# Gray-code and Program Extraction based on pre-Gray code

Joint work with Ulrich Berger, Kenji Miyamoto, and Helmut Schwichtenberg

Hideki Tsuiki

Interval Analysis and Constructive Mathematics
11/13-18, 2016, CMO Workshop, Oaxaca

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.
- The code space is not $\Sigma^\omega$ for some alphabet $\Sigma$, but we allow at most one $\perp$ in each sequence.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.
- The code space is not $\Sigma^\omega$ for some alphabet $\Sigma$, but we allow at most one $\perp$ in each sequence.
- A two-head nondeterministic machine is used for computation over such "bottomed" sequences.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.
- The code space is not $\Sigma^\omega$ for some alphabet $\Sigma$, but we allow at most one $\perp$ in each sequence.
- A two-head nondeterministic machine is used for computation over such "bottomed" sequences.
- Algorithms for simple functions like average are given, but it does not have enough logical treatment.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.
- The code space is not $\Sigma^\omega$ for some alphabet $\Sigma$, but we allow at most one $\perp$ in each sequence.
- A two-head nondeterministic machine is used for computation over such "bottomed" sequences.
- Algorithms for simple functions like average are given, but it does not have enough logical treatment.
- In this talk, we try to give logical background to such computation by formalizing Gray-code in logical systems.

# Summary of the talk

- Computable representation for real numbers need to be redundant, i.e., most number have more than one code.
- (Pure) Gray-code for real numbers [T 2002] is a non-redundant coding that can be used for computation.
- The code space is not $\Sigma^\omega$ for some alphabet $\Sigma$, but we allow at most one $\perp$ in each sequence.
- A two-head nondeterministic machine is used for computation over such "bottomed" sequences.
- Algorithms for simple functions like average are given, but it does not have enough logical treatment.
- In this talk, we try to give logical background to such computation by formalizing Gray-code in logical systems.
- We consider coalgebra of Gray-code and extract Gray-code algorithms from proofs.

1. Gray code of real number
2. Algebra/coalgebra of (pre-)Gray code
3. Program extraction based on pre-Gray code
4. Pure Gray code

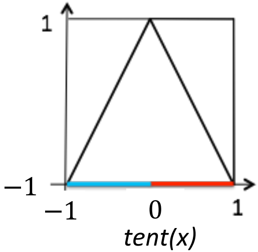
# Gray code

- (Binary-reflected) Gray-code is a coding of natural numbers.
- The Hamming distance between adjacent numbers is always 1.
- We consider expansion of the unit interval [-1, 1] based on Gray-code.

| | Binary | Gray |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 10 | 11 |
| 3 | 11 | 10 |
| 4 | 100 | 110 |
| 5 | 101 | 111 |
| 6 | 110 | 101 |
| 7 | 111 | 100 |
| 8 | 1000 | 1100 |
| 9 | 1001 | 1101 |
| 10 | 1010 | 1111 |
| 11 | 1011 | 1110 |
| 12 | 1100 | 1010 |
| 13 | 1101 | 1011 |
| 14 | 1110 | 1001 |
| 15 | 1111 | 1000 |

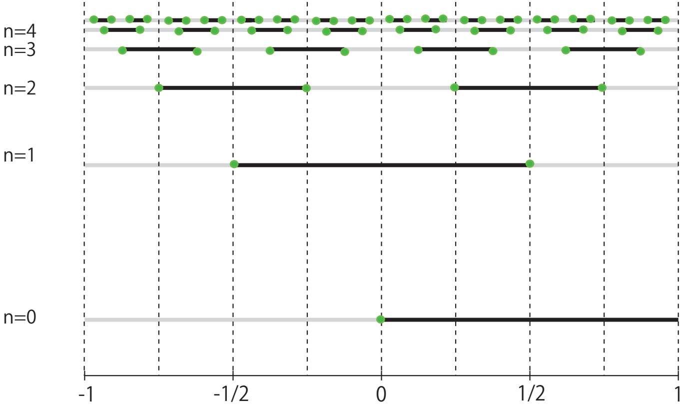
# Pure Gray-code for real number

- We use $\{\bar{1}(=-1), 1\}$ instead of $\{0, 1\}$.
- $tent(x) = \begin{cases} 1 + 2x & (-1 \leq x \leq 0) \\ 1 - 2x & (0 < x \leq 1) \end{cases}$.
- $P(x) = \begin{cases} \bar{1} & (x < 0) \\ \perp & (x = 0) \\ 1 & (x > 0) \end{cases}$.

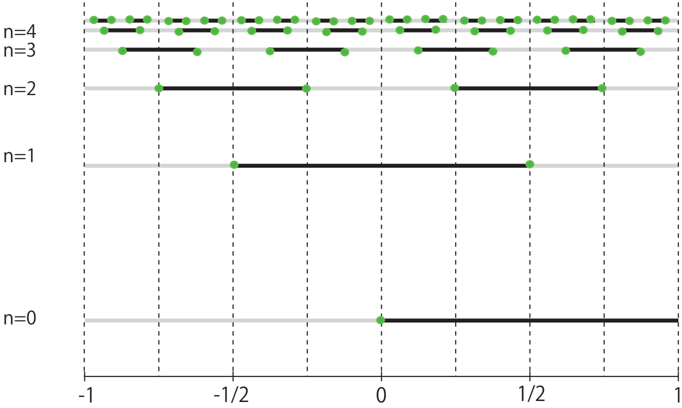



- The pure Gray code $\varphi(x) \in \{\perp, \bar{1}, 1\}^\omega$ of $x$ is defined as the itinerary of the tent function. That is, $\varphi(x)(n) = P(tent^n(x))$ $(n = 0, 1, \ldots)$

# Gray-code (gray for $\bar{1}$, black for 1, green ball for $\perp$)

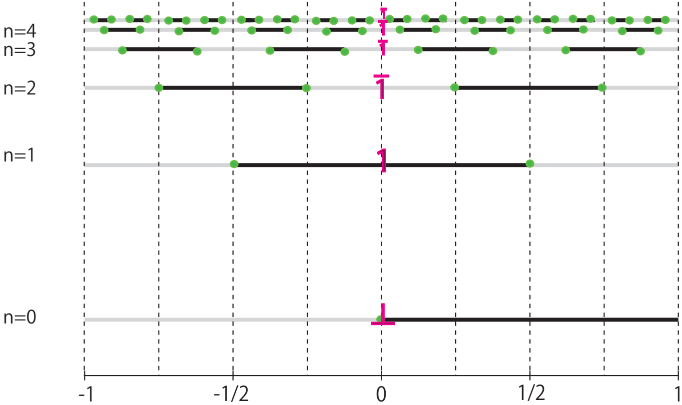


# Gray-code (gray for $\bar{1}$, black for 1, green ball for $\perp$)



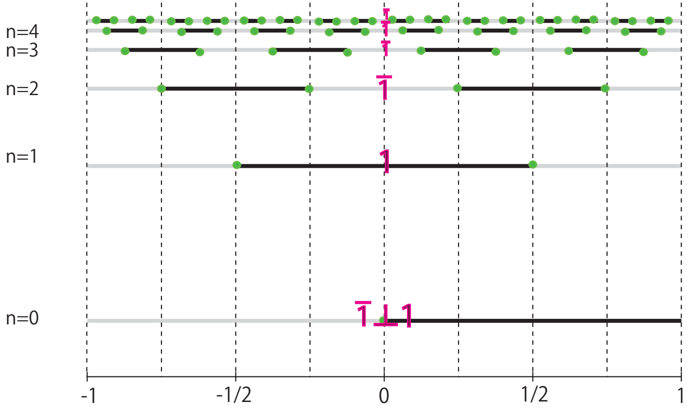

- $\varphi$ is a topological embedding from $[-1,1]$ to $\{\perp,\bar{1},1\}^\omega$, with the Scott topology of the domain $\{\perp,\bar{1},1\}^\omega$. (equal to the product of the topology on $\mathbb{T}=\{\perp,\bar{1},1\}$ generated by $\{\{\bar{1}\},\{1\}\}$.)

# Gray-code (gray for $\bar{1}$, black for 1, green ball for $\perp$)




- The pure Gray-code $\varphi(x)$ of a dyadic rational $x$ contains one $\perp$ and, after that, the sequence is always $1\bar{1}^{\omega}$ .

