Real Number Computation with Committed Choice Logic Programming Languages

Hideki Tsuiki

Graduate School of Human and Environmental Studies, Kyoto University
606-8501, Sakyo-ku, Kyoto, Japan

Abstract

As shown in [10], the real line can be embedded topologically in the set $\Sigma^{\omega}_{\perp,1}$ of infinite sequences of $\{0, 1, \perp\}$ containing at most one $\perp$. Moreover, there is a nondeterministic multi-headed machine, called an IM2-machine, which operates on $\Sigma^{\omega}_{\perp,1}$ and which induces the standard notion of computation over the reals via this embedding. In this paper, we study how the behavior of an IM2-machine can be expressed in "real" programming languages. When we use a lazy functional language like Haskell and represent a sequence as an infinite list, we cannot express the behavior of an IM2-machine. However, when we use a logic programming language with guarded clauses and committed choice, such as Concurrent Prolog, PARLOG, and GHC (Guarded Horn Clauses), we can express the behavior of IM2-machines naturally and execute them on an ordinary computer. We show that GHC-computability implies IM2-computability but not vice versa when we consider functions defined on $\Sigma^{\omega}_{\perp,1}$ in general, but they coincide when we only consider functions defined on the reals. We give some GHC program examples, such as the conversions between Gray-code and the signed digit representations, and the addition function on reals.

1 Introduction

It is easy to show that the real line cannot be embedded in Cantor space $\{0, 1\}^{\omega}$; Cantor space is totally disconnected and 0-dimensional whereas the real-line is connected and 1-dimensional. For this reason, in order to define a suitable notion of computation on the reals in terms of machines acting on infinite sequences (Type-2 machines), we need to use a redundant representation like the signed-digit representation[17].

On the other hand, the present author has shown that real numbers are represented uniquely as infinite sequences if we allow at most one undefined cell in a sequence. Indeed, the real line can be topologically embedded in the space $\Sigma^{\omega}_{\perp,1}$

of infinite sequences of $\{0, 1, \perp\}$ containing at most one $\perp$ by the Gray-code embedding[10]. He also defined the notion of an IM2-machine (indeterministic multi-head Type-2 machine) which makes generalized stream access to $\Sigma^{\omega}_{\perp,1}$ using two heads on each input and output tape, and has indeterministic (i.e. nondeterministic) behavior depending on which head is used to input a character. The notion of IM2-computability over the reals induced by the Gray-code embedding is equivalent to the standard notion of Type-2 computability over the reals induced by the signed-digit representation. More generally, the set $\Sigma^{\omega}_{\perp,n}$ of $n\perp$-sequences, which are infinite sequences of $\{0, 1, \perp\}$ with at most n copies of $\perp$, is itself a n-dimensional topological space into which any n-dimensional separable metric space can be embedded [9,11]. Therefore, we can compute over a n-dimensional separable metric space with a generalized IM2-machine with $n + 1$ heads, and this is the smallest number of heads that will do.

In this paper, we study how the behavior of an IM2-machine can be expressed in real programming languages. If we represent an $n\perp$-sequence as an infinite list some of whose components may cause non-terminating computation, we can express the rules of an IM2-machine in the syntax of a lazy functional programming language like Haskell[4]. Actually the present author used the syntax of Haskell to express Gray-code based real-number algorithms in [10]. However, the syntax requires a semantics different from the ordinary one, and those algorithms do not behave correctly if implemented in Haskell. The problem is that, in functional languages, we can only make sequential access to an infinite list, and computation stalls whenever a bottom is encountered. Contrary to this, if we use a logic programming language with guarded clauses and committed choice such as Concurrent Prolog[8], PARLOG[3], and (Flat) GHC (Guarded Horn Clauses)[14,15], we can directly express the behavior of an IM2-machine as a program, and execute it on an ordinary computer. The ability of an IM2-machine to wait for the next character from multiple heads corresponds to parallel execution of guards, and the indeterminism of an IM2-machine corresponds to committed-choice nondeterminism. Among those programming languages, we adopt (Flat) GHC, which is based on simple formalism and which has an efficient implementation. We give some examples of GHC programs operating on $n\perp$-sequences such as the conversions between Gray-code and the signed digit representation, and the addition function on reals with respect to the Gray-code embedding.

We compare the expressive powers of IM2-machines and GHC programs on $\Sigma^{\omega}_{\perp,n}$ and its subspaces. When $\Sigma^{\omega}_{\perp,n}$ is considered, IM2-computability implies GHC-computability but not vice versa. However, they are equivalent when we only consider functions defined on subspaces of $\Sigma^{\omega}_{\perp,n}$ composed of minimal limit elements of what we call an $n\perp$-domain. Since the image of the Gray-code embedding has such a structure, IM2-computability and GHC-computability coincide for real number computation realized in $\Sigma^{\omega}_{\perp,1}$. This equivalence means that we can use GHC as a language to define IM2-computable functions on $\Sigma^{\omega}_{\perp,n}$, instead of IM2-machines. Thus, we can use the expressive power of GHC in defining such functions; for example, we can define a function as a composition of recursively defined processes in GHC. Furthermore, we can execute them on an ordinary computer.

This research on computation over $\Sigma^{\omega}_{\perp,n}$ shows a difference between the expressive powers of functional and logic programming languages on infinite data. It also shows how logic programming languages may be exploited to implement continuous computation over topological structures.

In Sections 2,3, and 4, following [10], we introduce the signed-digit representation, Gray-code embedding, and IM2-machines. In Section 5, we show that it is impossible to express the behavior of an IM2-machine if a sequential lazy functional language is used and a sequence is represented as an infinite list. We introduce the language GHC in Section 6, and show that IM2-machines can be translated into GHC programs in Section 7. We study the relation between IM2-computability and GHC-computability in Section 8. In this section, we assume familiarity with basic domain theory. See, for example, [1,5] for expositions of the theory of domains. In Section 9, we present another translation of IM2-machines into GHC programs. This time, the obtained programs are written in the demand-driven fashion. Finally, we discuss implementation in other programming languages in Section 10.

Notations

In this paper, we consider the closed unit interval $\mathbb{I} = [0, 1]$ instead of the whole real line. We write Σ^* (and Σ^{ω}) for the sets of finite (and infinite) sequences of a finite character set Σ, respectively, and Σ^{∞} for $\Sigma^* \cup \Sigma^{\omega}$. We write $q[k]$ for the k-th character of a sequence q, with $q[0]$ the first element of the sequence.

We write $\Sigma^{\omega}_{\perp}$ for $(\Sigma \cup \{\perp\})^{\omega}$. We call a member of $\Sigma^{\omega}_{\perp}$ a *bottomed sequence*. We define an order relation on $\Sigma^{\omega}_{\perp}$ so that $p \leq q$ iff q is obtained by substituting some copies of $\perp$ in p with characters in Σ. We call a bottomed sequence in which at most n copies of $\perp$ appears an *$n\perp$-sequence*, and write $\Sigma^{\omega}_{\perp,n} \subset \Sigma^{\omega}_{\perp}$ for the set of $n\perp$-sequences. Though we present the theory without fixing Σ, one may think of Σ as $\{0, 1\}$ when we are concerned with bottomed sequences. From this definition, we have $\Sigma^{\omega} = \Sigma^{\omega}_{\perp,0} \subset \Sigma^{\omega}_{\perp,1} \subset \Sigma^{\omega}_{\perp,2} \subset \ldots \subset \Sigma^{\omega}_{\perp}$. We call $p \in \Sigma^{\omega}_{\perp}$ which contains finite number of Σ characters a *finite bottomed sequence*, and write $\Sigma^{*}_{\perp}$ for the set of them. We write $\Sigma^{*}_{\perp,n}$ for the set of finite bottomed sequences which contain at most n copies of $\perp$ when we discard the infinite sequence of $\perp$ at the end of the sequence. In other word, it is the set of $p \in \Sigma^{\omega}_{\perp}$ such that $p[k] = \perp$ for $k \geq r$ where r is the index of the $(n+1)$-th appearance of $\perp$.

We write $A \rightharpoonup B$ for the set of partial functions from A to B, and $A \rightrightarrows B$ for the set of *multi-valued partial functions* from A to B, that is, a subset of $A \times B$ considered as a partial function from A to the set of nonempty subsets of B. We write $f :\subseteq A \rightharpoonup B$ or $f :\subseteq A \rightrightarrows B$ when f belongs to these sets. When $n_1 < n_2$ and $m_1 < m_2$, a partial function in $\Sigma^\omega_{\bot,n_1} \rightharpoonup \Sigma^\omega_{\bot,m_1}$ can also be considered as an element of $\Sigma^\omega_{\bot,n_2} \rightharpoonup \Sigma^\omega_{\bot,m_2}$. Similarly for $\Sigma^\omega_{\bot,n_1} \rightrightarrows \Sigma^\omega_{\bot,m_1}$. Therefore, we have the relation $\Sigma^\omega_{\bot,0} \rightharpoonup \Sigma^\omega_{\bot,0} \subset \Sigma^\omega_{\bot,1} \rightharpoonup \Sigma^\omega_{\bot,1} \subset \ldots \subset \Sigma^\omega_{\bot} \rightharpoonup \Sigma^\omega_{\bot}$, and $\Sigma^\omega_{\bot,0} \rightrightarrows \Sigma^\omega_{\bot,0} \subset \Sigma^\omega_{\bot,1} \rightrightarrows \Sigma^\omega_{\bot,1} \subset \ldots \subset \Sigma^\omega_{\bot} \rightrightarrows \Sigma^\omega_{\bot}$.

2 Real-number computation by Type-2 machines

A *representation* of a set X is a partial surjective function from Σ^ω to X for a finite alphabet Σ. When δ is a representation and $\delta(p) = x$, we say that p is a *δ-name* of x. The decimal expansion δ_{10} is a standard representation of the real numbers. However, when we use this representation, we cannot express such a simple algorithm as multiplication by three with a Type-2 machine [17]. A *Type-2 machine* is, from programming point of view, a program which makes stream access to input/output infinite sequences. Because a machine can only read a finite prefix of the input when it outputs a character, a machine to multiply by three cannot write the first digit when the input is $0.33333\ldots$, and thus it does not work properly. It is also the case for the binary expansion.

Definition 1 *Suppose that δ_i is a representation of X_i $(i = 0, 1, \ldots, k)$. We say that a partial multi-valued function $f :\subseteq X_1 \times \ldots \times X_k \rightrightarrows X_0$ is $(\delta_1, \ldots, \delta_k, \delta_0)$-Type2-computable if there is a Type2-machine with n inputs which converts every $\delta_1 \times \ldots \times \delta_k$-name of $(x_1, \ldots, x_k) \in dom(f)$ to a δ_0-name of a member of $f(x_1, \ldots, x_k)$.*

As we have seen, "multiplication by three" is not $(\delta_{10}, \delta_{10})$-Type2-computable. In order to obtain a more natural notion of computability, we need to use a representation with higher redundancy.

