Streams with a Bottom in Functional Languages

Hideki Tsuiki and Keiji Sugihara

Graduate School of Human and Environmental Studies
Kyoto University, Kyoto, Japan
tsuiki, sugihara@i.h.kyoto-u.ac.jp
Fax: +81-75-753-6744

Abstract. When an infinite sequence contains a bottom cell, we cannot access the rest of the sequence with the ordinary stream access. On the other hand, when we consider an extended stream access with two heads, we can read or write $1\perp$-sequences, which are infinite sequences with at most one bottom cell. In this paper, we present a way of extending a lazy functional language with such an extended stream access in the realm of sequential computation. It has an application in real number computation in that the set of real numbers is topologically embedded in the set of $1\perp$-sequences [16], and therefore we can consider a program with such an extended stream access as directly manipulating real numbers. We implemented this mechanism by modifying the runtime of the Hugs system, which is a graph-reduction based implementation of the Haskell language. We present programming examples like addition and multiplication on real numbers in this extended Haskell.
For this implementation, we extended Haskell with the `gamb` operator, which works just as McCarthy's bottom-avoiding nondeterministic choice operator "amb". The difference is that it is realized in the realm of sequential computation, and that it is applicable only when the graph representations of the arguments share the same redex. In order to show that programs corresponding to two-head stream accesses satisfy this condition, we introduce a PCF-based calculus of term-graphs and define a data-type of $1\perp$-streams as a subtype of `[Bool]`.

1 Introduction

Stream is a useful data structure used in expressing, for example, process communication, and can be manipulated easily in functional languages. We are interested in boolean streams, so a stream in this paper means an infinite sequence of 0 and 1, which is accessed from left to right. One way of representing a stream in a functional language is to use the list type `[Bool]`. However, the type `[Bool]` includes infinite sequences with bottoms and if a program tries to make a stream access to input such an infinite sequence, it will be stuck at the bottom cell because the computation to obtain the value of the cell will not terminate. Therefore, with stream access, the part of the sequence after the first bottom is discarded, though the rest of the sequence may have valuable information.

The first author has found that streams with bottoms are useful in representing continuous topological spaces like $\mathcal{R}$, and performing computation over them.

We will call an infinite sequence which may contain at most n copies of bottom an $n\perp$-sequence, and denote by $\Sigma^{\omega}_{\perp,n}$ the set of $n\perp$-sequences. It is shown in [15] that any n-dimensional separable metric space can be topologically embedded in $\Sigma^{\omega}_{\perp,n}$, and in particular, $\mathcal{R}$ can be topologically embedded in $\Sigma^{\omega}_{\perp,1}$ by what we call the Gray-code embedding. It means that each real number has a unique representation as a $1\perp$-sequence, and through this representation, the approximation structures of $\Sigma^{\omega}_{\perp,1}$ and $\mathcal{R}$ coincide. Note that $\mathcal{R}$ cannot be embedded in Σ^{ω} and therefore the existence of $\perp$ is essential; $\mathcal{R}$ is a 1-dimensional connected space whereas Σ^{ω} is a 0-dimensional totally disconnected space. Thus, if we have a computation which fills (or reads) a $1\perp$-sequence infinitely, then we can consider that it is outputting (or inputting) a real number. As such, he considered a machine called an IM2-machine. This machine has two heads on each input/output tape to make an extended stream access on $1\perp$-sequences. It is shown that the induced computability notion of the real functions coincides with the standard one [19].

In this paper, we present a way of extending a functional language with the two-head stream access of an IM2-machine in the realm of sequential computation. In [17], it is shown that we can express the behavior of an IM2-machine naturally with a logic programming language with guarded clauses and committed choice, such as Concurrent Prolog, PARLOG, and GHC (Guarded Horn Clauses). Therefore, we can already execute our real-number computation algorithms on ordinary computers. However, since what we are expressing are "functions" over the reals like addition and multiplication, it is more desirable that we can express them as functions in functional programming languages. In addition, if they are implemented in functional languages, we can also apply higher-order functions like "map" and "foldr" to real number functions, which is impossible with the above logic programming languages.

It is easy to show that the two-head stream access of an IM2-machine can be implemented if we consider parallel computation and use McCarthy's "amb" operator [11]. The amb operator is a bottom-avoiding nondeterministic choice operator, defined so that `amb` M N is reduced to V if M has the value V, V' if N has the value V', and its computation does not terminate only when both M and N do not have normal forms. Note that if the computations of both of the arguments are terminating, `amb` M N has two possibilities. Therefore, `amb` is a nondeterministic multi-valued function. In order to compute `amb` M N, we need to compute the values of M and N in parallel, and we can express the "parallel or" operator with "`amb`." There are some researches extending parallel functional languages with the amb operator [2]. However, such an implementation requires complicated control and scheduling over threads. Moreover, the motivation and goal of such parallel operator is different from ours. Nondeterminism and multi-valuedness are known to be essential in real-number computation, and, correspondingly, IM2-machines are defining nondeterministic multi-valued functions. However, IM2-machines are performing sequential computation, and real-number computation is not related with parallelism in this context. Therefore, it is more natural to implement it in the realm of sequential computation,

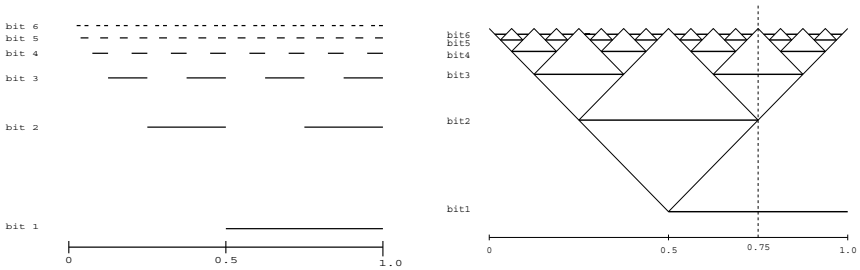


Fig. 1. The binary expansion and the Gray-code expansion of real numbers. Here, horizontal line means that the corresponding bit has value 1 [16].

and see what causes nondeterminism and multi-valuedness without using parallelism.

