserves the topological structure.

Consider the space $P \subset \{0,1,\perp\}^\omega$ of finite and infinite outputs of NM2-machines. The states of output tapes after a finite time of execution are $\{0,1\}^*\perp^\omega$ or $\{0,1\}^*\perp 10^*\perp^\omega$. Therefore, we define $P = \{0,1\}^*\perp^\omega + \{0,1\}^*\perp 10^*\perp^\omega + \{0,1\}^\omega + \{0,1\}^*\perp 10^\omega - \{0,1\}^*0^\omega$.

Since teh character $\perp$ may be overwritten by 0 or 1, it is not appropriate to consider Cantor topology on P. Instead, we define the order structure $\perp < 0$ and $\perp < 1$ on our alphabet and consider the Scott topology of $\{0,1,\perp\}$, i.e. $\{\{\},\{0\},\{1\},\{0,1,\perp\}\}$, and then consider its product topology on $\{0,1,\perp\}^\omega$. The set $\{\uparrow(d\perp^\omega) \mid d \in \{0,1,\perp\}^*\}$ is a base of this topology and therefore, when we consider its subspace topology on P, then $\{\uparrow(d\perp^\omega) \mid d \in \{0,1\}^* + \{0,1\}^*\perp 10^*\}$ forms a base. Note that $d\perp^\omega$ is a state of an output tape we can observe in finite time, and $\uparrow(d\perp^\omega)$ is the set of states we may obtain in finite or infinite time after we observe $d\perp^\omega$. This means that the observation that open sets are finitely observable properties in [Smyth 1992] holds for our NM2-machine when we consider this topology.

As we have noted, a strong NM2-machine M induces a partial function f_M from $Q = \{0,1\}^\omega + \{0,1\}^*\perp 10^\omega$ to itself. This function becomes continuous for the same reason as in Proposition refcont1.

Since $Im(G) = Q - \{0,1\}^*0^\omega$ is the space of maximal elements of P, we consider the subspace topology on $Im(G)$. Then, $Im(G)$ has the base $\{\uparrow(d\perp^\omega)\cap Im(G) | \, d \in \{0,1\}^* + \{0,1\}^*\perp 10^*\}$. We consider their inverse image by G. When $d \in \{0,1\}^*$ and $e \in 0^*$, $G^{-1}(\uparrow(d\perp^\omega))$ ranges over open intervals of the form $m\times 2^i < x < (m+1)\times 2^i$, and $G^{-1}(\uparrow(d\perp 1e))$ ranges over open intervals of the form $m \times 2^{-i} - 2^{-(i+1)} < x <$

$m \times 2^{-i} + 2^{-(i+1)}$ for m and i integers. Since these open intervals form a base of the unit open interval $\mathcal{I}$, $\mathcal{I}$ and $Im(G)$ become homeomorphic through the function G.

As a direct consequence, we have the following:

Proposition 8 *A Gray code computable function $f : \mathcal{I}^k \to \mathcal{I}$ is continuous.*

The poset P is essentially the same as the cpo RD in [Gianantonio 1996a], which is composed from the signed binary representation. The difference is that RD has limit elements corresponding to $\{0,1\}^*0^\omega$ in Gray code. If we add such elements to our representation, P becomes a cpo and all the outputs of an NM2-machine becomes Gray code interpretations. Then, each dyadic number comes to have three representations $\{0,1\}^*\bot 10^\omega$, $\{0,1\}^*110^\omega$, and $\{0,1\}^*010^\omega$, and the interpretation of a finite sequence like $01\bot^\omega$ becomes a closed interval $[1/4, 1/2]$ instead of an open interval $(1/4, 1/2)$.

As an application of our representation, we give a simple proof of Theorem 3.4 of [Weihrauch 1995], which says that there is no effective enumeration of computable real numbers.