Definition 2 The *(modified) signed digit representation* δ_{sn} of $\mathbb{I}$ uses the character set $\Gamma = \{0, 1, \bar{1}\}$ and it is a partial function from Γ^ω to $\mathbb{I}$ defined as

$$dom(\delta_{sn}) = \Gamma^\omega \setminus (\Gamma^* 1^\omega \cup \Gamma^* \bar{1}^\omega \setminus \{1^\omega, \bar{1}^\omega\}),$$

$$\delta_{sn}(a_1 a_2 \ldots) = 1/2 + \Sigma_{i=1}^\infty \{a_i \cdot 2^{-(i+1)}\}.$$

Here, $\bar{1}$ represents -1.

This definition is a bit different from the ordinary one in that the signed digit representation is usually defined as a total function from Γ^ω to $[-1, 1]$ defined as $\delta_{sn}(a_1 a_2 \ldots) = \Sigma_{i=1}^\infty \{a_i \cdot 2^{-i}\}$. In Definition 2, we restrict the ordinary one by fixing the first digit as 1 and deleting it from the string, and not allowing names of the forms $\Gamma^*111\ldots$ and $\Gamma^*\overline{1}\overline{1}\overline{1}\ldots$ except for 111... and $\overline{1}\overline{1}\overline{1}$.... This modification is only for some technical reason, and does not change the induced Type-2 computability over the reals. The representation δ_{sn} has high redundancy in that replacements of substrings of the forms $1\overline{1}$ to 01 and $\overline{1}1$ to $0\overline{1}$ do not change the number. Actually, infinitely many numbers have infinitely many names with δ_{sn}.

The above mentioned function to multiply by three becomes computable with respect to this representation. $(\delta_{sn}, \delta_{sn})$-Type2-computability is the most standard computability notion over the reals in that it coincides with many other computability notions on the reals based on different approaches. All the representations equivalent to δ_{sn} are shown to be redundant, and redundancy is considered as a fundamental property of real-number representations. See [17] for the theory of Type2-computability.

3 Gray-code embedding

Gray-code expansion is an expansion of $\mathbb{I}$ as infinite sequences of $\{0, 1\}$, which is different from the ordinary binary expansion. Figure 1 shows the binary and Gray-code expansion of $\mathbb{I}$. In the binary expansion of x, the head h of the expansion indicates whether x is in [0, 1/2] or [1/2, 1], and the tail is the expansion of $f(x, h)$ for f the following function:

$$f(x,h) = \begin{cases} 2 * x & \text{(when } h = 0.) \\ 2 * x - 1 & \text{(when } h = 1.) \end{cases}$$

Note that the rest of the expansion depends on the choice of the head character h when $x = 1/2$. On the other hand, the head of the Gray-code expansion is the same as that of the binary expansion, whereas the tail is the expansion of $t(x)$ for t the so-called tent function:

$$t(x) = \begin{cases} 2 * x & (0 \le x \le 1/2) \\ 2 * (1 - x) & (1/2 < x \le 1) \end{cases} .$$

We have two binary expansions for a dyadic number (a rational numbers of the form $m/2^k$). For example, 3/4 has two expansions 110000... and 101111.... It is also the case for the Gray-code expansion, and 3/4 has two expansions 111000... and 101000.... Note that they differ only at one bit. Generally, the two Gray-code expansions of a dyadic number differ only at one bit and the sequence after the bit is always 1000.... In this way, the second bit does not contribute to the fact that the value is 3/4, and it is more natural not to

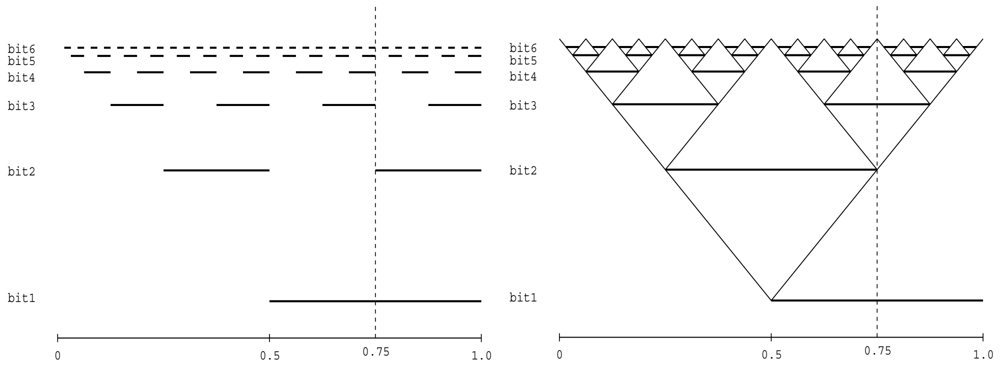

Fig. 1. The binary expansion and the Gray-code expansion of real numbers. Here, horizontal line means that the corresponding bit has value 1.

specify the bit and leave it undefined ($\perp$). Thus, we define the expansion of $3/4$ as $1\perp1000\ldots$, and define the modified Gray-code expansion as follows.

Definition 3 Let $\Sigma = \{0, 1\}$ and $P : \mathbb{I} \to \Sigma_\perp$ be the map

$$P(x) = \begin{cases} 0 & (x < {}^1\!/\!_2) \\ \perp & (x = {}^1\!/\!_2) \\ 1 & (x > {}^1\!/\!_2) \end{cases} .$$

Gray-code embedding G is a function from $\mathbb{I}$ to $\Sigma^\omega_{\perp,1}$ defined as $G(x)[n] = P(t^n(x))$ $(n = 0, 1, \ldots)$. We call $G(x)$ the *modified Gray-code expansion* of x.

We write $image(G) \subset \Sigma^\omega_{\perp,1}$ for the image of the Gray-code embedding. We have a topology on $\Sigma^\omega_{\perp,1}$, which is the subspace topology of the Scott topology on $\Sigma^\omega_\perp$. G is actually a topological embedding of $\mathbb{I}$ in $\Sigma^\omega_{\perp,1}$ with respect to this topology.

4 IM2-machine

We consider that information about a real number x is given incrementally as shrinking open intervals $a_1 < x < b_1$, $a_2 < x < b_2$, $\ldots$ converging to x. We study how a machine can output the modified Gray-code expansion of x on a tape based on this. When the information $x < 1/2$ or $1/2 < x$ is given, it can write 0 or 1 on the first cell, respectively. However, when $x = 1/2$, neither information is given and therefore it cannot fill the first cell eternally. However, in this case, it obtains the information $1/4 < x < 3/4$ at some time, and it can write 1 on the second cell skipping the first one if it is allowed to write a character not only on the leftmost unfilled cell but also on the next unfilled cell. After that, if the information $1/4 < x < 1/2$ or $1/2 < x < 3/4$

is given, it can write 0 or 1 on the skipped cell, respectively, and if it has the information $3/8 < x < 5/8$, it can write 0 on the third (i.e., the second unfilled) cell. In this way, when $x = 1/2$, the first cell is left unfilled and the sequence $1000\ldots$ is written from the second cell. Thus, if we consider that the output tape is filled with $\perp$ at the beginning, we can output the modified Gray-code expansion on the tape.

We can formulate this mechanism to write on the first or the second unfilled cell as an output with two heads. We consider two heads on each tape which move automatically after an output so that they are always located at the first and the second unfilled cell. That is, the two heads H_1 and H_2 are located at the first two cells at the beginning, and only H_2 moves to the next cell after an output from H_2, and H_1 moves to the position of H_2 and H_2 moves to the next cell after an output from H_1. This is a generalization of the ordinary stream access with one head, which moves to the next cell after an output.

As for the input, when the value of a cell is $\perp$, a machine cannot wait for it to be filled because it may not be filled eternally. Therefore, in order to skip a bottom cell and continue the input, we need two heads also on input tapes, which move the same way as the output-tape heads. Then, when both of the cells under the two heads are filled, a machine may have two possible inputs which will cause two different computations. Therefore, it has nondeterministic behavior and both of the computational paths must produce valid results. The author used the word indeterminism and called the machine an IM2-machine (indeterministic multi-head Type-2 machine) to distinguish it from the nondeterminism used for nondeterministic automatons. They are different in that a nondeterministic automaton accepts a word when one of the computational paths accepts the word, whereas our machine should produce a valid result not depending on the computational path it takes. Though the word nondeterminism is commonly used for both purposes when we are talking about programming languages, in order to identify the particular kind of nondeterminism which occurs in IM2-machines, I will use the word indeterminism for IM2-machines also in this paper.

Definition 4 We define an IM2-machine in a general form so that it can input/output $n\perp$-sequences with $n+1$ heads on each tape. We say that an IM2-machine has type $(\Sigma^\omega_{\perp,n_1}, \ldots, \Sigma^\omega_{\perp,n_k}, \Sigma^\omega_{\perp,n_0})$ when it has k input tapes $T_1, \ldots, T_k$ of types $\Sigma^\omega_{\perp,n_1}, \ldots, \Sigma^\omega_{\perp,n_k}$, and one output tape T_0 of type $\Sigma^\omega_{\perp,n_0}$. Though we use the same alphabet Σ for simplicity of the presentation, the character set Σ can be different for each tape, in practice. The tape T_i has $n_i + 1$ heads $H_1(T_i)$, $\ldots$,$H_{n+1}(T_i)$. The character set of T_i is Σ, and the bottom '$\perp$' can also appear on an input tape or an output tape. An IM2-machine has a set Q of states (with one initial state q_0) and a set of worktapes. Worktapes are the same as Turing-tapes in that only one head exists on each worktape which moves in both directions. For worktapes, we consider another character set Γ

which includes the blank character 'B'. A machine has a set of computational rules of the 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, $q, q' \in Q$, i_j are heads of different input tapes, o is an output head of the output tape, w_i, w'_i, w''_i are worktapes, c_j, c, d_j and d'_j are characters from the corresponding character sets, and M_j $(j = 1, \ldots, g)$ are '+' or '$-$'. Each part of the rule is optional; there may be a rule without $o(c)$, for example. Note that 'B' can appear in d_j and d'_j, but '$\perp$' cannot appear in c_j nor c.

The meaning of this rule is that if the state is q and the characters under i_j $(j = 1, \ldots, r)$ and the heads of 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 o and the heads of w'_j, respectively, move the heads of 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 $H_e(T_l)$ and the type of T_l is $\Sigma^\omega_{\perp,n}$, each head $H_d(T_l)$ $(e \le d \le n)$ moves to the position of $H_{d+1}(T_l)$ and $H_{n+1}(T_l)$ moves to the next cell.

At the beginning, the inputs $(p_1, \ldots, p_k)$ $(p_i \in \Sigma^\omega_{\perp,n_i})$ are put on the input tapes, the output tape is filled with '$\perp$', worktapes are filled with the blank character 'B', and the state set to the initial state. It repeats infinitely the selection of one of the applicable rules and its execution. We call a sequence of rules executed by a machine a computational path. We say that it outputs $q \in \Sigma^\omega_{\perp,n_0}$ when there is a computational path with which the output tape T_0 satisfies $T_0[k] = c$ at some finite time when $q[k] = c$ for $c \in \Sigma$, and $T_0[k] = \perp$ eternally when $q[k] = \perp$.

