Electronic Notes in Theoretical Computer Science 73 (2004) 3–43 

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

Sequentiality and Piecewise-affinity in Segments of Real-PCF

# Amin Farjudian[1](#_bookmark2)

*School of Computer Science University of Birmingham Birmingham, England*

Abstract

*Real PCF* (RPCF) was proposed by Mart´ın Escardo´ [[8](#_bookmark78)] as a language for Real number computation. One of the key — and most controversial — constants is *parallel-if* (pif *I* ), the existence of which causes a serious inefficiency in the language leading to RPCF being impractical. While search is being undertaken to replace pif *I* with a more efficient operator, one needs to be assured of the segment of RPCF without pif *I* being sequential. A positive answer to this question is the main result of this paper. On the other hand, we show that non-affine functions — such as *f* (*x*) := *x*2

— are not definable in RPCF without pif *I* .

*Keywords:* sequentiality, logical relations, real-number computation, Real-PCF, interval domain, PCF

# Contents

1. [A Reminder of the Definitions](#_bookmark0) 2
   1. [PCF](#_bookmark1) 2
   2. [Logical Relations](#_bookmark23) 16
   3. [Computation on Real Numbers: Real-PCF](#_bookmark28) 18
2. [Sequentiality and Parallelism in RPCF](#_bookmark35) 25
3. [Piece-wise affinity](#_bookmark56) 32
   1. [Discussion](#_bookmark66) 38
4. [Summary of the results and possible future investigations](#_bookmark68) 39

1 Email: [M.A.Farjudian@cs.bham.ac.uk](mailto:%20M.A.Farjudian@cs.bham.ac.uk)

1571-0661 © 2004 Published by Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2004.08.002

[Acknowledgement](#_bookmark69) 40

[References](#_bookmark70) 40

# A Reminder of the Definitions

* 1. *PCF*

In one of his seminal works [[13](#_bookmark79)], Plotkin introduced PCF — Programming lan- guage for Computable Functions. Here we give a description of the language, taken almost entirely from Plotkin’s original paper [[13](#_bookmark79)], slightly modified to match our framework.

The set of *types* of PCF is generated by the following grammar:

*σ* ::= bool | nat | *σ* → *σ*

bool and nat are the ground types of *truth values* and *natural numbers*, re- spectively.

For each type *σ* we assume the existence of denumerably many variables

*x*σ(*i* ≥ 0). C0, the set of *standard* constants of PCF, consists of the following:

i

true : bool false : bool

ifbool : bool → bool → bool → bool ifnat : bool → nat → nat → nat

Yσ : ((*σ* → *σ*) → *σ*) (one for each *σ*)

To perform arithmetic computations, we also add the following constants to C0, and call the new set of constants CA:

*n* : nat (one for each natural number *n*)

succ : (nat → nat) pred : (nat → nat) Zero : (nat → bool)

The set of *terms* of PCF is the least set T containing the following:

1. Every variable *x*σ is a term of type *σ*.

i

1. Every constant *c* ∈ CA of type *σ* is a term of type *σ*.
2. If *M* and *N* are terms of types (*σ* → *τ* ) and *σ*, respectively, then

(*MN* ) is a term of type *τ* .

1. If *M* is a term of type *τ* then (*λx*σ*.M* ) is a term of type (*σ* → *τ* ).

i

As the above rules impose an inductive structure on the set of terms T , we can define functions over T using recursion. For instance, take:

*Var* := {*x*σ | *i* ≥ 0 and *σ* ∈ }

i

to be the set of PCF variables, and:

P*fin* (*Var* ) := {*A* ⊆ *Var* | *A* is finite}

then the function FV: T → P*fin* (*Var* ) which returns the set of *free variables*

of any term *M* ∈T can be defined by:

FV(*x*σ) = {*x*σ} (*x*σ ∈ *Var* )

i i i

FV(*c*) = ∅ (*c* ∈ CA) FV((*MN* )) = FV(*M* ) ∪ FV(*N* )

FV((*λx*σ*.M* )) = FV(*M* ) \ {*x*σ}

i i

where \ is the relative complement symbol:

Notation 1.1 (relative complement (set difference) : *A* \ *B*) *For any two sets A and B we denote the relative complement of B in A by A* \ *B, i.e.*

*A* \ *B* := {*x* ∈ *A* | *x* ∈*/ B*}

A term *M* ∈T is said to be *closed* if FV(*M* ) = ∅ and *open* otherwise. Terms of the form (*MN* ) are called *applications* [2](#_bookmark4) and sometimes the brack-

ets are dropped, when they are understood as associating to the left. Terms of the form (*λx*σ*.M* ) are called *abstractions*.

i

*M* [*N*σ*/x*σ] is the result of *substituting* the term *N*σ (of type *σ*) for all free occurrences of *x*σ in *M* , making appropriate changes in the bound variables of *M* so that no free variables of *N* become bound.

i

i

*Programs* are closed terms of ground type. The idea is that the ground types are the datatypes, and programs produce data via *operational semantics*.

* + 1. *Operational Semantics of PCF*

We first define an *immediate reduction relation* → between terms:

2 Plotkin called them *combinations*.

Definition 1.2 [Immediate Reduction Relation →]

* + - 1. ⎧⎨ ifσ true *M N* → *M*

⎩ ifσ false *M N* → *N*

(*σ* ground)*.*

* + - 1. succ *m* → *m* +1 (*m* ≥ 0)
      2. pred *m* +1 → *m* (*m* ≥ 0)
      3. ⎧⎨ Zero 0 → true

⎩ Zero *m* +1 → false (*m* ≥ 0)

* + - 1. Yσ*M* → *M* (Yσ*M* )
      2. ((*λx*σ*.M* )*N* ) → *M* [*N/x*σ]

i i

*M* → *M* '

(*MN* ) → (*M* '*N* )



*N* → *N* '

(*MN* ) → (*MN* ')

(*M* ∈ {if*,* succ*,* pred*,* Zero})

Let →٨

denote the *reflexive and transitive* closure of →. Then we can define

the operational semantics by a partial function Eval which gives constants from programs:

Definition 1.3 [operational semantics for PCF : Eval]

Eval(*M* ) = *c* iff *M* →٨

*c,* for any program *M* and constant *c*

A closer look at the rules for → reveals that for each term there is *at most* one immediate reduction rule applicable. In particular → is a *partial function* which is undefined on constants. This implies that Eval is *well-deﬁned*, i.e.

*M* →٨ *c* and *M* →٨ *c*' implies that *c* and *c*' are identical.

* + 1. *Denotational Semantics of PCF*

We will use some mathematical structures called *cpo’s* in our treatment of the denotational semantics. Let us briefly go through some definitions and facts. For a more comprehensive account of cpo’s, see [[2](#_bookmark71)].