Theorem 9 *When $(x_i)_{i\in\omega}$ is a computable sequence, then a computable number x with $x \neq x_i$ for all $i \in \omega$ can be determined.*

Proof: Let s_i be the Gray code representation of x_i and suppose that there is a generalized NM2-machine M of type $(\{0,1\}^*, G)$ which strongly produces s_i when the binary code of i is given. M can be made deterministic because it does not have an input type G. Therefore, the order the output tape is filled is fixed. Since s_i is a representation of a Gray code, one of $s_i[2i]$ or $s_i[2i+1]$ is written in a finite time. When $s_i[2i]$ is written first, we put $t[2i] = \textit{not}\ s_i[2i]$ and $t[2i+1] = s_i[2i]$. When $s_i[2i+1]$ is written first, we put $t[2i] = s_i[2i+1]$ and $t[2i+1] = \textit{not}\ s_i[2i+1]$. Then, the resulting sequence t is computable and is not equal to s_i. Therefore, $G^{-1}(t)$ is not equal to x_i because of the injectivity of the representation. ∎

8 Conversions with the signed binary representation

As an example of an NM2-machine, we consider conversions between the Gray code and the signed binary representation. Recall that the signed binary representation of a number $x \in \mathcal{I}$ is given as a sequence $1\ xs$ with xs an infinite sequence of $\{0, 1, \overline{1}\}$. In this section, we consider xs as the signed binary representation and consider conversions between such an xs and the Gray code representation by a generalized NM2-machine.

Since finite prefixes express the same set of intervals in both representation, the conversions become simple automaton-like algorithms which do not use work tapes.

Example 1 *Conversion from signed binary representation to Gray code representation.* It has the type $(\{\overline{1},0,1\}^\omega, G)$. We call the input tape $\mathcal{I}$ and the output tape O. It has four states (i,j) $(i,j \in \{0,1\})$ with $(0,0)$ the initial state, and 12 rules:

$$
\begin{array}{ll}
(0,0), I(1) \Rightarrow (1,0), O1(1); & (0,1), I(1) \Rightarrow (1,0), O1(1); \\
(0,0), I(\overline{1}) \Rightarrow (0,0), O1(0); & (0,1), I(\overline{1}) \Rightarrow (1,0), O1(0); \\
(0,0), I(0) \Rightarrow (0,1), O2(1); & (0,1), I(0) \Rightarrow (0,1), O2(0); \\
(1,0), I(1) \Rightarrow (1,0), O1(0); & (1,1), I(1) \Rightarrow (1,0), O1(0); \\
(1,0), I(\overline{1}) \Rightarrow (0,0), O1(1); & (1,1), I(\overline{1}) \Rightarrow (1,0), O1(1); \\
(1,0), I(0) \Rightarrow (1,1), O2(1); & (1,1), I(0) \Rightarrow (1,1), O2(0);
\end{array}
$$

In order to express it simpler, we use the notation of the functional language Haskell as follows:

$$
\begin{array}{l}
\texttt{s2g(xs)} = \texttt{stog0(xs},0,0) \\
\texttt{stog0}(1:\texttt{xs},0,0) = 1:\texttt{stog0(xs},1,0) \\
\texttt{stog0}(\overline{1}:\texttt{xs},0,0) = 0:\texttt{stog0(xs},0,0) \\
\texttt{stog0}(0:\texttt{xs},0,0) = \texttt{c}:1:\texttt{ds} \quad \texttt{where c}:\texttt{ds} = \texttt{stog0(xs},0,1) \\
\texttt{stog0}(0:\texttt{xs},0,1) = \texttt{c}:0:\texttt{ds} \quad \texttt{where c}:\texttt{ds} = \texttt{stog0(xs},0,1) \\
\ldots
\end{array}
$$

