Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)

[Electronic Notes in Theoretical Computer Science 298 (2013) 383–402](http://dx.doi.org/10.1016/j.entcs.2013.09.023)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

A Stream Calculus of Bottomed Sequences for Real Number Computation

# Kei Terayama1 Hideki Tsuiki2

*Graduate School of Human and Environmental Studies Kyoto University*

*Kyoto, Japan*

Abstract

A calculus XPCF of 1⊥-sequences, which are infinite sequences of {0*,* 1*,* ⊥} with at most one copy of bottom, is proposed and investigated. It has applications in real number computation in that the unit interval I is

topologically embedded in the set Σ*ω* of 1⊥-sequences and a real function on I can be written as a program

⊥*,*1

which inputs and outputs 1⊥-sequences. In XPCF, one defines a function on Σ*ω*

⊥*,*1

only by specifying its

behaviors for the cases that the first digit is 0 and 1. Then, its value for a sequence starting with a bottom is calculated by taking the meet of the values for the sequences obtained by filling the bottom with 0 and 1. The validity of the reduction rule of this calculus is justified by the adequacy theorem to a domain-theoretic semantics. Some example programs including addition and multiplication are shown. Expressive powers of XPCF and related languages are also investigated.

*Keywords:* Bottom, stream, real number computation, domain model, PCF, adequacy, parallel or

# Introduction

Streams are a useful data structure used for expressing infinite sequences and one can implement real number computation with streams through signed digit expansion[1,2] or other expansions of real numbers[6]. However, since a stream can only be accessed one-way from left to right, if there is a bottom, i.e., a term whose evaluation does not terminate, in a stream, then a program get stuck when it tries to read in the value of the bottom cell and cannot input the rest of the sequence though it may contain valuable data.

Usually, a bottom is considered as a kind of programming error which should be avoided in a correct program. However, it is known that infinite sequences which may contain bottoms are useful in representing continuous topological spaces like

1 Email: [terayama@i.h.kyoto-u.ac.jp](mailto:myuserid@mydept.myinst.myedu)

2 Email: [tsuiki@i.h.kyoto-u.ac.jp](mailto:couserid@codept.coinst.coedu)

1571-0661 © 2013 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

<http://dx.doi.org/10.1016/j.entcs.2013.09.023>

R. Here, we call an infinite sequence of Σ∪{⊥} which may contain at most one copy of bottom a 1⊥-sequence. It is shown in [8] and [15] that R and I = [0*,* 1] can be

topologically embedded in the space Σ*ω*

⊥*,*1

of 1⊥-sequences of Σ for Σ = {0*,* 1} and

this embedding is called the Gray embedding in [15]. The signed-digit expansion and other admissible representations of R turn out to be redundant in the sense that infinitely many reals each satisfy the property of being represented by infinitely many codes[4,17]. On the other hand, with the Gray embedding, a unique code can be assigned to each real number by extending the code space with at most one copy of ⊥. This embedding result is extended in [16] to other topological spaces and it is shown that any *n*-dimensional separable metric space can be topologically

embedded in the space Σ*ω*

⊥*,n*

of *n*⊥-sequences.

[8] expressed a 1⊥-sequence as a function from N⊥ to {−1*,* 1*,* ⊥} and used the parallel if operator pif to access 1⊥-sequences and showed that real number algo- rithms can be expressed in PCF + pif. In order to evaluate pif *L M N* , one need to evaluate *L*, *M* , and *N* in parallel. Therefore, pif operator causes explosion of par- allel computations and it seems difficult to implement it efficiently. Martin Escard´o proposed Real PCF[6] which is an extension of PCF with real numbers. It is based on interval domains and a kind of parallel conditional operator is used.

On the other hand, [15] restricted the number of ⊥ to one and introduced an IM2-machine (Indeterministic Multiheads Type2 Machine) which enables extended stream access to 1⊥-sequences. However, the behavior of an IM2-machine needs to be specified through a set of overlapping rules and therefore functions expressible with IM2-machines are multi-valued functions in general. Moreover, a program of an IM2-machine is complicated because one needs to express its behaviors for inputs from extra heads.

In this paper, we introduce a calculus XPCF of 1⊥-sequences with which one can express extended stream accesses to them. It is an extension of PCF with a datatype S of 1⊥-sequences and is based on the algebraic domain BD of 1⊥-sequences[16]. The datatype S has, in addition to the constructors 0 : S → S and 1 : S → S to prepend a digit to a sequence, constructors 0 : S → S and 1 : S → S to insert a digit as the second element of a sequence. However, a function on S is defined by expressing its behaviors only for cases the argument has the form 0*N* and 1*N* with

the expression ⟨0*x* →*M*0; 1*x* →*M*1⟩. It means a function on Σ*ω*

⊥*,*1

to apply [[*λx.M*0]]

to *s* if the argument is 0*s*, to apply [[*λx.M*1]] to *s* if the argument is 1*s*, and apply both of them to *s* and take the meet of the results if the argument is ⊥*s*. XPCF can be considered as an algebraic domain variant of Real PCF. This calculus has the computational adequacy property with respect to its domain-theoretic model.

We give some example programs of XPCF including addition and multiplication on I through the Gray embedding. We also studied the expressive power of this language and showed that XPCF has the same expressive power as PCF + pif on types which do not contain S, that all computable elements of BD are expressible on type S, and that if we extend XPCF with the ∃ operator, then all the computable elements in the semantic domains are expressible.

As [7] showed, any real number calculus which is adequate to the interval do-

5 5

4 4

3 3

2 2

1 1

0.0 0.5 1.0

The Binary expansion

0.0 0.5 1.0

The Gray expansion

Fig. 1. The Binary and Gray expansion of I

main model and in which the average function can be represented, does not have a sequential reduction strategy. Their proof also applies to our model with some modifications and thus any sequential reduction strategy of this calculus is not ad- equate. We designed a sequential reduction strategy of XPCF and implemented it with Haskell. Though it is not adequate and some of the terms cannot be reduced to their denotations, it sequentially evaluates many of the terms like addition and multiplication.

In the next section, we start with explaining the Gray embedding of I in Σ*ω*

⊥*,*1

and

the domain BD of 1⊥-sequences. In Section 3, we define the syntax and semantics

of XPCF and, in Section 4, we show how real functions can be expressed in XPCF with some program examples. Then, we give reduction rules of XPCF in Section 5 and show the adequacy property in Section 6. In section 7, we study expressive powers of XPCF.

Notations: Recall that we fix Σ = {0*,* 1}. We denote by Γ∗ the set of finite sequences of a character set Γ and by Γ*ω* the set of infinite sequences of Γ. We define Γ∞ = Γ∗ ∪ Γ*ω*, which is a Scott domain, i.e., a bounded complete *ω*-algebraic

dcpo. Let Σ⊥ = Σ ∪ {⊥}, and Σ*ω*

⊥

be the set of infinite sequences of Σ⊥. Σ⊥ has

the order generated by ⊥ ⊑ 0 and ⊥ ⊑ 1. On Σ*ω* , we define the order ⊑ as *s* ⊑ *t*

⊥

if *s*(*n*) ⊑ *t*(*n*) for every *n*. (Σ*ω ,* ⊑) is a Scott domain. We define Σ*ω*

= {*s* ∈

⊥

*ω* | *s* contains at most one ⊥}.

Σ

⊥

⊥*,*1

# Real number computation and 1⊥-sequences

* 1. *Gray embedding*

The Gray expansion is an expansion of I as infinite sequences of Σ which is different from the ordinary binary expansion [15]. It is based on Gray code[10], which is a coding of natural numbers with Σ different from the binary code. Figure 1 shows the binary and Gray expansion of I. In the binary expansion of *x*, the head *h* of the expansion indicates whether *x* is in [0, 1/2] or in [1/2, 1] and the tail is the expansion of *f* (*x, h*) for *f* the function defined as

*f* (*x, h*) =  2*x* (if *h* = 0) *.*

 2*x* − 1 (if *h* = 1)

Thus, with the binary expansion, the tail of the expansion of 1*/*2 depends on the choice of the head character *h* and 1/2 has two expansions 1000... and 0111 On

the other hand, the head of the Gray 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*) =  2*x* (0 ≤ *x* ≤ 1*/*2) *.*

 2(1 − *x*) (1*/*2 *< x* ≤ 1)

Note that *t*(*x*) is continuous on *x* = 1*/*2 and therefore the tail of the expansion does not depend on the choice of the first digit. Actually, the two expansions of 1*/*2 are 01000... and 11000 which coincide from the second character. It means that

the value is half not depending on the first character. Therefore, we leave the first character undefined (⊥) and define a new expansion of 1/2 as ⊥1000 It is also

the case for expansions of dyadic numbers (rational numbers of the form *m/*2*k*) and therefore we assign codes of the form *p*⊥1000 *. . .* for *p* ∈ {0*,* 1}∗ to those numbers.