Let *D* be a set and ± ⊆ *D* × *D* a binary relation over *D*. (*D,* ±) is called a *partial order* if it satisfies the following:

* *reflexivity* ∀*x* ∈ *D* : *x* ± *x*
* *anti symmetry* ∀*x, y* ∈ *D* : *x* ± *y* and *y* ± *x* ⇒ *x* = *y*
* *transitivity* ∀*x, y, z* ∈ *D* : *x* ± *y* ± *z* ⇒ *x* ± *z*

Where there is no confusion, we simply write *D* instead of (*D,* ±).

Notation 1.4 *Throughout this paper a* и *b means a is strictly less than b, i.e:*

*a* и *b* ⇔ *a* ± *b* and *a* /= *b*

Definition 1.5 [bounded (consistent) subsets] Let (*D,* ±) be a partial order. Then *B* ⊆ *D* is bounded (or consistent) if it has an upper bound, i.e.:

∃*d* ∈ *D* : ∀*b* ∈ *B* : *b* ± *d*

We use the abbreviation *a* ↑ *b* for “{*a, b*} is bounded”.

Definition 1.6 [bounded complete] A partial order (*D,* ±) is said to be bounded complete if any of its bounded subsets has a *least* upper bound in *D*.

For *X* ⊆ *D* we write H*X* for the least upper bound (l.u.b.) of *X*, and H*X* for the greatest lower bound (g.l.b.) of *X* provided they exist. In case *X* has only two elements, we use the infix notation, i.e. we write *a*H*b* for H{*a, b*} and *a*H*b* for H{*a, b*}.

A subset *X* of *D* is said to be directed if it is non-empty and every pair of its elements have an upper bound in *X* itself, i.e:

∀*x, y* ∈ *X* : ∃*z* ∈ *X* : *x* ± *z* ∧ *y* ± *z*

We often write *X* ⊆*dir D* to abbreviate ‘*X* is a directed subset of *D*’.

Definition 1.7 [cpo] A partial order (*D,* ±) is said to be a complete partial order (cpo for short) if it satisfies the following:

1. *D* has a least element ⊥D under ±.
2. Every directed subset *X* of *D* has a least upper bound which we denote by HD*X* or just H*X* where there is no confusion.

Definition 1.8 [finite element] Let (*D,* ±) be a cpo and *a, b* ∈ *D*. *a* is said to be way-below *b* (or *a* approximates *b*) — written as *a* *b* — if the following is true:

∀*X* ⊆*dir D* : *b* ± HD*X* ⇒ ∃*x* ∈ *X* : *a* ± *x*

An element *d* ∈ *D* is called finite (or compact) if *d* *d*.

Definition 1.9 [basis] Let (*D,* ±) be a cpo. *B* ⊆ *D* is a basis for *D* if:

∀*x* ∈ *D* : ({*y* ∈ *B* | *y* *x*} ⊆*dir D*) and ( *x* = H{*y* ∈ *B* | *y* *x*} )

A cpo (*D,* ±) is called continuous if *D* is a basis for itself. Throughout this paper, by domain we mean continuous cpo. [3](#_bookmark12)

Definition 1.10 [algebraic cpo] A cpo (*D,* ±) is called algebraic if the col- lection of its finite elements forms a basis. In that case, we denote the basis by *K*(*D*).

Definition 1.11 [*ω*-continuous, *ω*-algebraic cpo] A cpo is called *ω*-continuous if it has a countable basis. If (*D,* ±) is algebraic and *K*(*D*) is countable, then (*D,* ±) is said to be *ω*-algebraic.

Definition 1.12 [continuous function between cpo’s] A function *f* : (*D,* ±D) →

(*E,* ±E) is continuous if:

1. *f* is monotone:

∀*x, y* ∈ *D* : *x*±D*y* ⇒ *f* (*x*)±E*f* (*y*)

1. *f* preserves the suprema of directed sets:

∀*X* ⊆*dir D* : *f* (HD*X*) = HE*f* (*X*)

The collection of all the continuous functions from *D* to *E* under the induced pointwise ordering forms a cpo which is often written as [*D* → *E*].

Of special interest are the two so-called *flat* cpo’s of truth values and natural numbers, ⊥ and N⊥ respectively, defined as follows:

Example 1.13 (i) Let N⊥ = {⊥} ∪{0*,* 1*,* 2*, ...*}, partially ordered as follows:

*x* ± *y* ⇔ (*x* = *y* or *x* = ⊥)

0¬zz1 z 2

. . . n . . .

zzz` z z

⊥

N⊥ is a cpo which is often called the flat domain of natural numbers.

3 Note that different people use different definitions for the concept of a domain.

(ii) Let ⊥ = {⊥*, tt , ff* } partially ordered as follows:

*x* ± *y* ⇔ (*x* = *y* or *x* = ⊥)

*tt ff*

` /’/

⊥

⊥ is also a cpo which is often called the flat Boolean domain.

By a *standard collection of domains for PCF* we mean a family {*D*σ} of cpo’s, one for each PCF-type *σ* ∈ such that:

* + *D*bool = ⊥
  + *D*nat = N⊥
  + *D*σ→τ = [*D*σ → *D*τ ]

The aim is to take {*D*σ} — the standard collection of domains for PCF — as a model and interpret the terms of PCF inside that model. Let us proceed step by step in order to make things clear. First we show how the constants are going to be interpreted via the function A which is type-respecting, i.e:

∀*c*σ ∈ CA : A(*c*) ∈ *D*σ

Definition 1.14 A: CA → ∪{*D*σ} is defined by:

A(true) = *tt*

A(false) = *ff*

A(*n*) = *n* (*n* ≥ 0)

⎧ *x* (if *p* = *tt* )

⎪⎪⎨

A(ifσ)(*p*)(*x*)(*y*) = *y* (if *p* = *ff* ) (*p* ∈ ⊥*, x, y* ∈ *D*σ and *σ* ground)

⎪⎪⎩ ⊥ (if *p* = ⊥)

A(succ)(*x*) = ⎧⎨ *x* +1 (*x* ≥ 0)

⎩⎧ ⊥ (*x* = ⊥)

(*x* ∈ N⊥)

A(pred)(*x*) = ⎨ *x* − 1 (*x* ≥ 1)

⎧⎩ ⊥ (*x* ∈ {⊥*,* 0})

⎪⎪⎨ *tt* (*x* = 0)

(*x* ∈ N⊥)

A(Zero)(*x*) =

*ff* (*x >* 0)

⎪⎪⎩ ⊥ (*x* = ⊥)

(*x* ∈ N⊥)

A(Yσ)(*f* ) = H{*f* n(⊥) | *n* ≥ 0} (*f* ∈ *D*σ→σ)

Terms are interpreted with respect to *environments*. An environment is simply a type-respecting function from the set of variables to the model

∪{*D*σ}. We let Env be the set of all the environments, ranged over by *ρ*. Hence for any *ρ* ∈ Env:

*ρ* : *Var* → ∪{*D*σ}

*ρ*(*x*σ) ∈ *D*σ

i

For any *ρ* ∈ Env*, x*σ ∈ *Var , d* ∈ *D*σ we let *ρ*x*σ* '→d denote the environment *ρ*'

such that:

i

*ρ*'(*x*) =

*i*

⎧⎨ *d* (*x* = *x*σ)

i

⎩ *ρ*(*x*) (*x* /= *x*σ)

i

Now we have all the necessary material to define the *denotational semantics*

*.*) : T → (Env → ∪{*D*σ}) by:

Definition 1.15 [denotational semantics of PCF]

*x*σ)(*ρ*) = *ρ*(*x*σ) (*x*σ ∈ *Var* )

i i i

*c*)(*ρ*) = A(*c*) (*c* ∈ CA)

(*MN* ))(*ρ*) = *M* )(*ρ*)( *N* )(*ρ*))

(*λx*σ*.M* ))(*ρ*)(*d*) = *M* )(*ρ*x*σ* '→d) (*d* ∈ *D*σ)

i *i*

* + 1. *Matching Operational and Denotational Semantics: the necessity of parallel operators*

Having both operational and denotational semantics side-by-side gives us a handy tool for studying properties of objects in one area by analysing the corresponding objects in the other. One such case — indeed an important one — is proving properties of programs via their corresponding object in the model. For instance, if *M* and *N* are programs written in PCF, one can prove their “equivalence” via their interpretation in the model. This in turn necessitates the two semantics to match up to a certain degree. This is an important subject with a rich literature available. Here we discuss this issue as far as needed for our own purposes. For a thorough treatment together with the proofs and details, again see [[13](#_bookmark79)] from which we will take much of our material, unless stated otherwise.

Perhaps the following theorem ( [[13](#_bookmark79)]) is a good place to start with. Here,

⊥ˆ denotes the environment which maps every variable to the bottom element of the corresponding cpo, i.e.

∀*x*σ ∈ *Var* : ⊥ˆ (*x*σ) = ⊥D*σ*

i i

Theorem 1.16 *For any PCF-program M* : *σ and constant c* : *σ:*

Eval(*M* ) = *c* ⇔ *M* )(⊥ˆ ) = A(*c*)

To prove (⇒) direction, one needs to observe the so-called soundness of the operational semantics with respect to the denotational one, i.e:

if *M* → *N* then *M* )(*ρ*) = *N* )(*ρ*) For (⇐) direction, see [[13](#_bookmark79)].

This theorem demonstrates how the *behaviour* of a program regarding its

*termination* and the constant it evaluates to is reflected in the denotational semantics. Now let us investigate the *equivalence* of programs. For that matter we need the following definition:

Definition 1.17 The set C of contexts of PCF with numbered holes — ranged over by *C* — is generated by the grammar:

*C* ::= [ ] | *x*σ | *c* | *CC* | *λx*σ*.C*

j i i

where *j* ∈ N*, x*σ ∈ *Var* and *c* ∈ CA.

i

If all the occurrences of the holes bear the same subscript in a term, we denote them [ ] for short.

In other words, contexts are just terms with holes in them. We usually write a context *C* as *C*[*.,... , .*] not to get confused with ordinary terms. These holes can be filled with terms of appropriate type to give a term. We denote a

context *C*[*.,... , .*] filled with the terms *M*1*,... , M*k by *C*[*M*1*,... , M*k]. Below we define the operation of *ﬁlling the holes of a context*. We abbreviate a vector

→−

of the form [*N*1*,... , N*n] simply as [*N* ]:

[ ]j[*N*1*,... , N*n] = *N*j

*x*σ →− σ

i [*N* ] = *x*i

→−

*c*[*N* ] = *c*

→− →− →−

(*C*1*C*2)[*N* ] = *C*1[*N* ] *C*2[*N* ]

(*λx*σ*.C* →− ] = *λx*σ*.*(*C*[→−

i )[*N* i *N* ])

As our main objects of interest are programs, we regard two terms *M*

and *N* as *operationally equivalent* — written as *M*

=∼ *N* — if they can be

substituted for each other in any program without affecting its behaviour:

Definition 1.18 [operational equivalence] *M* ∼= *N* if and only if for any con- text *C*[*.,... , .*] such that *C*[*M,... ,M* ] and *C*[*N,... ,N* ] are programs either both of Eval(*C*[*M,... ,M* ]) and Eval(*C*[*N,... ,N* ]) are undefined or else both are defined and equal.

It is easy to check that =∼ is an equivalence relation.

One of the reasons we define a denotational semantics for a language is to be able to resort to it as an easier alternative to the (usually) tedious op- erational semantics when it comes to proving the equivalence of programs, provided the equivalence is reflected in the denotational semantics. Unfortu- nately this is not the case with PCF and its model as we have defined it.

Take the two terms *M*0 and *M*1 defined as: [4](#_bookmark16)

*M*i = *λx.*ifnat(*x* true Ωbool){ifnat(*x* Ωbool true) [ifnat(*x* false false) Ωnat *i* ] [Ωnat]} {Ωnat}

4 taken from [[13](#_bookmark79)].

where *i* ∈ {0*,* 1}. Here *x* is of type bool → (bool → bool), Ωbool and Ωnat are non-terminating terms of types bool and nat respectively. They could be:

Ωbool = Ybool(*λx*bool*.x*) Ωnat = Ynat(*λx*nat*.x*)

It needs quite some effort to grasp what the *M*i’s do in the first place, and then the proof that they are operationally equivalent is a bit involved (see [[13](#_bookmark79)]). Anyway the important fact for us is that:

*M*0 =∼ *M*1

On the other hand:

*M*0)(⊥ˆ ) /= *M*1)(⊥ˆ ) (1)

The best way to verify the above inequality is to find and argument in the model over which *M*i)(⊥ˆ )*,* (*i* ∈ {0*,* 1}) disagree.

Definition 1.19 The function *p*ˆ*or* : ⊥ → ( ⊥ → ⊥) is defined by:

⎧ *tt* (*x* = *tt* ∨ *y* = *tt* )

ˆ ⎪

⎪⎪⎨

*por x y* := *ff* (*x* = *y* = *ff* )

⎪⎩ ⊥ otherwise

It is not difficult to see that *p*ˆ*or* ∈ *D*bool→(bool→bool), and to verify that:

*M*0)(⊥ˆ )(*p*ˆ*or* ) = 0

*M*1)(⊥ˆ )(*p*ˆ*or* ) = 1

hence the inequality ([1](#_bookmark17)).

Observing this mismatch, we say that the model is not *fully abstract* for the language. Full abstraction is an important criterion regarding the relation between a language and its model, and in our case in order to achieve full abstraction there could be two alternative ways ahead: either to purge the

model from troublesome objects like *por* or otherwise enrich the language.

ˆ

Both possibilities have been pursued but here we follow Plotkin’s direction of enriching the language [[13](#_bookmark79)] in order to get to the parallelism issues.