Thus, we consider our extension to sequential functional languages, and introduce a variant `gamb` of the `amb` operator to a graph-reduction based functional language. It is defined as an extension of the Haskell language and implemented by modifying the runtime system of the Hugs system, which is based on the notion of G-machines[7, 6, 8]. This implementation is available from the author's homepage[18], with some programming examples like addition and multiplication on real numbers.

Our `gamb` operator is a partial sequential realization of the `amb` operator. The difference is that `gamb` works sequentially, and that it works only for the case that the two term-graphs M and N given as the arguments share the same subgraph L as a redex and the normal form of one of M and N is composed from the weak head normal form of L only by list and boolean operations. In order to show that programs corresponding to two-head stream accesses satisfy this condition, we introduce a PCF-based calculus of term-graphs and define a datatype of $1\bot$-streams as a subtype of `[Bool]`. This datatype also brings out a set of primitive operators to manipulate $1\bot$-streams.

In Section 2, we overview Gray-code based real-number computation and IM2-machines, and show how it is expressed with McCarthy's "amb" operator. In Section 3, we introduce the operator `gamb` and explain how it works on term-graphs. In Section 4, we define $\mathrm{GPCF}^{\omega}_{\bot,1}$ and study its type system. In Section 5, we explain how `gamb` is implemented as an extension of Haskell. In Section 6, we explain programming examples of real number functions and higher order functions which use the `gamb` operator.

Notations: We consider the unit closed interval $\mathcal{I} = [0, 1]$ instead of the whole real line. We use 0 and 1 for the boolean values false and true, for simplicity. We fix the alphabet $\Sigma = \{0, 1\}$, and denote by Σ^{ω} the set of infinite sequences of Σ. We call an infinite sequence of $\{0, 1, \bot\}$ which may include at most one copy of $\bot$ an $1\bot$-sequence, and denote by $\Sigma^{\omega}_{\bot,1}$ the set of $1\bot$-sequences. Except for section 4, we use the word term-graph informally for a graph representation of a (lambda) term.

2 Gray-code and Real-number Computation

2.1 Gray-code embedding

Gray-code expansion is an expansion of $\mathcal{I} = [0, 1]$ as infinite sequences of $\{0, 1\}$, which is different from the ordinary binary expansion. Figure 1 shows the binary and Gray-code expansion of $\mathcal{I}$. 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:

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

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:

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

This expansion is based on Gray code[5], which is a binary coding of natural numbers different from the ordinary one.

We have two binary expansions for a dyadic number (a rational number 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. It means that the second bit does not contribute to the fact that the value is 3/4, and it is more natural to leave it undefined ($\perp$). Thus, we define the expansion of 3/4 as $1\perp 1000\ldots$, and define the modified Gray-code expansion as follows.

Definition 1 ([16], [4]). Let $\Sigma = \{0, 1\}$ and $P : \mathcal{I} \to \Sigma_\perp$ be the map

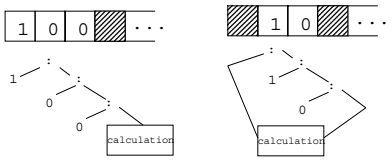
$$P(x) = \begin{cases} 0 & (x < 1/2) \\ \perp & (x = 1/2) \\ 1 & (x > 1/2) \end{cases}.$$

Gray-code embedding G is a function from $\mathcal{I}$ 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.

Note that $\perp$ appears only once in each modified gray-code expansion. G is a topological embedding of $\mathcal{I}$ in $\Sigma^\omega_{\perp,1}$, where the topology of $\Sigma^\omega_{\perp,1}$ is given as the subspace topology of the Scott topology of ${\Sigma_\perp}^\omega$.

2.2 IM2-machine

We study how the ordinary stream access can be extended to input/output $1\perp$-sequences. We explain it with the way the Gray-code of a real number is input or output by a program.



(a) Ordinary stream (b) $1\perp$-stream

Fig. 2. The process of outputting streams

For the output, we consider that a program (effectively) computes a real number x when it executes infinitely and produces better and better approximations of x as shrinking intervals. Therefore, we study how the Gray-code of x is output on a tape based on such information. When the information $x < 1/2$ or $1/2 < x$ is obtained by such a shrinking interval, a machine can write 0 or 1 on the first cell, respectively. However, when $x = 1/2$, neither information is given however long it waits and therefore it cannot fill the first cell forever. 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. After that, the computation of x will have the possibility to fill the first or the 4th cell of the string as Figure 2 shows. 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 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 as an output with two heads. We consider two heads H_1 and H_2 on each tape. They 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. In this way, the two heads are always located at the first and the second unfilled cell of the string. This is a generalization of the ordinary stream access with one head, which is located at the first unfilled cell and 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. 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 output-tape heads. Then, when both of the cells under the two heads are filled, a machine has two possible inputs which may cause two different computations. Therefore, it has nondeterministic behavior and both of the computational paths must produce valid results.