In this way, we have a mapping *ϕ* : I → Σ*ω*

⊥*,*1

called the Gray-embedding as follows.

Definition 2.1 ([15]) Let *P* : I → Σ⊥ be the map

*P* (*x*) =

⊥ (*x* = 1*/*2) *.*

 1 (*x >* 1*/*2)

 0 (*x <* 1*/*2)

*the Gray embedding ϕ* is a function from I to Σ*ω* defined as *ϕ*(*x*)(*n*) = *P* (*tn*(*x*))

⊥*,*1

(*n* = 0*,* 1*, . . .*).

An embedding of R in {−1*,* 1}*ω*

⊥*,*1

is defined in [8] independently by Gianantonio,

and the Gray embedding is essentially the same as the restriction of his embedding

to I. We call the 1⊥-sequence *ϕ*(*x*) the modified Gray expansion of *x*. The Gray

embedding *ϕ* is actually a topological embedding with the topology of Σ*ω*

⊥*,*1

the

subspace topology of the Scott topology of Σ*ω* .

⊥

* 1. *Domains of 1*⊥*-sequences*

We explain the domain BD of 1⊥-sequences [16]. Let Σ∗

⊥*,*1

be the set of finite 1⊥-

sequences of Σ. Here, *p* ∈ Σ⊥∗ is a finite 1⊥-sequence of Σ if ⊥ appears at most

once in *p* and ⊥ is not the final character of *p*. We have Σ∗

⊥*,*1

= {*ǫ,* 0*,* 1*,* ⊥0*,* ⊥1*, ...*}

with *ǫ* the empty sequence. We can regard Σ∗

⊥*,*1

as a subset of Σ*ω*

by identifying

*p* ∈ Σ∗ with *p*⊥*ω* ∈ Σ*ω* . We define BD = Σ∗

⊥

∪ Σ*ω*

, which is a Scott subdomain

⊥*,*1 ⊥

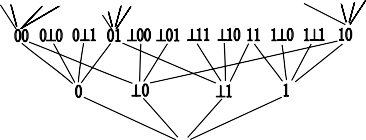
⊥*,*1

⊥*,*1

of Σ*ω* with the least element ⊥BD = *ǫ* as Figure 2 shows. For *c* ∈ Σ, we also denote

⊥

㼧••••••••••••••••••••••••••••••••••



••• ••••••••••••••

㻜㻜 㻜䎹㻜 㻜䎹㻝 㻜㻝 䎹㻜㻜 䎹㻜㻝 䎹㻝㻝 䎹㻝㻜 㻝㻝 㻝䎹㻜 㻝䎹㻝 㻝㻜㻜 䎹㻜 䎹㻝 㻝

䃔

䃨

䌥

䎹㻘㻝㼧



䃨

㻜

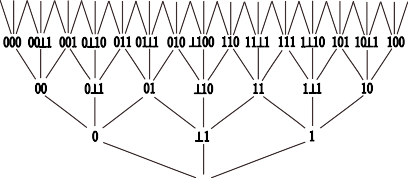
䃨 䃨

䃨

㻜㻝㻜 㻝㻝㻜

㻝㻜

䎹㻝㻜䃨



㻜㻜㻜 㻜㻜䎹㻝 㻜㻜㻝 㻜䎹㻝㻜 㻜㻝㻝 㻜㻝䎹㻝 㻜㻝㻜䎹㻝㻜㻜 㻝㻝㻜 㻝㻝䎹㻝 㻝㻝㻝 㻝䎹㻝㻜 㻝㻜㻝 㻝㻜䎹㻝 㻝㻜㻜

㻜㻜 㻜䎹㻝 㻜㻝 䎹㻝㻜 㻝㻝 㻝䎹㻝 㻝㻜

㻜

䎹㻝

㻝

䃔

ᵪᵰᵢ 㼧

*ω*



㻜㻜㻝㻜*ω* 㻜㻝㻝㻜*ω* 㻜㻝㻜*ω* 㻝㻝㻜*ω*

㻜*ω*

㻜䎹㻝㻜*ω*

*ω*

䎹㻝㻜

㻝䎹

㻝㻜

*ω*

䠍㻜

*••••••••••••••••••••••••••••••••••••••*

ᵠᵢ ᵰᵢ

䌥

䠆

䎹㻘㻝

Fig. 2. The domain BD Fig. 3. The domain RD

by *c* the continuous function from BD to BD to prepend *c* and denote by *c* the continuous function from BD to BD to insert *c* as the second character, where *c*(*ǫ*) is defined as ⊥*c*. We have the equation *b* ◦ *c* = *c* ◦ *b* for *b, c* ∈ Σ.

We regard that each finite sequence *s* = *d*0*d*1 *. . . dn*−1 of {0*,* 1*,* 0*,* 1} represents

the element *d*0(*d*1(*. . .* (*dn*−1(*ǫ*)))) of Σ∗

⊥*,*1

and each infinite sequence *s* = *d*0*d*1 *. . .* of

{0*,* 1*,* 0*,* 1} represents the limit of the infinite increasing sequence (*sn*)*n*=0*,*1*,...* in BD

for *sn* = *d*0(*d*1(*. . .* (*dn*−1(*ǫ*)))). Note that this limit exists in Σ*ω*

⊥*,*1

. In particular,

the sequence *b*0*b*1 *. . . bm*−1*c*0*c*1 *. . . cn*−1 represents *b*0*b*1 *. . . bm*−1⊥*c*0*c*1 *. . . cn*−1 ∈ Σ∗ (or *b*0*b*1 *. . . bm*−1 if *n* = 0), and the infinite sequence *b*0*b*1 *. . . bm*−1*c*0*c*1 *. . .* represents

⊥*,*1

*b*0*b*1 *. . . bm*−1⊥*c*0*c*1 *. . .* ∈ Σ*ω* .

⊥*,*1

Since BD is a Scott domain, the meet (i.e., the greatest lower bound) exists for any subset of BD. We show it explicitly, because it plays an important role in the semantics of XPCF. First, the meet on Σ⊥ = {0*,* 1*,* ⊥} is obviously defined. It is

naturally extended to the meet *s* ⊓Σ*ω t* in Σ*ω* as (*s* ⊓Σ*ω t*)(*n*) = *s*(*n*) ⊓ *t*(*n*). Let

⊥ ⊥ ⊥

trunc be the function from Σ*ω*

⊥

to BD to truncate the sequence after the second ⊥

to form a finite 1⊥-sequence if it contains more than one copies of ⊥, and returns itself if it does not.

Proposition 2.2 *The meet s* ⊓ *t of s, t* ∈ BD *is equal to* trunc(*s* ⊓Σ*ω t*)*.*

⊥

We define the subdomain RD of BD which is used for expressing I through the Gray representation. We define

RD = {*p*⊥10*n* : *p* ∈ Σ∗*, n* ∈ {0*,* 1*, . . . , ω*}} ∪ Σ∞*.*

It is a Scott domain. Let LRD be the subset {*p*⊥10*ω* : *p* ∈ Σ∗} ∪ Σ*ω* of Σ*ω*

⊥*,*1

. LRD

is the set of limit (i.e., non-compact) elements of RD as Figure 3 shows. LRD consists of *ϕ*(I) and those sequences obtained by filling a bottom of *s* ∈ *ϕ*(I) with 0 and 1. One can see that I is a retract of LRD and I is homeomorphic to the set of minimal elements of LRD with the retract map *δ* : LRD → I defined as *δ*(*s*) = *x* if *ϕ*(*x*) ⊑ *s*. One can see that the triple (RD*,* LRD*, δ*) is a retract domain representation of I in the sense of [3] and we call the map *δ* : LRD → I the Gray representation.

We can consider two codings of I based on the Gray embedding. The first one is obtained by identifying *x* with *ϕ*(*x*) through the embedding and the other one is the Gray representation *δ*. For example, for 1*/*2 ∈ I, ⊥10*ω* is the unique codes with *ϕ*. On the other hand, there are three codes ⊥10*ω,* 010*ω* and 110*ω* for 1*/*2 with respect to *δ*. Based on these codings, we have two notions that a function on BD realize a function on I.

Definition 2.3 Let *F* be a function from BD*n* to BD and *f* be a (partial) real function from I*n* to I.