As we have noted, an IM2-machine has indeterministic behavior and thus it may have more than one computational path to the same input, and thus it has the possibility to output many different results to the same input. Therefore, an IM2-machine defines a partial multi-valued function.

Definition 5 An IM2-machine M of type $(\Sigma^\omega_{\perp,n_1}, \ldots, \Sigma^\omega_{\perp,n_k}, \Sigma^\omega_{\perp,n_0})$ *realizes* a partial multi-valued function f from $\Sigma^\omega_{\perp,n_1} \times \ldots \times \Sigma^\omega_{\perp,n_k}$ to $\Sigma^\omega_{\perp,n_0}$ if, when a tuple of arguments in $dom(f)$ is given, M outputs infinitely under all the computational paths of M, and the set of outputs to $(p_1, \ldots, p_k) \in dom(f)$ is a subset of $f(p_1, \ldots, p_k)$. When such an M exists, we say that f is *IM2-computable.*

In this definition, a multi-valued function f is considered as a specification which M must satisfy. This specification is 'weak' in the following two senses. Firstly, the behavior of M to $p \notin dom(f)$ is not specified and it may output some value to $p \notin dom(f)$. Secondly, $a = f(q)$ means that M must produce an element of a but M need not have the possibility to produce all the elements of a. From these properties, the following lemma is apparent.

Lemma 6 *(1) If f is IM2-computable and g is its restriction to $S \subset dom(f)$, then g is also IM2-computable.*
(2) If f is IM2-computable, $dom(f) = dom(g)$, and $f(p) \subseteq g(p)$ for all $p \in dom(f)$, then g is also IM2-computable.

Definition 7 Let $\Sigma = \{0, 1\}$. An IM2-machine M of type $(\overbrace{\Sigma^{\omega}_{\perp,1}, \ldots, \Sigma^{\omega}_{\perp,1}}^{\text{k+1}})$ *realizes* a partial multi-valued function f from $\mathbb{I}^k$ to $\mathbb{I}$ if the embedding of f in $(\Sigma^{\omega}_{\perp,1})^k \rightrightarrows \Sigma^{\omega}_{\perp,1}$ by G is realized by M. In this case, we say that f is *Gray-code computable*. We define the computability of a partial function as a special case of that of a multi-valued function.

Theorem 8 ([10]) *A partial (multi-valued) function f from $\mathbb{I}^k$ to $\mathbb{I}$ is Gray-code computable iff it is $(\overbrace{\delta_{sn}, \ldots, \delta_{sn}}^{k+1})$-Type2-computable.*

In [9] and [11], it is shown that there is an embedding of any n-dimensional separable metric space in $\Sigma^{\omega}_{\perp,n}$. Therefore, when we fix such an embedding, we can also introduce computability notion on such spaces through IM2-machines.

5 Impossibility of implementing IM2-machines in functional languages

We consider how the behavior of an IM2-machine can be expressed in the syntax of a lazy functional programming language, as a recursively defined function which inputs and outputs infinite lists. The states and worktapes are treated in the same way as those of a functional-language implementation of a Turing machine. That is, the recursive function has extra arguments for the state and the contents of the worktapes before and after the head positions, with the order reversed on the before part. As for the inputs and outputs with multiple heads, we express as follows. Here, we explain with the two-head case.

First, the output. The return value of the function is an infinite list which consists only of unfilled cells of the output tape. Therefore, the two heads are considered as located at the first two members of the list. Thus, we can express the output from the first head as `c:foo()` with `c` the character 0 or 1 and `foo()` the recursive call to produce the rest of the output. The output from the second head is written as `x:c:xs where x:xs=foo()`, with the same meanings for `c` and `foo()`. Note that, in both cases, the new head positions of a machine are the first two members of `foo()`.

As an example, we consider the partial function *stog* $:\subseteq \Gamma^\omega \rightharpoonup \Sigma^\omega_{\perp,1}$ which converts the signed-digit representation to the modified Gray-code expansion for $\Gamma = \{0, 1, -1\}$ and $\Sigma = \{0, 1\}$. The IM2-machine which realizes *stog* has the type $(\Gamma^\omega, \Sigma^\omega_{\perp,1})$. It has 4 states $(0,0), (0,1), (1,0)$, and $(1,1)$ with $(0,0)$ the initial state, and it does not use worktapes. It has 12 computation rules with the following forms [10].

$$(0,0), I(0) \Rightarrow (0,1), H_2(O)(1);$$

$$\ldots$$

This line means, when the state is (0,0) and the input from the head of tape I is 0, the machine changes the state to (0,1) and outputs 1 from the second head of the output tape O. The Haskell program produced from this rule is as follows:

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

This program is equivalent to the following three-line program which is composed by considering the recursive structure of Gray-code embedding.

```
stog(1:xs) = 1:nh(stog xs)
stog(-1:xs) = 0:stog xs
stog(0:xs) = c:1:nh ds where c:ds= stog xs
```

Here, `nh` is the function to invert the value of the first element of an infinite list, defined as follows.

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

This is a correct Haskell program and it works as expected; the execution of `stog([0,0..])` has no output because it starts computing the value of the first cell, which is $\perp$, but the execution of `tail(stog([0,0..]))` will produce `[1,0,0,0...` infinitely.

Next, the input. As is the case for the output, the argument corresponding to an input tape is an infinite list composed of those cells which have not been read by the machine, and we consider that the two heads are located at the first two elements of the list. It looks natural to express such a function as follows, using pattern matching on the first and the second argument.

```
foo(0:xs,..) = bar1(xs,..)
foo(1:xs,..) = bar2(xs,..)
foo(c:1:xs,..) = bar3(c,xs,..)
foo(c:0:xs,..) = bar4(c,xs,..)
```

Note that this definition has an overlap between the first two lines and the rest, corresponding to the indeterministic behavior of our machine. As an example, consider the partial multi-valued function *gtos* $:\subseteq \Sigma^{\omega}_{\perp,1} \rightrightarrows \Gamma^{\omega}$ which converts the modified Gray-code expansion to the signed digit representation, and express the behavior of an IM2-machine which realizes *gtos*. Here, we only list a program which is written based on the recursive structure of Gray code embedding.

```
gtos(0:xs) = -1:gtos(xs)
gtos(1:xs) = 1:gtos(nh xs)
gtos(c:1:xs) = 0:gtos(c:nh xs)
```

Note that we do not need to express the case for `c:0:xs` because the second character of the argument to `gtos` is inverted on the third line.

This program uses the correct Haskell syntax. However, the meaning of this program in Haskell is different from our intention. The meaning of this program is as follows. Evaluate the argument to a cons cell and then evaluate the head of the cell. If the value is 0 or 1, then apply the first two rules. If it has other values, then it tries to apply the third rule. Thus, if the evaluation of the head of the argument does not terminate, it cannot apply the third rule even if the value of the second element is 1. Therefore, the execution of `gtos (stog [0,0..])` diverges instead of producing `[0,0,0,...`.

As another example, the addition algorithm with respect to Gray-code is presented in [10]. The algorithm is also written in the Haskell syntax, and it diverges for dyadic numbers when executed in Haskell.

Of course, if we express an $n\perp$-sequence as an infinite list of pairs of the form (head-number, character) such as `[(2,1),(1,0),...]`, we can implement the behavior of an IM2-machine in Haskell. We consider a character set $\Xi_{\Sigma,n} = \{a^{(i)} \mid a \in \Sigma, i \in \{1, 2, \dots, n{+}1\}\}$ and assign $a^{(i)}$ to the pair (i, a). The function which maps an infinite sequence in ${\Xi_{\Sigma,n}}^{\omega}$ to the corresponding $n\perp$-sequence is a representation of $\Sigma^{\omega}_{\perp,n}$. When this representation is combined with the Gray-code embedding, it comes to be a representation of $\mathbb{I}$ equivalent to the signed-digit representation. In particular, it is redundant and $1^{(2)}0^{(1)}\dots$ is equivalent to $0^{(1)}1^{(1)}\dots$. In this paper, we are interested in implementations which treat each $n\perp$-sequence as an infinite list of Σ which may contain at most n copies of $\perp$.

The multi-valued function *gtos* has no computable choice function, i.e., there is no single-valued computable function which is a subset of *gtos* and which has the same domain as *gtos*. If such a choice function exists, by composing it with the *stog* function, we have a computable function $\Gamma^\omega \rightharpoonup \Gamma^\omega$ which determines the normal form of the signed-digit representation, which is known to be impossible[17]. This fact means that the *gtos* function cannot be implemented in a deterministic language. In the following theorem, we also show the existence of a single-valued IM2-computable function not expressible in a sequential functional language. Here, a sequential functional language means a language in which the "parallel or" operator cannot be expressed.

Theorem 9 *When we express an $n\bot$-sequence directly as an infinite list which may contain at most n copies of $\bot$, there is an IM2-computable single-valued function not expressible in a sequential functional language.*

PROOF. Let $\Sigma = \{0, 1\}$. Consider the function $pors : \Sigma^\omega_{\bot,1} \to \Sigma^\omega$ defined with the following Haskell syntax.

$$dom(pors) = \Sigma^\omega \cup \bot 1\Sigma^\omega \cup 1\bot\Sigma^\omega$$

```
pors(0:0:xs) = 0:xs
pors(1:c:xs) = 1:xs
pors(c:1:xs) = 1:xs
```

pors is an IM2-computable single-valued function. Suppose that `pors` is a program which implements the *pors* function. Then, the parallel-or operator can be expressed as $\texttt{por}(\texttt{c},\texttt{d}) = \texttt{head}\ \texttt{pors}(\texttt{c} : \texttt{d} : [0, 0..])$. Therefore, *pors* cannot be expressed in a sequential functional language. ∎

It does not mean that we always need parallel execution mechanism for expressing the behavior of an IM2-machine. Actually, we have only one thread of computation in an IM2-machine, whose result is given at the first or the second cell of a tape. This theorem says that when a functional language is considered and $\Sigma^\omega_{\bot,n}$ is implemented as the (infinite) list type and therefore we can use the 'cons' operator on the type, then we need parallel execution mechanism for expressing IM2-computable functions. When a $1\bot$-sequence is given as an output of an IM2-machine, it has the property that the first or the second element is computed as the result of one sequential computation. However, when $\Sigma^\omega_{\bot,1}$ is implemented as the (infinite) list type, we cannot assume this property because we can construct, with the cons operation, an infinite list whose first two elements may be unrelated. This is actually what we did in the above proof.

6 The language GHC (Guarded Horn Clauses)

Our goal is to find a platform on which we can execute our real-number algorithms with Gray-code. As we studied in the previous section, we cannot use sequential functional languages. In the following sections, we study implementations of IM2-machines in committed choice logic programming languages.

GHC (Guarded Horn Clauses) is a simple parallel logic programming language for programming with communicating processes. It is based on Horn-clause logic programming, and has the notion of guarded clause and committed choice. In this language, the system does not search solutions by backtracking. Therefore, the value of a variable once computed is never revoked. In the following sections, we will study implementations of IM2-machines in the language KL1[13], which is based on the concept of Flat GHC [14,15]. Since Flat GHC is a sublanguage of GHC, we will simply call the language GHC.

