実数計算のGHCによる実現

Implementing Real Number Computation in GHC.

立木秀樹
Hideki TSUIKI
京都大学総合人間学部
Faculty of Integrated Human Studies, Kyoto University

実数のグレイコード表現は，位相構造を保ったまま実数の集合を不定元 $\perp$ を一つだけ許す無限列 ($1\perp$-列) の集合 ($\Sigma^{\omega}_{\perp,1}$) に埋め込む表現である．無限列へのマルチヘッドでのアクセスと非決定性 (実数計算では不決定性と呼ばれている) を持つタイプ 2 計算 (IM2 マシン) を考えれば，$1\perp$-列を入出力することができ，実数上に標準的な計算概念を導けることが知られている [11]．この論文では，IM2 マシンによる計算が，プログラミング言語によりいかに実装できるか考察する．IM2 マシンによる計算は，n 個まで $\perp$ を許す無限列 ($n\perp$-列) を無限リストとして表現した場合，Haskell の様な遅延評価の関数型言語では実行できない．しかし，GHC(Guarded Horn Clauses) を用いれば，極めて自然に表現し実行することができる．さらに，$n\perp$-列の集合 $\Sigma^{\omega}_{\perp,n}$ 上の関数としては，GHC によって計算可能という概念と IM2 マシンによる計算可能の概念が一致すること，$\Sigma_{\perp}{}^{\omega}$ 上の関数としては，前者の方がより広い概念となることを示す．

1 はじめに

実数を近似値としてではなく無限列への展開として表現し，無限列を入出力する，すなわち，ストリーム入出力を行うプログラム (タイプ 2 マシンと呼ばれている) として実数上の計算を考える枠組みは，計算可能性解析学の一つの標準的なアプローチとなっている [17]．

その場合，実数を無限列に展開する方法は複数存在し，そのどれを採用するかによって，導かれる計算可能の概念は異なってくる．その中でも，符合付き 2 進表現，および，それと同値な表現が導く実数の計算可能性は，他の実数の計算可能性へのアプローチ，例えば [4] や [6] の点列に対する計算可能性と実効連続性を用いる方法や，[3] などの領域的な方法を用いたものと同じであり，実数上のもっとも標準的な計算可能性概念の一つだと考えられている．符合付き 2 進表現，および，それと同値な表現は，アドミシブルと言われる性質を持っている．そして，アドミシブルな表現の性質として，単射ではないことが知られている．すなわち，一つの実数に対して，複数の表現が存在するという冗長性を必ずもっている．

それに対して著者は，通常の無限列ではなく，不定元 $\perp$ を 1 つまで許す様な無限列 (以下，$1\perp$-列と呼ぶことにする) の集合 ($\Sigma^{\omega}_{\perp,1}$ と書くことにする) を考えると，冗長性がなく，その中に実数の集合を位相的に埋め込む様な表現 (グレイコード表現) が考えられること，それぞれの入出力テープに 2 つのヘッドを持ち，どちらから入力があるかによって複数の可能な動作があるが，それらは全て正しい結果を導くという非決定性 (実数計算では不決定性と呼ばれている) を持つマシン (IM2 マシン) を考えれば $1\perp$-列の入出力が可能になること，そして，IM2 マシンとグレイコード表現により実数上に導出される計算可能概念が，前述の実数上の標準的な計算可能性概念と一致することを示した [11, 18]．グレイコード表現は実数の集合から $\Sigma^{\omega}_{\perp,1}$ への埋め込み，すなわち中への同相写像である．よって，IM2 マシンは，この埋め込みを通じて実数を直接扱う計算の仕組みと言える．

本論文では，$1\perp$-列を一般化して高々n個の $\perp$ を許す無限列 $n\perp$-列を考え，IM2 マシンの $n\perp$-列に対する入出力が，いかにプログラミング言語で実現できるか調べる．IM2 マシンの動作は，$n\perp$-列を遅延無限リストとして，関数型言語 Haskell [5] の文法で自然に記述することができる．実際，[11] では，IM2 マシンの動作の記述のために Haskell の文法を用いていた．しかし，その記述は，通常の関数型言語とは異なる意味論を想定しており，Haskell の処理系では意図した通りに実行されない．関数型言語は無限列に対して逐次的なアクセスしか許しておらず，$\perp$ を入力しようとしたところで計算が終了しないことになる．それに対し，本論文では，並列論理型言語 GHC (Guarded Horn Clauses) を用いれば，$n\perp$-列を無限リストとして表現したまま，IM2 マシンによる計算を自然に表現し実行することができることを示す．IM2 マシンの複数のヘッドの位置の入力を待つという機能は，GHC の Guard を並列に実行する機能と対応している．また，IM2 マシンの不決定的な実行は，GHC の committed choice により生じる非決定性に対応している．

さらに，入出力を，$\Sigma_{\perp,n}^{\omega}$ に制限して考えれば，$n+1$ ヘッド IM2 マシンで計算可能という概念と，GHC で計算可能という概念が一致すること，また，${\Sigma_{\perp}}^{\omega}$ 上の関数として考えれば，前者よりも後者の方が広い概念となることを示す．

最後に，Prolog での IM2 マシンの実装についても述べる．Prolog でも，式の値が具体化するまでブロックするプリミティブである wait を使えば，IM2 マシン定義可能な関数が表現可能であることを示す．この $\Sigma_{\perp,n}^{\omega}$ 上の計算可能性の研究は，実数計算の基礎としてだけではなく，関数型言語と論理型言語の無限データに対する記述力の違いを説明する例としても興味深い．

本論では，簡単のため，実数全体ではなく，開区間 $\mathcal{I}=(0,1)$ のみを考えることにする．

2, 3, 4, 5 章で，[11] にしたがって，符号付き 2 進展開表現による表現，グレイコード表現，IM2 マシン，Haskell の文法による IM2 マシンの表現について述べた後，6 章で IM2 マシンの動作は GHC で実現できることを述べる．7 章では，要求駆動的に IM2 マシンの動作を書けることを述べ，8 章で，IM2 マシン定義可能性と GHC によるプログラム可能性が，$n\perp$-列に対しては一致することを述べる．9 章で，他のプログラミング言語での実装について述べ，10 章でまとめる．

2　ストリームによる実数計算

実数を無限列として表現するもっとも代表的な方法は，10 進展開だろう．しかし，この表現を用いたのでは，引数を 3 倍するといった単純な関数も計算可能とはならない．入力ストリームから得られた文字列を実数の 10 進展開による表現と考え，それを 3 倍した数の表現を出力ストリームに出すプログラムが存在したとする．$\mathcal{I}$ に限っているので，先頭の"0." は省略することにする．すると，そのプログラムは，例えば，$03333\ldots$ という無限列を入力として得た時に，$10000\ldots$ または $09999\ldots$ という出力を行わないといけない．しかし，マシンが有限時間では有限の接頭辞しか読み込めないことと，その先に 4 が来る可能性も 2 が来る可能性もあることより，有限時間内では 1 文字も出力することができない．2 進展開でも同様である．それを解消するためには，表現をより冗長性の高い、すなわち、同じ数を表現する無限列の数が多いものにする必要がある．

定義 1　$\mathcal{I}$ の符号付き 2 進表現 δ_{sn} は，$\Gamma=\{0,1,\overline{1}\}$ という記号を用いた表現であり，$\overline{1}$ が -1 を表記しているとした時，$a_1a_2\ldots$ という文字列が，$\Sigma_{i=1}^{\infty}\{a_i\cdot 2^{-i}\}$ を意味するものである．

