Real Number Computation through Gray Code Embedding

Hideki Tsuiki

Division of Mathematics,
Faculty of Integrated Human Studies,
Kyoto University, Kyoto 606, Japan
tsuiki@i.h.kyoto-u.ac.jp

Abstract

We propose an embedding G of the unit open interval to the set $\{0,1\}^{\omega}_{\perp,1}$ of infinite sequences of $\{0,1\}$ with at most one undefined element. This embedding is based on Gray code and it is a topological embedding with a natural topology on $\{0,1\}^{\omega}_{\perp,1}$. We also define a machine called an IM2 machine (indeterministic multihead type 2 machine) which input/output sequences in $\{0,1\}^{\omega}_{\perp,1}$, and show that the computability notion induced on real functions through the embedding G is equivalent to the one induced by the signed digit representation and Type-2 machines. We also show that basic algorithms can be expressed naturally with respect to this embedding.

1 Introduction

One of the ways of defining computability of a real function is by representing a real number x as an infinite sequence called a name of x, and defining the computability of a function by the existence of a machine, called a Type-2 machine, which inputs and outputs the names one-way from left to right. This notion of computability dates back to Turing[Tur36], and is the basis of effective analysis [Wei85,Wei00].

This notion of computability depends on the choice of representation we use, and signed digit representation and equivalent ones such as the Cauchy representation and the shrinking interval representation are most commonly used; they have the property that every arbitrarily small rational interval including x can be obtained from a finite prefix of a name of x, and therefore induces computability notion that a function f is computable if there is a machine which can output arbitrary good approximation information of $f(x)$ as a rational interval when arbitrary good approximation information of x as a rational interval is given. The naturality of this computability notion is also justified by the fact that it coincides with those de-

Preprint submitted to Elsevier Science *18 March 2006*

fined through many other approaches such as Grzegorczyk's ([Grz57]), Pour-El and Richards ([PER89]), and domain theoretic approaches([ES98], [Gia99]).

One of the properties of these representations is that they are not injective [Wei00]. More precisely, uncountably many real numbers have infinitely many names with respect to representations equivalent to the signed digit representation [BH00]. This kind of redundancy is considered essential in many approaches to exact real arithmetic [BCRO86,EP97,Gia96,Gia97,Vui90].

Thus, computability of a real function is defined in two steps: first the computability of functions over infinite sequences is defined using Type-2 machines, and then it is connected with the computability of real functions by representations. The redundancy of representations means that we cannot define the computability of a real function more directly by considering an embedding of real numbers into the set of infinite sequences on which a Type-2 machine operates. In this paper, we consider such a direct definition by extending the notion of infinite sequences and modifying the notion of computation on infinite sequences.

Our embedding, called the Gray code embedding, is based on the Gray code expansion, which is another binary expansion of real numbers. The target of this embedding is the set $\{0,1\}^\omega_{\perp,1}$ of infinite sequences of $\{0,1\}$ in which at most one $\perp$, which means undefinedness, is allowed. We define the embbeding G of the unit open interval $\mathcal{I}$, and then explain how it can be extended to the whole real line in the final section. $\{0,1\}^\omega_{\perp,1}$ has a natural topological structure as a subspace of $\{0,1,\perp\}^\omega$. We show that G is a topological embedding from $\mathcal{I}$ to the space $\{0,1\}^\omega_{\perp,1}$.

Because of the existence of $\perp$, a machine cannot have sequential access to inputs and outputs. However, because $\perp$ appears only at most once, we can deal with it by putting two heads on a tape and by allowing indeterministic behavior to a machine. We call such a machine Indeterministic Multihead Type 2 machine (IM2-machine for short). Here, indeterministic computation means that there are many computational paths which will produce valid results [She75,Bra98]. Thus, we define computation over $\{0,1\}^\omega_{\perp,1}$ using IM2-machines, and consider the induced computational notion on $\mathcal{I}$ through the embedding G. We show that this computational notion is equivalent to the one induced by the signed digit representation and Type-2 machines.

We also show how basic algorithms like addition can be expressed with this 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.

We introduce Gray code embedding in Section 2 and an IM2 machine in Section 3. Then, we define the Gray code computability of real functions in Section 4, and show that it is equivalent to the computability induced by the signed digit representation and Type 2 machines in Section 5. In Section 6, we study topological structure.

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

Fig. 1. Binary code and Gray code of integers

In Section 7 and 8, we consider basic algorithms with respect to this embedding. We will discuss how this embedding can be extended to $\mathcal{R}$, give some experimental implementations, and give conclusion in Section 9.

Notation: Let Σ be an alphabet which does not include $\perp$. We write Σ^* for the set of finite sequences of Σ, Σ^ω for the set of infinite sequences of Σ, and $\Sigma^\omega_{\perp,n}(n = 1, 2, \ldots)$ for the set of infinite sequences of Σ in which at most n instances of the undefinedness character $\perp$ are allowed to exist. We write $f :\subseteq X \to Y$ when f is a partial function from X to Y, and $F :\subseteq X \rightrightarrows Y$ when F is a multi-valued function from X to Y, that is, F is a subset of $X \times Y$ considered as a partial function from X to the power set of Y. We call a number of the form $m \times 2^{-n}$ for integers m and n a dyadic number.

2 Gray Code Embedding

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 [ASD90] and finding minimal digital circuits [Dew93].