Let us first add constants:

pifo : bool → *o* → *o* → *o* (*o* ∈ {bool*,* nat})

to the set CA to get the extended set of constants CA+ and call the extended language based on that PCF+. Then in order to get an operational semantics for PCF+ we extend the relation → of Definition [1.2](#_bookmark3) (see page [4](#_bookmark3)) by the follow-

ing rules for pifo, (*o* ∈ {bool*,* nat}), and denote the new immediate reduction

relation by →+:

⎧ pifo *P M M* →+ *M*

⎪⎪⎨

(*ix*) pifo true *M N* →+ *M*

⎪⎪⎩ pifo false *M N* →+ *N*

⎪ pifo *P* →+ pifo *P* '

⎧⎪ *P* →+ *P* '

(*x*)

⎪⎪⎨

*M* →+ *M* '

⎪ *N* → *N* '

⎪

pifo *P M* →+ pifo *P M* '

+

⎪⎩ pifo *P M N* →+ pifo *P M N* '

⎪

Let us mention some notes about the constants pifo, (*o* ∈ {bool*,* nat}). As

you might have guessed, the prefix p stands for *parallel*, hence we read pif as “parallel-if”. A closer look at the rules (*ix*) and (*x*) reveals the parallelism as

pifo looks at its three arguments at the same time. Rule (*x*) consists of three rows, with the first one being the only one having a counterpart for if, see rule (*viii*), Definition [1.2](#_bookmark3), page [4](#_bookmark3).

The extension of →٨ to →٨ and Eval to Eval+ is straightforward and left

+

to the reader. Although →+ is *non-deterministic*, still it can be proved that

٨

→+ has the so-called *Church-Rosser* property, i.e.

∀*M ,M ,M*

: *M* →٨ *M* and *M* →٨ *M*

1 2 3

1 + 2

1 + 3

⇒ ∃*M, M* ' : *M* =∼ *M* ' and (*M* →٨ *M* and *M* →٨ *M* ')

α 2 + 3 +

where ∼=α is the *α*-equivalence between terms, i.e. equivalence up to renaming of bound variables.

This implies that Eval+ is again well-defined.

We extend A of Definition [1.14](#_bookmark13) (page [7](#_bookmark13)) to A+ : CA+ → ∪{*D*σ} by:

A+(*c*) = A(*c*) (*c* ∈ CA)

⎧⎪ *x* (*p* = *tt* ) (A)+(pif )(*p*)(*x*)(*y*) = ⎪⎨ *y* (*p* = *ff* )

o

*x* (*p* = ⊥ and *x* = *y*)

⎪⎪⎩ ⊥o (*p* = ⊥ and *x* /= *y*)

(*p* ∈

⊥ and *x, y* ∈ *D*o)

The definition of the denotational semantics *.*) for PCF+ based on A+ should

+

be straightforward.

Remark 1.20 Here we tried to follow Plotkin’s original definitions and there- fore added both pifbool and pifnat where any of the two would suffice. In fact we could have just as well added a constant por: bool → (bool → bool) with

the following immediate reduction rule:

⎧ por true *P* → true por *P* true → true

⎪⎪⎨

⎪⎪⎩ por false false → false

and extended A to A+ by:

A+(por) = *por*

ˆ

where *por* is the function defined in Definition [1.19](#_bookmark18), page [11](#_bookmark18). For a proof of the interdefinability of pifbool, pifnat and por in PCF, see [[15](#_bookmark85)].

ˆ

Now the relation between the operational and denotational semantics is much better as the following theorem shows:

Theorem 1.21 *[*[*13*](#_bookmark79)*]*

*For any* PCF+*-terms M*σ *and N*σ*:*

*M*σ ∼= *N*σ ⇔ ∀*ρ* ∈ Env : *M*σ)+(*ρ*) = *N*σ)+(*ρ*) In fact we have more:

Theorem 1.22 *[*[*13*](#_bookmark79)*]*

*Every ﬁnite element (see Deﬁnition* [*1.8*](#_bookmark7)*, page* [*5*](#_bookmark7)*) of any D*σ *is deﬁnable by a* PCF+*-term.*

It seems intuitively reasonable to take the l.u.b.’s of recursively enumerable sets of finite elements of the model as the collection of *computable* elements. Any element defined by a PCF+-term is computable (see [[13](#_bookmark79)]) and as Theorem

[1.22](#_bookmark19) says, any finite element of the model is captured by the language PCF+.

for in the language. One such object is ^∃∈ *D*(nat→bool)→bool defined by: But it turns out that there are computable objects of the model not accounted Definition 1.23 [continuous existential quantifier : ^∃]

⎧⎪ *ff* (*g*(⊥) = *ff* )

⎪⎨

^∃(*g*) = *tt* (*g*(*n*) = *tt* for some *n* ≥ 0)

⎪

⎪⎩ ⊥ (otherwise)

We then proceed by adding a constant ∃: (nat → bool) → bool to CA+

and denote the new set of constants as CA++. We call the language based on this set PCF++. The following reduction rule is added to →+ to obtain →++.

Note that →٨ is the transitive-reflexive closure of → :

++

++

⎧⎪ *F* Ωnat

→++

٨

false

(*xi*) ⎪⎨

∃*F* →++ false

⎪⎩ ∃*F* →++

⎪ ++

*Fm* →٨ true

(*m* ≥ 0)

true

It can be shown that ∃*F* →++ true and ∃*F* →++ false cannot both hold at the

same time, moreover →٨ satisfies Church-Rosser. Therefore Eval defined

++

++

over PCF++ programs by:

Eval (*M* ) = *c* iff *M* →٨ *c*

++ ++

is well-defined.

Finally we define A++ : CA++ → ∪{*D*σ} by:

A++(*c*) = A+(*c*)*,* (*c* ∈ CA+)

A++(∃) = ^∃

Again extending *.*)+ to a denotational semantics *.*)++

is straightforward.

for PCF++ using A++

Now the language is rich enough to define all computable elements of the model:

Theorem 1.24 ( [[13](#_bookmark79)]) *An element of D*σ *is computable if and only if it is deﬁnable by a* PCF++*-term.*

* + 1. *Theory versus Practice*

Designing a language and its operational semantics, presenting a model, in- terpreting the language inside it and studying the relation between the oper- ational and denotational semantics — one might like to categorize them as

*theoretical issues* — are just part of a bigger challenge, though quite an im-

portant one. In practice other issues arise such as efficiency regarding time and/or space. Though in Part [1.1.3](#_bookmark15) we tried to summarize the theoretical issues and demonstrate the success in that regard, we never cared about the efficiency of computations. Whenever we felt a deficiency, we did not hesitate to remedy by any means.