この計算式は 2 進表現の時と同じであり，$\overline{1}$ を使わなければ，この両者の表現は一致する．0.5 を表現するのに $10000\ldots$，$11\overline{111}\ldots$，$1\overline{1}111\ldots$ などの方法がある．さらに，$1\overline{1}$，$\overline{1}1$ という部分文字列を 01，$0\overline{1}$ にそれぞれ置き換えても同じ数となる．このように，δ_{sn} は高い冗長性を持っている．本論文では，符号付き 2 進表現は $a_1=1$ と固定し，途中から $111\ldots$ あるいは $\overline{111}\ldots$ と無限に続く表現は除くことにする．こう制限しても，$\mathcal{I}$ の数を表現するのに十分であるし，計算の表現可能性も変わらない．

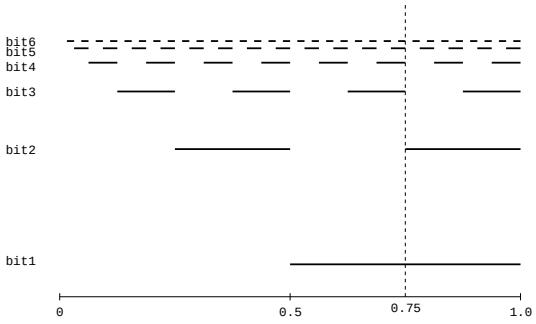

図 1: 実数の 2 進展開

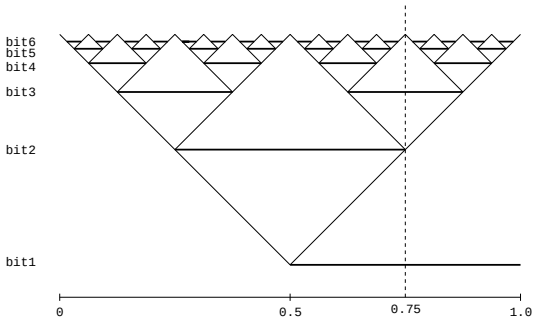

図 2: 実数のグレイコード 展開

この表現を用いると，上記の 3 倍する計算は表現可能となる．また，符号付き 2 進表現により導かれる実関数の計算可能性の概念は他の多くのアプローチによるものと一致し，最も標準的なものと考えられている．δ_{sn} の持つ冗長性は，無限列への展開により実数の計算可能性を考える際，表現が必ず持つ性質と考えられている．実際，δ_{sn} と同値な表現は全て冗長性を持つことが知られている [17]．

3　グレイコード 実数計算

実数のグレイコード 展開は，通常の 2 進展開とは異なる $0, 1$ の無限列としての実数の展開である．図 1 と 図 2 に，$\mathcal{I}$ の 2 進展開とグレイコード 展開を図示した．(線のある部分は 1 を表し，線のない部分は 0 を表している．斜めは補助線である．) 通常の 2 進展開では，先頭文字 (head) が $(0, 1/2]$ と $[1/2, 1)$ のどちらの区間に属しているかを意味しており，先頭を除いた残り (tail) が，その数に

$$f(x) = \begin{cases} 2 * x & (0 < x \leq 1/2) \\ 2 * x - 1 & (1/2 \leq x < 1) \end{cases}$$

という関数を適用した数の展開を意味している．それに対して，グレイコード 展開では，head の意味は同じだが， tail の意味する関数が，

$$f(x) = \begin{cases} 2 * x & (0 < x \leq 1/2) \\ 2 * (1 - x) & (1/2 \leq x < 1) \end{cases}$$

といういわゆるテント関数となっている．

2 進展開は，2 進有理数 (自然数 m, k を用いて $m/2^k$ と表現できる数) において 2 つ存在したが，グレイコード展開においても 2 つ存在する．例えば，$3/4$ の 2 進展開は，$0.110000...$ と $0.101111...$ である．それに対応して，グレイコード展開も $0.111000....$ と $0.101000...$ の 2 つが存在する．これらは 1 ビットしか違わないことに注意されたい．一般に，2 進有理数の 2 つのグレイコード展開は 1 ビットしか違わず，その異なるビットの後は，必ず $10000...$ となる．

このように，実数の展開は数を表現するための文字列なのに，$3/4$ に対しては，2 ビット目の値はその数が $3/4$ だということを表現するのに貢献していない．よって，そのビットの値は不定であると考え，$3/4$ の表現は $0.1\perp 10000...$ とするのが自然だと考えられる．$\perp$ は，不定，あるいはそのビットの値を求める計算が止らないことを表している．このように，グレイコード展開の 2 進有理数における値を修正することにより，冗長性のない実数のグレイコード表現が定義される．以下は，より厳密な形でこの定義を述べたものである．

定義 **2** 実数 x $(0<x<1)$ のグレイコード表現 $G(x)$ は記号集合 $\{\perp, 0, 1\}$ の列 $a_1 a_2 \ldots$ で，以下のように a_i が構成されたものである．$m \times 2^{1-i} - 2^{-i} < x < m \times 2^{1-i} + 2^{-i}$ が奇数 m に対して成り立つときには $a_i = 1$，偶数 m に対して成り立つときには $a_i = 0$，どちらも成り立たないときには $a_i = \perp$．

以後，Σ は，特に断らない限り，そこの文脈で一番適した文字集合とする．グレイコードを考えている時は，$\Sigma = \{1, 0\}$ であるし，符合付き 2 進表現を考えている時は $\Sigma = \{\overline{1}, 1, 0\}$ である．Σ の有限列全体の集合を Σ^*，無限列全体の集合を Σ^ω と書くことにする．$\Sigma \cup \{\perp\}$ の無限列の集合を ${\Sigma_\perp}^\omega$ と書くことにする．Σ の無限列に n 個まで $\perp$ が含まれることを許した列を $n\perp$-列と呼ぶことにし，$n\perp$-列全体の集合を $\Sigma^\omega_{\perp,n}$ と書くことにする．$\Sigma^\omega_{\perp,0} = \Sigma^\omega$ である．A から B への部分関数全体の集合を $A \rightharpoonup B$, A から B への部分多値関数全体の集合，すなわち，$A \times B$ の部分集合を A から B のべき集合から空集合を除いたものへの部分関数として見たもの全体の集合を $A \overrightarrow{\rightharpoonup} B$ と書くことにする．無限列 q の k 番目の文字を $q[k]$ と書くことにする．

$n_1 < n_2, m_1 < m_2$ とする．部分関数空間 $\Sigma^\omega_{\perp,n_1} \rightharpoonup \Sigma^\omega_{\perp,m_1}$ は，拡張された引数に対しては未定義とすることにより，$\Sigma^\omega_{\perp,n_2} \rightharpoonup \Sigma^\omega_{\perp,m_2}$ に自然に埋め込まれる．また，${\Sigma_\perp}^\omega \rightharpoonup {\Sigma_\perp}^\omega$ にも埋め込まれる．$\Sigma^\omega_{\perp,n_1} \overrightarrow{\rightharpoonup} \Sigma^\omega_{\perp,m_1}$ についても同様である．