In this way, we define a machine, called an IM2-machine (Indeterministic Multi-head Type2-machine), which has two heads on each input/output tape and which has nondeterministic behavior depending on the order it inputs. See [16] for the detailed definition of an IM2-machine.

2.3 IM2-machine outputs with functional languages

Ordinary stream access can be expressed in a lazy functional language as a recursively defined function of type `[Bool]` $\rightarrow$ `[Bool]`. As for the stream access with two heads, it is trivially impossible to express it in functional languages because multi-valued functions are not definable in functional languages. It is also shown in [17] that some IM2-computable single valued functions are not expressible in functional languages when $1\perp$-sequences are implemented as `[Bool]`. Therefore, for such an implementation, we need some extension to the language.

For the output, we need no extension and we can express the output of a $1\perp$-sequence with two heads in a functional language.

The output from the first head is expressed 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 the new head positions (i.e. `x` and the head of `xs`) comes to be the first two positions of the output of the recursive call of `foo()`. Therefore, we can consider that the tape is composed only of unfilled cells and the two heads are always located at the first two cells of the output. Note that `c` is a constant and therefore, when a term denoting a $1\perp$-stream is reduced to a weak head normal form (i.e., a cons cell), it must have one of the four forms $0 : M'$, $1 : M'$, $x : 0 : M'$, and $x : 1 : M'$.

As for the recursive call, the recursive function may have additional arguments to convey the internal state of the computation. However, in some cases, we can simplify (or even omit) such arguments by allowing the function to modify the result of the recursive call. As such a modification on $1\perp$-sequences, we consider inversion of boolean values. We use the function *nh* to invert the first character of a string defined as follows:

```
not 0 = 1
not 1 = 0
nh c:a = not c:a
```

and allow expressions like `c:nh foo()` and `not x:c:nh xs where x:xs=foo()` for the programs in the above paragraph.

As an example, we consider the function *stog* which converts the signed-digit representation to the Gray-code representation. The signed-digit representation is an expansion of $[-1, 1]$ as an infinite sequence of $\Gamma = \{0, 1, -1\}$, defined as

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

It is equal to the ordinary binary expansion if we do not use -1, and highly redundant in that every real number has infinitely many representations. For our purpose of writing the conversion with Gray-code, we fix the first character as 1 and discard it from the sequence so that every sequence denotes a real number in $\mathcal{I}$. We also change the definition of the Gray-code representation so that when $G(x)$ contains a bottom, then the two sequences obtained by filling the bottom cell with 0 and 1 are also representing x.

When the first digit of a signed-digit representation is $-1, 1$ and 0, it means that the number is in the intervals $[0, 1/2]$, $[1/2, 1]$, and $[1/4, 3/4]$, respectively. Note that these three intervals are expressed in Gray-code representation as the output of 0 and 1 from the first head, and the output of 1 from the second head, respectively. We can write in Haskell the conversion from signed-digit to Gray-code representations as follows considering the recursive structures of both representations [16].

```
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
```

When we execute `stog([0,0..])`, it will have no output on the display because the result is `[⊥,1,0,0..]`, but when we execute `tail(stog([0,0..]))`, it will produce `[1,0,0,0...` infinitely.

2.4 IM2-machine inputs with the amb operator

As for the input, we can express it if we can use McCarthy's "amb" operator[11], as follows. In this paper, we consider a variant of the amb operator of type

```
amb:: a -> a -> Amb a
```

where the datatype `Amb a` is defined as `data Amb a = Left a | Right a`. The term `amb M N` is reduced to `Left` V if M has the normal form V, `Right` V' if N has the normal form V', and its computation does not terminate only when both M and N do not have normal forms. When both M and N have normal forms, we have two possibilities and thus it is a nondeterministic operator.

Since an IM2-machine waits for one of the two cells to be filled, we can express it with the `amb` operator of type `Bool -> Bool -> Amb Bool` as follows.

```
foo(a:b:xs) = case (amb a b) of
      Left 0 -> ... foo(b:xs) ...
      Left 1 -> ... foo(b:xs) ...
      Right 0 -> ... foo(a:xs) ...
      Right 1 -> ... foo(a:xs) ... .
```

Note again that the argument of the recursive call is an infinite list without the cells we have read in. Also for this case, we allow the modification of the argument of the recursive calls with `nh` and `not` such as `foo (not b:nh xs)`.

As an example, we consider the *gtos* function which converts the Gray-code representation to the signed-digit representation. This program is also constructed from the recursive structures of both representations.

```
gtos(a:b:xs) = case (amb a b) of
      Left 0 -> -1:(gtos (b:xs))
      Left 1 -> 1:(gtos (nh (b:xs)))
      Right 1 -> 0:(gtos (a:nh xs))
      Right 0 -> case a of 0 -> -1: -1:(gtos xs)
                           1 -> 1:1:(gtos (nh xs))
```

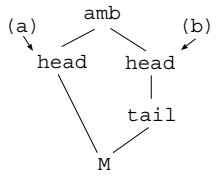


Fig. 3. The sharing structure of the term-graph of `amb`

3 Partial Sequential Realization of the Amb Operator

As we showed in the previous section, we can express the two-head stream access of an IM2-machine with the amb operator. However, in order to implement this operator, we need to execute two threads for both arguments in parallel. Parallel execution is a heavy mechanism, which is not easy to implement.