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 [HJ92] as follows:

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

This `conv` function has type `[Int] -> [Int]`, where `[Int]` is the Haskell type of (possibly infinite) list of integers. `a:b` means the list composed of `a` as the head and `b` as the tail, `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) = [],
```

with `[]` the empty list.

We will extend this coding to real numbers. Since the function `conv` is applicable 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 expansion of real numbers in the unit interval $\mathcal{I} = (0, 1)$ is visualized in Figure 2. Here, a horizontal line means that the corresponding bit has value 1 on the line and value 0 otherwise. This figure has a fine fractal structure and shows symmetricity of bits greater than n at every dyadic number $m \times 2^{-n}$.

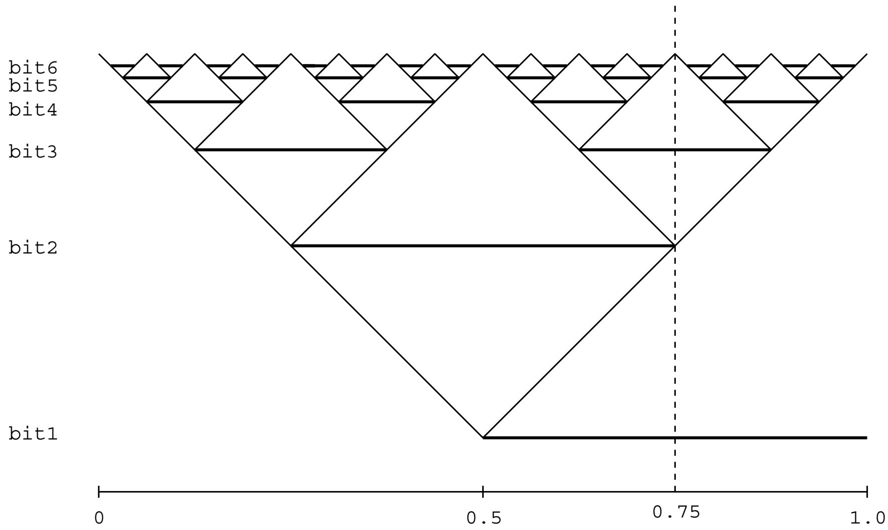

Fig. 2. Gray code of real numbers

In the usual binary expansion, we have two expansions for dyadic numbers. For example, $3/4$ can be expressed as 0.110000... and also as 0.101111.... This is also the case for the Gray code expansion. For example, by applying `conv` to these sequences, we have the two sequences 0.101000... and 0.111000... 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 $0.1\perp1000...$ as the unique representation of $3/4$. Note that the sequence after the bit where they differ is always 1000.... Thus, we define Gray code embedding of $\mathcal{I}$ as a modification of the Gray code expansion in that a dyadic number is represented as $s\perp1000...$ with $s \in \{0,1\}^*$.d

Definition 1 The Gray code embedding of the unit open interval $\mathcal{I}$ is an injective function G from $\mathcal{I}$ to $\Sigma^{\omega}_{\perp,1}$ which maps x to an infinite sequence $a_0 a_1 \ldots$ 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, $a_i = 0$ if the same holds for an even number m, and $a_i = \perp$ if $x = m \times 2^{-i} - 2^{-(i+1)}$ for some integer m. We call $G(x)$ the modified Gray code expansion of x, or simply Gray code of x.

When $G(x) = a_0 a_1 \ldots$, a_0 is $1, \perp$, or 0, according as x is bigger than, equal to, or less than $1/2$. The tail function which maps x to $G^{-1}(a_1 a_2 \ldots)$ denotes the so-called tent map

$$f(x) = \begin{cases} 2x & (x \leq 1/2) \\ 2(1-x) & (x > 1/2) \end{cases}$$

It is in contrast to the binary expansion in that the tail function of the binary

expansion denotes the function

$$f(x) = \begin{cases} 2x & (x \le 1/2) \\ 2x - 1 & (x > 1/2) \end{cases}$$

Note that Gray code expansion coincides with the itinerary by the tent map which is essential for symbolic dynamical systems [HY84].

3 Indeterministic Multihead Type 2 Machine

Consider calculating a real number x $(0 < x < 1)$ as the limit of approximations and output the result as the modified Gray code expansion. More precisely, we consider a calculation which produces shrinking intervals (r_n, s_n) $(n = 0, 1, \ldots)$ successively so that $\lim_{n\to\infty} s_n = \lim_{n\to\infty} r_n = x$.

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 that $1/4 < x < 3/4$, we can skip the first and write 1 as the second digit, and when we know that $3/8 < x < 5/8$, 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 of x 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 define $\perp$ as the "blank character" of the output tape and consider that the output tape is filled with $\perp$ at the beginning. 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 $x < 1/2$ 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 $\mathrm{H}_1(O)$ and $\mathrm{H}_2(O)$ on an output tape O which move automatically after an output. At the beginning, $\mathrm{H}_1(O)$ and $\mathrm{H}_2(O)$ are located above the first and the second cell, respectively. After an output from $\mathrm{H}_2(O)$, $\mathrm{H}_2(O)$ is moved to the next cell, and after an output from $\mathrm{H}_1(O)$, $\mathrm{H}_1(O)$ is moved to the position of $\mathrm{H}_2(O)$ and $\mathrm{H}_2(O)$ is moved to the next cell. Thus, in order to fill the output tape as $\perp\perp\perp\perp\perp\ldots \to 0\perp\perp\perp\perp\ldots \to 0\perp1\perp\perp\ldots \to 0\perp10\perp\ldots \to 0110\perp\ldots \to 01101\ldots$, we output $\mathrm{H}_1(O)(0), \mathrm{H}_2(O)(1), \mathrm{H}_2(O)(0), \mathrm{H}_1(O)(1), \mathrm{H}_1(O)(1)$. Here, $H(j)$ (H is $\mathrm{H}_1(O)$ or $\mathrm{H}_2(O)$ and $j = 0, 1$) means to output j from H . With this head movement rule, each cell is filled at most once and a cell is not filled eternally