It is clear that the behavior of an NM2-machine can be expressed using this notation with the state and the contents of the work tapes before and after the head positions passed as additional arguments. In the program `stog0`, the states are used to invert the output: the result of $\texttt{stog0}(xs,1,0)$ is that of $\texttt{stog0}(xs,0,0)$ with the first character inverted, and the result of $\texttt{stog0}(xs,0,1)$ is that of $\texttt{stog0}(xs,0,0)$ with the second character inverted. Therefore, we can simplify the above program as follows:

$$
\begin{array}{l}
\texttt{stog}(1:\texttt{xs}) = 1:\texttt{nh (stog xs)} \\
\texttt{stog}(\overline{1}:\texttt{xs}) = 0:\texttt{stog xs} \\
\texttt{stog}(0:\texttt{xs}) = \texttt{c}:1:\texttt{nh ds} \quad \texttt{where c}:\texttt{ds} = \texttt{stog xs}
\end{array}
$$

Here, `nh` is the function to invert the first element of an infinite list. That is,

```
not 0 = 1
not 1 = 0
nh (s:ds) = not s:ds
```

The behavior of `stog` with input `0:xs` is given in Figure 1. Here, a small circle means to invert the output.

The program `stog` is a correct Haskell program and it works on a Haskell system. However, if we evaluate `stog([0,0..])`, there will be no output because it tries to calculate the first digit, which is $\perp$. Of course, `tail(stog([0,0..]))` produces the answer `[1, 0, 0, 0, ...`.

Next, we consider the inverse conversion, which is an example of Gray code input.

Example 2 *Conversion from Gray code to signed binary representation.* It has the type $(G, \{\overline{1},0,1\}^\omega)$.

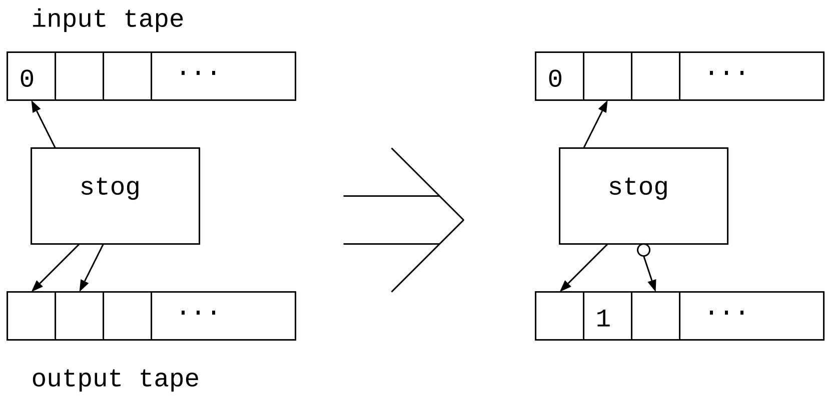

Figure 3: The behavior of stog NM2-machine when it reads 0

$$
\begin{aligned}
&\texttt{gtos}(1:\texttt{xs}) = 1:\texttt{gtos}(\texttt{nh xs})\\
&\texttt{gtos}(0:\texttt{xs}) = \overline{1}:\texttt{gtos}(\texttt{xs})\\
&\texttt{gtos}(\texttt{c}:1:\texttt{xs}) = 0:\texttt{gtos}(\texttt{c}:\texttt{nh xs})
\end{aligned}
$$

In this case, nondeterminism occurs and it yields many different results. The results are actually all signed binary representations of the same number. This is also a correct Haskell program. However, it fails to calculate, for example, `gtos(stog([0,0..]))` because `gtos`, from the first two rules, starts its non-terminating calculation to pattern match the head of the argument and fails to use the third rule. This is a limitation of the use of an existing functional language, and an implementation of such a calculation with multiple heads is a future work.

Looking at these examples, one may think it difficult to write such a program. However, it is based on the recursive structures Gray code has and it is not difficult to write such a program. One can see from Figure 3 the following three recursive equations:

$$
\begin{aligned}
[\![0:p]\!] \quad &= [\![p]\!] * 0.5,\\
[\![1:p]\!] \quad &= 0.5 + [\![\texttt{nh}\ p]\!] * 0.5 (= 1.0 - [\![p]\!] * 0.5]\!]),\\
[\![c:1:p]\!] &= 0.25 + [\![c:(\texttt{nh}\ p)]\!] * 0.5.
\end{aligned} \tag{1}
$$