First consider PCF and the immediate reduction rules over its terms (Defi- nition [1.2](#_bookmark3), page [4](#_bookmark3)). For any term, there is *at most* one rule that applies. Hence the operational semantics is *deterministic*. Imagine a machine implementing PCF, in the middle of a computation, reducing a term *C*[*M*1*,... , M*i*,... , M*k].

If *M*i is the subterm being worked on at the moment, there is no way the pro- cess will jump to another subterm *M*j (*j* /= *i*) unless the computation on *M*i is finished off with *M*i being evaluated to a constant. We try to formulate a prop- erty that captures this intuition and call a language satisfying this property

*sequential*.

Sequentiality can be studied both syntactically and semantically. Plotkin’s *activity lemma* [[13](#_bookmark79)] is an example of a syntactic formulation, Berry’s *syntactic sequentiality theorem* [[4](#_bookmark74)] [5](#_bookmark22) is another, which for the reader can serve as a good

motivation for a semantic investigation of sequentiality first introduced by Vuillemin [[16](#_bookmark86)], though in reality Vuillemin’s work preceded that of Berry’s.

Here in this paper we pursue Vuillemin’s semantic approach originally de- fined for functions over *flat cpo’s*:

Definition 1.25 [flat cpo] Given any nonempty set *X*, (*X*⊥*,* ±) defined by:

*X*⊥ = *X* ∪ {⊥} (where ⊥ ∈*/ X*)

and

∀*x, y* ∈ *X*⊥ : *x* ± *y* ⇔ (*x* = ⊥ ∨ *x* = *y*)

is a cpo. We call cpo’s of this shape flat cpo’s.

Examples of flat cpo’s we use are ⊥ and N⊥ (Example [1.13](#_bookmark11), page [6](#_bookmark11)) and (Definition [3.2](#_bookmark59), page [33](#_bookmark59)).

5 see [[1](#_bookmark72), section 2.4, page 41] for an English version.

Definition 1.26 [sequential function (Vuillemin)] Let *D, D*1*,... , D*n be flat cpo’s, and let *f* : *D*1 × ··· × *D*n → *D* be continuous. Let →−*x* = (*x*1*,... , x*n) ∈ *D*1 × ··· × *D*n, and suppose that *f* (→−*x* ) = ⊥. We say that *f* is sequential at

→−*x* if either *f* (→−*z* ) = ⊥ for all →−*z* ± →−*x* , or there exists *i* such that *x*i = ⊥ and:

∀→−*y* = (*y*1*,... , y*n): (→−*y* N →−*x* and *f* (→−*y* ) /= ⊥) ⇒ *y*i /= ⊥

We say then that *i* is a sequentiality index for *f* at →−*x* .

*f* is sequential if it is sequential at all →−*x* in its domain.

Note that Vuillemin-sequentiality is defined *only for ﬁrst order functions over flat cpo’s*. In fact the definition as it is cannot be generalized to higher types because a function like ∃ which is intuitively of an infinite parallel nature

^

would become Vuillemin-sequential. But at first order the definition works well as it can be shown that for compact first order functions *f* in ∪{*D*σ}, Vuillemin-sequentiality coincides with PCF-definability (Theorem [1.31](#_bookmark27), page [18](#_bookmark27)).

Also of interest is the Vuillemin-sequentiality of all unary first order func- tions over flat cpo’s. This can be used to show that although all first order PCF-definable functions in ∪{*D*σ} are Vuillemin-sequential [[1](#_bookmark72), Exercise 6.5.5, page 137], the converse is not true as there are uncountably many elements in

*D*nat→nat, all of which are Vuillemin-sequential, while there are only countably

many PCF-definable elements in there.

Equally interesting, at least for our purposes, is Sieber’s approach [[14](#_bookmark80)] which proves to have a tight relation to Vuillemin’s definition, as we shall see

in Theorem [1.31](#_bookmark27), page [18](#_bookmark27). But before that, we need to take a look at an important tool called *logical relations*.

* 1. *Logical Relations*

Notation 1.27 *Throughout this paper, by* Λ(*C*) *we mean the extension of the simply-typed λ-calculus with a set of constants C.*

Definition 1.28 [Logical Relations] Let Mi, (1 ≤ *i* ≤ *k*) be *k* models of

Λ(*C*), and *D*σ = *σ*)

i

M

*i*

(where *σ* ranges over types), and for any ground type

*o*, *R*o ⊆ *D*o × ··· × *D*o. Then a *k*-ary logical relation *R*k between M1*,... ,* Mk

k 1 k

can be built up from *R*o’s by the following definition for function type cases: for any *f*1*,... , f*k ∈ *D*σ→τ :

k

(*f*1*,... , f*k) ∈ *R*σ→τ ⇔

k

∀(*x*1*,... , x*k) ∈ *R*σ : (*f*1(*x*1)*,... , f*k(*x*k)) ∈ *R*τ

k k

*f* ∈ *D*σ is said to *preserve* the logical relation *R*k if and only if:

(*f,... ,f* ) ∈ *R*σ

k

*R* is said to be a *C*-logical relation if for any *c* : *σ* in *C*:

( *c*)M1

*,... ,* *c*)M*k*

) ∈ *R*σ

Logical relations are useful especially when it comes to establishing links between syntax and semantics as in Jung-Tiuryn’s theorem on lambda-definability [[11](#_bookmark81)]. Of course Jung and Tiuryn used a more powerful class of logical relations called

k

*Kripke logical relations* which unlike our definition, have varying arities.

Anyway Definition [1.28](#_bookmark24) is enough for our purposes. The important part of defining a logical relation is over the ground types, as that is the part over which we have control. Then having defined a suitable logical relation, we make extensive use of the following important lemma:

Lemma 1.29 (Fundamental lemma of logical relations) *Let R be a k- ary C-logical relation, C a set of constants, between k models of* Λ(*C*)*, namely* M1*,... ,* Mk*. Then for any closed* Λ(*C*)*-term M of type σ we have:*

( *M* ) *,... ,* *M* ) ) ∈ *R*σ

M M

1 *k*

Note 1 *To ﬁnd out more about logical relations as presented here, including a proof of Lemma* [*1.29*](#_bookmark25)*, see [*[*1*](#_bookmark72)*, chapter 4, section 5].*

For an example of logical relations, let us mention an important class of logical relations known as *Sieber-sequential relations*:

Definition 1.30 [Sieber-sequential relations] Let *A* ⊆ *B* ⊆ {1*,... , n*} and

consider the following *n*-ary relations Sn over ground types *o* ∈ {nat*,* bool}:

A,B

(*d*1*,... , d*n) ∈ Sn

A,B

⇔ (∃*i* ∈ *A* : *d*i = ⊥) or (∀*i, j* ∈ *B* : *d*i = *d*j)

A Sieber-sequential relation is an *n*-ary logical relation S such that So is

an intersection of relations of the form Sn .