only when $\mathrm{H}_1(O)$ is located on that cell and output is made solely from $\mathrm{H}_2(O)$. When $\mathrm{H}_1(O)$ and $\mathrm{H}_2(O)$ 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, $\mathrm{H}_1(O)$ and $\mathrm{H}_2(O)$ 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 modified Gray code expansion of a real number. We define our input mechanism so that finite input contains only approximation information. Therefore, our machine should not recognize that the cell under the head is $\perp$, because the character $\perp$ with its preceding prefix specifies the number exactly. 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 $\perp$. 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, two heads are sufficient for our purpose. Therefore, we consider two heads $\mathrm{H}_1(I)$ and $\mathrm{H}_2(I)$ on an input tape 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, unlike the blank character B used by a Turing machine.

Thus, we define a machine which has two heads on each input/output tape. Though we have explaind this idea based on the modified Gray code expansion, this machine can input/output sequences in $\Sigma^{\omega}_{\perp,1}$ generally. In order to give the same computational power as a Turing machine, we consider a state machine controlled by a set of computational rules, which has some ordinary work tapes in addition to the input/output tapes.

In order that the machine can continue working even when the cell under $\mathrm{H}_1(I)$ or $\mathrm{H}_2(I)$ for an input tape I is $\perp$, we need, at each time, a rule applicable only reading from $\mathrm{H}_1(I)$ or $\mathrm{H}_2(I)$. Therefore, the condition part of each rule should not include input from both $\mathrm{H}_1(I)$ and $\mathrm{H}_2(I)$. This also means that, if both head positions of an input tape are filled, we may have more than one applicable rules. Since a machine may execute both rules, both computational paths should produce valid results.

To summarize, we have the following definition.

Definition 2 Let Σ be the input/output alphabet. Let Γ be the work-tape alphabet which includes a blank character B. An *indeterministic multihead Type 2 machine* (IM2-machine in short) with k inputs is composed of the following:

(i) k input tapes named $I_1, I_2, \ldots, I_k$ and one output tape named O. Each tape T

has two heads $\mathrm{H}_1(T)$ and $\mathrm{H}_2(T)$,

(ii) several work tapes with one head,

(iii) a finite set Q of states with one initial state $q_0 \in Q$,

(iv) computational rules of the following form:

$$q, i_1(c_1), \ldots, i_r(c_r), w_1(d_1), \ldots, w_s(d_s) \Rightarrow \\ q', o(c), w'_1(d'_1), \ldots, w'_t(d'_t), M_1(w''_1), \ldots, M_u(w''_u).$$

Here, q and q' are states in Q, i_j are heads of different input tapes, o is a head of the output tape, w_j, w'_j, and w''_j are heads of work tapes, c_j $(j = 1, \ldots, r)$ and c are characters from Σ, d_j and d'_j are characters from Γ, and M_j $(j = 1, \ldots, u)$ are '+' 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 i_j $(j = 1, \ldots, r)$ and w_e $(e = 1, \ldots, s)$ are c_j and d_e, respectively, then change the state to q', write the characters c and d'_j $(j = 1, \ldots, t)$ under the heads o and w'_j, respectively, move the heads w''_j $(j = 1, \ldots, u)$ forward or backward depending on whether M_j = '+' or '−', and move the heads of input/output tapes as follows. For each i_j $(j = 1, \ldots, r)$ and o, when it is a head $\mathrm{H}_1(T)$ of a tape T, $\mathrm{H}_1(T)$ is moved to the position of $\mathrm{H}_2(T)$ and $\mathrm{H}_2(T)$ is moved to the next cell, and when it is $\mathrm{H}_2(T)$, the position of $\mathrm{H}_1(T)$ is left unchanged and $\mathrm{H}_2(T)$ 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 the heads $\mathrm{H}_1(T)$ and $\mathrm{H}_2(T)$ 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 an indeterministic multihead Type 2 machine more generally in that each input/output tape may have $n+1$ heads $\mathrm{H}_1(T), \ldots, \mathrm{H}_{n+1}(T)$ and it can input/output sequences in $\Sigma^\omega_{\perp,n}$ $(n = 0, 1, \ldots)$. We define the head movements after an input/output operation as follows. If input/output is made from $\mathrm{H}_l(T)$ $(l \leq n)$ then $\mathrm{H}_j(T)(l \leq j \leq n)$ are moved to the position of $\mathrm{H}_{j+1}(T)$ and $\mathrm{H}_{n+1}(T)$ is moved to the next cell. If input/output is made from $\mathrm{H}_{n+1}(T)$ then $\mathrm{H}_{n+1}(T)$ is moved to the next cell. Note that when $n = 0$, $\Sigma^\omega_{\perp,0}$ is nothing but Σ^ω and a tape has only one head which moves to the next cell after an input/output.

Note 2. Here, we acted as if the full contents of the input tapes were given at the beginning. However, an input is usually generated as an output of another machine, and given incrementally. In this case, the machine behaves like this: it repeats executing an applicable rule until no rule is applicable, and waits for input tapes to be filled so that one of the rules become applicable, and repeats this process indefinitely.

Note 3. A machine can have different input/output types on the tapes. The input/output types we consider are $\Sigma^\omega_{\perp,n}$ $(n \geq 0)$ and Σ^*, where we may write Σ^ω for