# Gray-code (gray for $\bar{1}$, black for 1, green ball for $\perp$)




- The pure Gray-code $\varphi(x)$ of a dyadic rational $x$ contains one $\perp$ and, after that, the sequence is always $1\bar{1}^{\omega}$ .
- We mainly consider Gray-code which is a little redundant in that all the three codes $sa1\bar{1}^{\omega}$ for $a \in \{\bar{1}, 1, \perp\}$ for dyadic rationals.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $1$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $1$ | $\bar{1}$ | $1$ | $\perp$ | | | | | | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $\perp$ | $1$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $\perp$ | $1$ | $\bar{1}$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\perp$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\perp$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\perp$ | $\cdots$ |

- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\cdots$ |

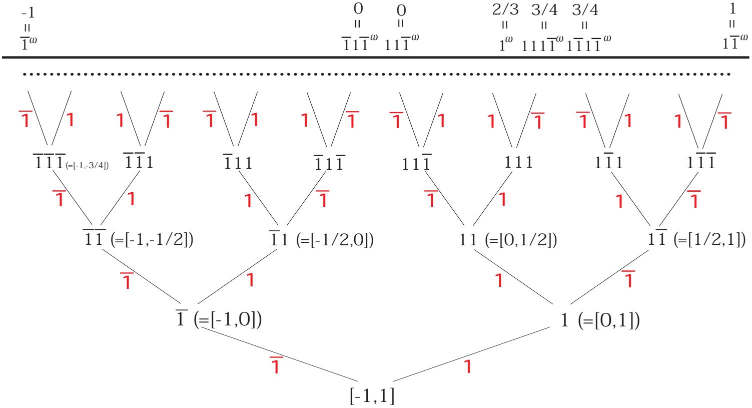
- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.

# IM2-machine

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\bar{1}$ | $1$ | $1$ | $\bar{1}$ | $\bar{1}$ | $1$ | $\perp$ | $1$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\cdots$ |

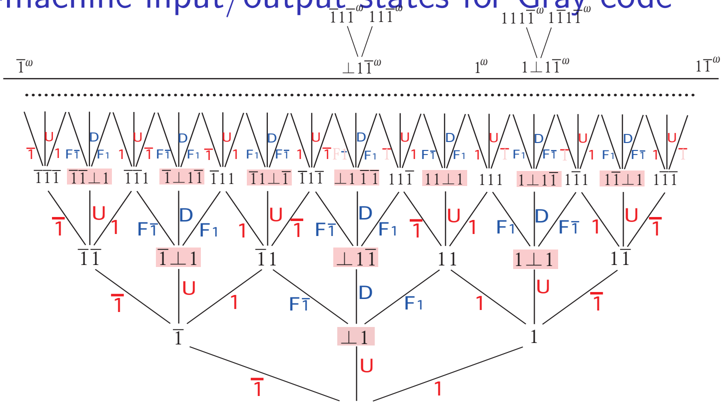
- $\perp$ means undefinedness and it is not an ordinary character. One cannot read or write a $\perp$.
- If one tries to read a $1\perp$-sequence from left to right, one cannot access the subsequence after $\perp$.
- IM2-machine is a machine which has two heads on each input/output tape to acccess $1\perp$-sequences.
- With the two heads, one can leave a cell undefined and go ahead, and fill/read it later.
- If there is an undefined cell, then one cannot make another skip until the undefined cell is filled. In this way, it is guaranteed to have at most one unfilled cell.
- If a cell is left undefined eternally, then it is $\perp$ in the infinite $1\perp$-sequence.

# Gray expansion




- Each node is denoting an interval, which is shrinking according to the input.

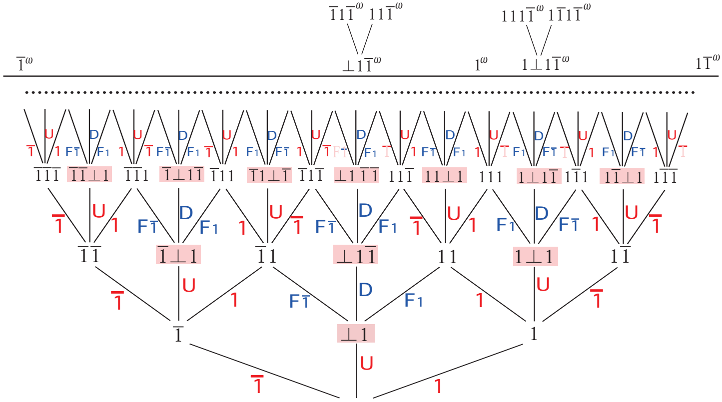
# IM2-machine input/output states for Gray code



Two states **G** (normal state) and **H** (auxiliary state, red in picture).

- $\mathrm{LR}_{\bar{1}}, \mathrm{LR}_1$ : fill the next cell with 1 or $\bar{1}$. **G** $\Rightarrow$ **G**
- $\mathrm{U}$(*undefined*): skip one cell and fill the next cell with 1. **G** $\Rightarrow$ **H**
- $\mathrm{D}$(*delay*): fill yet next cell with $\bar{1}$. **H** $\Rightarrow$ **H**
- $\mathrm{Fin}_{\bar{1}}, \mathrm{Fin}_1$: fill the skipped cell with 1 or $\bar{1}$. **H** $\Rightarrow$ **G**.

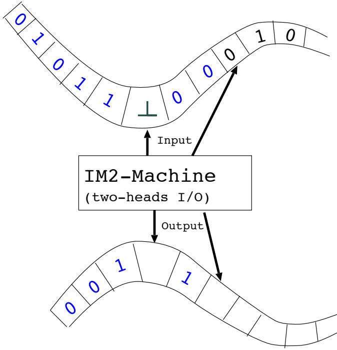
# IM2-machine input/output states for Gray code



- Finite states correspond to signed digit intervals.
- Limits of this finite states corresponds to ideal completion.
- It is a domain representation of the unit interval [Blanck].

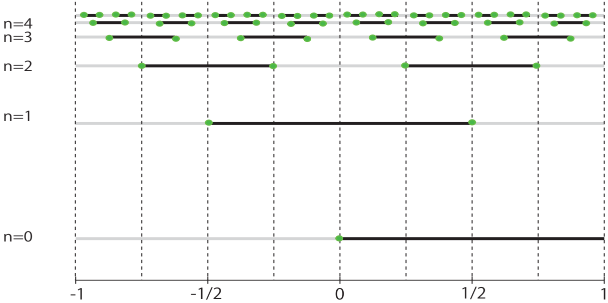
# IM2-machine = skip and fill later

- It is nondeterministic depending on which head is used when both of the heads have values.
- IM2-machine algorithms are directly executable in committed choice logic programming languages.
- We express such a manipulation of $1\perp$-sequence in Haskell syntax.
- Note that $\perp$ is a valid data of type Int in Haskell, and [Int] contain $1\perp$-sequences.



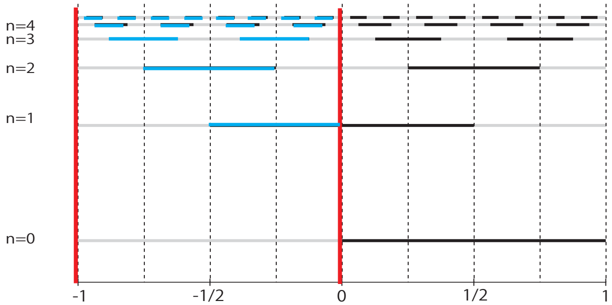
# Example1: Signed digit to Gray-code conversion

$$\mathtt{itog}(-1 : \mathtt{xs}) = \bar{1} : \mathtt{itog}\ \mathtt{xs}$$
$$\mathtt{itog}(1 : \mathtt{xs}) = 1 : \mathtt{nh}\ (\mathtt{itog}\ \mathtt{xs})$$
$$\mathtt{itog}(0 : \mathtt{xs}) = \mathtt{c} : 1 : \mathtt{nh}\ \mathtt{ds}\ \ \mathtt{where}\ \mathtt{c} : \mathtt{ds} = \mathtt{itog}\ \mathtt{xs}$$
$$\mathtt{nh}(\mathtt{s} : \mathtt{ds}) = -\mathtt{s} : \mathtt{ds}$$



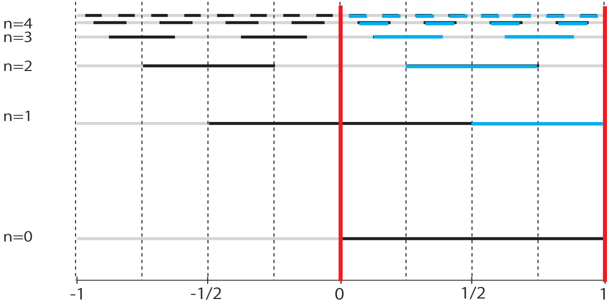
# Example1: Signed digit to Gray-code conversion

$\texttt{itog}(-1 : \texttt{xs}) = \bar{1} : \texttt{itog xs}$
$\texttt{itog}(1 : \texttt{xs}) = 1 : \texttt{nh}\ (\texttt{itog xs})$
$\texttt{itog}(0 : \texttt{xs}) = \texttt{c} : 1 : \texttt{nh ds} \quad \texttt{where c} : \texttt{ds} = \texttt{itog xs}$
$\texttt{nh}(\texttt{s} : \texttt{ds}) = -\texttt{s} : \texttt{ds}$