グレイコード表現では，$\perp$ は 1 回しか現れず，$\perp$ の後の文字列は常に $1000\ldots$ であることに注意されたい．よって，$\Sigma = \{0, 1\}$ に対し，G は $\mathcal{I}$ から $\Sigma^\omega_{\perp,1}$ への単射である．$\Sigma^\omega_{\perp,1}$ には，${\Sigma_\perp}^\omega$ の部分空間として，自然に位相が入る．その位相に関して，G は $\mathcal{I}$ から $\Sigma^\omega_{\perp,1}$ への位相的な埋め込みになっている．G を実数全体 $\mathcal{R}$ から $\Sigma^\omega_{\perp,1}$ への埋め込みに拡張することも容易である．

4 IM2-マシン

実数計算では，実数値そのものを有限時間内に求めるのではなく，無限に動き続け，計算とともに近似の精度が高くなっていく様な計算を考える．計算によって，$a_1 < x < b_1$, $a_2 < x < b_2$, $\ldots$ と実数の存在する開区間が狭まっていく計算過程を元に，その計算結果をグレイコードで出力することを考える．$3/4$ を出力する計算を考えると，有限時間ではその値が $3/4$ と確定できないため，$\perp$ を 2 ビット目に出力することはできず，ここで出力がとまってしまう．しかし，もし，計算によって，$5/8 < x < 7/8$ だということが分かったなら，その時点で 3 ビット目は 1 と確定する．よって，もし，マシンは前から順に出力するという制限を緩めたなら，2 ビット目をとばして 3 ビット目に 1 を出力することができる．その後，$5/8 < x < 3/4$ と分かれば飛ばしていた 2 ビット目に 1 を，$3/4 < x < 7/8$ と分かれば 2 ビット目に 0 を，そして，$11/16 < x < 13/16$ と分かれば 4 ビット目に 0 を出力するという具合に計算を進めることができる．出力値が $3/4$ の時には，最後まで 2 ビット目を埋めず，3 ビット目から先に $1000\ldots$ と出力することになる．よって，このマシン

では，$\perp$ はその桁の値が出力されてないことを意味し，テープは最初，$\perp$ で埋められていると考える．各ビットの出力は，近似値の情報が得られたことを示しており，その情報は書き換えられない．すなわち，出力位置は行ったり来たりするが，上書きは許されず，高々 1 文字を除いてすべて出力される出力過程となる．このような出力過程は，出力とともにヘッドが前に移動していく通常のストリームを拡張し，テープに 2 つ 1 方向性のヘッドが存在しており，それらがテープのまだ書き込まれていない最初の 2 つのセルに存在する様に，出力とともに移動する形で表現することができる．すなわち，通常のヘッド H_1 と先出しヘッド H_2 を考え，H_1 と H_2 は最初はテープの先頭の 2 つのセルに置かれており，H_2 から出力された時には H_2 だけが 1 つ前に移動し，H_1 から出力された時には H_1 が H_2 の位置に移動し，H_2 が 1 つ前に移動するとする．

入力も同様である．入力テープは，上の出力の過程と同じように埋められると考える．すなわち，最初は $\perp$ で埋まっており，1 箇所だけ飛ばすことを許しながら，文字が書き込まれてゆくと考える．$\perp$ のセルは後に書き込まれるかもしれないので，ヘッド位置の文字が $\perp$ であることに応じた動作はできない．片方のヘッドの位置の文字が $\perp$ である時も他のセルの値を読み込み計算を続けるため，入力テープにも 2 つのヘッドを用意する．そうすると，2 つのヘッドの両方とも書き込みがなされている時，マシンに可能な動作が 2 つ考えられることになる．しかも，それらは全て正しい結果を出力しないといけない．このように，複数の計算のパスが存在するが，それらのどれを用いても，求めたい性質を満たす結果が得られるという非決定性をもつマシンを考える．この性質は，どれか一つのパスが正解にたどり着くという (非決定性オートマトンにおける) 非決定性 (nondeterminism) と区別するために，実数計算では不決定性 (indeterminism) と呼ばれている [1]．

このようなマシンの定義を，より一般に，ヘッドの数が n 個 $(n \geq 1)$ であり，n の値はそれぞれの入出力テープによって異なっていても構わない様に拡張した形で IM2 マシンを定義する．そして，i 番目の入力テープの型が $Y_i (i = 1, 2, \ldots, k)$ であり，出力テープの型が Y_0 である IM2 マシンを，型 $(Y_1, \ldots, Y_k, Y_0)$ の IM2 マシンと呼ぶことにする．ここで，$Y_i (i = 0, 1, \ldots, k)$ は $\Sigma^{\omega}_{\perp,n} (n = 0, 1, \ldots)$ のどれかである．IM2 マシンは，状態や内部テープなどの扱いはチューリングマシンと同じであり，上記の様に入出力の方法だけが複数ヘッドと不決定性を持つように拡張されたものである．細かい定義は付録を参照されたい．

定義 **3** 型 $(Y_1, \ldots, Y_k, Y_0)$ の *IM2* マシンが $(p_1, \ldots, p_k)$ $(p_i \in Y_i)$ を入力として $q \in Y_0$ を出力するとは，$p_1, \ldots, p_k$ を入力テープに置き，内部テープを空白文字 *'B'* で，出力テープを不定文字 *'⊥'* で埋め，型 $\Sigma^{\omega}_{\perp,n}$ の入出力テープ T_l に対しては，ヘッド $H_1(T_l), \ldots, H_{n+1}(T_l)$ をテープの最初の $n+1$ 個のセルに置き，内部テープ T に対してはヘッド $H(T)$ をテープの最初のセルに置いた状態から，M の適用可能な規則を繰り返し適用することにより，出力テープが q の状態になる無限の計算のパスが存在することである．より正確には，出力テープを O とした時，$c \in \Sigma$ に対し，$q[k] = c$ ならばいつかは $O[k] = c$ となり，$q[k] = \perp$ ならば，いつまでも $O[k] = \perp$ である計算のパスが存在することである．

空白文字 'B' は，通常のチューリングマシンにおけるものと同じく，空白を意味する文字であり，マシンによって認識されるし，マシンが内部テープに書くこともできる．一方，不定文字 '⊥' は，まだ意味のある文字が書かれていないことを意味し，マシンが認識することもできないし，テープに書くこともできない．

定義 **4** *IM2* マシンを，通常のチューリングマシンと同様の有限文字列の入出力も許すように，以下のように拡張する．マシンの型 $(Y_1, \ldots, Y_k)$ において，Y_i として Σ^* も許すことにする．そして，$Y_i = \Sigma^*$ の時には，対応するテープは，$\Sigma \cup \{B\}$ をテープ文字とし，その入力テープの余白や出力テープは最初 B で埋められているものとする．さらに，$Y_0 = \Sigma^*$ の時には，終了状態が存在し，その状態になれば，マシンは停止するものとする．

Y_i が全て Σ^* のマシン，および，Y_i が全て Σ^* か $\Sigma^{\omega}_{\perp,0} (= \Sigma^{\omega})$ のマシンは，それぞれチューリングマシン，および，タイプ 2 マシンと同等である．上に述べたように，IM2 マシンは，不決定的な動作を行い，