A,B

It can be shown that *C*-logical relations, where *C* is the set of all PCF con- stants CA (see page [2](#_bookmark1)) are exactly the Sieber-sequential relations of Definition

[1.30](#_bookmark26) (see [[1](#_bookmark72), Exercise 6.5.3, page 136]).

Now let us take some special cases of Sieber-sequential relations, namely

k+1

*S*k+1 := S

{1,...,k},{1,...,k+1}

, (*k* ≥ 1):

(*x*1*,... , x*k+1) ∈ *S*k+1 ⇔ (∃*j* ≤ *k* : *x*j = ⊥) or (*x*1 = ··· = *x*k+1 /= ⊥)

which help us demonstrate the relation between Sieber’s and Vuillemin’s ap- proaches to sequentiality:

Theorem 1.31 *For a compact ﬁrst order function f in* ∪{*D*σ}*, the following* *are equivalent:*

1. *f is Vuillemin-sequential.*
2. *f is deﬁnable in PCF.*
3. *f is invariant under all k* + 1*-ary relations S*k+1*.*

For a proof of this theorem the reader can refer to [[1](#_bookmark72), Theorem 6.5.4, page 137].

* 1. *Computation on Real Numbers: Real-PCF*

PCF-programs are meant to output constants of the ground types bool or nat. Of course it is obvious that there are many more collection of objects over which we like to do computation. One such important case is the set of *real numbers*.

Traditionally we are used to computation on real numbers via *floating point approximations* which is satisfactory for everyday business but can prove to be extremely unreliable in special circumstances. Floating point computa- tion carries the problem of *round-off errors* with it, which we try to ignore in everyday life applications for a variety of reasons. In [[12](#_bookmark82)] this subject is ex- plored together with two interesting examples demonstrating the *unreliability of floating point approach*. Accordingly the idea of *exact real number computa- tion* has been put forward which is, as opposed to floating point computation, reliable, i.e. the output produced is guaranteed to be correct. Moreover the results can be computed *effectively* (e.g. as opposed to BSS approach [[5](#_bookmark75)]) to within any desired degree of accuracy.

Exact real number computation itself can be approached in two ways. At first people focused on *representation* while neglecting the issue of *datatypes* for real numbers, among which [[3](#_bookmark73)] is considered seminal. On the other hand,

perhaps [[10](#_bookmark83)] is among the earliest works where there is a clear distinction be- tween a representation-dependent operational semantics and a representation- independent denotational semantics. Di Gianantonio added to PCF a ground type which is interpreted as a domain of real numbers. This domain turns

out to be algebraic (see Definition [1.10](#_bookmark8), page [6](#_bookmark8)) and therefore *cannot have the*

*real line as its subspace of maximal elements*. This creates the possibility of defining functions not extensional over real numbers ( [[10](#_bookmark83), page 62] [6](#_bookmark29) ).

6 Of course on the same page, Di Gianantonio himself claims to have fixed the problem,

Mart´ın Escard´o introduced Real-PCF following similar ideas [[8](#_bookmark78)]. He added to PCF a ground type for real numbers interpreted as the so-called *unit in- terval domain* (see Definition [1.34](#_bookmark31), page [20](#_bookmark31)) which has the interval [0*,* 1] as its subspace of maximal elements. Also the problem of non-extensionality with Di

Gianantonio’s approach is avoided in Real-PCF. Of course there is much more to both Di Gianantonio’s and Escardo´’s works. Here we present an overview of Real-PCF as it is the necessary background to the rest of the paper.

Definition 1.32 [Real-PCF] The set of Real-PCF types is generated by the grammar:

*σ* ::= bool | nat | *I* | *σ* → *σ*

and the set of Real-PCF constants RCA is the extension of CA (see page [2](#_bookmark1)) with the following constants:

consa : *I* → *I* taila : *I* → *I* headr : *I* → bool

pif*I* : bool → *I* → *I* → *I*

The aim is to take the ground type *I* as the type of *real numbers in the unit interval, i.e.* [0*,* 1] and use the constants introduced in the definition for computation over them. In the above definition *a* ranges over intervals with

rational end-points in [0*,* 1], i.e.

*a* ∈ {[*p, q*] | *p, q* ∈ Q ∩ [0*,* 1]*,p* ≤ *q*}

These end-points must be distinct when *a* is a subscript for tail, i.e:

*p < q*

and

*r* ∈ Q ∩ (0*,* 1)

Notation 1.33 *We freely use the abbreviation RPCF for Real-PCF.*

The definition of the following terms for RPCF setting should be straight- forward now and we omit them here. Moreover we abuse the notation where there is no confusion and use these terms for the meanings mentioned below:

but the author still believes that is not the case.

1. T : the set of *RPCF-terms*.
2. *Var* : the set of *RPCF-variables*.
3. FV(*M* ) : the set of the *free variables of the RPCF-term M* .
   * 1. *Denotational Semantics: Interval Domain Model*

Let us begin with the ground type *I* whose denotation *DI* we take to be:

Definition 1.34 [unit interval domain] The cpo (I*,* ±I) defined by:

I = {[*r, s*] | *r, s* ∈ R*,* 0 ≤ *r* ≤ *s* ≤ 1}

and

∀*a, b* ∈I : *a* ±I*b* ⇔ *a* ⊇ *b*

is called the unit interval domain.

For simplicity, we denote [*r, r*] by r, and in the other direction as well we talk about an element like *r* ∈ [0*,* 1] where we really mean [*r, r*] ∈ I.

We want the elements of [0*,* 1] to be *maximal*, and use the *rational intervals*

to approximate them. To make a proper distinction, we refer to the maximal elements as *total* real numbers whereas we call the others *partial* real numbers. As a smaller interval is a better approximation to a number than a bigger one, we want to have the *superset* relation to be the order on the intervals, hence the definition of ±I.

It is not difficult to show that (I*,* ±I) is a cpo where supremum operation is simply defined to be *the set-theoretic intersection*, i.e.

∀*X* ⊆*dir* I : H*X* = ∩*X*

In fact (I*,* ±I) is bounded complete as well:

∀*X* ⊆ I : *X* bounded ⇒ *l .u.b. X* = ∩*X*

Moreover the countable set:

Io := {[*r, s*] ∈I | *r, s* ∈ Q}

forms a basis (Definition [1.9](#_bookmark9), page [6](#_bookmark9)) for I and makes it an *ω-continuous cpo*

(Definition [1.11](#_bookmark10), page [6](#_bookmark10)).

We let I denote the datatype *I* , hence:

Definition 1.35 [collection of domains for RPCF] {*D*σ} is *a collection of domains for RPCF* if:

*D*bool = ⊥ *D*nat = N⊥ *DI* = I

*D*σ→τ = [*D*σ → *D*τ ]

As we did for PCF (Part [1.1.2](#_bookmark6)) we first try to interpret the constants via a function RA : RCA → ∪{*D*σ} and then extend it in a natural way to a denotational semantics. For any constant *c* ∈ CA, we simply define RA(*c*) := A(*c*), where A is defined as in Definition [1.14](#_bookmark13), page [7](#_bookmark13).