1. *F exactly realizes f* if, for every (*x*1*, . . . , xn*) ∈ dom(*f* ),

*F* (*ϕ*(*x*1)*, . . . , ϕ*(*xn*)) = *ϕ*(*f* (*x*1*, . . . , xn*))*.*

1. *F realizes f* if, for every (*p*1*, . . . , pn*) ∈ (*δn*)−1(dom(*f* )),

*δ*(*F* (*p*1*, . . . , pn*)) = *f* (*δ*(*p*1)*, . . . , δ*(*pn*))*.*

# Syntax and denotational semantics of XPCF

Throughout this paper, we write types and constants of syntactic entities in Sans- serif font and program names and the names of semantic domains in Bold font.

* 1. *PCF*

We review the syntax, the semantics, and the reduction rules of the language PCF in Table 1. See [11] or some textbooks like [14] for the details of PCF. PCF has ground types B for boolean values and N for integers. For a term *M* , *FV* (*M* ) denotes the free variables of *M* and *M* is *closed* if *FV* (*M* ) is empty. A program is a closed term of a ground type. An *environment ρ* is a type-respecting map from the set of variables to {*Dσ*|*σ* type} and, for *a* ∈ *Dσ*, *ρ*[*a/xσ*] is the environment which maps *xσ* to *a* and any other variable *yσ* to *ρ*(*yσ*). If *M* is a closed term, then [[*M* ]](*ρ*) does not depend on *ρ* and we write [[*M* ]] for [[*M* ]](*ρ*).

S

The operational semantics of PCF is given by the *immediate reduction relation*

in Table 1. The result of evaluation of a program *M* is a constant *c* defined as

EvalPCF(*M* ) = *c* iff *M ⊲*∗ *c.*

The following theorem is often referred to as the *Adequacy Property* of PCF. It asserts that the operational and denotational semantics coincide.

Theorem 3.1 ([11, Theorem 3.1]) *For any PCF program M and constant c,*

EvalPCF(*M* ) = *c* iff [[*M* ]] = [[*c*]]*.*

* 1. *Syntax and semantics of XPCF*

The syntax and denotational semantics of XPCF is listed in Table 2. We list only the differences compared with the PCF specification. It has a ground type S such

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | |
| Syntax of PCF |
| Types: *σ* ::= B | N | *σ* → *σ*  Variables(of type *σ*): *xσ* ::= *xσ, yσ, zσ, ...*  Constants: *c* ::= tt*,* ff*,* if*σ,* Y*τ ,* k*n,* inc*,* dec*,* zero (*σ*:ground type, *τ* :type, *n* ∈ N)  Terms: *M* ::= *xσ* | *c* | (*MM* ) | (*λxσ .M* )  Typing Rules:  *xσ* : *σ* tt : B ff : B k*n* : N inc : N → N  dec : N → N zero : N → B if*σ* : B → *σ* → *σ* → *σ*  *M* : *τ M* : *σ* → *τ N* : *σ*  Y*σ* : (*σ* → *σ*) → *σ λxσ.M* : *σ* → *τ MN* : *τ* | | | |
| Denotational semantics of PCF | | |  |
| Domains: *D*B :the flat domain {⊥B*, tt, ff* } of truth values.  *D*N :the flat domain {⊥N*,* 0*,* 1*, ...*} of natural numbers.  *Dσ*→*τ* :the domain [*Dσ* → *Dτ* ] of continuous functions from *Dσ* to *Dτ* , with the least element denoted by ⊥*σ*→*τ* .  Interpretation of constants:  [[tt]] = *tt* [[ff]] = *ff* [[k*n*]] = *n*  [[inc]] = *λn* ∈ *D*N*.* (*n* + 1 (*n* /= ⊥N) [[dec]] = *λn* ∈ *D*N*.* (*n* − 1 (*n* ≥ 1)  ⊥N (*n* = ⊥N) ⊥N (*n* ∈ {⊥N*,* 0})  *tt* (*n* = 0) .  [[zero]] = *λn* ∈ *D*N*. ff* (*n >* 0) [[Y*σ*]] = *λF* ∈ *Dσ*→*σ. n*∈N *Fn*(⊥*σ*)  ⊥B (*n* = ⊥N)  *x* (*b* = *tt*) [[if*σ*]] = *λb* ∈ *D*B*.λx* ∈ *Dσ.λy* ∈ *Dσ.* *y* (*b* = *ff* )  ⊥*σ* (*b* = ⊥B)  Denotational semantics:  (i) [[*xσ*]](*ρ*) = *ρ*(*xσ*) (ii) [[*c*]](*ρ*) = [[*c*]]  (iii) [[*M N* ]](*ρ*) = [[*M* ]](*ρ*)([[*N* ]](*ρ*)) (iv) [[*λxσ.M* ]](*ρ*) = *λa* ∈ *Dσ.*[[*M* ]](*ρ*[*a/xσ*]) | | | |
| Operational semantics of PCF | |  | |
| Reduction rules:  (*λxσ.M* )*N ⊲ M* [*N/xσ*] Y*σ M ⊲ M* (Y*σ M* ) inc k*n ⊲* k*n*+1 dec k*n*+1 *⊲* k*n* zero k0 *⊲* tt zero k*n*+1 *⊲* ff if*σ* tt *M N ⊲ M* if*σ* ff *M N ⊲ N*  *M ⊲ M* ′ *N ⊲ N* ′ (if *M* is if *,* inc*,* dec or zero)  *M N ⊲ M* ′ *N M N ⊲ M N* ′ *σ* | | | |

Table 1

Syntax and semantics of PCF

Syntax of XPCF Syntax of PCF extended with the followings.

Types: S

Constants: 0*,* 1*,* 0*,* 1

Terms: ⟨0*x*S →*M* ; 1*x*S →*M* ⟩ | ⟨⟨0*x*S →*M* ; 1*x*S →*M* ⟩⟩

Typing Rules:

0 : S → S 1 : S → S 0 : S → S 1 : S → S

*M*0 : *σ M*1 : *σ*

⟨0*x*S →*M*0; 1*x*S →*M*1⟩ : S → *σ*

Denotational semantics of XPCF

*M*0 : *σ M*1 : *σ*

⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩ : S → *σ*

the followings.

Domains: *D*S = BD Interpretation of constants:

Semantics of PCF extended with

[[0]] = 0 ∈ *D*S→S (where 0(*s*) = 0*s* for *s* ∈ *D*S)

[[1]] = 1 ∈ *D*S→S (where 1(*s*) = 1*s* for *s* ∈ *D*S)

[[0]] = 0 ∈ *D*S→S (where 0(*as*) = *a*0*s* for *a* ∈ Σ⊥ and *s* ∈ *D*S, 0(*ǫ*) = ⊥0) [[1]] = 1 ∈ *D*S→S (where 1(*as*) = *a*1*s* for *a* ∈ Σ⊥ and *s* ∈ *D*S, 1(*ǫ*) = ⊥1)

Denotational semantics:

* + 1. [[⟨0*x*S →*M*0; 1*x*S →*M*1⟩]](*ρ*) =

⊥*σ* (if *s* = *ǫ*)



[[*M*0]](*ρ*[*s*′*/x*S]) (if *s* = 0*s*′)

*λs* ∈ *D*S*.*

[[*M*1]](*ρ*[*s*′*/x*S





]) (if *s* = 1*s*′)

[[*M*0]](*ρ*[*s*′*/x*S]) ⊓ [[*M*1]](*ρ*[*s*′*/x*S]) (if *s* = ⊥*s*′)

* + 1. [[⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩]](*ρ*) =



[[*M*0]](*ρ*[*ǫ/x*S]) ⊓ [[*M*1]](*ρ*[*ǫ/x*S]) (if *s* = *ǫ*)

[[*M*0]](*ρ*[*s*′*/x*S]) (if *s* = 0*s*′)

*λs* ∈ *D*S*.*

[[*M*1]](*ρ*[*s*′*/x*S





]) (if *s* = 1*s*′)

[[*M*0]](*ρ*[*s*′*/x*S]) ⊓ [[*M*1]](*ρ*[*s*′*/x*S]) (if *s* = ⊥*s*′)

Table 2

Syntax and denotational semantics of XPCF

that *D*S = BD with constants 0*,* 1*,* 0*,* 1 of type S → S which denote the functions 0*,* 1*,* 0*,* 1, respectively. For a variable of type S, we omit the type and write *x* for *x*S, for simplicity. We have function terms ⟨0*x*→*M*0; 1*x*→*M*1⟩ and ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩ of type S → *σ* for *M*0 and *M*1 terms of type *σ*. The variable *x* is a bound variable of ⟨0*x*→*M*0; 1*x*→*M*1⟩ and ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩.