同一の入力に対し，複数の出力の可能性を持つ．すなわち，IM2 マシンが定義するのは，部分多値関数である．

定義 **5** 型 $(Y_1,\ldots,Y_k,Y_0)$ の *IM2* マシン M が $Y_1\times\ldots\times Y_k$ から Y_0 への部分多値関数 f を実現するとは，$(p_1,\ldots,p_k)\in dom(f)$ を入力とした時の全ての計算のパスが無限に文字の出力を行い，M の出力の集合が $f(p_1,\ldots,x_k)$ の部分集合となることである．また，そのような M が存在する時，f を $(Y_1,\ldots,Y_k,Y_0)$ マシン定義可能と呼ぶ．

IM2 マシン M が同じ入力に対して持つ多くの動作のパスの中で，いくつかは無限列を出力して，いくつかは無限に時間をかけても有限列しか出力しない場合がある．その場合，M はその入力値を定義域に含むどの部分多値関数も定義していないこととする．この考え方は，スミス・パワー領域と同じである．

命題 **1** $n_1\leq n_2, m_1\leq m_2$ であり，等号は高々片方しか成り立たないとする．$\Sigma_\perp\stackrel{\rightarrow}{\rightarrow}\Sigma_\perp$ への埋め込みに関して，$(\Sigma^\omega_{\perp,n_1},\Sigma^\omega_{\perp,m_1})$ マシン定義可能関数の集合より，$(\Sigma^\omega_{\perp,n_2},\Sigma^\omega_{\perp,m_2})$ マシン定義可能関数の集合の方が真に大きい．

定義 **6** 型 $(\overbrace{\Sigma^\omega_{\perp,1},\ldots,\Sigma^\omega_{\perp,1}}^{k+1})$ の *IM2* マシン M が $\mathcal{I}^k$ から $\mathcal{I}$ への部分多値関数 F を実現するとは，G による F の $\Sigma^\omega_{\perp,1}\times\ldots\Sigma^\omega_{\perp,1}\stackrel{\rightarrow}{\rightarrow}\Sigma^\omega_{\perp,1}$ への埋め込みが計算可能なことである．この時，F をグレイコード計算可能という．

定義 **7** 上の定義において，F が部分関数である特別な場合として，*IM2* マシン M が $\mathcal{I}^k$ から $\mathcal{I}$ への部分関数 f を実現する，および，f がグレイコード計算可能であるという概念を定義する．

定理 **2** ([**11**]) グレイコード計算可能と，タイプ *2* マシンと符号付き *2* 進表現による計算可能 *[17]* とは，同値な概念である．

関数ではなく多値関数を基本としたことや不決定性を考えたことは，一見不自然に思われるかも知れないが，これらは計算可能な実数関数の本質であると考えられている ([2] など)．実数の上の関数は，関数の零点を求める操作など，通常の関数ではなく多値関数を考えることが多い．それは，実用的な計算を考える時により顕著になる．例えば，実数上の計算可能関数は全て連続であり，非連続な関数は (定理 2 の意味で) 計算可能ではないことはよく知られている．一方で，40 度でブザがなるセンサーは，実用上有用であり，そういう非連続なものも扱えることが好ましいと考えられる．しかし，物理的なセンサは，丁度 40 度という温度で状態が変化するというのは不自然であり，小さな実数 ϵ に対し，$40-\epsilon$ から $40+\epsilon$ までの入力に対しては，ブザが鳴っても鳴らなくてもいいという仕様が与えられているのが普通である．さらに，実際に実現されたセンサは，$40-\epsilon$ と $40+\epsilon$ の間のある決まった温度で必ずいつもブザが鳴るとは限らず，毎回動作が変化しても構わない．つまり，この仕様は，$x<40-\epsilon$ では 0 を返し，$x>40+\epsilon$ では 1 を返し，$40-\epsilon\leq x\leq 40+\epsilon$ では 0 か 1 のどちらかを返すという多値関数であり，実際に実現されるのも，その仕様を満たす不決定的計算である．

本章では，Gray Code 表現という $\mathcal{I}$ (または $\mathcal{R}$) から $\Sigma^\omega_{\perp,1}$ への埋め込みを通して，IM2 マシンによって実数の上の計算の概念を導出できることを説明した．[12] では，任意の n 次元可分距離空間が $\Sigma^\omega_{\perp,n}$ に埋め込めることが証明されている．よって，より一般に，IM2 マシンを用いると任意の可分距離空間の間の計算の概念を導くことができる．

5 IM2 マシンの動作の Haskell の文法による表現

IM2 マシンの動作は，Haskell の文法を用いて，無限リストを入出力する再帰プログラムとして表現できる．状態や内部テープの扱いは，チューリングマシンの動作の関数型言語による表現と同様である．つ

まり，状態，内部テープの現在のヘッド位置より左の部分，および，現在のヘッド位置より右の使われている部分を関数に対する引数として表現する．複数ヘッドによる入出力は以下のように記述する．ここでは 2 ヘッドで説明する．

まず，出力を考える．出力の無限リストは，まだ出力が行われていないセルだけからなっており，先頭の 2 つのセルにヘッドが存在していると考える．すると，1 つ目のヘッドからの出力は，`c:foo()` と書くことができる．ここで，`c` は 0 か 1, `foo()` の部分は，残りの出力を得るための計算である．2 つ目のヘッドからの出力は，`x:c:xs where x:xs=foo()` と書くことができる．ここでも，`c` は 0 か 1, `foo()` は残りの出力を得るための計算である．常に残りの計算 `foo()` の出力の最初の 2 文字が，新しいヘッド位置に対応していることに注意されたい．

出力の例として，符号付き 2 進表現からグレイコード表現への変換 (`stog`) を実現する $(\Sigma^\omega, \Sigma^\omega_{\perp,1})$ 型の IM2 マシンの動作を Haskell の文法で記述しよう．ここでは，$\mathcal{I}$ の符号付き 2 進表現は，最初の “1” を取り除いた残りを指すことにする．この変換は，4 つの状態 $(0,0), (0,1), (1,0), (1,1)$ ($(0,0)$ が初期状態) と 0 本の内部テープを持つ IM2 マシンで実現可能である．その計算規則は 12 個あり，次のような形をしている．詳細は [11] を参照されたい．

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

これは，状態が (0,0) の時に 入力テープ I のヘッドから 0 を読めば，状態を (0,1) に変更し，出力テープ O のヘッド 2 から 1 を出力することを意味している．この規則の詳細は付録を参照されたい．これに上記の翻訳を機械的に行った Haskell のプログラムは以下のようになる．

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

このプログラムは，グレイコードの持つ再帰的な構造を考えて書いた以下のプログラムと等価である．

```
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
```

ここで，`nh` は，無限リストの先頭の要素を反転させる関数である．つまり，

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

と定義される．この `stog` は，Haskell のプログラムとして正常に動作する．`stog([0,0..])` は，1 文字目 ($\perp$) を出力しようとして，何も出力せず止らないが，`tail(stog([0,0..]))` は `[1,0,0,0...` と結果を表示し続ける．

次に，入力を考える．入力の無限リストも同様に，まだ読み込まれていないセルだけからなっており，先頭の 2 つのセルにヘッドが存在していると考える．その様な関数は，次の様にパターンマッチングを用いて記述するのが自然だろう．

```
bar(0:xs,..)  = foo1(xs,..)
bar(1:xs,..)  = foo2(xs,..)
bar(c:1:xs,..)  = foo3(c,xs,..)
bar(c:0:xs,..)  = foo4(c,xs,..)
```

```
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)
```