When we are implementing the two-head stream access of an IM2-machine, we always use the amb operator in the form

`amb a b where a:b:x =` M`.`

Here, M represents a "producer process" which makes the two-head output access to an $1\bot$-stream. The point is that the calculation of the two arguments of `amb` share the same redex M and therefore we do not need to compute them in parallel. If a functional language is implemented based on graph reduction [7], the above term is represented as shown in Figure 3. Here, an application node is labelled with a combinator name when it is an application of a combinator, for simplicity. In this way, the two term-graphs representing the two arguments share the same subgraph as a redex. Therefore, if we can reduce this term using this sharing structure, it is expected that we can implement the partial amb operator we need for $1\bot$-stream access in the realm of sequential computation. As such an operator, we introduce `gamb`, which has the type

```
gamb:: Bool -> Bool -> Amb Bool.
```

We explain how `gamb` works with an example of the reduction of `gtos(stog [0,0..])`, where the program `gtos` is modified so that it uses the `gamb` operator instead of `amb`. From the definition of `gtos`, the evaluation of `gamb a b` in the definition of `gtos` will produce the term-graph Figure 4(A). The arguments `a b` of `gtos` are marked with (a) and (b), respectively. In this graph, there is only one redex node `stog`, which is shared by both of the arguments. Therefore, it is evaluated and we have the term-graph (B). It has three redexes, two of them are put the marks (*) and (**) and the other one is stog. If we use the leftmost outermost reduction strategy as usual functional languages do, the redex (*) is reduced and then the redex `stog` is reduced. After that, it starts a non-terminating computation of the node (a).

However, since the term-graph `stog` is the producer process of an $1\bot$-stream, it is reduced to one of the four forms $0 : M'$, $1 : M'$, $x : 0 : M'$, and $x : 1 : M'$ as we noted in Section 2.3. Therefore, we can obtain one of the normal forms

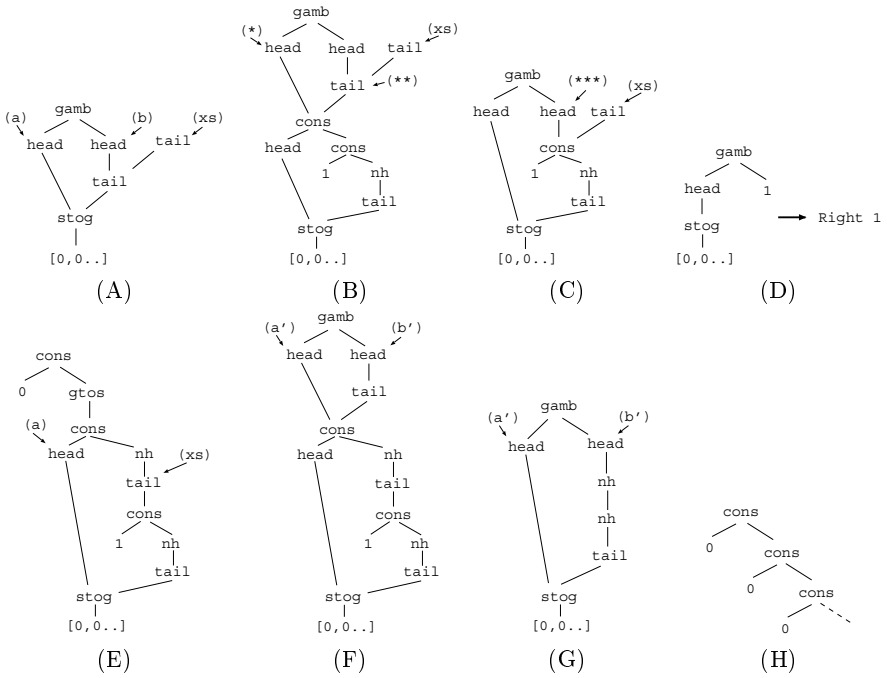


Fig. 4. The evaluation of `gtos(stog([0,0..]))`

of (a) and (b) by applying only the reduction rules `head` $(B : L) \to B$ and `tail` $(B : L) \to L$. Thus, we consider the reduction strategy of `gamb` MN to reduce both M and N with these two rules as long as they are applicable. Therefore, the redex (**) is reduced before `stog`, and we obtain the term-graph (C). Then, the redex (***) is reduced and we have (D). In this way, the argument (b) is reduced to a normal form 1 and `gamb` will return the value `Right` 1. As the result, the whole program `gtos(stog [0,0..])` is reduced to the term-graph Figure 4(E), which is a head normal form.

Next, consider the reduction of `gtos` in (E), which is a bit more complicated. Note that subgraphs of (xs) are reduced through the reduction of (**) in (B) because of the sharing, and we can use this simplified graph from the beginning. As we noted in Section 2.3 and 2.4, a function to output (input) a $1\bot$-stream can modify the result (arguments) of the recursive call by inserting `not` and `nh`. Correspondingly, the argument-subgraphs (a') and (b') of the term-graph (F) have nodes with labels `not` and `nh`, above their shared redex `stog`. Thus, we modify the reduction strategy of `amb` MN we mentioned above, so that M and N are reduced with the following reduction rules (I).