We call ⟨0*x* → *M*0; 1*x* → *M*1⟩ and ⟨⟨0*x* → *M*0; 1*x* → *M*1⟩⟩ *extended conditional terms*. For the two functions *f*0 = [[*λx*S*.M*0]] and *f*1 = [[*λx*S*.M*1]] from *D*S to *Dσ*,

the function *f* = [[⟨0*x*S →*M*0; 1*x*S →*M*1⟩]] : *D*S → *Dσ* returns *f*0(*s*) if the argument is 0*s* and *f*1(*s*) if the argument is 1*s*. For the case the argument starts with ⊥, we define *f* (⊥*s*) = *f*0(*s*) ⊓ *f*1(*s*), which is the meet of *f*0(*s*) and *f*1(*s*) in *Dσ*. Here, meets on *D*N and *D*B are obviously defined, meets on *D*S are explained in Section 2.2, and the meet of two functions *g, h* ∈ *Dσ*→*τ* is the pointwise meet function (*g* ⊓*h*)(*x*) = *g*(*x*)⊓*h*(*x*). We define *f* (*ǫ*) = ⊥*σ*. Thus, *f* is a strict function. It means that we adopt call by value semantics to an application of ⟨0*x*S →*M*0; 1*x*S →*M*1⟩.

The meaning of the term ⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩ is different from that of ⟨0*x*S → *M*0; 1*x*S → *M*1⟩ only for the case of *ǫ*, and [[⟨⟨0*x*S → *M*0; 1*x*S → *M*1⟩⟩]] is not a strict function. Note that if we identify *ǫ* and ⊥*ω* and match ⊥*s*′ with ⊥*ω* for *s*′ = *ǫ*, then the last case of the semantics of ⟨⟨0*x*S → *M*0; 1*x*S → *M*1⟩⟩ subsumes the first case. Note that both functions [[⟨0*x*S →*M*0; 1*x*S →*M*1⟩]] and [[⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩]] are continuous. Our intention in introducing two kinds of extended conditional terms is that ⟨0*x*S →*M*0; 1*x*S →*M*1⟩ is used in writing a program and ⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩ is used only in reduction steps, which we explain in Section 5. We call a closed ground type term a program if it does not contain extended conditional terms of the form ⟨⟨0*x*S →*M*0; 1*x*S →*M*1⟩⟩ as subterms.

# Program examples of XPCF

The function nh to invert the first digit is written as

nh = ⟨0*x*→1*x*; 1*x*→0*x*⟩*.*

Note that [[nh]](⊥*s*) = 0*s* ⊓ 1*s* = ⊥*s* for *s* ∈ Σ*ω*.

The function ns to invert the second digit is written as

ns = ⟨0*x*→0(nh *x*); 1*x*→1(nh *x*)⟩*.*

The following terms head : S → B and tail : S → S are the head and the tail function on *D*S.

head = ⟨0*x*→ff; 1*x*→tt⟩*,*

tail = ⟨0*x*→*x*; 1*x*→*x*⟩*.*

Here, we identify 0*,* 1*,* ⊥ ∈ Σ⊥ with *ff , tt,* ⊥B ∈ *D*B, respectively. Note that there is no cons function: B → S → S because if we prepend ⊥ to a 1⊥-sequence, then the result may not be a 1⊥-sequence. The function inv to invert all the digits is written as

inv = YS→S(*λf* S→S*.*⟨0*x*→1(*f x*); 1*x*→0(*f x*)⟩)*.*

For simplicity, we use the recursive definition notation to abbreviate a term defined with the Y operator. For example, inv is written as

inv = ⟨0*x*→1(inv *x*); 1*x*→0(inv *x*)⟩*.*

We show how real numbers and real functions are expressed in XPCF. Since

*ϕ*(0) = 0*ω*, *ϕ*(1) = 10*ω* and *ϕ*(1*/*2) = ⊥10*ω*, we can express these numbers as

0 = YS 0*,*

1 = 1(YS 0)*,*

1*/*2 = 1(YS 0).

In Section 2.1, we defined notions that a function on BD (exactly) realizes a function on I. We say that a closed XPCF term (exactly) realizes a real function if it denotes a function which (exactly) realizes the function. The program

div2 = *λx*S*.*0*x.*

realizes the function div2(*x*) = *x/*2 but does not exactly realize it because [[div2]](10*ω* ) = 010*ω* whereas *ϕ*(1*/*2) = ⊥10*ω*. There is also a program which exactly realizes div2, which is given later. Since the complement function comp(*x*) = 1 − *x* is realized by the function to invert the first digit, comp is exactly realized by the program nh. The tent function *t* is exactly realized by tail.

Programs which realize addition (average) av, subtraction sub, multiplication

mult and a program div2b which exactly realizes div2 can be written as follows.

av = ⟨0*x*→⟨0*y* →0(av *x y*); 1*y* →1(ns(av *x* (nh *y*)))⟩;

1*x*→⟨0*y* →1(ns(av (nh *x*) *y*)); 1*y* →1(av *x y*)⟩⟩

sub = ⟨0*x*→⟨0*y* →0(sub *x y*); 1*y* →YS0⟩;

1*x*→⟨0*y* →nh(av *x y*); 1*y* →0(sub *y x*)⟩⟩

mult = ⟨0*x*→⟨0*y* →0(0(mult *x y*)); 1*y* →0(mult *x* (1*y*))⟩;

1*x*→⟨0*y* →0(mult (1*x*) *y*);

1*y* →av (nh(av *x y*)) (1(nh(mult (nh *x*) (nh *y*))))⟩⟩

div2b = ⟨0*x*→0(0*x*); 1*x*→1(f *x*)⟩

f = ⟨0*x*→0(f *x*); 1*x*→0(1*x*)⟩

Here, [[f ]] is a function which satisfies [[f ]](0*ω* ) = ⊥0*ω* and [[f ]](*x*) = 0*x* if *x* contains the character 1.

# Operational semantics of XPCF

* 1. *Operational semantics of XPCF*

Table 3 shows the reduction rule of XPCF. For d ∈ {0*,* 1*,* 0*,* 1}, we say that a term

*M* of type S outputs d if *M* is reduced to d*M* ′.

We explain how the reduction of a term ⟨0*x* → *M*0; 1*x* → *M*1⟩ *N* proceeds. The first lines of rules (COND 0), (COND 1), (COND 0), and (COND 1) are for the reduction of an application term ⟨0*x* → *M*0; 1*x* → *M*1⟩ *N* . Note that a closed term *N* is reduced by (APP-R) to one of these four forms if [[*N* ]] is not ⊥ by the adequacy theorem in the next section. If *N* has the form 0*N* ′, (COND 0) is applied and then we have a term *M*0[0*x/x*] and *M*1[1*x/x*]. After that, *M*0 and *M*1 are evaluated by (LEFT) and (RIGHT) only with the additional information that the first character of *x* is 0. Note that if *M*0 contains *x*, then *M*0[0*x/x*] also contains *x* and therefore it is expected that this evaluation terminates when it requires the

|  |  |
| --- | --- |
|  |  |
| Reduction rule of XPCF |
| In addition to the reduction rule of PCF, we have the following rules. (COND 0) ⟨0*x*→*M*0; 1*x*→*M*1⟩(0*N* ) *⊲ M*0[*N/x*]  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩(0*N* ) *⊲ M*0[*N/x*] (COND 1) ⟨0*x*→*M*0; 1*x*→*M*1⟩(1*N* ) *⊲ M*1[*N/x*]  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩(1*N* ) *⊲ M*1[*N/x*]  (COND 0) ⟨0*x*→*M*0; 1*x*→*M*1⟩(0*N* ) *⊲* ⟨⟨0*x*→*M*0[0*x/x*]; 1*x*→*M*1[0*x/x*]⟩⟩*N*  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩(0*N* ) *⊲* ⟨⟨0*x*→*M*0[0*x/x*]; 1*x*→*M*1[0*x/x*]⟩⟩*N*  (COND 1) ⟨0*x*→*M*0; 1*x*→*M*1⟩(1*N* ) *⊲* ⟨⟨0*x*→*M*0[1*x/x*]; 1*x*→*M*1[1*x/x*]⟩⟩*N*  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩(1*N* ) *⊲* ⟨⟨0*x*→*M*0[1*x/x*]; 1*x*→*M*1[1*x/x*]⟩⟩*N*  (OUT 1) ⟨⟨0*x*→d*M*0; 1*x*→d*M*1⟩⟩*N ⊲* d(⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩*N* ) (d ∈ {0*,* 1*,* 0*,* 1}) (OUT 2) ⟨⟨0*x*→b(c*M*0); 1*x*→b′(c*M*1)⟩⟩*N ⊲* c(⟨⟨0*x*→b*M*0; 1*x*→b′*M*1⟩⟩*N* )  (b*,* b′ ∈ {0*,* 1} and b /= b′) (OUT 3) ⟨⟨0*x*→b*M*0; 1*x*→c(b*M*1)⟩⟩*N ⊲* b(⟨⟨0*x*→*M*0; 1*x*→c*M*1⟩⟩*N* ) (b*,* c ∈ {0*,* 1}) (OUT 4) ⟨⟨0*x*→c(b*M*0); 1*x*→b*M*1⟩⟩*N ⊲* b(⟨⟨0*x*→c*M*0; 1*x*→*M*1⟩⟩*N* ) (b*,* c ∈ {0*,* 1}) (OUT 5) ⟨⟨0*x*→*c*; 1*x*→*c*⟩⟩ *⊲ c* (*c* ∈ {tt*,* ff*,* k*n*})  (BAR) b(c*M* ) *⊲* c(b*M* ) (b*,* c ∈ {0*,* 1})  (PERM) ⟨0*x*→*M*0; 1*x*→*M*1⟩*NL ⊲* ⟨0*x*→*M*0*L*; 1*x*→*M*1*L*⟩*N*  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩*N L ⊲* ⟨⟨0*x*→*M*0*L*; 1*x*→*M*1*L*⟩⟩*N*  (If *x* ∈ *FV* (*L*), then rename the bound variable *x* to avoid variable collision.)  (LEFT) *M*0 *⊲ M* ′  0  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩ *⊲* ⟨⟨0*x*→*M* ′ ; 1*x*→*M*1⟩⟩  0  (RIGHT) *M*1 *⊲ M* ′  1  ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩ *⊲* ⟨⟨0*x*→*M*0; 1*x*→*M* ′ ⟩⟩  1  (APP-R) *N ⊲ N* ′ (if *M* is 0*,* 1*,* 0*,* 1*,* ⟨0*x*→*M* ; 1*x*→*M* ⟩ or ⟨⟨0*x*→*M* ; 1*x*→*M* ⟩⟩) *MN ⊲ MN* ′ 0 1 0 1 | |