Here, $[\![p]\!]$ denotes $G^{-1}(p)$. The first equation shows that on the interval with the first bit 0, i.e. the left half of Figure 2, the remaining bits form a $1/2$ reduction of Figure 2. The second equation shows that on the interval with the first bit 1, i.e. the right half of Figure 2, the remaining bits with the first bit inverted form a $1/2$ reduction of Figure 2, and if we use the equation with parenthesis, we can also state that the remaining bits form the reversal of Figure 2. These two equations characterize Figure 2. One interesting fact about this representation is that we also have the third equation, which shows that on the interval with the second bit 1, i.e. the middle half of Figure 2, the remaining bits with the second bit inverted form a $1/2$ reduction of Figure 2.

From Equations (1), we have the following recursive scheme.

$$\begin{aligned}
f(0:p) &= g_1(f(p)),\\
f(1:p) &= g_2(f(\mathtt{nh}\ p)),\\
f(a:1:p) &= g_3(f(a:\mathtt{nh}\ p)).
\end{aligned}$$

Here, g_1 is a function to calculate $f(x)$ from $f(x/2)$, g_2 is a function to calculate $f(x)$ from $f(x/2+1/2)$, g_3 is a function to calculate $f(x)$ from $f(x/2+1/4)$. `gtos` is derived immediately from this scheme.

On the other hand, Equations (1) can be rewritten as follows:

$$\begin{aligned}
G(x/2) &= 0:G(x),\\
G(1/2+x/2) &= 1:(\mathtt{nh}\ G(x)),\\
G(1/4+x/2) &= c:1:(\mathtt{nh}\ p) \quad \text{where} c:p=G(x).
\end{aligned}$$

`stog` uses this scheme to calculate the gray code output. These recursive schemes are used to derive the algorithm for addition in the next section.

9 Some simple algorithms in Gray code

We write some algorithms with respect to Gray code.

Example 3 *Multiplication and division by 2.* They are simple shifting operations.

```
mul2 (0:s) = s   (suppose that the input is 0 < x < 1/2)
div2 xs = 0:xs
```

Example 4 *The complement $x \mapsto 1-x$.* It is a simple operation to invert the first digit, i.e.,the `nh` function in Example 1. Note that with the usual binary representation and the signed binary representation, we need to invert all the bits to calculate $1-x$ and thus this operation needs to be defined recursively. We can also see that the reversal operation $(x \mapsto k/2^n - x)$ with respect to a dyadic number $k/2^{n+1}$ for $(k-1)/2^{n+1} < x < (k+1)/2^{n+1}$ is a operation to invert one digit.

Example 5 *Shifting $x \mapsto x + 1/2$ $(0 < x < 1/2)$* . Addition with a dyadic number is nothing but two continuous reversal operations over dyadic numbers. For the case of $+1/2$, the first axis is $1/2$ and the second axis is $3/4$. Therefore, the function

```
AddOneOfTwo x:y:xs = (n x):(n y):xs
```

operates as $x \mapsto x + 1/2$ if $0 < x < 1/2$ and as $x \mapsto x - 1/2$ if $1/2 < x < 1$.

Example 6 *Addition*

We consider an addition $x+y$ with $0 < x, y < 1$. Since the result is in $(0,2)$, we consider a function to calculate $(x+y)/2$, instead.

```
pl (0:as) (0:bs) = 0:pl as bs
pl (1:as) (1:bs) =1:pl as bs
pl (0:as) (1:bs) = c:1:nh cs where c:cs = pl as (nh bs)
pl (1:as) (0:bs) = c:1:nh cs where c:cs = pl (nh as) bs
pl (a:1:as) (b:1:bs) = c:1:nh cs where c:cs = pl (a:nh as) (b:nh bs)
pl (a:1:0:as) (0:0:bs) = 0:pl (a:1:as) (1:nh bs)
pl (a:1:0:as) (1:0:bs) = 1:pl (not a:1:as) (1:nh bs)
pl (a:1:0:as) (0:b:1:bs) = 0:1:pl (not a:nh as) (not b:nh bs)
pl (a:1:0:as) (1:b:1:bs) = 1:1:pl (a:nh as) (not b:nh bs)
pl (0:0:as) (b:1:0:bs) = 0:pl (1:nh as) (b:1:bs)
pl (1:0:as) (b:1:0:bs) = 1:pl (1:nh as) (not b:1:bs)
pl (0:a:1:as) (b:1:0:bs) = 0:1:pl (not a:nh as) (not b:nh bs)
pl (1:a:1:as) (b:1:0:bs) = 1:1:pl (not a:nh as) (b:nh bs)
```