A GHC program is a set of *guarded clauses* of the following form:

$$H \;\texttt{:-}\; G_1, \ldots, G_n \;\;\texttt{|}\;\; B_1, \ldots, B_m\,.$$

Here, H, G_i's, and B_i's are atomic formulas. H is called a *clause head*, G_i's are called *guard goals*, and B_i's are called *body goals*. The part of a clause before "|" (including the clause head) is called the *guard*, and the rest is called the *body*. The guard specifies the condition which needs to be satisfied to apply the clause. The body specifies the action to be taken when it is selected. When a goal G satisfies the guard of some program clause, we say that G is *ready*, and otherwise, G is *suspended*. Each variable in GHC is a logical variable; a data some of whose part includes variables can be assigned.

In implementing IM2-machines, we only use integers and lists. In addition, we do not use guard goals and thus use clauses of the form

$$H \;\texttt{:-}\; B_1, \ldots, B_m.$$

In this case, a goal is ready when it matches the head of one of the clauses. As body goals, we only use unifications (i.e. goals with predicate =) and invocations of user-defined predicates. Body unifications are used to generate a substitution and constrain the possible values of variables.

For the execution of a program, we consider a multi-set of goals, called the *goal-set*. The execution of a program starts with the initial goal-set "{main}", and proceeds as follows. (1) One ready goal A is selected from the goal-set. (2, the reduction step) Choose one clause C whose guard is satisfied by A, execute the unification goals in it, and replace the goal A in the goal-set with the body of C. (3) Repeat these steps until the goal-set becomes empty. In GHC, the word

```
main :- pinf(Y), sum(X,Y), naturals(0,X).
naturals(N, X) :- X = [N|XX], N1 := N + 1, naturals(N1, XX).
sum([X|XX],Y) :- X mod 2 =:= 0 | Y = [X|YY], sum(XX,YY).
sum([X|XX],Y) :- X mod 2 =\= 0 | sum(XX,Y).
pinf([N|Y]) :- builtin:print(N),  pinf(Y).
```

Program 1.   A program example of GHC

process is used informally as a goal whose reduction is recursively defined and thus produces a goal with the same predicate. Processes communicate with each other through variables shared by them.

Program 1 is a simple GHC program which outputs even numbers infinitely. It is composed of three processes. `naturals(0,X)` produces an infinite list $0, 1, 2, 3, 4, \ldots$. `sum(X,Y)` writes the input value of `X` to `Y` when it is even, and discards it when it is odd. `pinf(Y)` displays the value obtained from the input stream. `=:=` and `=\=` are equality and in-equality predicates, respectively. We explain how this program is executed. $\{\mathtt{main}\}$ is the initial goal-set, which is reduced to $\{\mathtt{pinf(Y)}, \mathtt{sum(X,Y)}, \mathtt{naturals(0,X)}\}$. In this set, only $\mathtt{naturals(0,X)}$ is ready and therefore it is selected. After the reduction, `X` is bound to `[0|XX]` and $\mathtt{naturals(0,X)}$ is replaced with $\mathtt{naturals(1,XX)}$ in the goal set. Then, since $[0|\mathtt{XX}]$ matches `[X|XX]` and `0 mod 2 =:= 0`, the first clause of `sum` becomes ready and is selected. Thus, `Y` is bound to $[0|\mathtt{YY}]$ and $\mathtt{pinf}([0|\mathtt{YY}])$ becomes ready.

Note that this results from a particular order of evaluation to evaluate from left to right in the goal-set. It is adopted in klic-3.003[16], which is an implementation of KL1 for non-parallel computers. However, the language GHC itself does not specify the order ready goals are selected. In particular, the language GHC does not prescribe fair scheduling and thus a ready goal may not be selected forever. In this example, since a goal with the predicate `naturals` is always in the goal-set and it is also possible that this clause is always selected and `sum` and `pinf` are never executed. In this sense, this is not a correct GHC program. We will explain in Section 9 how to write a program which does not rely on a particular scheduling policy.

GHC has two kinds of nondeterminism. One is caused by the order in which a goal is selected from the set of ready goals as we mentioned above. The other one is caused by the choice of a clause when multiple clauses satisfy a goal. The former one is usually called “and-nondeterminism.” The latter one is usually called “don’t-care” or committed-choice nondeterminism. In KL1, we can specify goal priority and clause priority to control these nondeterminism to some extent. In this paper, we do not consider this priority mechanism and study how the indeterminism of IM2-machines is related to the nondeterminism of GHC programs.

```
main             :- pinf(ZZ),gtos(YY,ZZ),stog(XX,YY),inf0(XX).

stog([-1|X],YY) :- YY=[0|Y],  stog(X,Y).
stog([1|X],YY)  :- YY=[1|Y],nh(Z,Y),stog(X,Z).
stog([0|X],YY)  :- YY=[C,1|Y],nh(Z,Y),stog(X,[C|Z]).

gtos([0|Y],XX)  :- XX = [-1|X],gtos(Y,X).
gtos([1|Y],XX)  :- XX=[1|X],nh(Y,Z),gtos(Z,X).
gtos([C,1|Y],XX):- XX=[0|X],nh(Y,Z),gtos([C|Z],X).

inf0(XX)        :- XX = [0|X], inf0(X).
pinf([X|Y])     :- io:outstream([print(X),flush]),pinf(Y).
nh(X,XX)        :- X=[X0|X1],not(X0,Z),XX=[Z|X1].
not(0,X)        :- X = 1.
not(1,X)        :- X = 0.
```
Program 2. Conversions between Gray-code and the signed-digit representations.

7 Implementation in GHC

As is explained in the previous section, we can express process communications through streams, simply by assigning the same logical variable to an argument of the producer process and that of the consumer process. Since a logical variable is used, we can assign to it a term which may contain variables. In particular, we can instantiate the second member of a stream leaving the head as a variable. Therefore, it is expected that an extended stream with multiple heads can also be expressed and thus the behavior of an IM2-machine can be implemented in this language.

Program 2 is an implementation of *stog* and *gtos* functions written in this way in GHC. These programs have the same problem as Program 1, and it works only with fair scheduling of ready goals, which will be discussed in Section 9. This program is composed of 4 processes. `inf0` produces an infinite list `[0,0,0,...]`, `stog` and `gtos` realize *stog* and *gtos* functions, respectively, and `pinf` outputs the stream to the display. These processes are connected in this order and the connections between `inf0` and `stog` and between `gtos` and `pinf` are ordinary streams and between `stog` and `gtos` is an extended stream with two-head accesses.

As the last body-goal of the third clause of `stog` shows, we can leave the first character unbound and instantiate the second character with 1. On the other hand, since the check of the guard parts of the clauses are done in parallel, we can naturally express and execute conditions of the form "if the first character is 0 then execute something and if the second character is 1 then execute another thing," as the clauses of `gtos` show. In lazy functional languages, we have demand-driven (or top-down) control in that the computation of an expression starts when a request is given by the context. On the other hand, in GHC, computation is done rather in a bottom-up fashion; it starts when there is enough information to make it, and as the result, more information is given as the bindings of the variables.

The computational rules of an IM2-machine can be translated into a GHC program with one predicate as follows.

Translation 1: The translation is almost the same as the translation into Haskell syntax in Section 5. We explain it for the following rule of an IM2-machine of type $(\Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2})$ for $\Sigma = \{0,1\}$, which has the state-set $Q \subset \mathbb{N}$ (with the initial state 0) and one worktape W.

$$5, H_2(I1)(0), W(0) \Rightarrow 7, H_2(O)(1), W(1), -(W). \tag{1}$$

This rule says that if the state is 5, the character under the second head of the input tape $I1$ is 0, and the character under the head of W is 0, then move to state 7, output 1 from the second head of the output tape, write 1 to the head position of W, and move the head of W to the left. The GHC clause corresponding to this rule is as follows:

$$\begin{aligned} &\mathtt{mm(5,[D,0|X],YY,[T|W1],[0|W2],ZZ)}\ \text{:-} \\ &\qquad \mathtt{ZZ=[C,1|Z],mm(7,[D|X],YY,W1,[T,1|W2],[C|Z])}. \end{aligned} \tag{2}$$

By adding the following clause `m` which assigns the initial values, we complete the translation.

```
m(XX,YY,ZZ) :- mm(0,XX,YY,[],ALLB,ZZ).
```

Here, `ALLB` is the infinite sequence $[-\mathtt{1},-\mathtt{1},-\mathtt{1},...]$ with -1 representing the blank character. Though GHC does not have the power to express infinite sequences, we consider here an extended operational semantics which allows bindings of infinite sequences to variables. Later in Definition 19, we will define another realization which does not use such an extended semantics.

Definition 10 A partial multi-valued function $f :\subseteq (\Sigma^{\omega}_{\perp})^k \rightrightarrows \Sigma^{\omega}_{\perp}$ is *realized* by a GHC-goal $\mathtt{m(X1,\ldots,Xk,X0)}$ if every execution of $\mathtt{m}(p_1,p_2,\ldots,p_k,\mathtt{Z})$ for $(p_1,\ldots,p_k) \in dom(f)$ will produce on $\mathtt{Z}$ an element p of $f(p_1,\ldots,p_k)$. That is, for every infinite computational path of the program, there is a $p \in f(p_1,\ldots,p_k)$ such that a character c is assigned to $\mathtt{Z}[k]$ when $p[k]=c$ and no value is assigned to $\mathtt{Z}[k]$ when $p[k]=\perp$. If f is realized by a GHC-goal, we say that f is *GHC-computable.*

Through the comparison of the operational semantics of IM2-machines and GHC programs, we have the following.

Theorem 11 *Suppose that an IM2-machine M of type $(\Sigma^{\omega}_{\perp,n_1},\ldots,\Sigma^{\omega}_{\perp,n_k},\Sigma^{\omega}_{\perp,n_0})$*

is translated into the GHC program `m(X1, X2,...,Xk, ZZ)` *by Translation 1. Then, a multi-valued function* $f :\subseteq \Sigma^\omega_{\perp,n_1} \times \ldots \times \Sigma^\omega_{\perp,n_k} \rightrightarrows \Sigma^\omega_{\perp,n_0}$ *is realized by* M *iff* f *is realized by* `m`*.*

The program obtained through Translation 1 contains only one non-unification goal in the body of each clause. Therefore, the goal-set contains only one goal throughout the execution, and parallel execution is done only in the check of the guards. Thus, "and-nondeterminism" is not used in implementing IM2-machines as GHC programs and indeterminism of an IM2-machine corresponds to the committed-choice nondeterminism of GHC programs, among the two kinds of nondeterminism of GHC we explained in Section 6. We will discuss this again in Section 9.