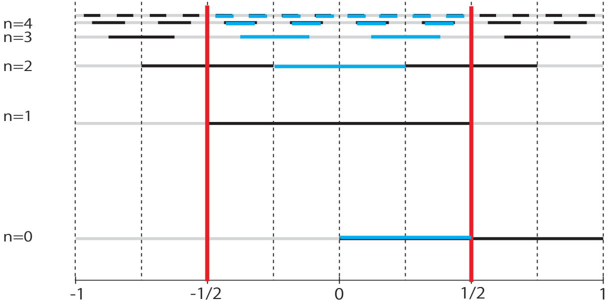
# Example1: Signed digit to Gray-code conversion

$$\mathtt{itog}(-1 : \mathtt{xs}) = \bar{1} : \mathtt{itog\ xs}$$
$$\mathtt{itog}(1 : \mathtt{xs}) = 1 : \mathtt{nh}\ (\mathtt{itog\ xs})$$
$$\mathtt{itog}(0 : \mathtt{xs}) = \mathtt{c} : 1 : \mathtt{nh\ ds}\ \ \mathtt{where\ c} : \mathtt{ds} = \mathtt{itog\ xs}$$
$$\mathtt{nh}(\mathtt{s} : \mathtt{ds}) = -\mathtt{s} : \mathtt{ds}$$



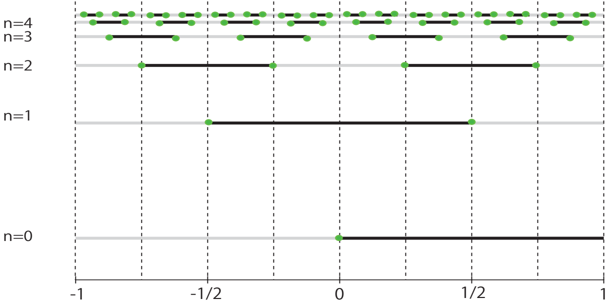
# Example1: Signed digit to Gray-code conversion

$$\mathtt{itog}(-1 : \mathtt{xs}) = \bar{1} : \mathtt{itog\ xs}$$
$$\mathtt{itog}(1 : \mathtt{xs}) = 1 : \mathtt{nh}\ (\mathtt{itog\ xs})$$
$$\mathtt{itog}(0 : \mathtt{xs}) = \mathtt{c} : 1 : \mathtt{nh\ ds} \quad \mathtt{where\ c : ds = itog\ xs}$$
$$\mathtt{nh}(\mathtt{s} : \mathtt{ds}) = -\mathtt{s} : \mathtt{ds}$$



# Example1: Signed digit to Gray-code conversion

$$\mathtt{itog}(-1 : \mathtt{xs}) = \bar{1} : \mathtt{itog\ xs}$$
$$\mathtt{itog}(1 : \mathtt{xs}) = 1 : \mathtt{nh}\ (\mathtt{itog\ xs})$$
$$\mathtt{itog}(0 : \mathtt{xs}) = \mathtt{c} : 1 : \mathtt{nh\ ds}\ \ \mathtt{where\ c} : \mathtt{ds} = \mathtt{itog\ xs}$$
$$\mathtt{nh}(\mathtt{s} : \mathtt{ds}) = -\mathtt{s} : \mathtt{ds}$$




- It is a correct Haskell program.
- `itog([0,0,..])` does not output the first digit because it is $\perp$.
- `tail(itog([0,0,..]))` outputs $[1, \bar{1}, \bar{1}, \bar{1}, \ldots$

# Example2: Gray-code to signed digit conversion

$$\begin{aligned}
&\mathtt{gtoi}(1 : \mathtt{xs}) = 1 : \mathtt{gtoi}(\mathtt{nh}\ \mathtt{xs})\\
&\mathtt{gtoi}(\bar{1} : \mathtt{xs}) = -1 : \mathtt{gtoi}\ \mathtt{xs}\\
&\mathtt{gtoi}(\mathtt{c} : 1 : \mathtt{xs}) = 0 : \mathtt{gtoi}(\mathtt{c} : \mathtt{nh}\ \mathtt{xs})
\end{aligned}$$

- It is not correct as a Haskell program in that when the argument is $[\perp, 1, \bar{1}, \bar{1}, \ldots]$, Haskell tries to evaluate the first digit and it starts a non-terminating computation and fails to apply the third rule.
- It is correct as equations, and one can execute it as term-rewriting rule.

# Example3: Average function

$$\begin{array}{l}
\mathtt{av}\ (\bar{1} : \mathtt{as})\ (\bar{1} : \mathtt{bs})\ =\ \bar{1} : \mathtt{av\ as\ bs} \\
\mathtt{av}\ (1 : \mathtt{as})\ (1 : \mathtt{bs})\ =\ 1 : \mathtt{av\ as\ bs} \\
\mathtt{av}\ (\bar{1} : \mathtt{as})\ (1 : \mathtt{bs})\ =\ \mathtt{c} : 1 : \mathtt{nh\ cs}\quad \mathtt{where\ c : cs}\ =\ \mathtt{av\ as\ (nh\ bs)} \\
\mathtt{av}\ (1 : \mathtt{as})\ (\bar{1} : \mathtt{bs})\ =\ \mathtt{c} : 1 : \mathtt{nh\ cs}\quad \mathtt{where\ c : cs}\ =\ \mathtt{av\ (nh\ as)\ bs} \\
\mathtt{av}\ (\mathtt{a} : 1 : \mathtt{as})\ (\mathtt{b} : 1 : \mathtt{bs})\ =\ \mathtt{c} : 1 : \mathtt{nh\ cs\ where\ c : cs}\ =\ \mathtt{av}\ (\mathtt{a} : \mathtt{nh\ as})\ (\mathtt{b} : \mathtt{nh\ bs})
\end{array}$$

$$\begin{array}{l}
\mathtt{av}\ (\mathtt{a} : 1 : \bar{1} : \mathtt{as})\ (\bar{1} : \bar{1} : \mathtt{bs})\ =\ \bar{1} : \mathtt{av}\ (\mathtt{a} : 1 : \mathtt{as})\ (1 : \mathtt{nh\ bs}) \\
\mathtt{av}\ (\mathtt{a} : 1 : \bar{1} : \mathtt{as})\ (1 : \bar{1} : \mathtt{bs})\ =\ 1 : \mathtt{av}\ (\mathtt{not\ a} : 1 : \mathtt{as})\ (1 : \mathtt{nh\ bs}) \\
\mathtt{av}\ (\mathtt{a} : 1 : \bar{1} : \mathtt{as})\ (\bar{1} : \mathtt{b} : 1 : \mathtt{bs})\ =\ \bar{1} : 1 : \mathtt{av}\ (\mathtt{not\ a} : \mathtt{nh\ as})\ (\mathtt{not\ b} : \mathtt{nh\ bs}) \\
\mathtt{av}\ (\mathtt{a} : 1 : \bar{1} : \mathtt{as})\ (1 : \mathtt{b} : 1 : \mathtt{bs})\ =\ 1 : 1 : \mathtt{av}\ (\mathtt{a} : \mathtt{nh\ as})\ (\mathtt{not\ b} : \mathtt{nh\ bs})
\end{array}$$

$$\begin{array}{l}
\mathtt{av}\ (\bar{1} : \bar{1} : \mathtt{as})\ (\mathtt{b} : 1 : \bar{1} : \mathtt{bs})\ =\ \bar{1} : \mathtt{av}\ (1 : \mathtt{nh\ as})\ (\mathtt{b} : 1 : \mathtt{bs}) \\
\mathtt{av}\ (1 : \bar{1} : \mathtt{as})\ (\mathtt{b} : 1 : \bar{1} : \mathtt{bs})\ =\ 1 : \mathtt{av}\ (1 : \mathtt{nh\ as})\ (\mathtt{not\ b} : 1 : \mathtt{bs}) \\
\mathtt{av}\ (\bar{1} : \mathtt{a} : 1 : \mathtt{as})\ (\mathtt{b} : 1 : \bar{1} : \mathtt{bs})\ =\ \bar{1} : 1 : \mathtt{av}\ (\mathtt{not\ a} : \mathtt{nh\ as})\ (\mathtt{not\ b} : \mathtt{nh\ bs}) \\
\mathtt{av}\ (1 : \mathtt{a} : 1 : \mathtt{as})\ (\mathtt{b} : 1 : \bar{1} : \mathtt{bs})\ =\ 1 : 1 : \mathtt{av}\ (\mathtt{not\ a} : \mathtt{nh\ as})\ (\mathtt{b} : \mathtt{nh\ bs})
\end{array}$$

- Correct program (equality of the both sides, covering over all the patterns, productivity check).
- How can we formally prove its correctness?
- What is the theory of computation over $1\bot$-sequences.
- Our goal is to study coalgebra of $1\bot$-sequences and consider logic to manipulate real number through Gray-code, and extract this kind of programs from proofs.