Table 3 Operational semantics of XPCF

value of *x*. Then, (BAR) rule is used to arrange outputs of *M*0[0*x/x*] and *M*1[0*x/x*] to the form b0b1 *. . .* b*k*c0c1 *. . .* c*j* for b*i,* c*i* ∈ {0*,* 1}. After that, if they coincide on the first or the second digit, then it makes an output with rules (OUT 1) to (OUT 5) and repeat it until no more output is possible. Thus, we obtain a term of the

form d0d1 *. . .* d*i*(⟨⟨0*x* → *M* ′ ; 1*x* → *M* ′ ⟩⟩ *N* ′) and we can continue this process to the

0 1

subterm ⟨⟨0*x* → *M* ′ ; 1*x* → *M* ′ ⟩⟩ *N* ′ with (APP-R) since all the rules applicable to

0 1

⟨0*x*→*M*0; 1*x*→*M*1⟩ *N* are also applicable to ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩ *N* .

One can see that the above reduction procedure fails to reduce

⟨0*x*→*M*0; 1*x*→*M*1⟩ *N L* for *M*0 and *M*1 function type terms and [[*N* ]] = ⊥*s* because the output of *L* cannot be fed to function terms *M*0 and *M*1. For the evaluation of

this term, we need the (PERM) rule. Suppose that ⟨0*x*→*M*0; 1*x*→*M*1⟩ : S → S → S and *M*0 and *M*1 are extended conditional terms of the form ⟨0*y* → *...*; 1*y* → *...*⟩. We first reduce the term ⟨0*x* → *M*0; 1*x* → *M*1⟩ *N L* to ⟨0*x* → (*M*0 *L*); 1*x* → (*M*1 *L*)⟩ *N* with the (PERM) rule and then reduce it as we explained. The (PERM) rule corre- sponds to reducing the lambda term (*λx.M* ) *N L* to (*λx.*(*M L*)) *N* , and it is similar to the permutative conversion rule used for proof normalization in proof theory [12]. One may wonder why we distinguish ⟨0*x* → *M*0; 1*x* → *M*1⟩ with ⟨⟨0*x* → *M*0; 1*x* → *M*1⟩⟩ because we can make the same reduction if we replace the former with the latter. However, strictness of ⟨0*x*→*M*0; 1*x*→*M*1⟩ plays an important role in writing recursively defined functions. Many of the functions on S are defined with the Y operator as *F* = YS→S(*λf.M* ) with *M* = ⟨0*x* → *M*0; 1*x* → *M*1⟩ and it is reduced to *M* [YS→S(*λf.M* )*/f* ] which cannot be reduced any more. If *M* = ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩ instead, then copies of YS→S(*λf.M* ) in *M*0 and *M*1 can be reduced with the rules (LEFT) and (RIGHT) and therefore it causes an infinite computation even if no

argument is given to the function term *F* .

* 1. *A sequential strategy of XPCF*

Though one needs to evaluate *M*0, *M*1, and *N* in parallel for the evaluation of *M* =

⟨⟨0*x* →*M*0; 1*x* →*M*1⟩⟩ *N* , the procedure we mentioned above is almost sequential in that the evaluations of *M*0 and *M*1 are expected to terminate because they contain the free variable *x* in many cases. There are some cases that the evaluation of *M*0 does not terminate and it outputs infinitely many digits. However, if *M*0 has the form d0d1*M* ′, then, from the forms of (OUT 1) to (OUT 4), one can consider that *M*0 has enough outputs for *M* to make an output and terminate its reduction and proceed to the evaluation of *M*1. We also need to take care of the case *M*0 has the form ⟨⟨0*y* → *M*00; 1*y* → *M*11⟩⟩ *L*. In this case, if we reduce *M* according to the

procedure we mentioned above, and *L* is reduced to 0*L*1

*⊲*∗ 02*L*2

*⊲*∗ · · · *⊲*∗ 0*nLn*

*⊲*∗ · · · ,

for example, then one repeats the application of (COND 0) without instantiating the outputs of *N* to *x*. However, we can handle many of the cases by defining that

⟨⟨0*y* →*M*00; 1*y* →*M*11⟩⟩ *L* cannot be reduced if *M*00 and *M*11 cannot be reduced and all the appearances of *y* in *M*00 and *M*11 have the form c0c1 *. . .* c*ky* for *k >* 1 and c*i* ∈

{0*,* 1}. Note that, in this case, further digits of *y* do not change the situation that *M*00 cannot be reduced. In this way, we designed a sequential reduction strategy of XPCF. We implemented it with Haskell. As it is proved in [7], in an interval domain model, an adequate real number calculus in which average function is definable does not have a sequential reduction strategy. It is also the case in our model and this sequential strategy is not adequate. Therefore, it does not evaluate all the terms to their denotations. However, we observed that applications of terms in Section 4 are reduced with our implementation and we expect that it evaluates many of the ”meaningful” terms to their semantics.

# Computational adequacy of XPCF

We show the soundness and completeness properties of XPCF. We first show that two kinds of substitutions in the reduction rule of XPCF preserve meanings.

Lemma 6.1 (i) *For terms M* : *τ and N* : *σ, a variable xσ, and an environment*

*ρ,* [[*M* [*N/xσ*]]](*ρ*) = [[*M* ]](*ρ*[[[*N* ]]*/xσ* ])*.*

(ii) *For a term M and* b ∈ {0*,* 1*,* 0*,* 1}*,* [[*M* [b*x/x*]]](*ρ*) = [[*M* ]](*ρ*[[[b]]*ρ*(*x*)*/x*])*.*

Proof. By structural induction on *M* . ✷

From Lemma 6.1, the following proposition holds.

Proposition 6.2 *For XPCF terms M, N and an environment ρ, if M ⊲ N then*

[[*M* ]](*ρ*) = [[*N* ]](*ρ*)*.*

Proof. It is proved by showing that the denotational semantics of the left side and the right side coincide for every reduction rule. ✷

In PCF, the result of evaluation of a program *M* of type *σ* is a constant of type *σ* if it exists. On the other hand, in XPCF, we consider non-terminating computations which output digits in {0*,* 1*,* 0*,* 1} as *M ⊲ . . . ⊲* d0(d1 · · · (d*n*−1*M* ′)) *⊲ . . .*. Note that the sequence d0*,* d1*,* · · · is not determined uniquely by *M* . For example, the term *M* = ⟨⟨0*x* → 0(Y0); 1*x* → 1(Y0)⟩⟩(1ΩS) for ΩS = YS→S(*λx*S*.x*S) is reduced to terms of the forms 10*nM* ′ and 0*nN* ′ for every *n*. However, from Proposition 6.2, if *M ⊲*∗ d0(d1 · · · (d*n*−1*M* ′)), then we have *d*0(*d*1*...*(*dn*−1*ǫ*)) ⊑ *d*0(*d*1*...*(*dn*−1[[*M* ′]])) = [[*M* ]] and thus the outputs are bounded by the denotation [[*M* ]] of *M* and have the least upper bound. Therefore, we define an evaluation function Eval from XPCF programs of type *σ* to elements of *Dσ* as follows.