$\Sigma^{\omega}_{\perp,0}$. We extend an IM2-machine with a sequence $(Y_1, \ldots, Y_k, Y_0)$ indicating 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 $\Sigma^{\omega}_{\perp,n}$, the corresponding tape has the properties written in Note 1. When Y_i is Σ^*, the corresponding tape has the alphabet $\Sigma \cup \{B\}$ and it has one head which moves to the next cell when it reads/writes a character. In this case, the blank cells are initialized with B. In addition, when Y_0 is Σ^*, we consider that the machine has a halting state at which the machine stops execution.

4 Gray Code Computability of Real Functions

As we have seen, an IM2-machine has a nondeterministic behavior and thus it has many possible outputs to the same input. Therefore, we consider that an IM2-machine computes a multi-valued function. Note that multi-valued functions appear naturally when we consider computation over real numbers [Bra98].

Definition 3 *An IM2-machine M with k inputs realizes a multi-valued function $F :\subseteq {\Sigma^{\omega}_{\perp,1}}^k \rightrightarrows \Sigma^{\omega}_{\perp,1}$ if all the computational paths M have with the input tapes filled with $(p_1, \ldots, p_k) \in dom(F)$ produce infinite outputs, and the set of outputs forms a subset of $F(p_1, \ldots, p_k)$. We say that F is IM2-computable when it is realized by some IM2-machine.*

This definition can be generalized to a multi-valued function $F :\subseteq Y_1 \times \ldots \times Y_k \rightrightarrows Y_0$ for the case Y_i is Σ^* or $\Sigma^{\omega}_{\perp,n}$ $(n = 0, 1, \ldots)$.

Note that our nondeterministic computation is different from nondeterminism used, for example, in a non-deterministic Turing machine; a non-deterministic Turing machine accepts a word when one of the computational paths accepts the word, whereas all the computational paths should produce valid results in our machine. To distinguish, we use the word indeterminism instead of nondeterminism following [She75] and [Bra98].

Definition 4 A multi-valued function $F :\subseteq \mathcal{I}^k \rightrightarrows \mathcal{I}$ is realized by M if $G(F)$ is realized by M. We say that F is Gray-code-computable if $G \circ F \circ G^{-1}$ is IM2-computable.

Definition 5 A partial function $f :\subseteq \mathcal{I}^k \to \mathcal{I}$ is Gray-code-computable if it is computable as a multi-valued function.

5 Equivalence to the Computability induced by the Signed Digit Representation

Now, we prove that Gray code computability is equivalent to the computability induced by a Type-2 machine and the (restricted) signed digit representation.

Definition 6 *A Type-2 machine is an IM2-machine whose type includes only Σ^ω and Σ^*, and whose computational rule is deterministic.*

This definition is equivalent to the one in [Wei00].

Proposition 1 *Let Y_i be Σ^ω or Σ^* $(i = 0, \ldots, k)$. There is an IM2-machine which computes $F :\subseteq Y_1 \times \ldots \times Y_k \rightrightarrows Y_0$ iff there is a deterministic IM2-machine which computes F.*

Proof: The if part is immediate. For the only if part, we need to construct a deterministic machine from an indeterministic machine for the case that the input/output tapes have only one head. Suppose that M is an IM2-machine which realizes F. Since the set of rules of M is finite, we give a numbering to them. We can determine whether or not each rule is applicable because the input tapes do not have the character $\perp$. Therefore, we can modify M to construct a deterministic machine M' which chooses the first applicable rule with respect to the numbering. The result of M' to $x \in dom(F)$ is uniquely determined and is in $F(x)$. ∎

Definition 7 *A representation of a set X is a surjective partial function from Σ^ω to X.*

If ρ is a representation of X and $\rho(p) = x$, we call p a ρ-name of x.

Definition 8 *Let $\delta :\subseteq \Sigma'^\omega \to \mathcal{I}$ and $\delta' :\subseteq \Sigma^\omega \to \mathcal{I}$ be representations. We say that δ is reducible to δ' ($\delta \leq \delta'$) when there is a computable function $f :\subseteq \Sigma^\omega \to \Sigma'^\omega$ such that $\delta(p) = \delta'(f(p))$ for all $p \in dom(\delta)$. We say that δ and δ' are equivalent ($\delta \equiv \delta'$) when $\delta \leq \delta'$ and $\delta' \leq \delta$.*

Definition 9 *1) The signed digit representation ρ_{sd} of $\mathcal{I}$ uses the alphabet $\Sigma = \{0, 1, \overline{1}\}$ with $\overline{1}$ denoting -1, and it is a partial function $\rho_{sd} :\subseteq \Sigma^\omega \to \mathcal{I}$ defined on*

$$\{a_1 a_2 \ldots \mid a_1 = 1 \text{ and } \exists j \geq 2,\ \exists l \geq 2 \text{ such that } a_j \neq 1 \text{ and } a_l \neq \overline{1}\}$$

and returns $\Sigma_{i=1}^{\infty} a_i \cdot 2^{-i}$ to $a_1 a_2 \ldots$.

2) The restricted signed digit representation ρ_{sdr} of $\mathcal{I}$ is a restriction of ρ_{sd} to a

smaller domain