1. Gray code of real number
2. Algebra/coalgebra of (pre-)Gray code
3. Program extraction based on pre-Gray code
4. Pure Gray code

# Algebra and coalgebra of ordinary sequences.

- Two constructors $\mathsf{cons}_a$ ($a \in \{\bar{1}, 1\}$) meaning to prepend $a$, in addition to nil denoting the empty sequence.
- The term $\mathsf{cons}_1(\mathsf{cons}_1(\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1\,\mathsf{nil})))$ denotes $11\bar{1}1$:

$$\begin{array}{rcr} \mathsf{nil} & \text{denotes} & \epsilon, \\ (\mathsf{cons}_1\,\mathsf{nil}) & \text{denotes} & 1, \\ (\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1\,\mathsf{nil})) & \text{denotes} & \bar{1}1, \\ (\mathsf{cons}_1(\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1\,\mathsf{nil}))) & \text{denotes} & 1\bar{1}1, \\ (\mathsf{cons}_1(\mathsf{cons}_1(\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1\,\mathsf{nil})))) & \text{denotes} & 11\bar{1}1. \end{array}$$

- It is a free algebra.
- For coalgebraic treatment, we read an infinite sequence of constructors from left to right.
- Starting with $\perp^\omega$ on an infinite tape, $\mathsf{cons}_a$ is an operation to fill the leftmost $\perp$ with $a$.
  $\perp^\omega \to 1\perp^\omega \to 11\perp^\omega \to 11\bar{1}\perp^\omega \to 11\bar{1}1\perp^\omega$

# Algebra and coalgebra of $1\perp$-sequences.

- Finite $1\perp$-sequence: an infinite sequence with $\perp^\omega$ at the end and at most one $\perp$ before that.
- We use two constructors $\mathsf{ins}_a$ ($a \in \{\bar{1}, 1\}$) meaning to insert $a$ as the 2nd character, in addition to $\mathsf{cons}_a$ ($a \in \{\bar{1}, 1\}$) and nil (meaning $\perp^\omega$).
- The term $\mathsf{ins}_1(\mathsf{ins}_{\bar{1}}(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\,\mathsf{nil})))$ denotes $\bar{1}1\bar{1}\perp 1$:

$$\begin{array}{rlr}
\mathsf{nil} & \text{denotes} & \perp^\omega, \\
(\mathsf{ins}_1\,\mathsf{nil}) & \text{denotes} & \perp 1 \perp^\omega, \\
(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\,\mathsf{nil})) & \text{denotes} & \bar{1}\perp 1 \perp^\omega, \\
(\mathsf{ins}_{\bar{1}}(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\,\mathsf{nil}))) & \text{denotes} & \bar{1}\bar{1}\perp 1 \perp^\omega, \\
(\mathsf{ins}_1(\mathsf{ins}_{\bar{1}}(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\,\mathsf{nil})))) & \text{denotes} & \bar{1}1\bar{1}\perp 1 \perp^\omega.
\end{array}$$

- $(\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\,\mathsf{nil}))))$ also denote the same sequence.
- $\mathsf{ins}_a \circ \mathsf{cons}_b = \mathsf{cons}_b \circ \mathsf{cons}_a$. It is not a free algebra.
- When read from left to right, one can prove that $\mathsf{ins}_a$ is an operation to fill the 2nd $\perp$ from the left with $a$.
$\perp^\omega \to \perp 1 \perp^\omega \to \perp 1 \bar{1} \perp^\omega \to \bar{1} 1 \bar{1} \perp^\omega \to \bar{1} 1 \bar{1} \perp 1 \perp^\omega$

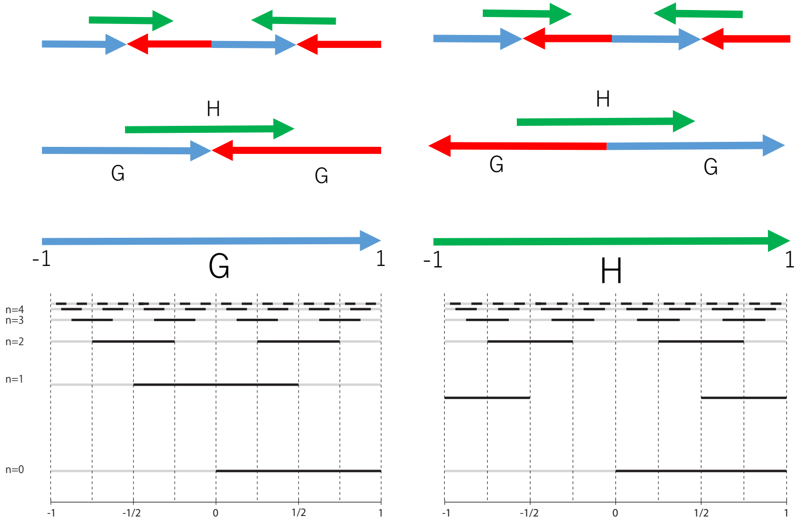
# Algebra and coalgebra of Gray code

- We need to restrict so that only $1\bar{1}^\omega$ appear after $\bot$.
- We consider mutually recursively defined subalgebras **G** and **H** of the algebra of $1\bot$-sequences

| | $\mathsf{cons}_a$ $(a \in \{\bar{1}, 1\})$ | $\mathsf{ins}_{\bar{1}}$ | $\mathsf{ins}_1$ | nil |
|---|---|---|---|---|
| **G** | $\mathrm{LR}_a : \mathbf{G} \to \mathbf{G}$ | | $\mathrm{U} : \mathbf{H} \to \mathbf{G}$ | $\mathsf{nil}_{\mathbf{G}} : \mathbf{G}$ |
| **H** | $\mathrm{Fin}_a : \mathbf{G} \to \mathbf{H}$ | $\mathrm{D} : \mathbf{H} \to \mathbf{H}$ | | $\mathsf{nil}_{\mathbf{H}} : \mathbf{H}$ |

- The carrier set of **G** is the set of finite Gray-codes.
- Example: $(\mathsf{ins}_1(\mathsf{ins}_{\bar{1}}(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\ \mathsf{nil}))))$, $(\mathsf{cons}_{\bar{1}}(\mathsf{cons}_1(\mathsf{cons}_{\bar{1}}(\mathsf{ins}_1\ \mathsf{nil}))))$.
  $\mathrm{U}(\mathrm{D}(\mathrm{Fin}_{\bar{1}}(\mathrm{U}(\mathsf{nil}_{\mathbf{H}}))))$ and $\mathrm{LR}_{\bar{1}}(\mathrm{LR}_1(\mathrm{LR}_{\bar{1}}(\mathrm{U}(\mathsf{nil}_{\mathbf{H}}))))$
  both are terms of type **G** representing $\bar{1}1\bar{1}\bot 1\bot^\omega$.
- Their meanings as left-to-right operation on $1\bot$-sequences are
  - $\mathrm{LR}_a : \bar{1}1\bot^\omega \mapsto \bar{1}1a\bot^\omega$
  - $\mathrm{U} : \bar{1}1\bot^\omega \mapsto \bar{1}1\bot 1\bot^\omega$
  - $\mathrm{Fin}_a : \bar{1}1\bot 1\bar{1}\bot^\omega \mapsto \bar{1}1a1\bar{1}\bot^\omega$
  - $\mathrm{D} : \bar{1}1\bot 1\bar{1}\bot^\omega \mapsto \bar{1}1\bot 1\bar{1}\bar{1}\bot^\omega$
- We call an infinite term of type **G** a pre-Gray code.

# **G** and **H** as codings of [-1, 1]




We study through these mutually-recursively defined codes of [-1, 1].

# Meaning of Gray code

- For each constructor $C$, define $f_C : [-1,1] \to [-1,1]$ as

$$f_{\mathrm{LR}_a}(x) = -a\frac{x-1}{2} \quad f_{\mathrm{Fin}_a}(x) = a\frac{x+1}{2} = f_{\mathrm{LR}_a}(-x), \tag{1}$$

$$f_{\mathrm{U}}(x) = \frac{x}{2}. \qquad f_{\mathrm{D}}(x) = \frac{x}{2}. \tag{2}$$

- We define the meaning of a finite term $v = [a_1 \dots a_n]$ $(= a_1(a_2 \dots (a_n \ \mathrm{nil}_*)))$ of **G** as the interval

$$f_{a_1}(f_{a_2}(\dots f_{a_n}(\mathbb{I}) \dots))$$

- We define the meaning $[\![v]\!]_{\mathbf{G}}$ of pre-Gray code $v = [a_1, a_2, \dots]$ as the unique real number in the intersection of the intervals denoted by its finite truncations.

$$[\![p]\!]_{\mathbf{G}} = \bigcap_{n=1}^{\infty} f_{a_1}(f_{a_2}(\dots f_{a_n}(\mathbb{I}) \dots))$$

- Similarly for $[\![v]\!]_{\mathbf{H}}$ and $[\![v]\!]_{\mathbf{I}}$.