図 3: 足し算のプログラム

不決定的な動作に対応し，1,2 行目と 3,4 行目の定義の間に，オーバーラップが存在することに注意して欲しい．

入力の例として，グレイコード表現から符号付き 2 進表現への変換 (gtos) を考える．ここでも，IM2 マシンの定義を翻訳したものではなく，より効率的に書いたプログラムを考える．

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

ここでは入力のストリームが任意の $1\perp$-列ではなくグレイコード表現の場合だけを考えていることと，3 行目の再帰呼び出しで 2 文字目を引っくり返していることから，`c:0:xs` の場合の処理は必要ない．

これも，Haskell の文法にのっとったプログラムである．ただし，この gtos の Haskell による実行意味は，まず第 1 要素を評価し，0 か 1 が得られたら最初の 2 つのルールを適用し，それ以外が得られたなら第 2 要素を評価し，それが 1 なら 3 つ目の規則を適用するということになる．よって，第 1 要素が $\perp$ で第 2 要素が 1 の時には第 1 要素の評価が止らないため，3 番目の規則は使われず，`gtos (stog [0,0..])` を実行しても，`[0,0,0,...` という結果を得ることはできず，何も出力されない．つまり，意図したものとは異なる関数となっている．

より本格的なプログラムとして，グレイコード表現により，実数の足し算を実現したプログラムを図 3 に示す．ただし，結果を $\mathcal{I}$ に適合させるために，正確に言うと，このプログラムが計算しているのは足し算ではなく，平均 $(\mathtt{pl}(\mathrm{x},\mathrm{y}) = (\mathrm{x}+\mathrm{y})/2)$ である．これも，Haskell の文法を用いて表現しているが，関数型言語の遅延評価の枠組みで実行したのでは最初の 4 つの規則しか用いられず，2 進有理数が入力された時にはとまらない．このような関数が，どういう考えに基づいて記述されるかは [11] を参照されたい．

$n\perp$-列を `[(1,1),(2,0),...]` という具合いに，(ヘッドの番号, 文字) というペアの無限列として表現し，タイプ 2 マシンで $n\perp$-列上のプログラムを記述することもできる．この実現は，同じ $n\perp$-列を表現する文字列が複数存在するという冗長性を持っており，符合付き 2 進表現と等価となる。よって，定理 2 でも述べたように，IM2 マシンにより計算可能な関数はすべてこの形でかける．このように，$n\perp$-列の表現方法を選べば，IM2 マシンにより計算可能な関数はすべて関数型言語で表現することができる．しかし，$n\perp$-列をそのまま $\perp$ を含む無限リストとして表現したのでは，$n\perp$-列の入出力を行う IM2 マシン定義可能な非多値関数で，逐次的な関数型言語で表現できないものが存在することがいえる [13]．ここで，逐次的

な関数型言語とは，parallel or を表現できない言語のことである．

6 GHC による実装

前節の最後に述べたように，逐次的な関数型言語では $n\perp$-列を無限リストとして扱って，IM2 マシンの動作を表現することができない．無限リストは全体の値を求めることができないので，必要な所だけしか値を求めないために遅延評価の様な要求駆動的なコントロールが必要である．しかし，関数型言語の様に「この値が欲しい」という形でしか要求が出せないのでは，$n\perp$-列の様な，中に $\perp$ を含んだ文字列を扱うのは困難である「この部分の計算をこれかこれが求まるまで行え」という形で要求を出し，値を利用する側のパターンマッチも，未定義な部分を中に含んだまま，並行して調べられる機構のある言語が必要と考えられる．

そのような考察のもと，Flat GHC [16][15] の概念に基づいたプログラミング言語 KL1 および，そのインプリメンテーションである klic [14] を用いて前章のプログラムの実装を試みた．実際に用いたのは KL1 であるが，GHC の考えに基づいた言語では同様なことができると考えられるので，ここで用いる言語は GHC と呼ぶことにする．GHC は，commited choice に基づいた，バックトラックのない，並行論理型言語である．論理型言語なので，式の一部が未定義な状態を，式の一部を変数のまま残しておくことにより表現することが可能であるし，パターンマッチの時に行われる commited choice により，複数の変数のうちのどれかが具体化されるのを待つこともできる．さらに，言語自体に非決定的な動作があるので，IM2 マシンの非決定的な計算が素直に表現できると考えられる．GHC では，プロセス間のストリーム通信を自然に表現することができる．本章と次章で，$n\perp$-列に拡張されたマルチヘッドのストリーム通信が GHC で自然に表現できることを示す．

ここではまず，パターンマッチについての説明に集中するため，klic の非並列マシン向けの実装 (klic-3.003) が採用している，ブロックしていないゴールを左から順に実行するというゴールの選択順に依存したコードを例に説明する．下記のプログラムは，[0,0,0,...] に stog を行い，さらに gtos を行い，結果を出力するプログラムである．

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

このプログラムを例に，GHC の動作について簡単に説明する．各行のことを節といい，:- の左辺を節のヘッド，右辺をボディという．一般に GHC の節はガード部を持っているが，このプログラムでは，ガー

ドで調べられるべき条件はすべてヘッドの引数のパターンの中に表現されているのでガードが存在しない．KL1 では，プログラムの実行は main という初期ゴールから始まる．そして，main をヘッドとして持つ節が選ばれてリダクションが行われ，そのボディである`pinf(ZZ),gtos(YY,ZZ),stog(XX,YY),inf0(XX)` の 4 つが次のゴールとなる．このうちの最初の 3 つは，それにマッチする引数パターンをヘッドに持つ節がプログラムの中に存在しないので，引数の変数が具体化されてその条件が満たされるまでブロックされる．しかし，`inf0(XX)` はそのような節が存在するので，リダクションが行われ，その節のボディに書かれた`XX = [0|X]` という束縛が行われ，`inf0(X)` がかわりにゴールに追加される．この束縛により，ブロックしていたゴール `stog(XX,YY)` が，3 番目の`stog` 節のヘッドとマッチするようになる．よって，リダクションが行われ，`YY=[C,1|Y]` という束縛が行われ，`stog(XX,YY)` がゴールから取り除かれ，かわりに `nh(Z,Y)`, `stog(X,[C|Z])` がゴールに加えられる．この束縛により，ブロックしていたゴール `gtos(YY,ZZ)` が，プログラム中 3 番目の`gtos` 節のヘッドとマッチするようになる．よって，リダクションが行われる．以下，同様である．

この様に，リストの 1 文字目を変数 C に残したまま，2 文字目だけを 1 に具体化するということが可能であり，一方，ガード部分の検査は並列に行われるため，入力ストリームの 1 文字目に 0 が現れたら，あるいは，2 文字目に 1 が現れたらといった条件を自然に表現することができる．関数型言語の様に要求駆動的に特定の式の弱頭正規形を求めることを目標に計算が行われるのではないことに注意されたい．むしろ，引数が具体化されてブロックしていたゴールが動作可能になることにより，つまり，引数の値がそろうことにより計算がすすんでいく．さらに，その際に複数の適用可能な節が存在すれば，どれが適用されるかは非決定的に決まる．これは，IM2 マシンの不決定的動作とよく対応している．