$$\{a_1 a_2 \ldots \mid a_1 = 1 \text{ and } \forall k\ \exists j \geq k,\ \exists l \geq k \text{ such that } a_j \neq 1 \text{ and } a_l \neq \bar{1}\}$$

without the first character $a_1 (= 1)$.

By ρ_{sd}, 3/8 has infinitely many names $10\bar{1}000\ldots$, $1\bar{1}1000\ldots$, $10\bar{1}\bar{1}111\ldots$, $10\bar{1}1\bar{1}\bar{1}\bar{1}\ldots$, $10\bar{1}01\bar{1}\bar{1}\bar{1}\ldots$. The domain of ρ_{sdr} means that we do not use a name which lasts as $111\ldots$ or $\bar{1}\bar{1}\bar{1}\ldots$, and therefore 3/8 has only two ρ_{sdr}-names $0\bar{1}000\ldots$ and $\bar{1}1000\ldots$.

Proposition 2 $\rho_{sdr} \equiv \rho_{sd}$.

Proof: It is an easy exercise to give an algorithm that converts a ρ_{sd}-name to a ρ_{sdr}-name.

∎

Definition 10 *Let* $\delta :\subseteq \Sigma^\omega \to \mathcal{I}$ *be a representation of* $\mathcal{I}$. *A multi-valued function* $F :\subseteq \mathcal{I} \rightrightarrows \mathcal{I}$ *is* (δ, δ)*-computable if there is a Type-2 machine* M *of type* $(\Sigma^\omega, \Sigma^\omega)$ *such that if* $\delta(p) \in dom(F)$, *then* M *with input* p *produces an infinite sequence* q *such that* $\delta(q) \in F(\delta(p))$.

A partial function is (δ, δ)-computable if it is computable as a multi-valued function. This definition can easily be extended to a function with several arguments.

Equivalent representations induce the same computability notion on $\mathcal{I}$. As we explained in the introduction, the equivalence class to which signed digit representation belongs induces a suitable notion of computability on real numbers.

Proposition 3 *Let* M *be an IM2-machine which realizes a multi-valued function* $F :\subseteq Y_1 \times \ldots \times Y_k \rightrightarrows Y_0$ *and* N_i $(i = 1, \ldots, k)$ *be IM2-machines which realize multi-valued functions* $G_i :\subseteq Y_1' \times \ldots \times Y_n' \rightrightarrows Y_i$. *Suppose that* $Im(\langle G_1, \ldots, G_k \rangle) \subset dom(F)$. *Then, there is an IM2-machine* $M \circ \langle N_1, \ldots, N_k \rangle$ *which realizes the multi-valued function* $F \circ \langle G_1, \ldots, G_k \rangle :\subseteq Y_1' \times \ldots \times Y_n' \rightrightarrows Y_0$. *Here, the composition of multi-valued functions* F *and* G *is defined to be* $y \in (F \circ G)(x)$ *if* $\exists z.\ z \in G(x)$ *and* $y \in F(z)$.

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 $\Sigma \cup \{B\}$ which connects the parts representing N and M, and work tapes to simulate the head movements of the input tape of M and 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 also need to modify the rules so that it first looks for an applicable rule coming from M and if there is no such rule, then looks for a rule coming from N. It is possible because the former rules do not access to the input

tapes and therefore a machine can determine whether a particular rule is applicable or not.

When $k > 1$, we need to copy the input tapes onto work tapes so that they can be shared by the parts representing $N_1, \ldots, N_k$. We define that it executes rules coming from N_i until it outputs a character, and then switch to the next part. ∎

As we will show in Section 7, we have the followings.

Lemma 4 *There is an IM2-machine of type $(\{\bar{1}, 0, 1\}^\omega, \{0, 1\}^\omega_{\perp,1})$ which converts a ρ_{sdr}-name of x to $G(x)$ for all $x \in I$.*

Lemma 5 *There is an IM2-machine of type $(\{0, 1\}^\omega_{\perp,1}, \{\bar{1}, 0, 1\}^\omega)$ which converts $G(x)$ to the ρ_{sdr}-name of x for all $x \in I$.*

Now, we prove the equivalences.

Theorem 6 *A multi-valued function $F :\subseteq \mathcal{I}^k \to \mathcal{I}$ is Gray-code-computable iff it is $((\rho_{sdr})^k, \rho_{sdr})$-computable.*

Proof: Suppose that M is an IM2-machine which Gray code computes F. By composing it with the IM2-machines in Lemma 4 and Lemma 5, we can form, by Proposition 3, an IM2-machine of type $((\{\bar{1}, 0, 1\}^\omega)^k, \{\bar{1}, 0, 1\}^\omega)$ which outputs a ρ_{sdr}-name of a member of $F(x_1, \ldots, x_k)$ when ρ_{sdr}-names of x_i are given. Therefore, we have a desired Type-2 machine by Proposition 1.

On the other hand, suppose that there is a Type-2 machine which $((\rho_{sdr})^k, \rho_{sdr})$-computes F. Since a Type-2 machine is a special case of an IM2-machine, again, by composing the IM2-machines in Lemma 4 and Lemma 5, we can form an IM2-machine which Gray code computes F. ∎

6 Topological Properties

Let $\Sigma = \{0, 1\}$. In this section, we show that G from $\mathcal{I}$ to $\Sigma^\omega_{\perp,1}$ is homeomorphic, and therefore is a topological embedding.

Since the character $\perp$ may be overwritten by 0 or 1, it is not appropriate to consider Cantor topology on $\Sigma^\omega_{\perp,1}$. Instead, we define the order structure $\perp < 0$ and $\perp < 1$ on our alphabet and consider the Scott topology on $\{0, 1, \perp\}$, i.e. $\{\{\}, \{0\}, \{1\}, \{0, 1\}, \{0, 1, \perp\}\}$. We consider its product topology on $\{0, 1, \perp\}^\omega$, and consider its subspace topology on $\Sigma^\omega_{\perp,1}$. Let $\uparrow p$ denote the set $\{x \mid p \leq x\}$. Then, the set $\{\uparrow (d\perp^\omega) \mid d \in \{0, 1, \perp\}^*\}$