Reduction rules (I)

`head` $(B : L)$	$\to B$	(R-head)
`tail` $(B : L)$	$\to L$	(R-tail)
`not` 0	$\to 1$	(R-not0)
`not` 1	$\to 0$	(R-not1)
`nh` $(B : L)$	$\to$ `not` $B : L$	(R-nh)

This is the minimum list of rules which works. We had better also use the rules `nh nh` $L \to L$ and `not not` $B \to B$ in practice so that the term-graphs do not become significantly large.

Thus, the term-graph (F) is reduced to (G), and then, the two arguments of gamb share the same subgraph `stog`, and thus it is reduced and after that, reduction rules (I) become applicable, and thus it is also reduced to `Right 1`. In this way, if we continue the reduction, we obtain an infinite graph (H).

To summarize, the term-graph `gamb` M N is reduced as follows.
(1) Rules in (I) are applied to subgraphs of M and N which are reachable from the root through nodes labelled with `cons`, `head`, `tail`, `not`, and `nh` until no more rules are applicable. Rules are applied in some fixed order; in our implementation, the leftmost outermost reduction order on M, and then on N.
(2) If one of them become a normal form (that is, 0 or 1), then `gamb` returns the corresponding value. For example, if M is reduced to 0, `gamb` M N is reduced to `Left` 0. If both are normal forms, `gamb` returns the left value.
(3) Compare the leftmost outermost redexes of M and N. If they are not identical, then raise a runtime error.
(4) Reduce the shared redex in (3) to a weak head normal form.
(5) Repeat (1) to (4) until it returns in (2) or it raises an error in (3).

We need the repetition (5) because the shared redex can be reduced to a shared redex again by rules in (I). Rules in (I) are graph reductions implemented as follows. When (R-not0) or (R-not1) is applied, the `not` node is simply overwritten with 1 or 0. It is also the case for the (R-nh) rule; the `nh` node is overwritten with a new `not` node. However, in (R-head) and (R-tail) rules, we cannot overwrite the `head` node with the node B because the node B already exists. Instead, we overwrite `head` with a new indirection node which points to B. This is actually the way term-graphs are reduced in graph-based functional language implementations [9, 7, 6]. We omit an indirection node in the figures.

The `gamb` operator is a partial realization of `amb`. That is, (1) when `gamb` M N is reduced to L, `amb` M N can also be reduced to L, and (2) when the reduction of `gamb` M N does not terminate, the reduction of `amb` M N does not terminate.

4 A Term-graph Calculus of $1\bot$-streams

In the previous section, we defined the reduction of `gamb` MN for arbitrary terms M and N of type **Bool**, and therefore `gamb` MN may cause a runtime error in (3) depending on the ways graph implementations of M and N share a subgraph. In this section, we define a term-graph calculus $\mathrm{GPCF}^{\omega}_{\bot,1}$ which has the type of $1\bot$-streams as a subtype of `[Bool]`, and show that such a runtime error does not happen when `gamb` is used for $1\bot$-stream access.

We start with defining a term-graph. Let Γ be a set of labels with arity in $\mathbb{N}$. A term-graph over a signature Γ is a quadruple $g = \langle N, symb, args, r\rangle$ such that (1) N is a finite set of nodes, (2) $symb : N \to \Gamma$ is a function which assigns a label to a node, (3) $args : N \to N^*$ is the list of successor nodes such that $length(args(n)) = arity(symb(n))$, (4) $r \in N$ is the root of g, and (5) g is acyclic as a graph. Note that we only consider acyclic term-graphs in this paper.

$\mathbf{GPCF}^{\omega}_{\perp,1}$

Types: $\sigma, \tau \quad ::= \mathbf{Bool} \mid \mathbf{Stream} \mid \sigma \to \tau \mid \mathbf{AmbBool}$
Variables(of type τ): $x^\tau,\ y^\tau,\ z^\tau$
Term-Graphs: $B, L, M, N ::= 0 \mid 1 \mid \mathtt{head}\ L \mid \mathtt{tail}\ L \mid B : L \mid \mathtt{not}\ B \mid \mathtt{nh}\ B$
$\mid \lambda x^\tau.M \mid M\ N \mid \mu x^\tau.M \mid \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N$
$\mid \mathtt{gambr}\ L \mid \mathtt{Left}\ B \mid \mathtt{Right}\ B \mid M||N$

Typesystem : $x^\sigma :: \sigma \qquad 0 :: \mathbf{Bool} \qquad 1 :: \mathbf{Bool} \qquad \dfrac{B :: \mathbf{Bool}}{\mathtt{not}\ B :: \mathbf{Bool}}$

$$\frac{L :: \mathbf{Stream}}{\mathtt{head}\ L :: \mathbf{Bool}} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{tail}\ L :: \mathbf{Stream}} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{nh}\ L :: \mathbf{Stream}}$$

$$\frac{M :: \tau}{\lambda x^\sigma.M :: \sigma \to \tau} \qquad \frac{B :: \mathbf{Bool}, M :: \sigma, N :: \sigma}{\mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N :: \sigma} \qquad \frac{M :: \sigma \to \tau, N :: \sigma}{MN :: \tau} \quad \text{(T-tail)}$$

$$\frac{M :: \sigma}{\mu x^\sigma.M :: \sigma} \quad \text{(T-mu)} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{not}^n\ c : L\ ::\ \mathbf{Stream}} \quad (n \geq 0, c = 0 \text{ or } c = 1) \quad \text{(T-cons)}$$