Remark 1.36 With each RPCF-type *σ* which is not a PCF-type, a new fix- point combinator Yσ is added to the language, often without our notice!.

Anyway, the interpretation is formulated as in Definition [1.14](#_bookmark13), page [7](#_bookmark13) for PCF-types, i.e.

RA(Yσ)(*f* ) = H{*f* n(⊥) | *n* ≥ 0} (*f* ∈ *D*σ→σ)

For the proper RPCF-constants, let us present their denotations in a more intuitive fashion. From a geometric point of view, it does not matter if we choose two other numbers *r < s*, rather than 0 and 1, and build our interval domain upon it. Let us suppose 0 ≤ *r* ≤ *s* ≤ 1 and denote the interval [*r, s*] by *a*, and the interval domain built upon it by *a*I, bearing in mind that *a*I is

the singleton domain in the case *r* = *s* . If *r < s* then RA(consa) is defined to

be the scaling isomorphism — denoted by *Cons* a — from I to *a*I, otherwise (i.e. *r* = *s*) it is simply the unique constant function available. We can simply

consider the codomain to be the whole of I, hence say RA(consa): I → I. There is apparently another scaling isomorphism from *a*I back to I (if *r* /= *s*)

that we call *Tail* a : *a*I → I. We can extend the domain of the function from *a*I to I so that it remains a morphism in the category CPO. In fact we consider the maximal extension (under the order relation on [I → I]) and for

simplicity denote it by the same name *Tail* a, hence *Tail* a : I → I, and we put

RA(taila) = *Tail* a. The following figure may give a better intuition:

Consa

*a*

Taila

*I I* *I*

Now perhaps the following formulae are easier to follow:

RA(consa)(*b*)= [*r* + (*s* − *r*)*x, r* + (*s* − *r*)*y*] (2)

RA(taila)(*b*)= [(*x*' − *r*)*/*(*s* − *r*)*,* 1 − (*s* − *y*')*/*(*s* − *r*)] (3) where:

*a* = [*r, s*]

*b* = [*x, y*]

⎧⎨ *x*' = min(max(*r, x*)*, s*)

⎩ *y*' = max(min(*s, y*)*, r*)

The denotation of the constants headr and pif*I* can be easily described

explicitly while their corresponding immediate reduction rule (see Definition [1.37](#_bookmark34), page [24](#_bookmark34)) tells all about their expected behaviour. Let *r* be a rational number 0 *< r <* 1:

⎧ *tt* if (*x* = [*x*1*, x*2] ∧ *x*2 *< r*)

⎪⎪⎨

∀*x* ∈I : [RA(headr)](*x*) = *ff* if (*x* = [*x*1*, x*2] ∧ *r < x*1)

⎪⎪⎩ ⊥ otherwise i*.*e *r* ∈ *x*

⎧ *x* if *p* = *tt*

⎪⎪⎨

∀*p* ∈ ⊥*,* ∀*x, y* ∈ I : [RA(pif*I* )](*p*)(*x*)(*y*) = *y* if *p* = *ff*

⎪⎪ *x*H*y* if *p* = ⊥

⎩

Having defined RA it is straightforward to define a denotational semantics

*.*) : T → (Env → ∪{*D*σ})

for RPCF, following the same style as we did for PCF in Definition [1.15](#_bookmark14), page

[9](#_bookmark14). Note that we have not modified any symbol from PCF to RPCF (except A to RA) as we are only dealing with RPCF throughout the rest of the paper, hence there should be no confusion.

Consider any two intervals *a, b* ∈ I, where *a* = [*a*1*, a*2] and *b* = [*b*1*, b*2] and define:

*ab* = [*a*1 + (*a*2 − *a*1)*b*1*, a*1 + (*a*2 − *a*1)*b*2] It is easy to verify that:

RA(consab) = RA(consa)*.*RA(consb) where *.* is just functional composition.

Also, consider *a, b* ∈ I, where *a* = [*a*1*, a*2] and *b* = [*b*1*, b*2] and this time subject to the conditions:

*a*1 /= *a*2 and *a* ± *b*

then there exists a unique *c* ∈ I such that *ac* = *b*, which we denote by *b* \ *a*. In fact:

*c* = [(*b*1 − *a*1)*/*(*a*2 − *a*1)*,* (*b*2 − *a*1)*/*(*a*2 − *a*1)]

Now it seems reasonable to use consa’s (*a* ∈ I , with rational end-points) as digits in order to represent real numbers in the interval [0*,* 1]. The idea is to represent any *shrinking* sequence of intervals with rational end-points by a sequence of the form:

(consa1 *,* consa1 consa2 *, ...,* consa1 *...*consa*n , ...*)

This sequence of intervals converges to an element *a* ∈I which can be partial or total depending on the nature of the sequence. We simply represent this element *a* by:

consa1 consa2 *...*consa*n ...*

* + 1. *Operational Semantics*

We extend the immediate reduction relation for PCF (Definition [1.2](#_bookmark3), page [4](#_bookmark3)) to one for RPCF and still denote it by →. The aim is to reduce any Real-PCF program *M* of type *I* to some consa*M* ' — where *a* has rational end-points —

and then continue reducing *M* ' to consa' *M* '', and so on. This way we produce

a stream of digits.

Before presenting the definition, take note of the following:

* + - 1. There are RPCF-types *σ* that are not PCF-types, correspondingly there are new *ﬁx-point combinators* Yσ : (*σ* → *σ*) → *σ* that are not PCF- constants. But still the reduction rule is the same as clause (*v*) of Defi-

nition [1.2](#_bookmark3), page [4](#_bookmark3), i.e.

Yσ*M* = *M* (Yσ*M* )

in particular:

Y*I* cons[0,1/2] = 0*,* Y*I* cons[1/2,1] = 1

* + - 1. For intervals *a, b* ∈I where *a* = [*a*1*, a*2] and *b* = [*b*1*, b*2] we define:

*a* ≤ *b* := *a*2 ≤ *b*1

* + - 1. Similarly for real number *r* and the interval *a* = [*a*1*, a*2] we define:

*a < r* := *a*2 *< r*

Definition 1.37 [immediate reduction relation for RPCF] The immediate reduction relation → is the extension of the corresponding relation from PCF (Definition [1.2](#_bookmark3), page [4](#_bookmark3)) to Real-PCF by the following rules:

* + - * 1. consa(consb*M* ) → consab*M*
        2. taila(consb*M* ) → Y*I* cons[0,1/2] (if *b* ≤ *a*)
        3. taila(consb*M* ) → Y*I* cons[1/2,1] (if *a* ≤ *b*)
        4. taila(consb*M* ) → consb\a*M* (if *a* ± *b* and *a* /= *b*)
        5. taila(consb*M* ) → cons(aHb)\a(tail(aHb)\b*M* ) (if *a* † *b, a* /± *b, b* /± *a, a* /≤ *b, b* /≤ *a*)
        6. headr(consa*M* ) → true (if *a < r*)
        7. headr(consa*M* ) → false (if *a > r*)
        8. pif true *M N* → *M,* pif false *M N* → *N*
        9. pif *L* (consa*M* ) (consb*N* ) →

consaHb(pif *L* (consa\(aHb)*M* ) (consb\(aHb)*N* )) (if *a*H*b* /= ⊥)

*N* → *N* '



*MN* → *MN* '

*M* → *M* '

(*M* ∈ {consa*,* taila*,* headr*,* pif*I* })

*N* → *N* '

pif*I L M* → pif*I L M* '

pif*I L M N* → pif*I L M N* '

We denote the reflexive and transitive closure of → by →∗.