To calculate the sum with respect to the signed binary representation, we, in general, need to look ahead two characters. As is shown in this example, it is also the case with the Gray code representation. Since it does not have redundancy, we can reduce the number of rules compared with the program wrriten in the same way, only with pattern matching and not using arithematic operations, with the signed binary representation, which needs to have 25 rules.

10 Conclusion and future works

We have defined an injective representation of $\mathcal{I} = (0, 1)$ using Gray code, and introduced a nondeterministic multihead Type 2 machine as a machine which can input/output this representation. [Escardo 1997] has shown that the redundancy of the signed binary representation is not an obstacle in expressing algorithms, and in this sense, injectivity may not contribute to practical programming methodology. However, injectivity means that real numbers are embedded into the set of infinite sequences, and thus we can consider our machine working directly on real numbers.

We discuss how our representation of the unit open interval $\mathcal{I} = (0, 1)$ can be extended to the set of real numbers $\mathcal{R}$. First, by using the first digit as the sign bit: 1 if positive, 0 if negative, and is left as $\perp$ if the number is zero, we can extend our representation to the interval $(-1, 1)$. We can also extend it to $(-2^k, 2^k)$ for arbitrary k by assuming that there is a decimal point after the k-th digit. However, there seems to be no direct extension of this representation to the whole real number without losing injectivity and without losing the simplicity of the algorithms in 8 and 8.

One possibility is to introduce the character "." denoting the decimal point and express the number as we do with the decimal expression. In order to allow an expression starting with $\perp$ (.i.e. integers other than 0), we also need to allow a expression starting with 0 so that we have the same computability notion as admissible representations. However, then, we loose the injectivitiy of the representation. We also have the same kind of difficulty if we represent a real number like the floating point expression: as a pair of a number indicating the decimal point and a gray code representation on $(-1, 1)$.

Another possibility is to use an embedding of $\mathcal{R}$ to $(-1, 1)$, such as the function $f(x) = 2 * arctan(x)/pi$. It is known that this function is computable, and therefore, there are generalized NM2-machines which convert between a gray code in $(-1, 1)$ and a signed binary representation in $\mathcal{R}$. Therefore, we can define our new representation $G'(x) = G(f(x))$ $(x \in \mathcal{R})$. It is clear that this representation is injective and that it induces the same computability notion as admissible representations on $\mathcal{R}$. However, then, algorithms for simple operations in 8 and 8 become complicated.

As for implementation, we consider three different ways of implementing NM2-machines.

One is to extend pattern matching of lazy functional languages as we explained in Example 2. The definition of `gtos` in Example 2 had the form

```
gtos(1 : xs) = ...
gtos(0 : xs) = ...
gtos(c : 1 : xs) = ...
```

Therefore, in order to evaluate `gtos(c:xs)`, we need to evaluate both `c` and `xs` in parallel so that the third rule is used when the evaluation of `c` does not terminate. In this sense, one may think that we need parallel execution for our purpose. However it seems that we do not need it in our case. The argument of `gtos` always represents a real number, and therefore, the author conjectures that it does not happen that the two parts `c` and `xs` are independant but they always share the same head redex.