$$\frac{L :: \mathbf{Stream}}{\mathtt{not}^l\ y : c_1 : \ldots : c_m : \mathtt{nh}^n\ x\ \ \mathtt{where}\ y : z_1 : \ldots : z_k : x = L\ ::\ \mathbf{Stream}}$$
$$(c_i = 0 \text{ or } c_i = 1 (i = 1, \ldots, m), l, m, n, k \geq 0) \quad \text{(T-b01)}$$

$$\frac{L :: \mathbf{Stream}}{\mathtt{gambr}\ L :: \mathbf{AmbBool}} \qquad \frac{L :: \mathbf{AmbBool},\quad M :: \mathbf{Bool} \to \sigma,\quad N :: \mathbf{Bool} \to \sigma}{(M||N)\ L :: \sigma}$$

$$\frac{B :: \mathbf{Bool}}{\mathtt{Left}\ B :: \mathbf{AmbBool}} \qquad \frac{B :: \mathbf{Bool}}{\mathtt{Right}\ B :: \mathbf{AmbBool}}$$

(I)-reduction: applying the following rules to subgraphs reachable from the root through nodes labelled with :, `head`, `tail`, `not`, and `nh` until no more rules are applicable.

$$\begin{array}{lll}
\mathtt{head}\ (B : L) & \to B & \text{(R-head)} \\
\mathtt{tail}\ (B : L) & \to L & \text{(R-tail)} \\
\mathtt{not}\ 0 & \to 1 & \text{(R-not0)} \\
\mathtt{not}\ 1 & \to 0 & \text{(R-not1)} \\
\mathtt{nh}\ (B : L) & \to \mathtt{not}\ B : L & \text{(R-nh)}
\end{array}$$

Reduction rules (II):

$$\begin{array}{lll}
(\lambda x^\tau.M)\ N & \to M\ \mathtt{where}\ x^\tau = N & \text{(R-app)} \\
\mu x^\tau.M & \to M\ \mathtt{where}\ x^\tau = \mu x^\tau.M & \text{(R-mu)} \\
\mathtt{if}\ 1\ \mathtt{then}\ M\ \mathtt{else}\ N & \to M & \text{(R-if0)} \\
\mathtt{if}\ 0\ \mathtt{then}\ M\ \mathtt{else}\ N & \to N & \text{(R-if1)} \\
f||g\ (\mathtt{Left}\ B) & \to f\ B & \text{(R-L)} \\
f||g\ (\mathtt{Right}\ B) & \to g\ B & \text{(R-R)}
\end{array}$$

Reduction rules (III):

$$\begin{array}{ll}
\mathtt{gambr}\ (0 : M) \to \mathtt{Left}\ 0 & \mathtt{gambr}\ (M : 0 : N) \to \mathtt{Right}\ 0 \\
\mathtt{gambr}\ (1 : M) \to \mathtt{Left}\ 1 & \mathtt{gambr}\ (M : 1 : N) \to \mathtt{Right}\ 1
\end{array}$$

Fig. 5. The term-graph calculus of 1⊥-streams $\mathrm{GPCF}^{\omega}_{\perp,1}$

GPCF$^{\omega}_{\perp}$ is a PCF-like term-graph calculus with **Stream** and **AmbBool** type (Figure 5). We consider the minimal set of types for our explanation and omit the integer type, for example. We consider typed variables to simplify the type system and let X^{τ} be a set of variables of type τ. The set Γ of labels (with arity) is defined as $\{0^{(0)}, 1^{(0)}, \mathtt{head}^{(1)}, \mathtt{tail}^{(1)}, :^{(2)}, \mathtt{not}^{(1)}, \mathtt{nh}^{(1)}, x^{\tau\,(0)}, \lambda x^{\tau\,(1)}, @^{(2)}$(application), $\mathtt{if\_then\_else\_}^{(3)}, \mu x^{\tau\,(1)}, \mathtt{gambr}^{(1)}, \mathtt{Left}^{(1)}, \mathtt{Right}^{(1)}, ||^{(2)}$(destructor for **AmbBool**)$\}$. Here, a variable x^{τ} is a member of X^{τ}. We have the condition that for each λx^{τ} and μx^{τ}, there exists at most one bounded variable node with label x^{τ}.

We sometimes omit the type τ when it is obvious from the context. Though `not` and `nh` are λ-expressible, we list them as primitives because we need special treatment. Note that the μ constructor does not produce a cyclic graph, and μx^{τ} is a label of a node for each x^{τ}.

When we express a term-graph in text, we use infix notation for :, ||, and @, and we omit the operator symbol @. We consider that multiple occurrences of the same bounded variable are expressing the same variable node.

In order to express sharing of nodes, we assign a variable to each node which has more than one parents, and use the notation M `where` $x^{\tau} = N$ for the graph M with the variable node x^{τ} substituted for N of type τ. We also use a notation with pattern matching for the **Stream** type; we write M `where` $y_1 : \ldots : y_n : x = N$ for M `where` $x = \mathtt{tail}\ z_n$ `where` $y_n = \mathtt{head}\ z_n$ $\ldots$ `where` $z_2 = \mathtt{tail}\ z_1$`where` $y_1 = \mathtt{head}\ z_1$`where` $z_1 = N$. It is close to the call-by-need calculus in [1], but they consider a term-calculus which simulates graph-based reduction, whereas the objects of the calculus are term-graphs themselves in our calculus. We also write $\mathtt{not}^n\ M$ for the n-times successive application of `not` to M.