IM2 マシンの実行規則は， Haskell の文法による記述と同様の方法で，1 つの述語だけからなる GHC のプログラムに翻訳できる．自然数で番号づけされた状態集合 (0 が初期状態) と，内部テープを 1 本もつ $(\Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2})$ 型の IM2 マシンの次のような計算規則を例に説明しよう．

$$5, H_2(I1)(0), W(0) \Rightarrow 7, H_2(O)(1), W(1), -(W)$$

これは，状態が 5 で，入力テープ 1 番の 2 番目のヘッドの値が 0 で，内部テープ W のヘッド位置の値が 0 なら，状態を 7 に移行し，出力テープ O の 2 番目のヘッドから 1 を出力し，W のヘッド位置に 1 を出力し，W のヘッドを左に 1 つ移動するという意味である．

これに対応する節は，以下のようになる．

```
m(5,[X1,0|X], YY, [T|W1], [0|W2], OO) :-
  OO=[O1,1|O],m(7,[X1|X],YY,W1,[T,1|W2],O)
```

この変換において，内部テープのヘッドの右が空リスト (すなわち，すべて 'B') の場合を考えて，ヘッド位置のブランク文字 B を読み込む操作，および，ヘッド位置の文字によらずにヘッドを右に移動する操作を書く時には，対応する引数の位置に空リストが与えられた時にも動作する様に書く必要があることに注意されたい．そして，状態などに初期値を与えるプログラム mm を以下のようにおく．

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

GHC には無限な式を変数に束縛するという概念はないが，プログラムの入力変数に無限式も束縛可能にする拡張をした操作的意味でプログラムの実行を考えることにする．すると，この様に変換されたプログラムは，元の IM2 マシンと同じ動作をすることが示される．

定理 3 $(Y_1, \ldots, Y_k, Y_0)$ 型の *IM2* マシン M を翻訳することにより作成された *GHC* の述語

```
mm(X1, X2,...,Xk, OO)
```

```
inf0([X1|X]) :- X1 = 0, inf0(X).
gtos(YY, [X1|X]) :- YY = [Y1,Y2|Y], gtos_r(YY, [X1|X]).
gtos_r([0|Y], [X1|X]) :-  X1 = -1, gtos(Y, X).
gtos_r([1|Y], [X1|X]) :- X1 = 1, nh(Y, Z), gtos(Z, X).
gtos_r([C,1|Y], [X1|X]) :- X1 = 0, nh(Y, Z), gtos([C|Z], X).
stog(XX, [Y1,Y2|Y]) :- XX = [X1|X], stog_r(XX, [Y1,Y2|Y]).
stog_r([-1|X], [Y1|Y]):- Y1 = 0,  stog(X, Y).
stog_r([1|X], [Y1|Y]):- Y1 = 1, nh(Z,Y),  stog(X, Z).
stog_r([0|X], [C,Y2|Y]):- Y2 = 1, nh(Z, Y),  stog(X, [C|Z]).
```

図 4: KL1 による gtos, stog の実装

を考える．M が $(p_1, \ldots, p_k)$ に対し p を出力することと，ゴール $\mathtt{mm}(p_1, p_2, \ldots, p_k,\ \mathtt{OO})$ を実行すると，OO に p が出力される，すなわち，$p[k] = c$ ならばいつかは $\mathtt{OO}[k] = c$ となり，$p[k] = \perp$ ならば，いつまでも $\mathtt{OO}[k] = \perp$ となる計算のパスが存在することとは同値である．

この変換で作られたプログラムは，それぞれの節のボディに 1 つしかゴールを含まないため，ゴール集合には常に 1 つしかゴールを含まない．よって，並列な実行を必要としていないことに注意されたい．

7　要求駆動的な実現

$n\perp$-列を入出力する GHC プログラムをより簡潔に書くためには，上記の stog や gtos の様に，右辺に複数のゴールを持った節を書く必要が出てくる．また，上記の定理では，入力変数の値として無限列が与えられていると仮定しているが，実際には，それを求めるためのゴールも定義し動作させる必要がある．よって，複数のゴールが存在する時にも正常に動作する様なプログラムの書き方を考える．

前節の stog を行ってから gtos を行うプログラム例では，ブロックしていないゴールが複数存在するときに，それらが左から順に実行されるというゴールの選択に依存したプログラムとなっていた．しかし，GHC の仕様では，ブロックしていないゴールが複数存在したときには，そのどれが次に実行されるかは規定されていない．どのような順で実行されても正しい答を出す様にプログラムを書くことが求められている．前節のプログラムでは，inf0(X) が常にゴールの集合の中に含まれており，常にこれが選ばれて，stog が一度も呼び出されない可能性もある．よって，正しいプログラムではない．

通常，そのようなプログラムは，個々のプロセスを要求駆動的に書くこと，つまり，値を生産するゴールは値の要求を受け取るまではブロックするように実行の順序を制御することによって書かれる．値の要求は，値を受け渡すストリームの変数をコンス・セルに具体化させることによって行うことができる [8]．

ここで実装したい，マルチヘッドのストリーム通信の場合にも，この方法で実装することができる．図 4 は，その方法で記述された stog と gtos である．ただし，マルチヘッド通信の場合には，値の要求は，値を受け渡すストリームの変数の先頭だけではなく，そのストリーム上に許されているボトムの数 $+\ 1$ 個目までを具体化する必要がある．そうしないと，先のヘッドから入力を行った時の，次の入力要求を行う規則の右辺の部分 (gtos_r の 3 行目の右辺で第 1 引数をコンス・セルに具体化しているところ) で誤ってそのゴールを駆動することになる．

この例について，簡単に解説する．gtos の入力と stog の出力が 2 ヘッドストリーム通信，gtos の出力と stog の入力が通常のストリーム通信である．gtos は，結果を返す側のストリーム (すなわち第 2 引数) が，結果を用いる他のプログラムによって，[X1|X] というコンス・セルに具体化されることにより動作可能となる．そして，第 1 引数 YY を [Y1,Y2|Y] に具体化することにより，第 1 引数のストリームから

```
pl(AA,BB,[Z1,Z2|Z]):-AA=[A1,A2,A3|A],BB=[B1,B2,B3|B],pl_r(AA,BB,[Z1,Z2|Z]).
pl_r([0|A],[0|B],ZZ) :-  ZZ = [0|Z], pl(A, B, Z).
pl_r([1|A],[1|B],ZZ) :- ZZ = [1|Z], pl(A, B, Z).
pl_r([0|A],[1|B],ZZ):-ZZ=[C,1|Z],nh(B,BN), pl(A,BN,R), [C|CS]=R, nh(CS, Z).
pl_r([1|A],[0|B],ZZ):-ZZ=[C,1|Z],nh(A,AN), pl(B,AN,R), [C|CS]=R, nh(CS,Z).
pl_r([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).
pl_r([C,1,0|A],[0,0|B],ZZ) :-  ZZ = [0|Z], pl([C,1|A], [1|BN],Z), nh(B,BN).
...
```

図 5: KL1 による pl の実装 (一部)

値を要求する．要求に応じて具体化される値が Y1 = 0,1, または Y2 = 1 に従い，それに対応する gtos_r が実行され，X1 の部分が具体化される．

このような形のマルチヘッドストリーム通信プログラムの書き方を，合成可能な形と呼ぶことにする．合成可能な形で関数を記述すると，関数の合成を，対応する引数の変数名を一致させて，ゴールを並べるだけで記述できる．[11] では，k 入力の IM2 マシン M および，n 入力の IM2 マシン $N_1, \ldots, N_k$ がそれぞれ $\Sigma^{\omega}_{\perp,1}$ を入出力する関数 $f, g_1, \ldots, g_k$ を実現している時，$f \circ < g_1, \ldots, g_k >$ を実現する n 入力の IM2 マシンが存在することを証明している．しかし，その合成マシンの構成は，単なるルールの和ではなく，それぞれの部分が交互に実行されることを保証するための工夫の入った複雑なものとなっていた．プログラムとしての実現を考える際には，単に M に対応するプログラムが存在するだけではなく，それが，そのプログラミング言語における関数合成の方法に従って，自然に合成できる必要がある．ここで定義した合成可能な形のマルチヘッドストリーム通信の GHC プログラムは，その条件を満たしている．