The second one is to use the continuity of a computable function and omit the third rule of `gtos` and 5-th and later rules of `pl`. Actually, if we assume the continuity of a function, the first two lines of `gtos` and the first 4 lines of `pl` determine the whole behavior of these functions to all the arguments. Therefore, we can consider execution based only on these lines. That is, when the first cell is skipped and the value 1 is given to the second cell, one can split the computation into two processes: computations supposing the first digit as '1' and '0', and execute them in parallel. When the inputs from the second head are obtained, then they are sent to these two processes, and the results of these two processes are merged to produce the output. Since our representation is injective, we only need to make bitwise comparison of the outputs of these two processes in this merge process, and it outputs those bits on which the two processes agree. When an input is given from the first head, then the other execution is killed. This way of implementation can be considered as a variant of REAL PCF[Escardo 1996]. We have implemented addition in this way using Java multi threads.

The third one is hardware implementation. Since our representation uses two characters plus $\perp$, it is not hard to implement it on hardware. Since FPGA (Field Programmable Gate Array) is becoming popular, hardware design is within our reach.

Acknowledgment

The author thanks Andreas Knobel for many interesting and illuminating discussions. He also thanks Mariko Yasugi and Computable Analysis group in Kyoto for

many discussions and comments.

References

[Amalraj *et al.* 1990] D. J. Amalraj, N. Sundararajan, and G. Dhar. (1990) A Data Structure based on Gray Code Encoding for Graphics and Image Processing. SPIE Vol. 1349 Applications of Digital Image Processing XIII.

[Boehm *et al.* 1986] H. J. Boehm, R. Cartwright, M. Riggle, and M. J. O'Donnel. (1986) Exact real arithmetic: A case study in higher order programming. In *ACM Symposium on Lisp and Functional Programming.*

[Brattka 1998] Vasco Brattka. (1998) Recursive and Computable Operations over Topological Structures. *Dissertation, Fern Universitat,.*

[Dewney 1993] A. K. Dewdney. (1993) The New Turing Omnibus, Computer Science Press. Computer Science Press.

[Edalat and Potts 1997] A. Edalat and P. J. Potts. (1997) A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, **6**, 1997.

[Escardo 1996] Martin Escardo. (1996) PCF extended with real numbers. *Theoretical Computer Science*, **162(1)**, 79–115.

[Escardo 1997] Martin Escardo. (1997) Effective and sequential definition by cases on the reals via infinite signed-digit numerals. In *Proc. Third Workshop on Computation and Approximation*, ENTCS Vol. 13.

[Gianantonio 1996a] Pietro Di Gianantonio. (1996) Real Number Computability and Domain Theory. *Information and Computation*,**127**, 11–25.

[Gianantonio 1996b] Pietro Di Gianantonio. (1997) A golden ratio notation for the real numbers. *Technical Report CS-R9602*, CWI Amsterdam.

[Hata and Yamaguti 1984] Masayoshi Hata and Masaya Yamaguti. (1984) The Takagi function and its generalization. *Japan J. Appl. Math.*, no. 1, 183-199.

[Pour-El and Richards 1989] Marian B. Pour-El and J. Ian Richards. (1989) Computability in Analysis and Physics. Springer, Berlin.

[Smyth 1992] M. B. Smyth. (1992) Topology. In *S. Abramsky, D. M. Gabbay, and T.S.E. Maibaum, editors, Handbook of Logic in Computer Science, volume 1*, 641–761. Clarendon Press, Oxford.

[Turing 1936] Alan M. Turing. On computable real numbers, with an application to the Entscheidungsproblem. In *Proc. of the London Mathematical Society* (2) **42**, 230–265 (1936), and (2) **43**, 544–546 (1937).

[Vuillemin 1990] J. Vuillemin. (1990) Exact real computer arithmetic with continued fractions. In *IEEE Transactions on Computers*, **39(8)** 1087–1105.

[Weihrauch 1985] K. Weihrauch. (1985) Type 2 recursion theory. *Theoretical Computer Science,* **38**, 17–33.

[Weihrauch 1995] K. Weihrauch. (1995) A simple introduction to computable analysis. Technical Report 171-7/1995, FernUniversiat.