In this type system, all the graphs are directed acyclic graphs, and therefore the type of a term-graph is uniquely defined inductively. Reduction rules are graph-reduction rules. Note that M `where` $x^{\tau} = N$, which is the right hand side of (R-app) is obtained by copying M so that edges pointing to x^{τ} are redirected to N. It is also the case for the (R-mu) rule. The set of reduction rules is composed of the (I)-reduction and rules in (II) and (III). The reduction rules are parallel to the evaluation rules of `gamb` in Section 3. We have defined the (I)-reduction in this form for two reasons. One is to provide the subject-reduction property, and the other one is to reduce B_2 to a (I)-normal form when $\mathtt{gambr}(B_1 : B_2 : L)$ is given. We consider leftmost outermost reduction order.

As operations to construct **Stream** term-graphs, we consider insertion of a constant to the head or next to the head of a **Stream** element. This is parallel to the constructor of ordinary streams to insert a constant to the head. We also allow the operation to remove the first or the second element from a **Stream** element to form a new $1\perp$-stream. The (T-cons) and (T-tail) rules are for the operations to the first element, and (T-b01) rule with $m = 1, k = 0$ (or $m = 0, k = 1$) is for the insertion (or removal, respectively) operation to the second element. The (T-b01) rule is applicable to terms-graphs obtained through successive applications of these operators. As for destructors, we use the `gambr` operator which corresponds to the following program.

```
gambr(M) = gamb a b where a:b:x = M.
```

Note that this is the way `gamb` is used to input from a $1\bot$-stream. By structural induction on M, we can prove the following.

Proposition 1. *Suppose that $M :: \sigma$ and M is reduced to N in GPCF$^{\omega}_{\bot,1}$. Then N is typable and $N :: \sigma$.*

Corollary 1. *Suppose that M is a term of type* **Stream** *and M is reduced to $L = N_1 : N_2$. Then, after applying (I)-reduction, one of the followings hold,*
(1) N_1 is a constant (i.e. 0 or 1),
(2) N_2 is $c : N_3$ for c a constant,
(3) L has the form $\texttt{not}^l\ y : \texttt{nh}^n\ x$ `where` $y : z_1 : \ldots : z_m : x = G$ *for $l, m, n \geq 0$.*

Corollary 1 shows that reduction of `gambr` M with the leftmost outermost reduction will produce a value (case (1) or (2)), or continue the reduction of G. For the latter case, when G is reduced to a cons cell, L is reduced by (I)-reduction to one of the forms (1), (2), and (3). Thus, the reduction of `gambr` M will produce a value `Left` c or `Right` c ($c = 0$ or 1) or it does not terminate.

Thus, we can say that runtime error does not occur for a typable program when `gamb` is added to a graph-reduction based lazy functional language with this kind of graph-representations. However, most implementations of Haskell use cyclic graphs for the representation of recursive structures. For this case, typing rules presented here do not have such good properties. We consider a variant GPCF$^{\omega,c}_{\bot,1}$ of GPCF$^{\omega}_{\bot,1}$ in which term-graphs are allowed to be cyclic, $\mu x^{\tau}.M$ is not a term-graph but a textual representation of a cyclic graph, and (T-mu) and (R-mu) do not exit. To see the difference, consider the term-graph K `=`μ`x. a:1:y where a:y = x` of type **Stream**. In GPCF$^{\omega}_{\bot,1}$, it is reduced to **Stream**-type terms of the form `a:1:1:1:...:1:y where a:y =` K. On the other hand, in GPCF$^{\omega,c}_{\bot,1}$, it is reduced by (I)-reduction to the term-graph `(`μ`x.x):(`μ`y.1:y)` which does not belong to **Stream** type. Here, μ`x.x` is an indirection node pointing itself. Roughly speaking, GPCF$^{\omega}_{\bot,1}$ step by step simulates two-head stream output of an IM2-machine, whereas cyclic graph representation enables us to make all the infinite outputs at a time and realizes the result of infinite-time computation, which our type system does not deal with. In the same way, the reduction of the term-graph `gambr(`μ`x. a:y where a:y = x)` of type **AmbBool** does not terminate in GPCF$^{\omega}_{\bot,1}$ whereas it is reduced to `gambr((`μ`a.a):(`μ`y.y))` by (I)-reduction and stops (i.e., causing a runtime error) in GPCF$^{\omega,c}_{\bot,1}$. Our implementation in the next section is close to GPCF$^{\omega,c}_{\bot,1}$ and has this behavior.

5 Implementation

We have implemented our `gamb` operator as an extension of the Hugs system, which is a graph-reduction based implementation of the Haskell language. First, we implemented it as an extension of Gofer ver2.30 which is an ancestor of the Hugs system. Because Gofer ver2.30 has a good documentation [6], in particular of the G-machine structure of the runtime system [7], it is not difficult to put a

hook on the `eval` operator of the G-machine of the Gofer system so that when it is a function application and the function is `gamb`, then reduce it as listed in Section 3. This implementation is available from the author's web page[18].