1. Gray code of real number
2. Algebra/coalgebra of (pre-)Gray code
3. Program extraction based on pre-Gray code
4. Pure Gray code

# Extraction of real number algorithms

- We represent Gray-code as pre-Gray code, that is, as an infinite sequence of constructors of **G** and **H**.
- We formalize pre-Gray code in $\mathrm{TCF}$ (the *Theory of Computable Functionals*) by means of coinductive definitions. In TCF, infinite structures like pre-Gray code are treated as cototal ideals.
- We use the proof assistant system Minlog, which is an implementation of $\mathrm{TCF}$, and make a constructive proof of a formula. Minlog system will extract from the proof a program as a term in an extension $T^+$ of Gödel's $T$ involving higher type recursion and corecursion operators. We do not go into the detail.
- I show the formulas to be proved and the extracted program as a readable stream-transforming program.

# Predicates ${}^{\mathrm{co}}G(x)$ and ${}^{\mathrm{co}}H(x)$

- We define the predicate ${}^{\mathrm{co}}I(x)$ saying that $x$ has a signed digit representation as the greatest fixed point of a strictly positive operator.
- We define the predicates ${}^{\mathrm{co}}G(x)$ and ${}^{\mathrm{co}}H(x)$ saying that $x$ has a **G** term (i.e., a pre-Gray code) and **H** term, respectively, as the greatest fixed points of a mutually-defined strictly positive operator.
- We use the following coalgebraic data type in programs

$$
\begin{aligned}
\mathbf{I} &= \mathrm{C}\ \{\bar{1}, 0, 1\} \times \mathbf{I}\\
\mathbf{G} &= \mathrm{LR}\ \{\bar{1}, 1\} \times \mathbf{G}\ +\ \mathrm{U}\ \mathbf{H},\\
\mathbf{H} &= \mathrm{Fin}\ \{\bar{1}, 1\} \times \mathbf{G}\ +\ \mathrm{D}\ \mathbf{H}.
\end{aligned}
$$

# Signed digit to Gray-code conversion

**Theorem** $\forall^{\mathrm{nc}}_x ({}^{\mathrm{co}}I(x) \to {}^{\mathrm{co}}G(x))$.

**Lemma** $\forall^{\mathrm{nc}}_x (\exists_a {}^{\mathrm{co}}I(ax) \to {}^{\mathrm{co}}G(x))$, $\forall^{\mathrm{nc}}_x (\exists_a {}^{\mathrm{co}}I(ax) \to {}^{\mathrm{co}}H(x))$.

**Extracted Program:** $\mathtt{itoPreG} : \mathbf{I} \to \mathbf{G}$, $\mathtt{g} \colon \{-1, 1\} \times \mathbf{I} \to \mathbf{G}$, $\mathtt{h} \colon \{-1, 1\} \times \mathbf{I} \to \mathbf{H}$

$$\begin{aligned}
&\mathtt{itoPreG}(\mathtt{v}) &&= \mathtt{g}(1, \mathtt{v}) && && \\
&\mathtt{g}(\mathtt{b}, \mathrm{C}_{-1}(\mathtt{v})) &&= \mathrm{LR}_{-\mathtt{b}}(\mathtt{g}(1, \mathtt{v})), \qquad && \mathtt{h}(\mathtt{b}, \mathrm{C}_{-1}(\mathtt{v})) &&= \mathrm{Fin}_{-\mathtt{b}}(\mathtt{g}(-1, \mathtt{v})), \\
&\mathtt{g}(\mathtt{b}, \mathrm{C}_{1}(\mathtt{v})) &&= \mathrm{LR}_{\mathtt{b}}(\mathtt{g}(-1, \mathtt{v})), \qquad && \mathtt{h}(\mathtt{b}, \mathrm{C}_{1}(\mathtt{v})) &&= \mathrm{Fin}_{\mathtt{b}}(\mathtt{g}(1, \mathtt{v})), \\
&\mathtt{g}(\mathtt{b}, \mathrm{C}_{0}(\mathtt{v})) &&= \mathrm{U}(\mathtt{h}(\mathtt{b}, \mathtt{v})), \qquad && \mathtt{h}(\mathtt{b}, \mathrm{C}_{0}(\mathtt{v})) &&= \mathrm{D}(\mathtt{h}(\mathtt{b}, \mathtt{v})).
\end{aligned}$$

Recall the original Gray-code program we had is

$$\begin{aligned}
&\mathtt{itog}(-1 : \mathtt{xs}) = -1 : \mathtt{itog\ xs} \\
&\mathtt{itog}(1 : \mathtt{xs}) = 1 : \mathtt{nh}\ (\mathtt{itog\ xs}) \\
&\mathtt{itog}(0 : \mathtt{xs}) = \mathtt{c} : 1 : \mathtt{nh\ ds} \ \ \mathtt{where\ c} : \mathtt{ds} = \mathtt{itog\ xs} \\
&\mathtt{nh}(\mathtt{s} : \mathtt{ds}) = -\mathtt{s} : \mathtt{ds}
\end{aligned}$$

Through some program transformation, we can show

$$\mathtt{preGtoG}(\mathtt{itoPreG}(\mathtt{v})) = \mathtt{itog}(\mathtt{v})$$

for preGtoG a program to transform a pre-Gray code to Gray code

$$\mathtt{preGtoG}(\mathtt{D} : \mathtt{p}) \ = \ \mathtt{a} : 0 : \mathtt{x} \ \ \mathtt{where\ a} : \mathtt{x} \ = \ \mathtt{preGtoG}(\mathtt{p})$$

# Gray-code to Signed digit conversion

**Theorem** $\forall^{\mathrm{nc}}_x({}^{\mathrm{co}}G(x) \to {}^{\mathrm{co}}I(x))$.

**Lemma** $\forall^{\mathrm{nc}}_x(\exists_a({}^{\mathrm{co}}G(ax) \vee {}^{\mathrm{co}}H(ax)) \to {}^{\mathrm{co}}Ix)$.

**Extracted Program:** $\mathtt{preGtoI} : \mathbf{G} \to \mathbf{I}$,
$[\mathtt{f}, \mathtt{g}] \colon \{-1, 1\} \times \mathbf{G} + \{-1, 1\} \times \mathbf{H} \to \mathbf{I}$

$$
\begin{aligned}
&\mathtt{preGtoI}(\mathtt{v}) &&= \mathtt{f}(1, \mathtt{v}) && && \\
&\mathtt{f}(\mathtt{a}, \mathrm{LR}_{\mathtt{b}}(\mathtt{p})) &&= \mathrm{C}_{\mathtt{a}*\mathtt{b}}(\mathtt{f}(-\mathtt{a}*\mathtt{b}, \mathtt{p})), &&\mathtt{g}(\mathtt{a}, \mathrm{Fin}_{\mathtt{b}}(\mathtt{p})) &&= \mathrm{C}_{\mathtt{a}*\mathtt{b}}(\mathtt{f}(\mathtt{a}*\mathtt{b}, \mathtt{p})), \\
&\mathtt{f}(\mathtt{a}, \mathrm{U}(\mathtt{q})) &&= \mathrm{C}_0(\mathtt{g}(\mathtt{a}, \mathtt{q})), &&\mathtt{g}(\mathtt{a}, \mathrm{D}(\mathtt{q})) &&= \mathrm{C}_0(\mathtt{g}(\mathtt{a}, \mathtt{q})).
\end{aligned}
$$

Recall the original Gray-code program we had is

$$
\begin{aligned}
&\mathtt{gtoi}(1 : \mathtt{xs}) = 1 : \mathtt{gtoi}(\mathtt{nh\ xs}) \\
&\mathtt{gtoi}(-1 : \mathtt{xs}) = -1 : \mathtt{gtoi\ xs} \\
&\mathtt{gtoi}(\mathtt{c} : 1 : \mathtt{xs}) = 0 : \mathtt{gtoi}(\mathtt{c} : \mathtt{nh\ xs})
\end{aligned}
$$

Through some program transformation, we can show

$$\mathtt{preGtoI}(\mathtt{v}) = \mathtt{gtoi}(\mathtt{preGtoG}(\mathtt{v}))$$

# Average

**Lemma** $\forall^{\mathrm{nc}}_x({}^{\mathrm{co}}G(-x) \to {}^{\mathrm{co}}Gx), \forall^{\mathrm{nc}}_x({}^{\mathrm{co}}H(-x) \to {}^{\mathrm{co}}Hx)$.

**Extracted Program:** `ming`: $\mathbf{G} \to \mathbf{G}$ and `minh`: $\mathbf{H} \to \mathbf{H}$

$$\begin{aligned} &\mathit{minf}(\mathrm{LR}_a(p)) = \mathrm{LR}_{-\mathtt{a}}(\mathrm{p}), && \mathit{minh}(\mathrm{Fin}_a(p)) = \mathrm{Fin}_{-\mathtt{a}}(\mathrm{p}),\\ &\mathit{minf}(\mathrm{U}(q)) \quad = \mathrm{U}(\mathit{minh}(q)), && \mathit{minh}(\mathrm{D}(q)) \quad = \mathrm{D}(\mathtt{minh}(\mathrm{q})). \end{aligned}$$

**Lemma** $\forall^{\mathrm{nc}}_x({}^{\mathrm{co}}Hx \to {}^{\mathrm{co}}Gx), \forall^{\mathrm{nc}}_x({}^{\mathrm{co}}Gx \to {}^{\mathrm{co}}Hx)$.