Definition 6.3 (i) For an XPCF program *M* of type N or B, we define

(

Eval(*M* ) = [[EvalPCF(*M* )]] if EvalPCF(*M* ) exists*,*

⊥ otherwise*.*

(ii) For an XPCF program *M* of type S, we define

Eval(*M* ) = .{*d*0(*d*1*...*(*dn*−1(*ǫ*))) | *M ⊲*∗ d0(d1 · · · (d*n*−1*M* ′)) for some *M* ′}

with d*i* ∈ {0*,* 1*,* 0*,* 1} and *di* = [[d*i*]] for 0 ≤ *i < n*.

The soundness of XPCF is derived from Proposition 6.2 immediately.

Theorem 6.4 (Soundness of XPCF) *For a program M,* Eval(*M* ) ⊑ [[*M* ]]*.*

To show the completeness, we use the computability method (see [11]). That is, define the set Comp*σ* of computable terms of type *σ* for each type *σ* and then show that all the XPCF terms are computable.

Definition 6.5 We define the predicate Comp*σ* for each type *σ* by induction on types.

1. Let *σ* be B or N. A program *M* : *σ* has property Comp*σ* if [[*M* ]] = Eval(*M* ).
2. A program *M* : S has property CompS if [[*M* ]] ⊑ Eval(*M* ). That is, for any

∗

*p* ∈ Σ

⊥*,*1

with *p* ⊑ [[*M* ]], *p* ⊑ Eval(*M* ) holds.

1. A closed term *M* : *σ* → *τ* has property Comp*σ*→*τ* if whenever *N* : *σ* is a closed term with property Comp*σ* then *MN* is a term with property Comp*τ* .
2. An open term *M* : *σ* with free variables *xσ*1 *, ..., xσn*

has property Comp*σ* if

1 *n*

*M* [*N*1*/xσ*1 ] · · · [*Nn/xσn* ] has property Comp*σ* whenever *N*1*, ..., Nn* are closed

1 *n*

terms having properties Comp*σ*1 ,...,Comp*σn* respectively.

We say that a term of type *σ* is *computable* if it has property Comp*σ*.

It is immediate to show the followings. (1) If *M* : *σ* → *τ* and *N* : *σ* are closed computable terms, so is *MN* . (2) For a ground type *τ* , a term *M* : *σ*1 → · · · → *σn* → *τ* is computable if and only if *M*˜ *N*1 · · · *Nn* is computable for all closed computable

terms *N*1 : *σ*1*, ..., Nn* : *σn* and closed instantiation *M*˜

of *M* by computable terms.

For *s* ∈ Σ∗

⊥*,*1

, we define the context *s*[*X*] as follows,

*s*[*X*] = b0(b1 · · · (b*n*−1*X*)) if *s* = *b*0*b*1 · · · *bn*−1*,*

(

b0(b1 · · · (b*n*−1(c0(c1 · · · (c*m*−1*X*))))) if *s* = *b*0*b*1 · · · *bn*−1⊥*c*0*c*1 · · · *cm*−1.

Here, b*i,* c*j* ∈ {0*,* 1}, *bi* = [[b*i*]], and *cj* = [[c*j* ]] for 0 ≤ *i < n* and 0 ≤ *j < m*. We

say that a term *M* of type S *outputs s* ∈ Σ∗ if there is a reduction *M ⊲*∗ *s*[*M* ′] for

⊥*,*1

some *M* ′.

Lemma 6.6 *Let M* : S *be a computable term such that x is the only free variable*

*and let s* ∈ Σ∗*. For any p* ⊑ [[*M* ]](*ρ*[*s/x*]) *with p* ∈ Σ∗ *, M* [*s*[*x*]*/x*] *outputs q* ∈ Σ∗

⊥*,*1

⊥*,*1

*such that p* ⊑ *q*

Proof. We have [[*M* ]](*ρ*[*ǫ/x*]) = [[*M* [ΩS*/x*]]] by structural induction on *M* . From the equation [[*s*[ΩS]]] = *s* and Lemma 6.1 (i), we have

[[*M* ]](*ρ*[*s/x*]) = [[*M* ]](*ρ*[[[*s*[ΩS]]]*/x*]) = [[*M* [*s*[ΩS]*/x*]]]*.*

Since *M* and *s*[ΩS] are computable, *M* [*s*[ΩS]*/x*] is computable. Therefore, for any

∗

*p* ∈ Σ

⊥*,*1

∗

*q* ∈ Σ

⊥*,*1

with *p* ⊑ [[*M* ]](*ρ*[*s/x*]), there exists a reduction *M* [*s*[ΩS]*/x*] *⊲*∗ *q*[*M* ′] with such that *p* ⊑ *q*. If there is a reduction sequence *M* [*s*[ΩS]*/x*] *⊲*∗ *q*[*M* ′], then

there is a reduction sequence *M* [*s*[*x*]*/x*] *⊲*∗ *q*[*M* ′′] by ignoring the reductions related

to ΩS. Therefore, *M* [*s*[*x*]*/x*] outputs *q* such that *p* ⊑ *q*. ✷

Proposition 6.7 *Every XPCF term is computable.*

Proof. We prove it by structural induction on terms.

In order to prove the computability of Y*σ* for an XPCF type *σ*, we use an extension of the syntactic information order in [11], which we omit here. We only explain the proof of the cases 0*,* 1*,* 0*,* 1*,* ⟨0*x*→*M*0; 1*x*→*M*1⟩ and ⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩.

The case d ∈ {0*,* 1*,* 0*,* 1}. We show that for any computable term *M* of type

S, d*M* is computable. Because the function [[d]] : *D*S → *D*S is continuous, for any

⊥*,*1

∗

*p* ∈ Σ

⊥*,*1

with *p* ⊑ [[d*M* ]] = [[d]]([[*M* ]]), there exists *q* ∈ Σ∗

with *q* ⊑ [[*M* ]] such

that *p* ⊑ [[d]](*q*). Since *M* is computable, *M* outputs *q*′ ∈ Σ∗ such that *q* ⊑ *q*′.

⊥*,*1

Therefore, d*M* outputs [[d]](*q*′) which satisfies *p* ⊑ [[d]](*q*) ⊑ [[d]](*q*′) and thus d is

computable.

We show that if terms *M*0 and *M*1 of type *σ* are computable, so is the term ⟨0*x*→ *M*0; 1*x* →*M*1⟩. It is enough to show that the term ⟨0*x* →*M*˜0; 1*x* →*M*˜1⟩*N*1*N*2 · · · *Nn* of type a ground type *τ* is computable when *N*1 : S*, N*2*, ..., Nn* are closed computable terms and *M*˜0 and *M*˜1 are instantiations of all free variables, except *x*, of *M*0 and *M*1 by closed computable terms, respectively. We only show the case *τ* = S.

Case [[*N*1]] = *ǫ*. From the reduction rule, we have the following equation:

[[⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩*N*1*N*2 · · · *Nn*]]

= [[⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩]](*ǫ*)([[*N*2]]) · · · ([[*Nn*]])

= ⊥*σ*([[*N*2]]) · · · ([[*Nn*]]) = ⊥S*.*

Therefore, ⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩*N*1*N*2 · · · *Nn* is computable.

Case [[*N*1]] = 0*s*. For any *p* ∈ Σ∗

⊥*,*1

such that *p* ⊑ [[⟨0*x* →

*M*˜0; 1*x* →

*M*˜1⟩*N*1 · · · *Nn*]] = [[*M*˜0]]*ρ*([*s/x*])([[*N*2]]) · · · ([[*Nn*]]), from the continuity, there ex-

ists *s*′ ∈ Σ∗

⊥*,*1

such that *p* ⊑ [[*M*˜0]]*ρ*([*s*′*/x*])([[*N*2]]) · · · ([[*Nn*]]) and 0*s*′ ⊑ [[*N*1]].