6 Some Algorithms with Gamb

As we explained in the introduction, the main application area of $1\perp$-stream programming is real number computation. We list two programs to compute real-functions over the unit interval [0,1] in [18]. One is the average function `pl` to compute $(x+y)/2$, and the other one is multiplication. They can also be expressed as $\mathrm{GPCF}^{\omega}_{\perp,1}$-terms of type $\mathbf{Stream} \to \mathbf{Stream} \to \mathbf{Stream}$.

We can also apply higher-order functions "map" and "foldr1" to real functions like `pl`. Therefore, for example, we can write and execute

```
sum a = gtos (foldr1 pl (map stog (map (code a))))
```

which calculates the sum of the elements of the finite list `a`. Here, the `code` function maps a real number with decimal representation to the binary representation. This time again, it can also be expressed in $\mathrm{GPCF}^{\omega}_{\perp,1}$.

7 Conclusion

We extended the notion of a stream to a stream with at most one bottom and implemented, as an extension of Haskell, input/output of such extended streams. This mechanism can be used for real number computation because the set of real numbers is topologically embedded in $\Sigma^{\omega}_{\perp,1}$.

We defined a datatype corresponding to a $1\perp$-stream as a subtype of the infinite list type `[Bool]`. There is another way of implementing computation over $1\perp$-streams. That is, to assign a name to each constructor and represent a $1\perp$-stream as an ordinary stream. However, because of the several ways of constructing the same $1\perp$-stream, such representation is not canonical. From the authors experience, the existence of multiple-representation complicates $1\perp$-stream programs. In addition, if we accept this approach, we need to consider the relation between the denotation as a $1\perp$-stream and its representation as an ordinary stream. Since a $1\perp$-stream itself is directly expressible in a programming language, the author thinks it natural to try to write a program which directly manipulates them, as we did in this paper.

We have two goals in this study of $1\perp$-stream calculi. One is to actually implement it and write and execute real-number programs. The other one is to study the computational structure of $1\perp$-streams and relate it to that of real numbers. One observation here is that, the nondeterminism and multi-valuedness of functions over $1\perp$-streams appear not because we perform parallel computation but because we access the *intensional* information how the arguments are represented as term-graphs. We need to investigate it with non-sequentiality feature of real number computation studied in [3].

In this paper, we have only presented the operational side of $\mathrm{GPCF}^{\omega}_{\perp,1}$. It is expected that, through the investigation of the denotational side of $\mathrm{GPCF}^{\omega}_{\perp,1}$, we

can study the structure of $1\perp$-streams from many aspects, including algebraic, domain-theoretic, and category-theoretic point of view. The author is interested in applying the semantics of a sequential nondeterministic language in [10] to our language. It is left as a future work.

Acknowledgments The authors thanks Martin Escardo for discussions about the use of amb operator for Gray-code computation. The first author was supported in part by Kayamori Foundation of Informatical Science Advancement.

References

1. Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky and Philip Wadler. A Call-by-Need Lambda Calculus, in *Proc. POPL '95, 22'nd Annual Symposium on Principles of Programming Languages, San Francisco, California*, 233-246, 1995.
2. A. Du Bois, R. Pointon, H.-W. Loidl, and P. Trinder. Implementing Declarative Parallel Bottom-Avoiding Choice. in *Proc. 14th Symposium on Computer Architecture and High Performance Computing*, 2002.
3. Martín Hötzel Escardó, Martin Hofmann, and Thomas Streicher. On the non-sequential nature of the interval-domain model of exact real-number computation. *Mathematical Structures in Computer Science*, to appear.
4. Pietro Di Gianantonio. An Abstract Data Type for Real Numbers. *Theoretical Computer Science*, 221:295–326, 1999.
5. F. Gray. Pulse code communications. U. S. Patent 2632058, March 1953.
6. Mark P. Jones. The implementation of the Gofer functional programming system. *Research Report YALEU/DCS/RR-1030*, Yale University, USA, 1994.
7. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
8. Simon Peyton Jones, Editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
9. J.R. Kennaway, J.W. Klop, M.R. Sleep and F.J. de Vries. An Introduction to Term Graph Rewriting. in *[14]*
10. J. Raymundo Marcial-Romero and Martín Hötzel Escardó. Semantics of a Sequential Language for Exact Real-Number Computation. in *Proceedings of the Annual IEEE Symbposium on Logic in Computer Science*. 426–435, 2004.
11. John McCarthy. A Basis for a Mathematical Theory of Computation. in P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, 33–70, North-Holland. 1963.
12. John Hughes and Andrew Moran. Making Choices Lazily. in *Conference Record of FPCA '95*, 108–119,ACM Press, 1995.
13. Kristoffer H. Rose Graph-based Operational Semantics of a Lazy Functional Language. in *[14]*
14. Ronan Sleep, Rinus Plasmeijer and Marko van Eekelen, Editors. *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.
15. Hideki Tsuiki. Computational dimension of topological spaces. In *Computability and Complexity in Analysis*, LNCS 2064, 323–335, Springer,2001.
16. Hideki Tsuiki. Real number computation through gray code embedding. *Theoretical Computer Science*, 284(2):467–485, 2002.
17. Hideki Tsuiki. Real Number Computation with Committed Choice Logic Programming. *Journal of Logic and Algebraic Programming*, to appear, 2004.
18. `http://www.i.h.kyoto-u.ac.jp/~tsuiki/bot_haskell`
19. Klaus Weihrauch. *Computable analysis, an Introduction*. Springer-Verlag, 2000.