**Extracted Program:** `htog`: $\mathbf{H} \to \mathbf{G}$ and `gtoh`: $\mathbf{G} \to \mathbf{H}$:

$$\begin{aligned} &\mathit{htog}(\mathrm{Fin}_a(p)) = \mathrm{LR}_{\mathtt{a}}(\mathtt{ming}(\mathrm{p})), && \mathit{gtoh}(\mathrm{LR}_a(p)) = \mathrm{Fin}_{\mathtt{a}}(\mathtt{ming}(\mathrm{p})),\\ &\mathit{htog}(\mathrm{D}(q)) \quad = \mathrm{U}(\mathrm{q}), && \mathit{gtoh}(\mathrm{U}(q)) \quad = \mathrm{D}(\mathrm{q}) \end{aligned}$$

**Lemma A** $\forall^{\mathrm{nc}}_{x,y\in{}^{\mathrm{co}}G}\exists^{\mathrm{r}}_{x',y'\in{}^{\mathrm{co}}G}\exists_i(\frac{x+y}{2}=\frac{x'+y'+i}{4})$.

**Extracted Program:** $\texttt{lemA}\colon \mathbf{G}\times\mathbf{G}\to\{-2,-1,0,1,2\}\times\mathbf{G}\times\mathbf{G}$

$$\begin{aligned}
&\mathit{lemA}(\mathrm{LR}_a(p),\mathrm{LR}_{a'}(p')) &&= (\mathtt{a}+\mathtt{a}',\mathtt{mult}(-\mathtt{a},\mathtt{p}),\mathtt{mult}(-\mathtt{a}',\mathtt{p}')),\\
&\mathit{lemA}(\mathrm{LR}_a(p),\mathrm{U}(q)) &&= (\mathtt{a},\mathtt{mult}(-\mathtt{a},\mathtt{p}),\mathtt{htog}(\mathtt{q})),\\
&\mathit{lemA}(\mathrm{U}(q),\mathrm{LR}_a(p)) &&= (\mathtt{a},\mathtt{htog}(\mathtt{q}),\mathtt{mult}(-\mathtt{a},\mathtt{p})),\\
&\mathit{lemA}(\mathrm{U}(q),\mathrm{U}(q')) &&= (\mathtt{0},\mathtt{htog}(\mathtt{q}),\mathtt{htog}(\mathtt{q}')).
\end{aligned}$$

**Lemma B** $\forall_i\forall^{\mathrm{nc}}_{x,y\in{}^{\mathrm{co}}G}\exists^{\mathrm{r}}_{x',y'\in{}^{\mathrm{co}}G}\exists_{j,d}(\frac{x+y+i}{4}=\frac{\frac{x'+y'+j}{4}+d}{2})$.

**Extracted Program:** $\texttt{lemB}\colon \{-2,-1,0,1,2\}\times\mathbf{G}\times\mathbf{G}\to$
$\{-2,-1,0,1,2\}\times\{-1,0,1\}\times\mathbf{G}\times\mathbf{G}$

$$\begin{aligned}
&\mathit{lemB}(i,\mathrm{LR}_a(p),\mathrm{LR}_{a'}(p')) &&= (\mathtt{J}(\mathtt{a},\mathtt{a}',\mathtt{i}),\mathtt{K}(\mathtt{a},\mathtt{a}',\mathtt{i}),\mathtt{mult}(-\mathtt{a},\mathtt{p}),\mathtt{mult}(\\
&\mathit{lemB}(i,\mathrm{LR}_a(p),\mathrm{U}(q)) &&= (\mathtt{J}(\mathtt{a},\mathtt{0},\mathtt{i}),\mathtt{K}(\mathtt{a},\mathtt{0},\mathtt{i}),\mathtt{mult}(-\mathtt{a},\mathtt{p}),\mathtt{htog}(\mathtt{q}\\
&\mathit{lemB}(i,\mathrm{U}(q),\mathrm{LR}_a(p)) &&= (\mathtt{J}(\mathtt{0},\mathtt{a},\mathtt{i}),\mathtt{K}(\mathtt{0},\mathtt{a},\mathtt{i}),\mathtt{htog}(\mathtt{q}),\mathtt{mult}(-\mathtt{a},\mathtt{p}\\
&\mathit{lemB}(i,\mathrm{U}(q),\mathrm{U}(q')) &&= (\mathtt{J}(\mathtt{0},\mathtt{0},\mathtt{i}),\mathtt{K}(\mathtt{0},\mathtt{0},\mathtt{i}),\mathtt{htog}(\mathtt{q}),\mathtt{htog}(\mathtt{q}')).
\end{aligned}$$

**Lemma C** $\forall^{\mathrm{nc}}_z (\exists^{\mathrm{r}}_{x,y \in {}^{\mathrm{co}}G} \exists_i (z = \frac{x+y+i}{4}) \to {}^{\mathrm{co}}G(z)), \forall^{\mathrm{nc}}_z (\exists^{\mathrm{r}}_{x,y \in {}^{\mathrm{co}}G} \exists_i (z = \frac{x+y+i}{4}) \to {}^{\mathrm{co}}H(z))$.

**Extracted Program:** $\texttt{lemCg}\colon \{-2,-1,0,1,2\} \times \mathbf{G} \times \mathbf{G} \to \mathbf{G}$, $\texttt{lemCh}\colon \{-2,-1,0,1,2\} \times \mathbf{G} \times \mathbf{G} \to \mathbf{H}$.

$$
\begin{aligned}
&\texttt{lemCg}(\texttt{i},\texttt{p},\texttt{p}') = \text{let } (\texttt{i}_1,\texttt{d},\texttt{p}_1,\texttt{p}'_1) = \texttt{lemB}(\texttt{i},\texttt{p},\texttt{p}') \text{ in} \\
&\qquad \text{case } (\texttt{d}) \text{ of} \\
&\qquad\quad \texttt{0} \to \mathrm{U}(\texttt{lemCh}(\texttt{i},\texttt{p}_1,\texttt{p}'_1)) \\
&\qquad\quad \texttt{a} \to \mathrm{LR}_{\texttt{a}}(\texttt{lemCg}(-\texttt{ai},\texttt{mult}(-\texttt{a},\texttt{p}_1),\texttt{mult}(-\texttt{a},\texttt{p}'_1))), \\
&\texttt{lemCh}(\texttt{i},\texttt{p},\texttt{p}') = \text{let } (\texttt{i}_1,\texttt{d},\texttt{p}_1,\texttt{p}'_1) = \texttt{lemB}(\texttt{i},\texttt{p},\texttt{p}') \text{ in} \\
&\qquad \text{case } (\texttt{d}) \text{ of} \\
&\qquad\quad \texttt{0} \to \mathrm{D}(\texttt{lemCh}(\texttt{i},\texttt{p}_1,\texttt{p}'_1)) \\
&\qquad\quad \texttt{a} \to \mathrm{Fin}_{\texttt{a}}(\texttt{lemCg}(-\texttt{a} * \texttt{i},\texttt{mult}(-\texttt{a},\texttt{p}_1),\texttt{mult}(-\texttt{a},\texttt{p}'_1))).
\end{aligned}
$$

**Theorem** $\forall^{\mathrm{nc}}_{x,y}({}^{\mathrm{co}}G(x) \to {}^{\mathrm{co}}G(y) \to {}^{\mathrm{co}}G(\frac{x+y}{2}))$.