合成可能な形のプログラムは，一般に，IM2 マシンの実行規則から，自動的に合成することができる．定理 3 の上にあげた GHC プログラムの生成例では，節の左辺の m を m1 に変えて，全体に以下様な形のルールを追加すれば良い．

```
m(Q,XX, YY, W1, W2, [O1,O2|O]) :-
       XX = [X1,X2|X], YY = [Y1,Y2|Y],
       m1(Q,XX, YY, W1, W2, [O1,O2|O]).
```

各 $n\perp$-列 $p \in \Sigma^{\omega}_{\perp,n}$ に対し p を出力するオラクル述語を考える．$oracle_p$(X) は，引数 X が $[\mathtt{Y}_0, \ldots, \mathtt{Y}_n|\mathtt{Y}]$ に束縛されると，$\mathtt{Y}_0, \ldots, \mathtt{Y}_n$ のうちの一つ $\mathtt{Y}_j$ を具体化し，$oracle_p([\mathtt{Y}_0, \ldots, \mathtt{Y}_{j-1}, \mathtt{Y}_{j+1}, \ldots, \mathtt{Y}_n|\mathtt{Y}])$ という新たなゴールを生成するという形で，p を出力するとする．各 p に対し，そのような述語は一意的ではないことに注意されたい．

定理 4 f は $Y_1 \times \ldots \times Y_k \rightrightarrows Y_0$ 型の多値関数とし，$(\Sigma^{\omega}_{\perp,n_1}, \ldots, \Sigma^{\omega}_{\perp,n_k}, \Sigma^{\omega}_{\perp,n_0})$ 型の *IM2* マシン M が f を実現しているとする．その時，合成可能な形の *GHC* のプログラム

```
mc(X1, X2,...,Xk, X0)
```

が存在し，mc は f を計算する．すなわち，$(p_1, \ldots, p_k) \in dom(f)$ に対し，

```
main:- oracle_{p_1}(X_1), ..., oracle_{p_k}(X_k),
          mc(X_1, X_2, ..., X_k, X), q(X).
q(YY) :- YY = [Y_1, ..., Y_{n_0}|Y], q1([Y1, ..., Y_{n_0}|Y]).
q1([0|Y]) :- q(Y).
```

```
q1([1|Y]) :- q(Y).
q1([C,1|Y]) :- q([C|Y]).
...
```

というプログラムは main の変数 X に無限の出力を行い，無限時間の実行の結果 X に得られる出力は，$f(p_1,\ldots,p_k)$ の要素である．ここで，... の部分には、q1 の引数のリストの最初の n_0 文字の中の 1 つだけ Σ の文字に具体化したものに対し同様な節が存在するとする．

この方式で，5 章のグレイコードによる実数の足し算 pl を合成可能な形の KL1 プログラムとして実装した．図 5 にその一部を示す．プログラム全体は [9] からダウンロードできる．ここでは，元とした Haskell の文法による pl の定義が，IM2 マシンの定義を翻訳したものではなく，複数の文字を先読みしてどの節を適用するか判断する形になっている．そのために，必要となる最大数 (3 個) のストリーム要素を要求する形になっている．

8 IM2 マシン定義可能性と GHC 定義可能との関係

6 章で述べた様に，IM2 マシンの動作を GHC で記述し，実行することができる．逆はどうだろうか．今，GHC のプログラム foo(X,Y) が，$\Sigma^{\omega}_{\perp,n}$ から $\Sigma^{\omega}_{\perp,m}$ への部分多値関数 f を実現しているとする．すなわち，$p \in dom(f)$ に対し，X に $p \in \Sigma^{\omega}_{\perp,n}$ を束縛してプログラムを実行すると，$f(p) \subset \Sigma^{\omega}_{\perp,m}$ の要素のどれかを Y に出力するとする．ここでも，無限式を入力変数に束縛するという概念を含んだ操作的意味の元で考えることにする．その時，型 $(\Sigma^{\omega}_{\perp,n}, \Sigma^{\omega}_{\perp,m})$ を持つ IM2 マシン M が存在し，M が f を実現することが言えるだろうか．IM2 マシンは，チューリングマシンの概念の拡張であり，GHC のインタープリタに相当するものも書くことができる．入出力だけは注意が必要である．元の foo の出力は，$\Sigma^{\omega}_{\perp,m}$ の文字列の IM2 マシンの出力順序のどれにも沿っていないかもしれない．しかし，一旦，内部テープに出力していき，先頭の $m+1$ 個のうちのどれかの値が確定した時点で出力を行う様にすればよい．入力も同様に，プログラムの実行のために必要な所まで，内部テープに読み込んでからルールの検索を行えばよい．よって，型 $(\Sigma^*, \Sigma^{\omega}_{\perp,n}, \Sigma^{\omega}_{\perp,m})$ の IM2 マシン M_{UNIV} が存在し，第 1 引数に foo のプログラムを有限列としてエンコードしたものを与え，第 2 引数に $p \in \Sigma^{\omega}_{\perp,n}$ を与えると，$f(p)$ の要素のどれかを出力する IM2 マシンが存在する．第 1 引数の代りに，foo のプログラムをエンコードしたものを内部テープに作成する処理を最初に行う IM2 マシンを考えることにより，目的のマシンを構成することができる．よって，次のことが示せた．

定理 5 *$n\perp$-列上の部分多値関数では，IM2 マシン定義可能関数の集合と，GHC のプログラムとして定義可能な関数の集合は一致する．*

ここでは，マシンの入出力は，$n\perp$-列，すなわち，$\Sigma^{\omega}_{\perp,n}$ に制限していた．この制限をはずしたらどうだろうか．すなわち，GHC のプログラム foo(X,Y) が，$\Sigma^{\omega}_{\perp}$ から $\Sigma^{\omega}_{\perp}$ への部分多値関数 f を実現している時，ある n,m が存在し，$dom(f) \in \Sigma^{\omega}_{\perp,n}$ であり，f は $(\Sigma^{\omega}_{\perp,n}, \Sigma^{\omega}_{\perp,m})$ マシン定義可能となるだろうか．答えは否定的である．例えば，f として，$\Sigma^{\omega}_{\perp}$ から $\Sigma^{\omega}_{\perp}$ への恒等関数を考えれば良い．

```
ap(X,Y) :- X = Y.
```

により，恒等関数は GHC プログラムとして定義可能であり，その定義域はどの $\Sigma^{\omega}_{\perp,n}$ $(n>0)$ より広い．命題 1 と合わせて，以下のことが言える．

定理 6 *$\Sigma^{\omega}_{\perp} \rightrightarrows \Sigma^{\omega}_{\perp}$ の上には，$(\Sigma^{\omega}_{\perp,1}, \Sigma^{\omega}_{\perp,1})$ マシン定義可能 $\subset$ $(\Sigma^{\omega}_{\perp,2}, \Sigma^{\omega}_{\perp,2})$ マシン定義可能 $\subset \ldots$ という階層があり，GHC のプログラムとして定義可能な関数の集合は，このどれよりも大きい．*