Corollary 12 *When we consider multi-valued functions in* $\Sigma^\omega_{\perp,n_1} \times \ldots \times \Sigma^\omega_{\perp,n_k} \rightrightarrows \Sigma^\omega_{\perp,n_0}$*,*

$$\textit{IM2-computable functions} \subset \textit{GHC-computable functions.}$$

This inclusion relation is strict in general. As a counter example, consider the identity function id_n from $\Sigma^\omega_{\perp,n}$ to $\Sigma^\omega_{\perp,n}$. It is realized by the GHC program

```
idghc(X,Z) :- X = Z.
```

However, it is not realizable by an IM2-machine. A candidate for an IM2-machine realizing the identity function on $\Sigma^\omega_{\perp,1}$ for $\Sigma = \{0, 1\}$ is the following:

```
id([0|X], Y) :- Y = [0|Z], id(X, Z).
id([1|X], Y) :- Y = [1|Z], id(X, Z).
id([C,0|X], Y) :- Y = [D,0|Z], id([C|X], [D|Z]).
id([C,1|X], Y) :- Y = [D,1|Z], id([C|X], [D|Z]).
```

Here, we present the GHC program obtained through Translation 1 for simplicity. This program may activate only the third rule for the input $[0, 0, 0, ...]$. That is, the first head may be left at the first cell and $[\perp, 0, 0, 0 \ldots]$ may be produced as the result. Therefore, this IM2-machine does not realize id_n.

Proposition 13 *1) When* $f :\subseteq \Sigma^\omega_{\perp,n} \rightrightarrows \Sigma^\omega_{\perp,n}$ *is IM2-computable and* $p < q$ *for* $p, q \in dom(f)$*, we have* $f(p) \subseteq f(q)$*.*
2) id_n *is not IM2-computable for* $n \geq 1$*.*
In particular, there is a GHC-computable function which is not IM2-computable.

PROOF. 1) Suppose that an IM2-machine M, when applied to p, produces r following a computational path L. Then, for every $q > p$, M can take the

same computational path and thus may produce the same output r.
2) Immediate from (1). ∎

This proposition means that we cannot show that a function is IM2-computable by giving a GHC program. However, as we show in the next section, when restricted to $image(G) \subset \Sigma^{\omega}_{\perp,1}$ of the image of the Gray code embedding, IM2-computability and GHC-computability coincide. Therefore, when we consider real number computation, we can use GHC programs instead of IM2-machines.

8 The equivalence of IM2- and GHC-computability for real number computation

In this section, we show the equivalence of IM2-computability and GHC-computability of real-valued functions encoded through the Gray-code embedding. We prove this fact more generally for the case that a function is defined on subspaces of $\Sigma^{\omega}_{\perp,n}$ composed of minimal limit elements of some domain structures [11].

First, we prepare some notions which we use in this section. In this paper, we use the word *domain* for an ω-algebraic pointed dcpo. We write $K(D)$ for the set of finite (i.e. compact) elements of D, and $L(D)$ for the set of limit (i.e. non-finite) elements of D. We write $K(x)$ for the set of finite elements below x. We consider domains with concrete structures. When P is a poset, we define the *level* of $d \in P$ as the maximal length of a chain $\perp_P = a_0 < a_1 < \ldots < a_n = d$, when it exists. We say that a domain D is *stratified* if each $e \in K(D)$ has a level. When D is a stratified domain, we write $K_n(D)$ for the set of level-n finite elements of D, $K_n(p)$ for $K(p) \cap K_n(D)$, and $K_n(S)$ for $\cup\{K_n(p) \mid p \in S\}$. Thus, when D is a stratified domain, $K(D)$ is stratified as $K(D) = K_0(D) \cup K_1(D) \cup \ldots$ and $K_0(D) = \{\perp_D\}$. The poset $(\Sigma^{\omega}_{\perp}, \leq)$ forms a stratified domain, with $K(\Sigma^{\omega}_{\perp}) = \Sigma^{*}_{\perp}$. For $\Sigma^{\omega}_{\perp}$, the level of d is just the number of Σ-characters in d. In a poset P, when $d < d'$ and there is no element e such that $d < e < d'$, we say that d' is an *immediate successor* of d and call the pair (d, d') a *successor pair* or an *edge* from d to d'. We write $succ(d)$ for the set of outgoing edges from d.

We say that a domain D has *enough minimal limit elements* if, for each $y \in L(D)$, there exists a minimal element x of $L(D)$ such that $x \leq y$. The domain $\Sigma^{\omega}_{\perp}$ does not have enough minimal limit elements. On the other hand, we can show that every finite-branching domain (i.e., stratified domain D such that $succ(d)$ is a finite set for every $d \in K(D)$) has enough minimal limit elements [11]. We write $M(D)$ for the set of minimal elements of $L(D)$ when D has enough minimal limit elements.

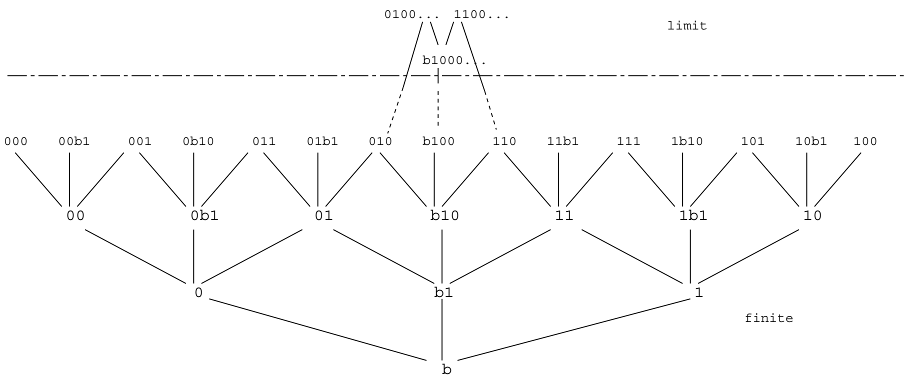

Fig. 2. The structure of $\mathcal{D}_{\mathcal{R}}$. In this figure, b means $\perp$.

We write $\mathcal{D}_{\Sigma,n}$ for the subdomain of $\Sigma_{\perp}^{\omega}$ with $K(\mathcal{D}_{\Sigma,n}) = \Sigma_{\perp,n}^{*}$ and $L(\mathcal{D}_{\Sigma,n}) = \Sigma_{\perp,n}^{\omega}$. Note that $K(\mathcal{D}_{\Sigma,n})$ and $L(\mathcal{D}_{\Sigma,n})$ are the sets of finite-time and infinite-time states of tapes of IM2-machines, respectively. $\mathcal{D}_{\Sigma,n}$ is a finite-branching domain and $M(\mathcal{D}_{\Sigma,n})$ is the set of bottomed sequences with just n copies of $\perp$. We write $\mathcal{D}_{\mathcal{R}}$ for the subdomain of $\mathcal{D}_{\Sigma,1}$ which corresponds to Gray-code. That is, $L(\mathcal{D}_{\mathcal{R}}) = \Sigma^{\omega} \cup \Sigma^{*}\perp 10^{\omega}$ and $K(\mathcal{D}_{\mathcal{R}}) = \Sigma^{*} \cup \Sigma^{*}\perp 10^{*}$ for $\Sigma = \{0,1\}$, as Figure 2 shows. $\mathcal{D}_{\mathcal{R}}$ is a finite-branching domain and $M(\mathcal{D}_{\mathcal{R}})$ is the image of the Gray-code embedding G, and thus homeomorphic to $\mathbb{I}$.

We introduce a labelling of edges of $K(\mathcal{D}_{\Sigma,n})$ with the character set $\Xi_{\Sigma,n}$ defined in Section 5, so that the label $a^{(i)}$ is assigned to an edge filling the i-th unfilled cell with a. For example, the edge from $\perp^{\omega}$ to $\perp 1 \perp^{\omega}$ is labeled with $1^{(2)}$, and the edges from $\perp 1 \perp^{\omega}$ to $\perp 10 \perp^{\omega}$ and from $\perp 10 \perp^{\omega}$ to $010\perp^{\omega}$ are labeled with $0^{(2)}$ and $0^{(1)}$, respectively. Then, this labelling induces, for a subdomain D of $\mathcal{D}_{\Sigma,n}$, a naming system (i.e., a partial surjective function) $\phi_{K(D)} : {\Xi_{\Sigma,n}}^{*} \rightharpoonup K(D)$ and a representation $\phi_{L(D)} : {\Xi_{\Sigma,n}}^{\omega} \rightharpoonup L(D)$.

Our goal is to use subdomains of $\mathcal{D}_{\Sigma,n}$ to restrict the behavior of IM2-machines. We say that D is a *$n\perp$-domain* when (1) $K(D) \subseteq K(\mathcal{D}_{\Sigma,n})$ and $L(D) \subseteq L(\mathcal{D}_{\Sigma,n})$, (2) the embedding of D in $\mathcal{D}_{\Sigma,n}$ is full (i.e., $d < e$ in D iff $d < e$ in $\mathcal{D}_{\Sigma,n}$) and preserves the least element, the levels of finite elements, and the supremums of directed sets, and (3) the set $\phi_{K(D)}^{-1}(K(D))$ is recursive.

From (2), the level of $d \in K(D)$ is just the number of Σ-characters in d, and an edge from d to d' corresponds to an operation to fill one unfilled cell of d. Therefore, $K(D)$ is defining a restriction on the way a tape is filled by an IM2-machine, and $L(D)$ is the set of $n\perp$-sequences obtained as an output of an IM2-machine which outputs an element of $K(D)$ at each finite time. The condition (3) is equivalent to the condition that for each $d \in K(D)$, the set $succ(d)$ is computable from the $\phi_{K(D)}$-name of d.

$\mathcal{D}_{\Sigma,n}$ is an $n\perp$-domain with $succ(d) = \Xi_{\Sigma,n}$ for every d, and $\mathcal{D}_{\mathcal{R}}$ is a $1\perp$-domain with $succ(d) = \{0^{(1)}, 1^{(1)}, 1^{(2)}\}$ when $d \in \Sigma^{*}$ and $succ(d) = \{0^{(1)}, 1^{(1)}, 0^{(2)}\}$

when $d \in \Sigma^*\perp 10^*$. When $x \in L(D)$, every infinite strictly increasing sequence in $K(x)$ converges to an element of $y \in L(D)$ such that $y \leq x$. Therefore, the following lemma is immediate.

Lemma 14 *When D is an $n\perp$-domain and $x \in M(D)$, every infinite strictly increasing sequence in $K(x)$ converges to x.*

Definition 15 Let D_i $(i = 0, \ldots, k)$ be $n_i\perp$-domains. A partial multi-valued function $f :\subseteq L(D_1) \times \ldots \times L(D_k) \rightrightarrows L(D_0)$ is $(D_1 \ldots, D_k, D_0)$-*IM2-realized* by an IM2-machine M iff f is realized by M and M inputs/outputs $n_i\perp$-sequences only following the structure of $K(D_i)$ on each input/output tape T_i $(i = 0, 1, \ldots, k)$. We say that f is $(D_1 \ldots, D_k, D_0)$-*IM2-computable* when there is an IM2-machine which $(D_1 \ldots, D_k, D_0)$-IM2-realizes f.