is a base of $\{0,1,\perp\}^\omega$. From this, we have a base $\{\uparrow (d\perp^\omega) \cap \Sigma^\omega_{\perp,1} \mid d \in P\}$ for $P = \{0,1\}^* + \{0,1\}^*\perp\{0,1\}^*$ of $\Sigma^\omega_{\perp,1}$.

Note that P corresponds to the states of output tapes of IM2-machines after a finite time of execution, and $\uparrow (d\perp^\omega)\cap\Sigma^\omega_{\perp,1}$ is the set of possible outputs of an IM2-machine after it outputs $d \in P$. Thus, if $q \in O$ for an open set $O \subset \Sigma^\omega_{\perp,1}$ and for an output q of an IM2-machine, then this fact is available from a finite time of execution of the machine. In this sense, the observation that open sets are finitely observable properties in [Smy92] holds for our IM2-machine. We can prove the following fundamental theorem in just the same way as we do for Type-2 computability and Cantor topology on $\{0,1\}^\omega$.

Theorem 7 *An IM2-computable function $f :\subseteq (\Sigma^\omega_{\perp,1})^k \to \Sigma^\omega_{\perp,1}$ is continuous.*

Now, $Im(G)$ is the set $\{0,1\}^\omega - \{0,1\}^*0^\omega + \{0,1\}^*\perp 10^\omega \subset \Sigma^\omega_{\perp,1}$. We also consider the subspace topology on $Im(G)$, which has the base $\{\uparrow (d\perp^\omega) \cap Im(G) \mid d \in \{0,1\}^* + \{0,1\}^*\perp 10^*\}$. We consider the inverse image of this base by G. When $d \in \{0,1\}^*$ and $e \in 0^*$, $G^{-1}(\uparrow (d\perp^\omega))$ and $G^{-1}(\uparrow (d\perp 1e))$ range over open intervals of the form $m \times 2^{-i} < x < (m+1) \times 2^{-i}$, and $m \times 2^{-i} - 2^{-(i+1)} < x < m \times 2^{-i} + 2^{-(i+1)}$, respectively, 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. Thus, we have the following:

Theorem 8 *The Gray code embedding G is a topological embedding of $\mathcal{I}$ into $\Sigma^\omega_{\perp,1}$.*

As a direct consequence, we have the following:

Corollary 9 *A Gray-code-computable function $f :\subseteq \mathcal{I}^k \to \mathcal{I}$ is continuous.*

As an application of our representation, we give a simple proof of Theorem 4.2.6 of [Wei00], which says that there is no effective enumeration of computable real numbers. Here, we define $(x_i)_{i\in\omega}$ to be a computable sequence if there is an IM2-machine of type $(\Sigma^*, \Sigma^\omega)$ which outputs $G(x_i)$ when a binary name of i is given.

Theorem 10 *If $(x_i)_{i\in\omega}$ is a computable sequence, then a computable number x with $x \neq x_i$ for all $i \in \omega$ exists.*

Proof: Let $s_i = G(x_i)$ and M be an IM2-machine which computes s_i to the binary name of i. By Proposition 1, we can assume that M is deterministic. This means that, by selecting one machine, the order the output tape is filled is fixed. Since $s_i \in \Sigma^\omega_{\perp,1}$, either $s_i[2i]$ 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 $t[2i+1] = \textit{not}\ s_i[2i+1]$. Here, *not* is defined as *not* $1 = 0$ and *not* $0 = 1$. Then, the resulting sequence t is computable and is in $Im(G)$, but is not equal to s_i for $(i \in \omega)$. Therefore, $G^{-1}(t)$ is not equal to x_i because of the injectivity

of the representation. ∎

7 Conversion with signed digit representation

As an example of an IM2-machine, we consider conversions between the Gray code and the restricted signed digit representation. Recall that a ρ_{sdr}-name of $x \in \mathcal{I}$ 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 ρ_{sdr}-name of x.

Since the intervals represented by finite prefixes of both representation coincide, the conversions become simple automaton-like algorithms which do not use work tapes.

Example 1 *Conversion from the signed digit representation to Gray code.* It has the type $(\{\overline{1}, 0, 1\}^\omega, \{0, 1\}^\omega_{\perp,1})$. We simply write the head of the input tape as I. It has four states (i, j) $(i, j \in \{0, 1\})$ with $(0, 0)$ the initial state, and 12 rules:

$$
\begin{array}{ll}
(0,0), I(1) \Rightarrow (1,0), \mathrm{H}_1(O)(1); & (1,0), I(1) \Rightarrow (1,0), \mathrm{H}_1(O)(0); \\
(0,0), I(\overline{1}) \Rightarrow (0,0), \mathrm{H}_1(O)(0); & (1,0), I(\overline{1}) \Rightarrow (0,0), \mathrm{H}_1(O)(1); \\
(0,0), I(0) \Rightarrow (0,1), \mathrm{H}_2(O)(1); & (1,0), I(0) \Rightarrow (1,1), \mathrm{H}_2(O)(1); \\
\\
(0,1), I(1) \Rightarrow (0,0), \mathrm{H}_1(O)(1); & (1,1), I(1) \Rightarrow (0,0), \mathrm{H}_1(O)(0); \\
(0,1), I(\overline{1}) \Rightarrow (1,0), \mathrm{H}_1(O)(0); & (1,1), I(\overline{1}) \Rightarrow (1,0), \mathrm{H}_1(O)(1); \\
(0,1), I(0) \Rightarrow (0,1), \mathrm{H}_2(O)(0); & (1,1), I(0) \Rightarrow (1,1), \mathrm{H}_2(O)(0);
\end{array}
$$

In order to express this more simply, we use the notation of the functional language Haskell as follows:

```
s2gxs = stog0(xs, 0, 0)
stog0(1 : xs, 0, 0) = 1 : stog0(xs, 1, 0)
stog0(1̄ : xs, 0, 0) = 0 : stog0(xs, 0, 0)
stog0(0 : xs, 0, 0) = c : 1 : ds   where c : ds = stog0(xs, 0, 1)
stog0(0 : xs, 0, 1) = c : 0 : ds   where c : ds = stog0(xs, 0, 1)
...
```

Here, `where` produces bindings of `c` and `ds` to the head and the tail of `stog0 (xs,0,1)`, respectively. It is clear that the behavior of an IM2-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 `stog0(xs, 1, 0)` is that of `stog0(xs, 0, 0)` with the first character inverted, and the result of `stog0(xs, 0, 1)` is

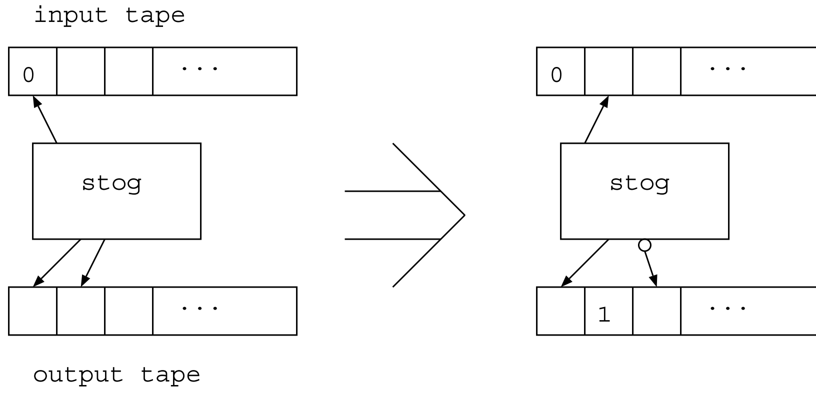

Fig. 3. The behavior of stog IM2-machine when it reads 0

that of $\texttt{stog0}(\texttt{xs}, 0, 0)$ with the second character inverted. Therefore, we can simplify the above program as follows:

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

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 on an output head means to invert the output from that head before filling the tape.

The program `stog` is a correct Haskell program and 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 digit representation.* Now, we only show a Haskell program. It has the type $(\{0,1\}^{\omega}_{\perp,1}, \{\overline{1},0,1\}^{\omega})$.

$$\texttt{gtos}(1 : \texttt{xs}) = 1 : \texttt{gtos}(\texttt{nh xs})$$
$$\texttt{gtos}(0 : \texttt{xs}) = \overline{1} : \texttt{gtosxs}$$
$$\texttt{gtos}(\texttt{c} : 1 : \texttt{xs}) = 0 : \texttt{gtos}(\texttt{c} : \texttt{nh xs})$$

In this case, indeterminism occurs and yields many different valid results: the results are actually signed digit representations of the same number. This is also a correct

Haskell program. However, it fails to calculate, for example, `gtos(stog([0,0..]))` because the program `gtos`, from the first two rules, tries to pattern match the head of the argument and starts its non-terminating calculation. Therefore, it fails to use the third rule. This is a limitation of the use of an existing functional language. We will discuss how to implement an IM2-machine as a program in Section 9.

These programs are based on the recursive structure of the Gray code and is not as difficult to write such a program as one might imagine. One can see from Figure 2 the following three recursive equations:

$$\begin{aligned} [\![0:p]\!] &= [\![p]\!]/2, \\ [\![1:p]\!] &= 1/2 + [\![\texttt{nh}\ p]\!]/2 (= 1.0 - [\![p]\!] * 0.5]\!]), \\ [\![c:1:p]\!] &= 1/4 + [\![c:(\texttt{nh}\ p)]\!]/2. \end{aligned} \tag{1}$$

Here, $[\![p]\!]$ is $G^{-1}(p)$. The first equation corresponds to the fact 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 corresponds to the fact 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. It says 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.

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

Here, g_1 is a function to calculate $f(x)$ from $f(2x)$ when $0 < x < 1/2$, g_2 is a function to calculate $f(x)$ from $f(2x-1)$ when $1/2 < x < 1$, and g_3 is a function to calculate $f(x)$ from $f(2x - 1/2)$ when $1/4 < x < 3/4$. `gtos` is derived immediately from this scheme.

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

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

`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.

8 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 digit 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 complement operation ($x \mapsto k/2^n - x$) with respect to a dyadic number $k/2^{n+1}$ for $(k-1)/2^{n+1} < x < (k+1)/2^{n+1}$ can be implemented as inverting one digit.

Example 5 *Shifting $x \mapsto x + 1/2$* $(0 < x < 1/2)$. Addition with a dyadic number is nothing but two continuous complement operations over dyadic numbers. In 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 = (not x):(not 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 addition $x + y$ with $0 < x, y < 1$. Since the result is in $(0, 2)$, we consider the average function $(x+y)/2$, 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 digit representation, we need to look ahead two characters. It is also the case with the Gray code representation. Since it does not have redundancy, we can reduce the number of rules from 25 to 13 compared with the program written in the same way with the signed digit representation.

9 Extension to the Whole Real Line, Implementation, and Conclusion

We have defined an embedding G of $\mathcal{I}$ to $\{0,1\}^{\omega}_{\perp,1}$ based on Gray code, and introduced an indeterministic multihead Type 2 machine as a machine which can input/output sequences in $\{0,1\}^{\omega}_{\perp,1}$. Since G is a topological embedding of $\mathcal{I}$ into $\{0,1\}^{\omega}_{\perp,1}$, our IM2-machines are operating on a topological space which includes $\mathcal{I}$ as a subspace. We hope that this computational model will propose a new perspective on real number computation.

In this paper, we only treated the unit open interval $\mathcal{I} = (0,1)$. We discuss here how this embedding can be extended to the whole real line $\mathcal{R}$. First, by using the first digit as the sign bit: 1 if positive, 0 if negative, and $\perp$ if the number is zero, we can extend it 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 to all of the real numbers without losing injectivity and without losing the simplicity of the algorithms in Section 7 and 8.