From the computability of *N*1, there exists 0*s*′′ ∈ Σ∗

⊥*,*1

such that *N*1 outputs

0*s*′′ and 0*s*′ ⊑ 0*s*′′. Then, *p* ⊑ [[*M*˜0]]*ρ*([*s*′′*/x*])([[*N*2]]) · · · ([[*Nn*]]) holds. Since we

have [[*M*˜0[*s*′′[ΩS]*/x*]*N*2 · · · *Nn*]] = [[*M*˜0]]*ρ*([*s*′′*/x*])([[*N*2]]) · · · ([[*Nn*]]) and *s*′′[ΩS] is com-

putable, *M*˜0[*s*′′[ΩS]*/x*]*N*2 · · · *Nn* is also computable and outputs *t* ∈ Σ∗

⊥*,*1

such that

*p* ⊑ *t*. Therefore, we have a reduction sequence ⟨0*x* → *M*˜0; 1*x* → *M*˜1⟩*N*1 · · · *Nn ⊲*∗

⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩0*s*′′[*N* ′ ] · · · *Nn ⊲*∗ *t*[*M* ′′] such that *p* ⊑ *t*.

1

Case [[*N*1]] = 1*s*. The proof is similar to the case [[*N*1]] = 0*s*.

Case [[*N*1]] = ⊥*u* with *u* ∈ Σ∞\{*ǫ*}. From the reduction rule, we have the following equation:

[[⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩*N*1 · · · *Nn*]]

= [[⟨0*x*→*M*˜0 *N*2 · · · *Nn*; 1*x*→*M*˜1*N*2 · · · *Nn*⟩*N*1]]

= [[⟨0*x*→*M*˜0 *N*2 · · · *Nn*; 1*x*→*M*˜1*N*2 · · · *Nn*⟩]](⊥*u*)

= [[*M*˜0*N*2 · · · *Nn*]](*ρ*[*u/x*]) ⊓ [[*M*˜1*N*2 · · · *Nn*]](*ρ*[*u/x*])*.*

Because of the continuity of [[*M*˜0*N*2 · · · *Nn*]] and [[*M*˜1*N*2 · · · *Nn*]], for any *p* ∈ Σ∗ with *p* ⊑ [[⟨0*x* → *M*˜0; 1*x* → *M*˜1⟩*N*1 · · · *Nn*]], there is *s* ∈ Σ∗ such that *s* ⊑ *u*, *p* ⊑ [[*M*˜0*N*2 · · · *Nn*]](*ρ*[*s/x*]), and *p* ⊑ [[*M*˜1*N*2 · · · *Nn*]](*ρ*[*s/x*]). Since *N*1 is computable,

⊥*,*1

*N*1 outputs ⊥*s*. By Lemma 6.6, (*M*˜0*N*2 · · · *Nn*)[*s*[*x*]*/x*] outputs *q*0 ∈ Σ∗

⊥*,*1

such that

*p* ⊑ *q*0 and (*M*˜1*N*2 · · · *Nn*)[*s*[*x*]*/x*] outputs *q*1 ∈ Σ∗

⊥*,*1

such that *p* ⊑ *q*1. Therefore,

⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩*N*1 · · · *Nn* has the following reduction:

⟨0*x*→*M*˜0; 1*x*→*M*˜1⟩*N*1 · · · *Nn*

*⊲*∗ ⟨0*x*→*M*˜0*N*2 · · · *Nn*; 1*x*→*M*˜1 *N*2 · · · *Nn*⟩⊥*s*[*N* ′ ]

1

*⊲*∗ ⟨⟨0*x*→(*M*˜0 *N*2 · · · *Nn*)[*s*[*x*]*/x*]; 1*x*→(*M*˜1 *N*2 · · · *Nn*)[*s*[*x*]*/x*]⟩⟩*N* ′

1

*⊲*∗ ⟨⟨0*x*→*q*0[*M* ′ ]; 1*x*→*q*1[*M* ′ ]⟩⟩*N* ′

0 1 1

*⊲*∗ *q*[*M* ′]

for some *q* ∈ Σ∗ such that *p* ⊑ *q* ⊑ (*q*0 ⊓ *q*1).

⊥*,*1

The computability proof of ⟨⟨0*x* → *M*˜0; 1*x* → *M*˜1⟩⟩ is that of ⟨0*x* → *M*0; 1*x* → *M*1⟩

without the case [[*N*1]] = *ǫ* and without restricting in the final case to *u* /∈ *ǫ*. ✷

Therefore, the completeness of XPCF holds.

Theorem 6.8 (Completeness of XPCF) *For a program M,* Eval(*M* ) ⊒ [[*M* ]]*.* Combining the soundness and completeness of XPCF, we have the computa-

tional adequacy of XPCF. That is, Eval(*M* ) = [[*M* ]] for every program *M* .

# Expressive power of XPCF

In this section, we often omit the type and write *x* for *xσ* and if for if*σ* when no confusion can arise.

We compare expressive powers of XPCF and PCF+. Here, PCF+ is the calculus PCF extended with the parallel conditional pif*σ* : B → *σ* → *σ* → *σ* as a constant for each *σ* ∈ {B*,* N}. The interpretation of pif*σ* is given as follows

[[pif*σ*

]] = *λb* ∈ *D*B

*.λx* ∈ *Dσ*

*.λy* ∈ *Dσ*

*. y* (*b* = *ff* )

*x* (*b* = ⊥B and *x* = *y*)



*x* (*b* = *tt*)

⊥*σ* (otherwise)*.*

The operational semantics of PCF + is the operational semantics of PCF together with:

pif*σ M c c ⊲ c,* pif*σ* tt *M N ⊲ M,* pif*σ* ff *M N ⊲ N,*

*M ⊲ M* ′

pif*σ M ⊲* pif*σ M* ′*,*

*N ⊲ N* ′

pif*σ M N ⊲* pif*σ M N* ′*,*

*L ⊲ L*′

pif*σ M N L ⊲* pif*σ M N L*′*.*

Consider the following XPCF term pif′

*σ*

of type B → *σ* → *σ* → *σ* for *σ* ∈ {B*,* N}.

pif′

*σ*

= *λu*B*.λyσ.λzσ.*⟨⟨0*x*→*y*; 1*x*→*z*⟩⟩(if S *u* (0ΩS) (1ΩS))*.*

It satisfies [[pif ′ ]] = [[pif*σ* ]] and therefore it expresses the pif*σ* operator. Note that one can also express it as an XPCF program

*σ*

*λu*B*.λyσ.λzσ.*⟨0*x*→*y*; 1*x*→*z*⟩0(ifS *u* (0ΩS) (1ΩS))

without using ⟨⟨*...*⟩⟩. Thus, PCF + terms can be translated into XPCF terms by replacing pif*σ* with pif′ .

*σ*

Theorem 7.1 *For a PCF* + *term M* : *σ and a PCF environment ρ, there exists an XPCF term M* ′ : *σ such that* [[*M* ]](*ρ*) = [[*M* ′]](*ρ*′)*. Here, ρ*′ *is any extension of ρ to an XPCF environment.*

On the other hand, there is an embedding-projection pair (e-p pair in short)

(*e, p*) between the domains *D*S = BD and *D*N→B ∼= Σ*ω*

⊥

where the projection *p* is

the function trunc in Section 2.2. Here, a pair of continuous functions *e* : *X* → *Y*

and *p* : *Y* → *X* is an e-p pair if they satisfy *p* ◦ *e* = *idX* and *e* ◦ *p* ⊑ *idY* . Terms e : S → (N → B) and p : (N → B) → S such that [[e]] = *e* and [[p]] = *p* can be written in XPCF as follows,

e : = YS→N→B(*λf* S→N→B*.λg*S*.λn*N*.*if (zero *n*) (head *g*) (*f* (tail *g*) (dec *n*)))

*.λn .λf .*⟨0*x*→0(*g* (inc *n*) *f* ); 1*x*→1(*g* (inc *n*) *f* )⟩

p : = YN→(N→B)→S(*λg*N→(N→B)→S N N→B

0(if(*f n*)(0ΩS)(1ΩS)))k0

where tail = ⟨0*x*→*x*; 1*x*→*x*⟩ and head = ⟨0*x*→tt; 1*x*→ff⟩.

We can extend the e-p pair (*e, p*) to higher order types. We inductively define

*σt* for every XPCF type *σ* as follows

B*t* = B*,* N*t* = N*,* S*t* = N → B*,* and (*σ* → *τ* )*t* = *σt* → *τ t.*

We inductively define e*σ* : *σ* → *σt* and p*σ* : *σt* → *σ* for every XPCF type *σ* as follows,

N B

eN = pN = *λx .x,* eB = pB = *λx .x,* eS = e*,* pS = p*,*

*t*

e = *λfσ*→*τ σ*

*σ*→*τ .λx .*e*τ* (*f* (p*σ*(*x*)))*,*

p = *λfσt*→*τt.λxσ.*p (*f* (e (*x*)))*.*

*σ*→*τ τ σ*

It is immediate to show that ([[e*σ*]]*,* [[p*σ*]]) is an e-p pair for every type *σ*.