For $a^{(i)} \in \Xi_{\Sigma,n}$, we say that an IM2-machine inputs/outputs as $a^{(i)}$ specifies when it inputs/outputs a from the i-th head.

Proposition 16 *When D is an $n\perp$-domain, the identity function on $M(D)$ is (D, D)-IM2-computable.*

PROOF. First, note that $succ(d) \subset \Xi_{\Sigma,n}$ and therefore $succ(d)$ is selected from the powerset of $\Xi_{\Sigma,n}$. We consider a machine M which has the state set the union of the powerset of $\Xi_{\Sigma,n}$ and $\{-1\}$. For each $\alpha \subset \Xi_{\Sigma,n}$ and $a \in \alpha$, M has a rule which says that if the state is α and it has an input specified by a, change the state to "-1", output as specified by a, and put a to the end of a worktape W. That is, W remembers $\phi_{K(D)}$-name of the sequence it has already input. When the state is "-1", it calculates $\alpha = succ(d)$ for the current input state d written on W, and change the state to α. When a sequence $p \in M(D)$ is given on the input tape, M inputs p infinitely, and therefore the way M inputs p forms an infinite increasing sequence in $K(D)$. Since it converges to p by Lemma 14, M reads all the characters of p, and copies it to the output tape. ∎

From the definition, when D_i is an $n_i\perp$-domain, $(D_1, \ldots, D_k, D_0)$-IM2-computability implies IM2-computability. The converse is also true if f is defined on $M(D_1) \times \ldots \times M(D_n)$:

Proposition 17 *Let $f :\subseteq M(D_1) \times \ldots \times M(D_k) \rightrightarrows M(D_0)$ for D_i an $n_i\perp$-domain. If f is IM2-computable, then f is $(D_1 \ldots, D_k, D_0)$-IM2-computable. In particular, f is realized by a machine which inputs characters from all the input tapes simultaneously, and behaves deterministicly inside in the sense that if two executions of M have the same order of inputs, then they have the same computational path.*

PROOF. Suppose that an IM2-machine M of type $(\Sigma^{\omega}_{\perp,n_1}, \ldots, \Sigma^{\omega}_{\perp,n_k}, \Sigma^{\omega}_{\perp,n_0})$ which realizes f is given. The new machine M' has worktapes to remember the contents and the head positions of the input tapes of M. M' simulates the behavior of M. That is, (1) if there is an applicable rule of M to the contents of the worktapes, then execute it. (2) If there is no applicable rule, M' reads one character form each input tape in the same way as we did in Proposition 16 and copy them on worktapes. M' repeats (1) and (2). If M' has the stage (2) infinitely many times, M' reads all the characters of the input tapes from the proof of Proposition 16. Therefore, at least one rule of M becomes applicable after a finite repetition of stage (2). This means that M' simulates an infinite computational path of M. In step (1), a machine can judge if each rule is applicable or not because M' uses only worktapes which includes 'B' instead of $\perp$. Therefore, we can define M' so that M' searches for an applicable rule of M in a deterministic way. When M has an output, the location of the output may not be allowed by the structure of $K(D_0)$. Therefore, M' first stores it on a worktape, and M' outputs only when the content of the worktape becomes a member of $K(D_0)$. ∎

In Proposition 17, we considered a simulation of an IM2-machine by another IM2-machine. In the same way, an IM2-machine can simulate a GHC-program when the function it realizes is restricted to $M(D_1) \times \ldots \times M(D_k) \rightrightarrows M(D_0)$. In this case, an IM2-machine can copy the inputs to the worktapes incrementally as we did in Proposition 17. Since an IM2-machine is a generalization of a Turing machine, we can also write an interpreter of GHC as an IM2-machine. After each step of the execution of a GHC-program, it checks what is bound to the variable corresponding to the output stream, and it outputs when the content of the variable is a member of $K(D_0)$.

Theorem 18 *1) When $f :\subseteq M(D_1) \times \ldots \times M(D_k) \rightrightarrows M(D_0)$ for D_i an $n_i\perp$-domain $(i = 0, \ldots, k)$, f is IM2-computable iff f is GHC-computable.*
2) In particular, $f :\subseteq \mathbb{I}^k \rightrightarrows \mathbb{I}$ is Gray-code computable iff its embedding in $(\Sigma^{\omega}_{\perp,1})^k \rightrightarrows \Sigma^{\omega}_{\perp,1}$ by G is GHC-computable.

This justifies the use of GHC, instead of IM2-machines, as a language to define effectivity on the reals. Therefore, GHC programs like Program 2 can be considered as defining Gray-code computable functions, and those programs written in the Haskell syntax which can be directly translated into GHC programs can also be considered as defining Gray-code computable functions.

```
main            :- pinf(ZZ),gtos_d(YY,ZZ),stog_d(XX,YY),inf0(XX).
inf0([X1|X]) :- X1 = 0, inf0(X).

gtos_d(YY, [X1|X]) :- YY = [Y1,Y2|Y], gtos1(YY, [X1|X]).
gtos1([0|Y], XX) :-  XX = [-1|X], gtos_d(Y, X).
gtos1([1|Y], XX) :- XX = [1|X], nh(Y, Z), gtos_d(Z, X).
gtos1([C,1|Y], XX) :- X1 = [0|X], nh(Y, Z), gtos_d([C|Z], X).

stog_d(XX, [Y1,Y2|Y]) :- XX = [X1|X], stog1(XX, [Y1,Y2|Y]).
stog1([-1|X], YY):- YY = [0|Y],  stog_d(X, Y).
stog1([1|X], YY):- YY = [1|Y], nh(Z,Y),  stog_d(X, Z).
stog1([0|X], YY):- YY = [C,1|Y], nh(Z, Y),  stog_d(X, [C|Z]).
```
Program 3. Demand-driven implementations of `gtos` and `stog` in GHC

9 Demand-driven implementation in GHC

We define the composition of GHC processes as we did in the main clause of Program 2. That is, by sharing a logical variable between a producer process and a consumer process. Suppose that $\mathtt{m_i}$ ($i = 1, \ldots, k$) are GHC processes realizing $f_i :\subseteq Y_1 \times \ldots \times Y_l \rightrightarrows Z_i$ and $\mathtt{n}$ is a GHC process realizing $g :\subseteq Z_1 \times \ldots \times Z_k \rightrightarrows Z$. Then, we write $\mathtt{n} \circ \langle \mathtt{m_1}, \ldots, \mathtt{m_k} \rangle$ for the process composed by connecting the i-th input of `n` with the output of $\mathtt{m_i}$ for $i = 1, \ldots, k$, and assigning the same variable to the j-th inputs of $\mathtt{m_1}, \ldots, \mathtt{m_k}$ for each $j = 1, \ldots, l$. Then, it is expected that $\mathtt{n} \circ \langle \mathtt{m_1}, \ldots, \mathtt{m_k} \rangle$ realizes $g \circ \langle f_1, \ldots, f_k \rangle$. However, it it not true; when we cannot expect fair scheduling, there is a computational path that executes only the clauses of $\mathtt{m_1}$ and $\mathtt{n} \circ \langle \mathtt{m_1}, \ldots, \mathtt{m_k} \rangle$ does not have an output.

In [10], it is proved that $g \circ \langle f_1, \ldots, f_k \rangle$ is IM2-computable if the component functions are IM2-computable. However, the set of rules of the IM2-machine for $g \circ \langle f_1, \ldots, f_k \rangle$ is not a simple collection of the rules of the components, but rather a complicated one in order to ensure that all the components are fairly scheduled. When we consider an implementation in a real programming language GHC, a function composition should be expressed as a composition of processes.

For the case of usual stream programming in GHC, this problem is solved by writing each process in a demand-driven way. That is, the producer process is kept suspended until the consumer process raises a demand for a value. A demand can be conveyed from the consumer process to the producer process by instantiating the variable representing the stream with a cons cell.

This programming technique applies also to the case of multi-head stream access. The clauses `stog_d` and `gtos_d` in Program 3 are the *stog* and *gtos* functions rewritten in this way, respectively. One thing to note is that we need to instantiate the variable representing the stream not with one cell but with a list of cells with the length $n+1$, where n is the number of bottoms which may appear on the stream. In this example, we instantiate it with a list of two cells on the first line of gtos_d. Otherwise, the goal is activated by mistake at the place one is preparing the new stream by removing the value given on the second head (in this example, the body of the third clause of stog1).

We formalize this notion of a demand-driven GHC process, which starts the calculation of the next output when a demand is raised by the consumer process. We consider that a stream is filled following the graph structure of $K(D)$ for some $n_i\bot$-domain D. For example, in the stog_d program, the output stream is filled following $K(\mathcal{D}_{\mathcal{R}})$. GHC-computability defined in Definition 10 was based on an extended operational semantics which allows bindings of infinite sequences to variables. The computability we define here does not use such an extended semantics.

Definition 19 Suppose that D is an $n\bot$-domain, $p \in L(D)$, and $S \subseteq L(D)$. We say that the goal m(X) *D-dd-realizes* p (or S) if it is suspended when X is not bound to a sequence of cons cells of length at least $n+1$, and for $l \geq n+1$, the execution of $\{\mathtt{m(X)}, \mathtt{X} = [\mathtt{Z_1}, \ldots, \mathtt{Z_l}|\mathtt{Z}]\}$ produces an element of $K_{l-n}(p)$ (or $K_{l-n}(S)$) on X and become suspended. In addition, the execution of $\{\mathtt{m(X)}, \mathtt{out(X)}\}$ with the following program produces infinite output on X, which is p (or a member of S).

```
out(X):-X = [Z1, ..., Zn+1|Z], dd(X).
dd([0|X]):-out(X).
dd([1|X]):-out(X).
dd([C, 0|X]):-out([C|X]).
...
dd([C1, ..., Cn, 1|X]):-out([C1, ..., Cn|X]).
```

Here, a clause with the head predicate dd exists for each one-character output of $\Sigma^{\omega}_{\bot,n}$.

In this definition, it is more natural to change the set of clauses with the head predicate dd depending on the string $d \in K(D)$ that m has already output, so that it accepts only the elements of $succ(d)$. We defined as above only because it is equivalent and much simpler. Next, we define demand-driven GHC processes which input and output $n\bot$-domain elements.

Definition 20 Suppose that D_i is an $n_i\bot$-domain ($i = 0, 1, \ldots, k$) and $f :\subseteq L(D_1)\times\ldots\times L(D_k) \rightrightarrows L(D_0)$. We say that the goal $\mathtt{m(X1, \ldots, Xk, Z)}$ $(D_1, \ldots, D_k, D_0)$-*dd-realizes* f (in short for $(D_1, \ldots, D_k, D_0)$-realizes f in a demand-driven way) if the following goal om(Z) D_0-dd-realizes the set $f(p_1, \ldots, p_k)$ when $oracle_{p_i}(\mathtt{Xi})$ is any oracle goal which D_i-dd-realizes p_i.

om(Z) :- $oracle_{p_1}$(X1), ..., $oracle_{p_k}$(Xk), m(X1, ..., Xk, Z).