One possibility is to use some computable embedding of $\mathcal{R}$ into $(-1,1)$, such as the function $f(x) = 2 * arctan(x)/\pi$. It is known that this function is computable, and therefore, we have IM2-machines which convert between the signed digit representation of $x \in \mathcal{R}$ and the Gray code of $f(x)$ in $(-1,1)$. Therefore, we can define our new representation as $G'(x) = G(f(x))$ $(x \in \mathcal{R})$. It is clear that this representation embeds $\mathcal{R}$ into $\Sigma^{\omega}_{\perp,1}$, and all the properties we have shown in Section 4 to 6 hold if we replace $\mathcal{I}$ with $\mathcal{R}$ and G with G'. In particular, the computability notion on $\mathcal{R}$ induced by G' and IM2-machines is equivalent to the one induced by the signed digit representation and Type-2 machiens. However, we will lose the symmetricity of the Gray code expansion and simplicity of the algorithms in Section 7 and 8.

Another possibility is to introduce the character "." indicating the decimal point into the sequence. In order to allow an expression starting with $\perp$ (i.e. integers of the form 2^n), we also need to consider an expression starting with 0 because it should be allowed to fill the $\perp$ with 0 or 1 afterwards. Thus, we lose the injectivity of the expansion because we have `0:1:xs = 1:xs`. We also have the same kind of difficulty if we adopt the floating-point-like expression: a pair of a number indicating the decimal

point and a Gray code on $(-1,1)$.

Although this expansion becomes redundant, the redundancy introduced here by preceding zeros is limited in that we only need at most one zero at the beginning of each representation and thus each number has at most two names. As is shown in [BH00], we need infinitely many names to infinitely many real numbers if we use representations equivalent to the signed binary representation. Therefore, the redundancy we need for this extension is essentially smaller than that of the signed binary representation.

Finally, we show some experimental implementations we currently have. As we have noted, though we can express the behavior of an IM2-machine using the syntax of a functional language Haskell, the program comes to have different semantics under the usual lazy evaluation strategy. We have implemented this Gray code input/output mechanism using logic programming languages. We have written `gtos`, `stog`, and the addition function `pl` of Section 8 using KL1 [UC90], a concurrent logic programming language based on Guarded Horn Clauses. We have also implemented them using the coroutine facility of SICStus Prolog. We are also interested in extending lazy functional languages so that programs in Section 7 and 8 become executable. The details about these implementations are given in [Tsu01].

Acknowledgements

The author thanks Andreas Knobel for many interesting and illuminating discussions. He also thanks Mariko Yasugi, Hiroyasu Kamo, and Izumi Takeuchi for many discussions.

References

[ASD90] D. J. Amalraj, N. Sundararajan, and G. Dhar. A data structure based on gray code encoding for graphics and image processing. In *SPIE Vol. 1349 Applications of Digital Image Processing XIII*, 1990.

[BCRO86] H. J. Boehm, R. Cartwright, M. Riggle, and M. J. O'Donnel. Exact real arithmetic: A case study in higher order programming. In *ACM Symposium on Lisp and Functional Programming*, 1986.

[BH00] Vasco Brattka and Peter Hertling. Topological properties of real number representations. *Theoretical Computer Science*, 2000. to appear.

[Bra98] Vasco Brattka. *Recursive and Computable Operations over Topological Structures.* PhD thesis, Fern Universitat, 1998.

[Dew93] A. K. Dewdney. *The New Turing Omnibus*. Computer Science Press, 1993.

[EP97] Abbas Edalat and Peter John Potts. A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

[ES98] Abbas Edalat and Philipp Sünderhauf. A domain-theoretic approach to computability on the real line. *Theoretical Computer Science*, 210(1):73–98, 1998.

[Gia96] Pietro Di Gianantonio. Real number computability and domain theory. *Information and Computation*, 127:11–25, 1996.

[Gia97] Pietro Di Gianantonio. A golden ratio notation for the real numbers. Technical Report Technical Report CS-R9602, CWI Amsterdam, 1997.

[Gia99] Pietro Di Gianantonio. An abstract data type for real numbers. *Theoretical Computer Science*, 221:295–326, 1999.

[Grz57] Andrzej Grzegorczyk. On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44:61–71, 1957.

[HJ92] P. Hudak and S. P. Jones. Haskell report. Technical Report 27(5), SIGPLAN Notices, 1992.

[HY84] Masayoshi Hata and Masaya Yamaguti. The takagi function and its generalization. *Japan J. Appl. Math.*, 1:183–199, 1984.

[PER89] Marian B. Pour-El and J. Ian Richards. *Computability in Analysis and Physics*. Springer-Verlag, 1989.

[She75] J. C. Shepherdson. Computation over abstract structures: serial and parallel procedures. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73, volume 80 of Studies in Logic and the Foundation of Mathematics*, pages 445–513. North-Holland, 1975.

[Smy92] M. B. Smyth. Topology. In S. Abramsky, D. M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 1*, pages 641–761. Clarendon Press, Oxford, 1992.

[Tsu01] Hideki Tsuiki. Implementing real number computation in GHC. *Computer Software (in Japanese)*, 18(2):40–53, 2001.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proc. of the London Mathematical Society 42(2)*, pages 230–265, 1936.

[UC90] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[Vui90] J. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.

[Wei85] Klaus Weihrauch. Type 2 recursion theory. *Theoretical Computer Science*, 38:17–33, 1985.

[Wei00] Klaus Weihrauch. *Computable analysis, an Introduction.* Springer-Verlag, Berlin, 2000.