**Extracted Program:** average: $\mathbf{G} \times \mathbf{G} \to \mathbf{G}$

$$\mathtt{average}(\mathrm{p}, \mathrm{p}') = \mathtt{lemCg}(\mathtt{lemA}(\mathrm{p}, \mathrm{p}'))$$

Recall the original Gray-code program we had is

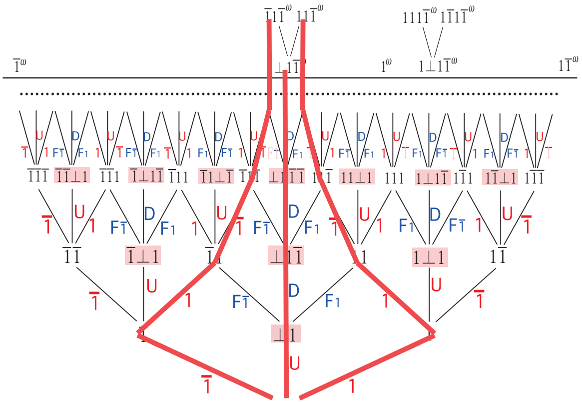
```
av (0:as) (0:bs) = 0:av as bs
av (1:as) (1:bs) = 1:av as bs
av (0:as) (1:bs) = c:1:nh cs where c:cs = av as (nh bs)
av (1:as) (0:bs) = c:1:nh cs where c:cs = av (nh as) bs
av (a:1:as) (b:1:bs) = c:1:nh cs where c:cs = av (a:nh as) (b:nh bs)

av (a:1:0:as) (0:0:bs) = 0:av (a:1:as) (1:nh bs)
av (a:1:0:as) (1:0:bs) = 1:av (not a:1:as) (1:nh bs)
av (a:1:0:as) (0:b:1:bs) = 0:1:av (not a:nh as) (not b:nh bs)
av (a:1:0:as) (1:b:1:bs) = 1:1:av (a:nh as) (not b:nh bs)

av (0:0:as) (b:1:0:bs) = 0:av (1:nh as) (b:1:bs)
av (1:0:as) (b:1:0:bs) = 1:av (1:nh as) (not b:1:bs)
av (0:a:1:as) (b:1:0:bs) = 0:1:av (not a:nh as) (not b:nh bs)
av (1:a:1:as) (b:1:0:bs) = 1:1:av (not a:nh as) (b:nh bs)
```

It seems like a non-equivalent program. Pre-Gray code is redundant and there are many different ways to output the same Gray-code.

1. Gray code of real number
2. Algebra/coalgebra of (pre-)Gray code
3. Program extraction based on pre-Gray code
4. Pure Gray code

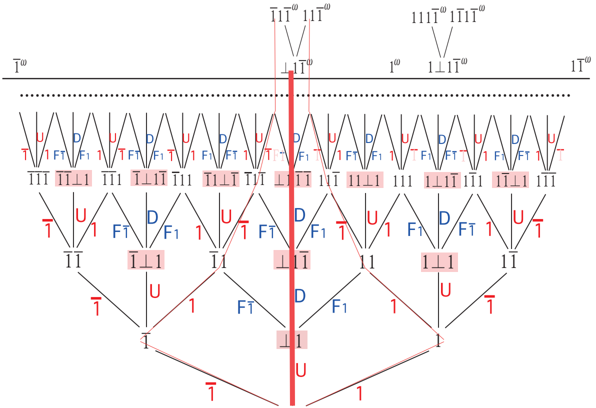
# Pure-Gray code

- We studied algorithms based on (a bit redundant) Gray-code rather than pure Gray-code. Gray-code allowed all the increasing sequences in the domain of finite pre-Gray codes.
- However, we are interested in Gray-code because it is not redundant.
- Can we input/output pure Gray-code?



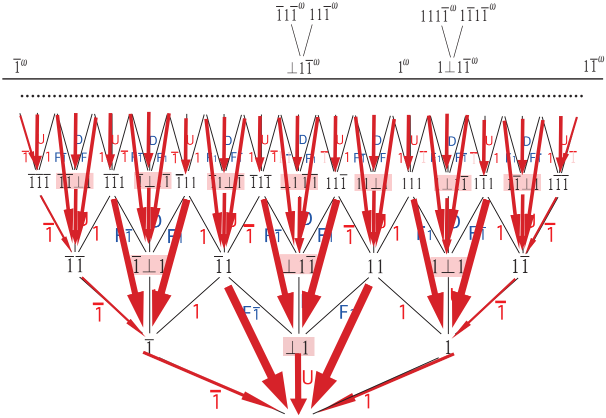
# Pure-Gray code

- We studied algorithms based on (a bit redundant) Gray-code rather than pure Gray-code. Gray-code allowed all the increasing sequences in the domain of finite pre-Gray codes.
- However, we are interested in Gray-code because it is not redundant.
- Can we input/output pure Gray-code?



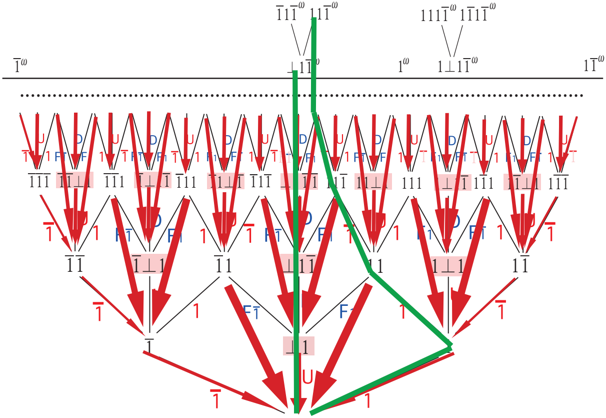
# Conversion from Gray code to pure-Gray code

- There is a conversion from Gray-code to Pure Gray-code.



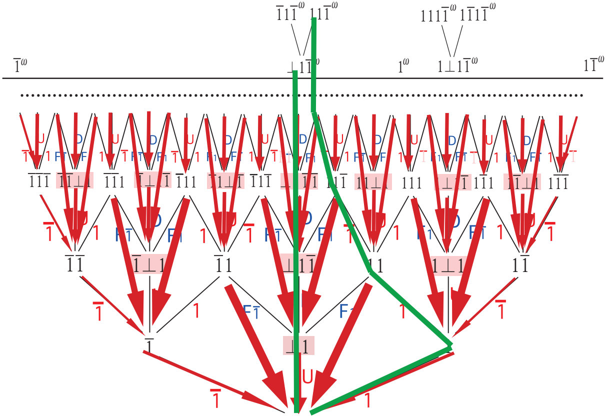
# Conversion from Gray code to pure-Gray code

- There is a conversion from Gray-code to Pure Gray-code.



# Conversion from Gray code to pure-Gray code

- There is a conversion from Gray-code to Pure Gray-code.



- We extract this conversion from a constructive proof.

# Conversion from Gray code to pure-Gray code

- We define variants $\Gamma'$, $\Delta'$ of the operators $\Gamma$, $\Delta$ by

$$\Gamma'(X,Y) := \{\, y \mid \exists^{\mathrm{r}}_{x\in X}\exists_a(y = -a\tfrac{x-1}{2} \wedge y \neq 0) \vee \exists^{\mathrm{r}}_{x\in Y}(y = \tfrac{x}{2} \wedge y \neq \pm\tfrac{1}{2})\,\},$$

$$\Delta'(X,Y) := \{\, y \mid \exists^{\mathrm{r}}_{x\in X}\exists_a(y = a\tfrac{x+1}{2} \wedge y \neq 0) \vee \exists^{\mathrm{r}}_{x\in Y}(y = \tfrac{x}{2} \wedge y \neq \pm\tfrac{1}{2})\,\}$$

- We define $({}^{\mathrm{co}}M, {}^{\mathrm{co}}N) := \nu_{(X,Y)}(\Gamma'(X,Y), \Delta'(X,Y))$.
- **Proposition** For cototal ideals $p$ in $\mathbf{G}$ and $x \in \mathbb{I}$

$$({}^{\mathrm{co}}M)^{\mathbf{r}}(p,x) \leftrightarrow \varphi(p) \text{ is a pure Gray code of } x.$$

# Extraction of the conversion from Gray to Pure Gray

**Theorem** $\forall^{\mathrm{nc}}_x({}^{\mathrm{co}}G(x) \to {}^{\mathrm{co}}M(x)), \forall^{\mathrm{nc}}_x({}^{\mathrm{co}}H(x) \to {}^{\mathrm{co}}N(x))$.

**Extracted Program:** $\mathtt{g}\colon \mathbf{G} \to \mathbf{G}$ and $\mathtt{h}\colon \mathbf{H} \to \mathbf{H}$, defined by (with $a$ for $\mathrm{LR}_a$)