Here, we allow an oracle goal which may not be expressible as a GHC program. We say that f is $(D_1, \ldots, D_k, D_0)$-*dd-computable* if there is a goal m(X1, ..., Xk, Z) which $(D_1, \ldots, D_k, D_0)$-dd-realizes f.

Lemma 21 *Suppose that the GHC goal* m(X1, ..., Xk, Z) $(D_1, \ldots, D_k, D_0)$*-dd-realizes* $f :\subseteq L(D_1) \times \ldots \times L(D_k) \rightrightarrows L(D_0)$*. Then, the following goal* oms(Z) D_0*-dd-realizes the set* $f(S_1, \ldots, S_k)$ *for* $S_1 \times \ldots \times S_k \subseteq dom(f)$*. Here,* $oracle_{S_i}$(Xi) *is any oracle goal which* D_i*-dd-realizes* S_i*.*

oms(Z) :- $oracle_{S_1}$(X1), ..., $oracle_{S_k}$(Xk), m(X1, ..., Xk, Z).

Proposition 22 *Suppose that* $\mathtt{m_i}$ $(i = 1, \ldots, k)$ *are GHC goals which* $(D_1, \ldots, D_l, E_i)$*-dd-realizes* $f_i :\subseteq L(D_1) \times \ldots \times L(D_l) \rightrightarrows L(E_i)$ *and* n *is a GHC goal which* $(E_1, \ldots, E_k, E)$*-dd-realizes* $g :\subseteq L(E_1) \times \ldots \times L(E_k) \rightrightarrows L(E)$*. Then,* $\mathtt{n} \circ \langle \mathtt{m_1}, \ldots, \mathtt{m_k} \rangle$ $(D_1, \ldots, D_l, E)$*-dd-realizes* $g \circ \langle f_1, \ldots, f_k \rangle$*.*

PROOF. Immediate from Lemma 21. ∎

Now, we define the following translation of an IM2-machine to a GHC program.

Translation 2: We explain the translation for the computational rule (1) of an IM2-machine of type $(\Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2})$ in Section 7. It is translated into the following clause:

$$\begin{aligned} &\mathtt{mr(5,[D,0|X],YY,[T|W1],[0|W2],ZZ)} \\ &\qquad \mathtt{:\text{-}ZZ = [C,1|Z], mm(7,[D|X],YY,W1,[T,1|W2],[C|Z]).} \end{aligned} \tag{3}$$

That is, only modifying the head predicate of the clause (2) obtained by Translation 1 from mm to mr. In addition, we add the following clauses to the whole program.

$$\begin{aligned} &\mathtt{m(X,Y,Z)\text{:-}mm(0,X,Y,[\,],ALLB,Z), allb(ALLB).} \\ &\mathtt{mm(Q,XX,YY,W1,W2,[Z1,Z2|Z])\text{:-}XX = [X1,X2|X], YY = [Y1,Y2|Y],} \\ &\qquad\qquad \mathtt{W2 = [W|W3], mr(Q,XX,YY,W1,W2,[Z1,Z2|Z]).} \\ &\mathtt{allb([A|B])\text{:-}A = 0, allb(B).} \end{aligned} \tag{4}$$

It is expected that if M is an IM2-machine which $(D_1 \ldots, D_k, D_0)$-IM2-realizes a multi-valued function f, then the GHC program m obtained by Translation 2 $(D_1 \ldots, D_k, D_0)$-dd-realizes f. However, it is true only for a special type of IM2-machine. For example, we can consider a Type2-machine which realizes $f :\subseteq \Sigma^\omega \to \Sigma^\omega$ as a special case of an IM2-machine which $(\mathcal{D}_{\Sigma,1}, \Sigma^\infty)$-IM2-realizes f; it has two heads on the input tape but only uses H_1. However, the GHC program m obtained through Translation 2 does not $(\mathcal{D}_{\Sigma,1}, \Sigma^\infty)$-dd-realize f, because when executed with a oracle predicate $oracle_p$(X) which $\mathcal{D}_{\Sigma,1}$-dd-realizes $p \in \Sigma^\omega$, $oracle_p$(X) may instantiate X with [C, 1|X] and m does not have a corresponding rule. Therefore, we define such a special kind of IM2-machine, which accepts all the elements of $succ(d)$ when it has already input d.

Definition 23 Suppose that D_i are $n_i\bot$-domains $(i = 0, 1, \ldots, k)$ and M is an IM2-machine which $(D_1, \ldots, D_k, D_0)$-IM2-realizes a partial multi-valued function f. We call each finite-time state of M obtained with the input $(p_1, \ldots, p_k) \in dom(f)$ a *configuration* of $(M, p_1, \ldots, p_k)$, and the strings $(d_1, \ldots, d_k) \in K(p_1) \times \ldots \times K(p_k)$ read at that time the input-state of the configuration. We say that M *strongly* $(D_1, \ldots, D_k, D_0)$*-IM2-realizes* f if, for each $(p_1, \ldots, p_k) \in dom(f)$ and for each configuration C of $(M, p_1, \ldots, p_k)$ with the input state $(d_1, \ldots, d_k) \in K(p_1) \times \ldots \times K(p_k)$, if $(a_1, \ldots, a_k) \in succ(d_1) \times \ldots \times succ(d_k)$, then there is a computational path starting from C such that the next input is a sub-tuple of $(a_1, \ldots, a_k)$. Here, the input of a computational rule can be expressed as $(b_1, \ldots, b_k)$ for $b_i \in \Xi_{\Sigma,n_i} \cup \{\bot\}$, with $b_i = a \in \Xi_{\Sigma,n_i}$ when it inputs from the i-th stream as indicated by a, and $b_i = \bot$ when it does not have an input from the i-th stream. A sub-tuple means a tuple obtained by replacing some components with $\bot$.

The proof of Proposition 17 shows that the machine M' constructed in the proof strongly $(D_1, \ldots, D_k, D_0)$-IM2-realizes f. Therefore, we have

Proposition 24 *Let $f :\subseteq M(D_1) \times \ldots \times M(D_k) \rightrightarrows M(D_0)$ for D_i an $n_i\bot$-domain $(i = 0, 1, \ldots, k)$. If f is $(D_1 \ldots, D_k, D_0)$-IM2-computable, then there is a machine M which strongly $(D_1 \ldots, D_k, D_0)$-IM2-realizes f and which behaves deterministicly inside.*

Theorem 25 *Let D_i be an $n_i\bot$-domain $(i = 0, \ldots, k)$. Suppose that an IM2-machine M of type $(\Sigma^\omega_{\bot,n_1}, \ldots, \Sigma^\omega_{\bot,n_k}, \Sigma^\omega_{\bot,n_0})$ is translated into a program for the goal* m(X1,...,Xk,Z) *by Translation 2. If a multi-valued function $f :\subseteq M(D_1) \times \ldots \times M(D_k) \rightrightarrows M(D_0)$ is strongly $(D_1, \ldots, D_k, D_0)$-IM2-realized by M, then f is $(D_1, \ldots, D_k, D_0)$-dd-realized by* m.

PROOF. Consider the case $k = 2$ and $n_1 = n_2 = n_0 = 1$. For $(p_1, p_2) \in dom(f)$, the execution of $\{oracle_{p_1}$(XX), $oracle_{p_2}$(YY), m(XX, YY, ZZ), ZZ = [Z1, Z2|Z]$\}$

will cause the reduction of `m`, which will perform unifications `XX = [X1,X2|X]`, `YY = [Y1,Y2|Y]`, and `W2=[W|W3]`, and thus one of `X1` or `X2` and one of `Y1` or `Y2` is instantiated by the oracle predicates, and `W` is instantiated to 0 by `allb`. Suppose that `X2` and `Y2` are instantiated to 0. Since M strongly (D_1, D_2, D_0)-IM2-realizes f, for each pair of values given on the input tapes, at least one clause of `mr` is executable. When it executes a `mr` clause with inputs, after the reduction of `mm`, the corresponding input streams are instantiated with lists of cons cells with one more length, and thus oracle predicates will fill one more characters. In this way, an element of $succ(d_1) \times succ(d_2)$ is given on the input tape for (d_1, d_2) the portions of the streams currently read by the predicate. This repetition stops when a clause with an output is activated and a list of cons cells with length 1 is bound to `zz`. ∎

Thus, we have two steps of translation. By Proposition 24, every IM2-machine which $(D_1, \ldots, D_k, D_0)$-IM2-realizes f can be translated into an IM2-machine which strongly $(D_1, \ldots, D_k, D_0)$-IM2-realizes f. Then, it is translated into a GHC program which $(D_1, \ldots, D_k, D_0)$-dd-realizes f by Translation 2, as Theorem 25 shows. Note that, in many interesting cases, an IM2-machine which strongly $(D_1, \ldots, D_k, D_0)$-IM2-realizes f is given from the beginning and thus we do not need the first translation, as the following proposition shows.

Proposition 26 *If a total multi-valued function $f : M(\mathcal{D}_{\mathcal{R}})^k \rightrightarrows M(D)$ is $(\mathcal{D}_{\mathcal{R}}, \ldots, \mathcal{D}_{\mathcal{R}}, D)$-IM2-realized by M, then it is strongly $(\mathcal{D}_{\mathcal{R}}, \ldots, \mathcal{D}_{\mathcal{R}}, D)$-IM2-realized by M.*

PROOF. First, consider the case $k = 1$. $\mathcal{D}_{\mathcal{R}}$ has the property that for each $d \in K_n(\mathcal{D}_{\mathcal{R}})$, there is an element $p \in M(\mathcal{D}_{\mathcal{R}})$ such that $K_n(p) = \{d\}$. Therefore, for a successor pair (d', d) in $K(\mathcal{D}_{\mathcal{R}})$ and a configuration C with the input state d', M must have a rule to read the input from d' to d so that it can input infinitely when p is given as the argument, or it runs infinitely and produces an output without reading the input any more. Since d and d' are arbitrary, a configuration with the input-state d' must have all of the three ways of reading the next character, or it can execute without reading the input forever. Therefore, we have the result. The same argument applies to the case f has more than one arguments. ∎