We define a syntactical translation (−)*t* from XPCF terms to PCF + terms so that *Mt* : *σt* for *M* : *σ*. Before that, we define a function *r* : *D*N→B → *D*N→B as *r* = *e* ◦ *p* and *rσ* : *Dσt* → *Dσt* as *rσ* = *eσ* ◦ *pσ*. The function *r* satisfies

*r*(*f* )(*n*) =

*ff* if *f* (*n*) = *ff* and ⊥ appears at most once in *f* (0)*,* · · · *, f* (*n* − 1)

⊥ otherwise.

*tt* if *f* (*n*) = *tt* and ⊥ appears at most once in *f* (0)*,* · · · *, f* (*n* − 1)

for *f* : *D*N → *D*B and *n* ∈ *D*N. Let r be any PCF + term such that [[r]] = *r*. For every XPCF type *σ*, we inductively define a PCF+ term r*σ* : *σt* → *σt* which satisfies [[r*σ*]] = *rσ* as follows

rB = *λx*B*.x*B*,* rN := *λx*N*.x*N*,* rS := r*,* and r*σ*→*τ* = *λfσt*→*τt .λx*(*σt* )*.*r*τ* (*f* (r*σx*))*.*

For an XPCF term *M* , we inductively define *Mt* : *σt* as follows,

*σ t* (*σt*) *t t t σt*→*σt*

(*x* ) = r*σx ,* c = c*,* if*σ* = if*σ,* Y*σ* = *λf*

*.*Y*σt* (r*σ*→*σf* )*,*

(*λxσ.M* )*t* = *λx*(*σt* )*.Mt,* (*MN* )*t* = (*MtNt*)*,*

0*t* = *λf* N→B*.λx*N*.*if (zero *x*) tt ((rS *f* ) (dec *x*))*,* 1*t* = *λf* N→B*.λx*N*.*if (zero *x*) ff ((rS *f* ) (dec *x*))*,* 0*t* = *λf* N→B*.λx*N*.*if (zero *x*) ((r *f* ) k )

S 0

(if (zero (dec *x*)) tt ((rS *f* ) (dec *x*)))*,*

1*t* = *λf* N→B*.λx*N*.*if (zero *x*) ((r *f* ) k )

S 0

(if (zero (dec *x*)) ff ((rS *f* ) (dec *x*)))*,*

⟨⟨0*x*→*M*0; 1*x*→*M*1⟩⟩*t* = *λf* N→B*.*pif ((rS *f* ) k0) *Mt*[*λy*N*.*(rS *f* )(inc *y*)*/x*N→B]

0

*Mt*[*λy*N*.*(rS *f* )(inc *y*)*/x*N→B]*,*

1

⟨0*x*→*M*0; 1*x*→*M*1⟩*t* = *λf* N→B*.*if(pif (if ((rS *f* ) k0) tt tt) tt (if ((rS *f* ) k1) tt tt))

(pif ((rS *f* ) k0) *Mt*[*λy*N*.*(rS *f* )(inc *y*)*/x*N→B]

0

*Mt*[*λy*N*.*(rS *f* )(inc *y*)*/x*N→B]) Ω*σ*

1

where c is a constant other than 0*,* 1*,* 0*,* 1*,* if*σ* or Y*σ* and the type of terms *M*0 and *M*1 is *σ*. Here, we assume that the same XPCF variable does not appear in different types to prevent conflictions in the translation to PCF + terms.

*t*

*t t σ σ*

We define a translation (-) of environments as *ρ* (*x* ) = *eσ*(*ρ*(*x* )).

Proposition 7.2 *For any term M* : *σ in XPCF and environment ρ, eσ*([[*M* ]](*ρ*)) = [[*M t*]](*ρt*) *holds.*

Proof. By structural induction on *M* . ✷

It is known that in PCF + all compact elements and all computable first-order functions are definable [11]. Through the translation (-)*t*, we can derive the following results on expressive power of XPCF.

Theorem 7.3 (i) *XPCF and PCF* + *have the same expressive power on PCF types.*

1. *All computable elements of D*S *are definable in XPCF.*
2. *The function exist is not definable in XPCF. Here, exist* : *D*N→B → *D*B *is the function*



*λf* ∈ *D*N→B*.*

*ff f* (⊥) = *ff*

*tt* ∃*n* ∈ N*.f* (*n*) = *tt*

⊥ *otherwise.*

Proof. (i) Since *eσ* is the identity function if *σ* does not contain *S*, Theorem 7.1 and Proposition 7.2 show that XPCF and PCF + have the same expressive power on PCF types.

1. For any computable element *x* ∈ *D*S, *e*S(*x*) ∈ *D*N→B is a computable element because *e*S is a computable function. Therefore, *e*S(*x*) is definable in PCF + [11]. Since *p*S is definable in XPCF, *p*S(*e*S(*x*)) = *x* is definable in XPCF.
2. Suppose that there exists a closed XPCF term *M* : (N → B) → B such that [[*M* ]] = *exist*. Since *eσ* is identity for a PCF type *σ*, we have [[*M* ]] = *e*(N→B)→B([[*M* ]]) = [[*M t*]] from Proposition 7.2. However, *exist* is not definable in PCF + [11] and this is a contradiction. Therefore, *exist* is not definable in XPCF.✷

In [11], Plotkin introduced the language PCF++ which is an extension of PCF+ by adding the existential quantifier ∃ : (N → B) → B as a constant such that [[∃]] = *exist*. [5] showed that Real PCF extended with ∃ is universal, based on a technique due to Thomas Streicher [13] to establish that PCF extended with recursive types, parallel-or and ∃ is universal. We define a calculus XPCF ∃, which is the extension of XPCF with the ∃ operator. XPCF ∃ is universal in the following sense.

Theorem 7.4 *For every XPCF type σ, all computable elements of Dσ are definable in XPCF* ∃*.*

Proof. For any XPCF type *σ* and computable element *x* ∈ *Dσ*, *eσ*(*x*) ∈ *Dσt* is a computable element because *eσ* is a computable function. Therefore, *eσ*(*x*) is definable in PCF ++ by [11]. Since *pσ* is definable in XPCF, *pσ*(*eσ*(*x*)) = *x* is definable in XPCF ∃. ✷

# Acknowledgement

The authors would like to thank to Takashi Sakuragawa for valuable comments. This work was partly supported by JSPS KAKENHI Grant Number 22500014.

# References

1. Berger, U., and Hou, T., *Coinduction for Exact Real Number Computation*. Theory of Computing Systems. 43 (2008), 394-409.
2. Boehm, H.J., Cartwright, R., Riggle, M., and O’Donnell, M.J., “Exact real arithmetic: a case study in higher order programming”, ACM Symposium on Lisp and Functional Programming, ACM, New York, 1986.
3. Blanck, J., *Domain representations of topological spaces*. Theoretical Computer Science. 247 (2000), 229–255.
4. Brattka, V., and Hertling, P., *Topological properties of real number representations*. Theoretical Computer Science. 284 (2002), 241-257.
5. Escard´o, M.H., *Real PCF extended with* ∃ *is universal*. Advances in Theory and Formal Methods of Computing:Proceedings of the Third Imperial Collage Workshop. (1996), 13-24.
6. Escard´o, M.H., *PCF extended with real numbers*. Theoretical Computer Science. 162 (1996), 79-115.
7. Escard´o, M.H., Hofmann, M., and Streicher, T., *On the non-sequential nature of the interval-domain model of real-number computation*, Math. Struct. in Comp. Science. 14(6) (2004), 803-814.
8. Di Gianantonio, P., *An abstract data type for real numbers*. Theoretical Computer Science. 221 (1999), 295-326.
9. Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., and Scott, D.S., “Continuous Lattices and Domains”, Cambridge University Press, 2003.
10. Gray, F., “Pulse code communication”, March 17, 1953 (filed Nov. 1947). U.S. Patent 2,632,058.
11. Plotkin, G., *LCF considered as a programming language*, Theoretical Computer Science. 5 (1977), 223-255.
12. Schwichtenberg, H., and Wainer, S.S., *Proofs and Computations*, Cambridge University Press, 2012.
13. Streicher, T., *A universality theorem for PCF with recursive types, parallel-or and* ∃. Mathematical Structures for Computing Science. 4(1) (1994), 111-115.
14. Streicher, T., “Domain-Theoretic Foundations of Functional Programming”, World Scientific, 2006.
15. Tsuiki, H., *Real number computation through gray code embedding*. Theoretical Computer Science.

284(2) (2002), 467-485.

1. Tsuiki, H., *Compact metric spaces as minimal-limit sets in domains of bottomed sequences*. Math. Struct. in Comp. Science. 14 (2004), 853-878.
2. Weihrauch, K., “An Introduction to Computable Analysis”, Springer, Berlin, 2000.