9　他の言語での実現

最後に，$n\perp$-列の入出力の，他の言語での実現について述べたい．

SICStus Prolog などの Prolog 言語には，block, wait といった，変数が具体化するのを待つためのプリミティブが存在する．wait の条件には，1 つの変数だけではなく，複数の変数のどれかが具体化するのを待つといった複雑なことも書ける．ガードの並列実行は書けないが，var という，変数が束縛されているかどうかを調べる述語が存在するので，SICStus Prolog でも，IM2 マシン定義可能な $\Sigma^{\omega}_{\perp,n}$ 上の関数を記述することができる．SICStus Prolog によるグレイコード実数の足し算のプログラムを [10] に置いてあるので，見て頂きたい．wait 文を多用しているので，かなり煩雑なものとはなっているが，それは，SICStus Prolog の，block 宣言の記述力が弱く，具体化するのを待つ変数の指定の所にパターンを書くことができないためであり，本質的ではない．GHC は並列論理型言語と言われているが，IM2 マシンの動作を実装する上では，並行な実行は本質ではないと述べた．Prolog でも，block, wait などを駆使すれば，並列な実行をシミュレートするプログラムも可能であるが，ここではそのようなプログラミングは行っていないことに注意されたい．

遅延評価を持つ関数型言語を拡張する実現方法も考えられる．5 章で述べた様に，IM2 マシンの動作は Haskell の構文で自然に表現できる．それをそのまま実行できる様な言語の拡張は，多値関数の定義を許すことになるので，慎重に行われる必要がある一方，実数計算の立場からは，もっとも自然な実装方法といえる．関数型言語の遅延評価は，要求駆動的に引数を弱頭正規形 (whnf) まで評価することによって実行が進む．拡張された無限列の入出力を許すには，例えばヘッドが 2 つの場合には「m と n のどちらかが whnf になるまで評価を行う」という形の要求駆動が必要である．一見それは，並列実行を必要とする様に見えるが，IM2 マシンを模倣したプログラムでは，m と n は独立した計算にはなり得ない．関数の引数となる無限列が IM2 マシンの出力を模倣したプログラムで作成されるのなら，m と n の計算は独立した計算ということはなく，m と n は常に同じ部分グラフを共有しており，その部分の簡約によって m か n の whnf が得られるはずである．関数型言語をこのように拡張する研究は，今後の課題である．

10　終わりに

実数計算を実現するための枠組みである，不決定的マルチヘッドマシンの動作は，GHC で記述できることを示した．さらに，両者の記述力は，$n\perp$-列の上では一致することを示した．

GHC は，並列のプロセスを記述することを目的に設計された言語である．そのために考案された，guard と commited choice の機構やそこから必然的に生じる非決定性が，実数計算という，全く異なった分野でのアルゴリズムの記述に必要とされている能力と一致していることは，大変興味深い．

謝辞

Prolog や KL1 がグレイコード実数の実現に向いているのではないかという示唆は，多くの方から頂いた．中でも，奈良女子大学の鴨浩靖氏，St.Andrews 大学の Ian Gent 氏に感謝したい．

付録

IM2 マシンの定義

Σ を，$\perp$ を含まない文字集合, Γ を，ブランク文字 B を含む文字集合とする．Y_i $(i = 0, \ldots, k)$ を，自然数 n に対し $\Sigma^{\omega}_{\perp,n}$ のどれかだとする．型 $(Y_1, \ldots, Y_k, Y_0)$ の IM2 マシンとは，k 本の入力テープ $T_l (l = 1, \ldots, k)$，1 本の出力テープ T_0，数本の内部テープ，内部状態の集合 Q (初期状態 q_0 を含む)，および，計算規則からなるマシンで，以下のようなものである．

$Y_l = \Sigma^{\omega}_{\perp,n}$ の時，入出力テープ T_l には $n+1$ 個のヘッド $H_1(T_l), \ldots, H_{n+1}(T_l)$ が存在し，T_l には Σ の文字および $\perp$ が置けるものとする．また，内部テープ T には 1 つのヘッド $H(T)$ が存在し，Γ の文字が

書けるものとする．

計算規則は，以下の形をしているとする．

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

ここで，q と q' は，Q の要素であり，h_i は異なる入力テープのヘッド，o は出力テープのヘッド，w_i, w'_i, w''_i は内部テープのヘッド，$c_i (i = 1, \ldots, j)$ および c は Σ の文字，d_i, d'_i は Γ の文字，M_i は $+$ か $-$ のどちらかとする．規則のそれぞれの部分は存在しなくてもよい: 例えば，$o(c)$ の部分の存在しない規則があってもよい．

この規則の意味は，以下の通りである．状態が q, ヘッド $h_i (i = 1, \ldots, j)$ と $w_e (e = 1, \ldots, l)$ の位置の文字がそれぞれ c_i と d_e の時規則を適用でき，状態が q' に変化し，ヘッド o および $w'_i (i = 1, \ldots, m)$ の元に c, d'_i の文字が書かれ，ヘッド w''_i が，M_i が $+$ なら前に，$-$ なら後ろに 1 セル移動し，ヘッド $h_i (i = 1, \ldots, j)$ および o が $H_e(T_l)$ であり，T_l の型が $\Sigma^{\omega}_{\perp,n}$ なら，$H_d(T_l)$ $(e \leq d \leq n)$ がそれぞれ $H_{d+1}(T_l)$ の位置に移動し，$H_{n+1}(T_l)$ が，1 セル前に移動する．

参考文献

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

[2] Vasco Brattka. A stability theorem for recursive analysis. In *Proc. of 3rd Workshop on Computability and Complexity in Analysis*, 1998.

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

[4] Andrzej Grzegorczyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.

[5] Report on the programming language haskell, version 1.2, 1992.

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

[7] Simon L. Peyton. *The Implementation of Functional Programming Languages.* Prentice/Hall, 1987.

[8] Ehud Shapiro and Akikazu Takeuchi. *Object oriented programming in Concurrent Prolog.* ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.

[9] Hideki Tsuiki. http://www.i.h.kyoto-u.ac.jp/~tsuiki/cca/klic/plus_drv.kl1.

[10] Hideki Tsuiki. http://www.i.h.kyoto-u.ac.jp/~tsuiki/cca/klic/plus_drv.pl.

[11] Hideki Tsuiki. Real number computation through gray code embedding. *Theoretical Computer Science*, 2000. to appear.

[12] Hideki Tsuiki. The Topological Dimension of a Topological Space. In *Proc. Workshop on Computability and Complexity in Analysis*, 2000, to appear.

[13] Hideki Tsuiki. The computational power of PCF extended with multi-head stream accesses. In preparation, 2001.

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

[15] Kazunori Ueda. Guarded horn clauses. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers, Vol.1, Chapter 4*. The MIT Press, 1986.

[16] Kazunori Ueda and Koichi Furukawa. Transformation rules for ghc programs. In *Proc. the International Conf. on Fifth Generation Computer Systems*, pages 582–591. ICOT, Tokyo, 1988.

[17] Klaus Weihrauch. *An introduction to computable analysis*. Springer-Verlag, to appear, 2000.

[18] 立木秀樹. 実数の表現とグレイコード. 数理科学, (11):26–33, 1999.