The goal `stog_d` in Program 3 $(\Gamma^\infty, \mathcal{D}_{\mathcal{R}})$-dd-realizes the *stog* function, and `gtos_d` $(\mathcal{D}_{\mathcal{R}}, \Gamma^\infty)$-dd-realizes the *gtos* function. Program 4 is a GHC program which $(\mathcal{D}_{\mathcal{R}}, \mathcal{D}_{\mathcal{R}}, \mathcal{D}_{\mathcal{R}})$-dd-realizes the addition function (more precisely, it is the average function $\mathtt{pl}(\mathtt{x}, \mathtt{y}) = (\mathtt{x} + \mathtt{y})/2$ in order to adjust the result to $\mathbb{I}$) with respect to the Gray-code embedding, which is based on the algorithm written

```
pl(A,B,[Z1,Z2|Z]):-A=[A1,A2,A3|A],B=[B1,B2,B3|B],pl1(A,B,[Z1,Z2|Z]).
pl1([0|A],[0|B],ZZ) :-  ZZ = [0|Z], pl(A, B, Z).
pl1([1|A],[1|B],ZZ) :- ZZ = [1|Z], pl(A, B, Z).
pl1([0|A],[1|B],Z):-Z=[C,1|W],nh(B,BN),pl(A,BN,R),[C|D]=R,nh(D, W).
pl1([1|A],[0|B],Z):-Z=[C,1|W],nh(A,AN),pl(B,AN,R),[C|D]=R,nh(D,W).
pl1([C,1|A],[D,1|B],ZZ) :- ZZ=[E,1|Z], nh(ES,Z), [E|ES]=R,
                            nh(A,AN), nh(B,BN), pl([C|AN],[D|BN],R).
pl1([C,1,0|A],[0,0|B],ZZ):-ZZ=[0|Z],pl([C,1|A],[1|BN],Z),nh(B,BN).
pl1([C,1,0|A],[1,0|B],ZZ):-ZZ=[1|Z],pl(AN,[1|BN],Z),nh([C,1|A],AN),
                            nh(B,BN).
pl1([C,1,0|A],[0,D,1|B],ZZ):-ZZ=[0,1|Z],pl(ANN,BNN,Z),nh([C|AN],ANN),
                              nh([D|BN], BNN), nh(A, AN), nh(B,BN).
pl1([C,1,0|A],[1,D,1|B],ZZ):-ZZ=[1,1|Z],pl([C|AN],BNN,Z),
                              nh([D|BN],BNN), nh(A, AN), nh(B,BN).
pl1([0,0|A],[D,1,0|B],ZZ) :- ...
pl1([1,0|A],[D,1,0|B],ZZ) :- ...   symmetric to the above 4 clauses.
pl1([0,C,1|A],[D,1,0|B],ZZ) :- ...
pl1([1,C,1|A],[D,1,0|B],ZZ) :- ...
```
Program 4. GHC implementation of `pl`

with the Haskell syntax in [10]. Though they are not direct translations of IM2-machines, the translations from the Haskell syntax are essentially the same as Translation 2. A slight difference is that, since the original algorithm of `pl` reads two characters from each input $1\bot$-stream to determine which clause to apply, Program 4 instantiates a list of cells with length three on the first line. This is because the guard of the predicate `pl1` requires two elements out of the first three cells of each input stream.

Finally, we consider how programs written in a demand-driven way operates not on $M(D)$ but on $L(D)$. Since an IM2-machine M which strongly (D,E)-IM2-realizes f can accept, at the input-state d, all the inputs in $succ(d)$, we can consider that a sequence of $succ(d)$ characters is the input. That is, we can give, instead of p, an infinite sequence $s \in {\Xi_{\Sigma,n}}^{\omega}$ such that $\phi_{L(D)}(s) = p$ as an input. When, in addition to this, the machine behaves deterministicly inside, the computational path of a machine is determined uniquely based on $s \in {\Xi_{\Sigma,n}}^{\omega}$. This means that the machine is equivalent to a Type2-machine. Therefore, by Proposition 24,

Proposition 27 *Suppose that D_i $(i = 0, 1, \ldots, n)$ are $n_i\bot$-domains. A partial multi-valued function $f :\subseteq M(D_1)\times\ldots\times M(D_k) \rightrightarrows M(D_0)$ is $(D_1, \ldots, D_n, D_0)$-IM2-computable iff f is $(\phi_{L(D_1)}, \ldots, \phi_{L(D_k)}, \phi_{L(D_0)})$-Type2-computable.*

Theorem 8 can be derived from this proposition because the signed digit representation is equivalent to $\phi_{L(D)}$. This proposition cannot be extended to $f :\subseteq L(D) \rightrightarrows L(E)$. A counter example is the id_n function in Section 7, which is obviously $(\phi_{L(\mathcal{D}_{\Sigma,n})}, \phi_{L(\mathcal{D}_{\Sigma,n})})$-Type2-computable. As another example, consider the total functions $\hat{gtos} : L(\mathcal{D}_{\mathcal{R}}) \rightrightarrows \Gamma^{\omega}$ and $\hat{stog} : \Gamma^{\omega} \to L(\mathcal{D}_{\mathcal{R}})$ for $\Gamma =$

$\{0, 1, \bar{1}\}$ and $L(\mathcal{D}_{\mathcal{R}}) = \Sigma^\omega \cup \Sigma^* \bot 10^\omega$. $\hat{stog}$ is the extension of $stog$ to Γ^ω defined by the same algorithm in Section 5, and $\hat{gtos}$ is an extension of $gtos$, which is the inverse of $\hat{stog}$. For example, we have $\hat{gtos}(010^\omega) = \{\bar{1}1^\omega, 0\bar{1}1^\omega, 00\bar{1}1^\omega, \ldots\}$. We can show that $\hat{gtos}$ is not $(\mathcal{D}_{\mathcal{R}}, \Gamma^\infty)$-IM2-computable by Proposition 13 because $\hat{gtos}(010^\omega)$ does not include $\hat{gtos}(\bot 10^\omega) = \{0^\omega\}$. On the other hand, $\hat{gtos}$ is Type2-computable with respect to $(\phi_{L(\mathcal{D}_{\mathcal{R}})}, id_{\Gamma^\infty})$.

Let D_i be an $n_i\bot$-domain ($i = 0, \ldots, k$) and consider that $f :\subseteq L(D_1) \times \ldots \times L(D_k) \rightrightarrows L(D_0)$ is realized by a Type2-machine M of type $(\phi_{L(D_1)}, \ldots, \phi_{L(D_k)}, \phi_{L(D_0)})$. Then, by considering an input character $a^{(i)} \in \Xi_{\Sigma,n}$ as indicating an input of a from the i-th head, we can consider the rules of M as rules of an IM2-machine of type $(\Sigma^\omega_{\bot,n_1}, \ldots, \Sigma^\omega_{\bot,n_k}, \Sigma^\omega_{\bot,n_0})$. Then, applying Translation 2, we have a GHC program `m`. Since a Type2-machine is deterministic, `m` has a unique computational path when the behaviors of the oracle predicates are fixed. We can express the output of an oracle predicate $oracle_p$ as a sequence in $\phi_{L(D)}(p)$. When the oracles output $(s_1, \ldots, s_k) \in \Xi^*_{\Sigma,n_1} \times \ldots \times \Xi^*_{\Sigma,n_k}$, then the behavior of `m` is just the same as the behavior of M to $(s_1, \ldots, s_k)$. Therefore, the GHC program `m` $(D_1, \ldots, D_k, D_0)$-dd-realizes f. This is a strong result in that it considers $L(D)$ instead of $M(D)$. For example, it says that the $\hat{gtos}$ function is $(\mathcal{D}_{\mathcal{R}}, \Gamma^\infty)$-dd-computable, whereas it is not $(\mathcal{D}_{\mathcal{R}}, \Gamma^\infty)$-IM2-computable. Actually, the `gtos_d` process in Program 3 $(\mathcal{D}_{\mathcal{R}}, \Gamma^\infty)$-dd-realizes $\hat{gtos}$.

In this way, when the consumer process is always waiting for the next input to come and the inputs are processed one by one by the consumer process in the order they arrive, we can consider that the computation proceeds in a deterministic way. On the other hand, it is known in the theory of real-number computation that nondeterminism and multi-valuedness are essential properties of real-number computation. Therefore, when we consider that a process inputs $\Sigma^\omega_{\bot,1}$, which includes the set of real numbers as a subspace, then we have nondeterminism and multi-valuedness. The above investigation shows that this nondeterminism is caused by the multiple possibilities of the order in which an input tapes is filled by an oracle predicate (i.e., the order in which information about a real number is given by the environment).

10 Implementations in other programming languages, and conclusions

As we have seen, our implementation of IM2-machines uses the notion of guarded clauses and committed choice. Therefore, other parallel logic programming languages with guarded clauses and committed-choice nondeterminism, like PARLOG[3] and Concurrent Prolog[8], can also be used instead of GHC. We can also investigate the behavior of IM2-machines in the framework of concurrent constraint programming [2,6], which also has the notion of logical variables and guarded-choice nondeterminism.

SICStus Prolog, which is a dialect of Prolog, contains the primitive "freeze" which blocks a goal until a variable is instantiated. Though "freeze" does not have enough expressive power, when we combine "when" and "nonvar" primitives, one can block a goal until at least one of a set of variables is instantiated. With these primitives, we can express IM2-computable functions.

One can consider an extension of a lazy functional language for implementating an IM2-machine[12]. Since we want to express (multi-valued) 'functions' over the reals like addition and multiplication, it is desirable if we can express them as functions in some functional programming languages. Then, we can use higher-order mechanism like "map" and "foldr" to real number functions, which is impossible with logic-based programming languages.

The notions of guarded clause and committed choice was introduced into logic programming languages in order to express processes which run in parallel. It is interesting that they are just the facilities we need for real number computation, and these mechanisms are applied to this completely different research area.

Acknowledgements

Many people suggested me the possibility of using logic programming languages for real number computation based on Gray-code. Among them, the author would like to thank to Hiroyasu Kamo and Ian Gent. He also thanks Alex Shimpson for reading a draft paper and giving me a lot of comments.

References

[1] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[2] Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37–78, 1995.

[3] Steve Gregory. Parallel Logic Programming in PARLOG. Addison-Wesley, 1987.

[4] Simon Peyton Jones, Editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[5] Gordon D. Plotkin. Post-graduate lecture notes in advanced domain theory. 1981.

[6] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. 18th Ann. ACM Symp. on Principles of Programming Languages,* New York, 1991.

[7] Ehud Shapiro editor. Concurrent Prolog: Collected Papers, Volume 1 and 2. The MIT Press, 1986.

[8] Ehud Shapiro. Concurrent Prolog: A progress Report. *IEEE Computer*, 19(8):44–58, 1986. Also in [7], Chapter 5.

[9] Hideki Tsuiki. Computational dimension of topological spaces. In Jens Blanck, Vasco Brattka, and Peter Hertling, editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 323–335, Berlin, 2001. Springer.

[10] Hideki Tsuiki. Real number computation through gray code embedding. *Theoretical Computer Science*, 284(2):467–485, 2002.

[11] Hideki Tsuiki. Compact metric spaces as minimal-limit sets in domains of bottomed sequences. *Mathematical Structure in Computer Science*, to appear, 2003.

[12] Hideki Tsuiki and Keiji Sugihara. Extending Haskell with Multi-head Stream Accesses. Submitted, 2004.

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

[14] Kazunori Ueda. Guarded Horn clauses. E. Wada, editor, *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pp 168–179. Springer, 1986. Revised version in [7] Chapter 4.

[15] Kazunori Ueda, Designing a Concurrent Programming Language. In *Proc. an International Conference organized by the IPSJ to Commemorate the 30th Anniversary (InfoJapan'90),* Information Processing Society of Japan, October 1990, pages 87–94.

[16] klic Home Page http://www.klic.org/ .

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