$$
\begin{array}{llll}
\mathtt{g}(\mathtt{a}(\bar{1}(\mathrm{p}))) & = \mathtt{a}(\mathtt{g}(\bar{1}(\mathrm{p}))) & \mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\bar{1}(\mathrm{p})))) & = \mathrm{D}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) \\
\mathtt{g}(\mathtt{a}(1(\bar{1}(\mathrm{p})))) & = \mathrm{U}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) & \mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(1(\mathrm{p})))) & = \mathrm{Fin}_\mathtt{a}(\mathtt{g}(\bar{1}(1(\mathrm{p})))) \\
\mathtt{g}(\mathtt{a}(1(1(\mathrm{p})))) & = \mathtt{a}(\mathtt{g}(1(1(\mathrm{p})))) & \mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{U}(\mathrm{q})))) & = \mathrm{Fin}_\mathtt{a}(\mathtt{g}(\bar{1}(\mathrm{U}(\mathrm{q})))) \\
\mathtt{g}(\mathtt{a}(1(\mathrm{U}(\mathrm{q})))) & = \mathtt{a}(\mathtt{g}(1(\mathrm{U}(\mathrm{q})))) & \mathtt{h}(\mathrm{Fin}_\mathtt{a}(1(\mathrm{p}))) & = \mathrm{Fin}_\mathtt{a}(\mathtt{g}(1(\mathrm{p}))) \\
\mathtt{g}(\mathtt{a}(\mathrm{U}(\mathrm{q}))) & = \mathtt{a}(\mathtt{g}(\mathrm{U}(\mathrm{q}))) & \mathtt{h}(\mathrm{Fin}_\mathtt{a}(\mathrm{U}(\mathrm{q}))) & = \mathrm{Fin}_\mathtt{a}(\mathtt{g}(\mathrm{U}(\mathrm{q}))) \\
\mathtt{g}(\mathrm{U}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) & = \mathrm{U}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) & \mathtt{h}(\mathrm{D}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) & = \mathrm{D}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\bar{1}(\mathrm{p})))) \\
\mathtt{g}(\mathrm{U}(\mathrm{Fin}_\mathtt{a}(1(\mathrm{p})))) & = \mathtt{a}(\mathtt{g}(1(1(\mathrm{p})))) & \mathtt{h}(\mathrm{D}(\mathrm{Fin}_\mathtt{a}(1(\mathrm{p})))) & = \mathrm{Fin}_\mathtt{a}(\mathtt{g}(\bar{1}(1(\mathrm{p})))) \\
\mathtt{g}(\mathrm{U}(\mathrm{Fin}_\mathtt{a}(\mathrm{U}(\mathrm{q})))) & = \mathrm{U}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\mathrm{U}(\mathrm{q})))) & \mathtt{h}(\mathrm{D}(\mathrm{Fin}_\mathtt{a}(\mathrm{U}(\mathrm{q})))) & = \mathrm{D}(\mathtt{h}(\mathrm{Fin}_\mathtt{a}(\mathrm{U}(\mathrm{q})))) \\
\mathtt{g}(\mathrm{U}(\mathrm{D}(\mathrm{q}))) & = \mathrm{U}(\mathtt{h}(\mathrm{D}(\mathrm{q}))) & \mathtt{h}(\mathrm{D}(\mathrm{D}(\mathrm{q}))) & = \mathrm{D}(\mathtt{h}(\mathrm{D}(\mathrm{q})))
\end{array}
$$

- When `f` is a program which input/output Gray-code, $g \circ f$ is a program which outputs pure Gray-code to pure Gray-code.
- Therefore, every program that handles Gray-code can be converted to a program handling pure Gray-code.

# Concluding remarks

- What kind of benefits do we have with non-redundant codes? Though it looks difficult to make efficient programs,
  - subspace is easier to imagine than quotient,
  - a program is directly operating on real number,
  - it is a direct working application of domain theory,
  - (I hope some practical meaning...)
- We used representation of Gray-code in pre-Gray code, i.e., ordinary sequences and applied the standard theory of coinduction and program extraction. Is there a theory that manipulate Gray-code and $1\perp$-sequences more directly? Ulrich's talk is in that direction.

# References

[1] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Hideki Tsuiki. Logic for Gray-code computation. In D. Probst and P. Schuster (eds), Concepts of Proof in Mathematics, Philosophy, and Computer Science, De Gruyter, 2017.
[2] Helmut Schwichtenberg and Stanley S. Wainer. Proofs and Computations. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, 2012.
[3] Hideki Tsuiki. Real number computation through Gray code embedding. Theoretical Computer Science, 284:467485, 2002.

Thank you very much.

1. Gray code of real number
2. Algebra/coalgebra of (pre-)Gray code
3. Program extraction based on pre-Gray code
4. Pure Gray code
5. Appendix

# Extraction of real number algorithms (Signed Digit case)

- For signed digit rep., we consider the strictly positive operator

$$\Phi(X) := \{\, x \mid \exists^{\mathrm{r}}_{x' \in X} \exists_{d \in \{-1,0,1\}} (x = \frac{x' + d}{2}) \,\}.$$

- We define ${}^{\mathrm{co}}I := \nu_X \Phi(X)$ as the greatest fixed point of $\Phi$.
- ${}^{\mathrm{co}}I$ satisfies the (strengthened) coinduction axiom. That is,

$$X \subseteq \Phi({}^{\mathrm{co}}I \cup X) \to X \subseteq {}^{\mathrm{co}}I.$$

- Next, we consider an operator on pairs $(v, x)$ where $v$ is a signed digit stream and $x$ is a real number.

$$\Phi^{\mathbf{r}}(Y) := \{\, (v, x) \mid \exists^{\mathrm{nc}}_{(v', x') \in Y} \exists_d (x = \frac{x' + d}{2} \wedge v = \mathrm{C}_d(v'))) \,\}.$$

- As its greatest fixed point, we have a relation $({}^{\mathrm{co}}I)^{\mathbf{r}}$ called the realizability extension of ${}^{\mathrm{co}}I$ between signed digit streams $v = [a_1 a_2 \ldots]$ and real numbers $x$.

$$({}^{\mathrm{co}}I)^{\mathbf{r}} := \nu_Y \Phi^{\mathbf{r}}(Y).$$

- **Proposition:** $({}^{\mathrm{co}}I)^{\mathbf{r}}(v,x) \leftrightarrow x = [\![v]\!]_{\mathbf{SD}}$.
- In order to extract a program that computes a function, for example the average function, we prove

$$\forall^{\mathrm{nc}}_{x,x'}({}^{\mathrm{co}}I(x) \to {}^{\mathrm{co}}I(x') \to {}^{\mathrm{co}}I(\frac{x+x'}{2})).$$

- Then, Minlog system will (by the Soundness theorem) extract from the proof a function term $f$ which satisfies

$$({}^{\mathrm{co}}I)^{\mathbf{r}}(v,x) \to ({}^{\mathrm{co}}I)^{\mathbf{r}}(v',x') \to ({}^{\mathrm{co}}I)^{\mathbf{r}}(f(v,v'),\frac{x+x'}{2}).$$

  From the above proposition, this term is a program for the average function,

# Extraction of real number algorithms (pre-Gray code case)

- For the case of pre-Gray code, **G** and **H** are mutually recursively defined cototal ideals. Therefore, we consider the binary strictly positive operator

$$\Gamma(X, Y) := \{\, y \mid \exists^{\mathrm{r}}_{x \in X} \exists_a (y = -a\frac{x-1}{2}) \vee \exists^{\mathrm{r}}_{x \in Y} (y = \frac{x}{2}) \,\},$$
$$\Delta(X, Y) := \{\, y \mid \exists^{\mathrm{r}}_{x \in X} \exists_a (y = a\frac{x+1}{2}) \vee \exists^{\mathrm{r}}_{x \in Y} (y = \frac{x}{2}) \,\}.$$

- Define $({}^{\mathrm{co}}G, {}^{\mathrm{co}}H) := \nu_{(X,Y)}(\Gamma(X, Y), \Delta(X, Y))$ as the greatest fixed point of $(\Gamma, \Delta)$.
- We have the (strengthened) simultaneous coinduction axiom.

$$(X, Y) \subseteq (\Gamma({}^{\mathrm{co}}G \cup X, {}^{\mathrm{co}}H \cup Y), \Delta({}^{\mathrm{co}}G \cup X, {}^{\mathrm{co}}H \cup Y))$$
$$\to (X, Y) \subseteq ({}^{\mathrm{co}}G, {}^{\mathrm{co}}H).$$

- The realizability extension $(({}^{\mathrm{co}}G)^{\mathbf{r}}, ({}^{\mathrm{co}}H)^{\mathbf{r}})$ is a pair of binary predicates on cototal ideals $p$ in **G** or $q$ in **H** (respectively) and real numbers $x$.

- For $x \in \mathbb{I}$ and cototal ideals $p$ in $\mathbf{G}$ and $q$ in $\mathbf{H}$

$$({}^{\mathrm{co}}G)^{\mathbf{r}}(p, x) \leftrightarrow x = [\![p]\!]_{\mathbf{G}},$$
$$({}^{\mathrm{co}}H)^{\mathbf{r}}(q, x) \leftrightarrow x = [\![p]\!]_{\mathbf{H}}$$

- From a proof of

$$\forall^{\mathrm{nc}}_{x,y}({}^{\mathrm{co}}G(x) \to {}^{\mathrm{co}}G(y) \to {}^{\mathrm{co}}G(\frac{x+y}{2})),$$

for exmaple, we obtain a program for the average, which transforms pre-Gray codes of the arguments to a pre-Gray code of the result.

- Coalgebras appearing in the program

$$\mathbf{I} = \mathrm{C}\ \{\bar{1}, 0, 1\} \times \mathbf{I}$$
$$\mathbf{G} = \mathrm{LR}\ \{\bar{1}, 1\} \times \mathbf{G}\ +\ \mathrm{U}\ \mathbf{H},$$
$$\mathbf{H} = \mathrm{Fin}\ \{\bar{1}, 1\} \times \mathbf{G}\ +\ \mathrm{D}\ \mathbf{H}.$$