Definition 1.38 [operational semantics for RPCF] The map Eval (Definition [1.3](#_bookmark5), page [4](#_bookmark5)) can be extended to a partial map over RPCF-programs (which we still denote by Eval) by the following case for programs *M* of type *I* :

Eval(*M* ) := {*a* ∈f | *M* →∗ consa*M* '*,* for some *M* '}

Note 2 *For a more precise and also comprehensive treatment of Real-PCF, see [*[*8*](#_bookmark78)*].*

# Sequentiality and Parallelism in RPCF

Unlike PCF, RPCF has a parallel operator as a constant, i.e. pif *I* . By the time Mart´ın Escard´o put forward RPCF [[8](#_bookmark78)], it was already speculated that

representation-independent real number computation needs parallel operators ( [[3](#_bookmark73),[10](#_bookmark83)]). Therefore pif*I* was included in RPCF right from the beginning. In [[8](#_bookmark78)]

Escardo´ shows that all computable elements of type at most 1 in the interval

domain model are definable in RPCF. Of course, like PCF, there are higher order objects such as ∃ (Definition [1.23](#_bookmark20), page [14](#_bookmark20)) not definable in RPCF. But by adding a constant ∃ for existential quantification, the language becomes

^

*universal* for the model, i.e. all computable objects of any order in the interval

domain model are definable ( [[7](#_bookmark77)]).

Though adding parallel operators to the language solves the definability

problems, they come at a heavy cost, i.e. the issue of efficiency in practice. Therefore we would rather have a more efficient substitute for pif *I* . We need

to analyze the language and its model more carefully to have a better view of our choices. One might think of getting rid of any kind of parallelism in the language. This idea was ruled out by Escardo´, Hofmann and Streicher in [[6](#_bookmark76)] where they proved that even if functions as basic as any continuous extension of mediation ⊕: [0*,* 1] → [0*,* 1] → [0*,* 1] defined by:

∀*x, y* ∈ [0*,* 1] : *x* ⊕ *y* := *x* + *y*

2

(4)

to the whole interval domain f, are going to be definable in the language, then the existence of some parallel mechanism is necessary. Of course this result crucially depends on some specific conditions, some of which need not necessarily hold in an ideal setting. For example, in the interval domain model

— the model in which this result was studied — all functions on real numbers are *extensional* at both partial and total real numbers, i.e. for any RPCF-type *σ*:

∀*f* ∈ *DI* →σ*,x* ∈ f*, y*1*, y*2 ∈ *D*σ : *f* (*x*) = *y*1 ∧ *f* (*x*) = *y*2 ⇒ *y*1=D*σ y*2

where =D*σ* is the equality on *D*σ. In practice, we generally do not care whether such a function is extensional at partial real numbers or not.

While there is a search for more efficient substitutes for pif *I* one needs to be assured of the segment of RPCF without pif *I* being sequential. Let *w* RCA denote the set of RPCF-constants with pif *I* removed:

Definition 2.1 [*w* RCA]

*w* RCA := RCA \ {pif*I* }

where RCA is as in Definition [1.32](#_bookmark30), page [19](#_bookmark30).

Definition 2.2 [weak-RPCF (wRPCF)] By wRPCF we mean the segment of RPCF built upon the set of constants *w* RCA (Definition [2.1](#_bookmark37)). In other words wRPCF is RPCF without pif*I* .

Remark 2.3 Note that the set of wRPCF-types is the same as the set of RPCF-types.

First we need to fix a criterion for sequentiality and test wRPCF against it. The one we consider in this paper is a generalization of Vuillemin-sequentiality (Definition [1.26](#_bookmark21), page [16](#_bookmark21)) to functions over the interval domain. This way we will show that the first order wRPCF-definable functions are sequential. [7](#_bookmark41)

Definition 2.4 [generalized Vuillemin-sequentiality] Suppose

∀*j* ∈ {0*,... , n*} : *D*j ∈{ ⊥*,* N⊥*,* f}

then

*f* : *D*1 × ··· × *D*n → *D*0

is said to be Vuillemin-sequential (or simply sequential) at →−*x* = (*x*1*,... , x*n) ∈

*D*1 × ··· × *D*n if either:

1.*f* (→−*z* ) = *f* (→−*x* ) for all →−*z* ± →−*x*

or otherwise

2. ∃*i* ∈ {1*,... , n*} : ∀→−*z* = (*z*1*,... , z*n) ∈ *D*1 × ··· × *D*n :

if →−*z* N →−*x* and *f* (→−*z* ) N *f* (→−*x* ) then *z*i N *x*i

This *i* is called a sequentiality index for *f* at →−*x* .

*f* is sequential if it is sequential at all →−*x* in its domain.

Remark 2.5 Note that by the above definition, all unary first order functions of the interval domain model are trivially sequential.

Although as discussed before (page [16](#_bookmark21)) Vuillemin-sequentiality cannot be freely generalized to any domain, our generalization can be made legitimate on the following accounts:

Intuition : Think of a first-order function *f* of *k* arguments as a black-box with *k* channels of input. The intuition behind sequentiality is that we want *f* to be called sequential if at any time and any stage of computation process, *f* is “looking at” only one of its arguments. If this argument is the *i*-th one we like to call *i* the index of sequentiality (at this stage in the

7 In [[9](#_bookmark84)] we considered another criteria, i.e. *conservativity over PCF*.

process). Now if the information from any other channel is increased, it cannot improve the output of *f* as *f* is focusing only on the *i*-th argument.

Matching the expectations : As we shall see (from Remark [2.6](#_bookmark43) on page [27](#_bookmark43) and Lemma [2.8](#_bookmark51) on page [29](#_bookmark51)) any function with intuitive parallel behaviour is not Vuillemin-sequential. In particular, constants like pif *I* or por are not

Vuillemin-sequential.

We prove that any first order wRPCF-definable function is sequential. Our proof is a generalization of the proof of Theorem [1.31](#_bookmark27) (page [18](#_bookmark27)) as presented in [[1](#_bookmark72), Theorem 6.5.4, page 137], which can give a much better idea as to why

we define the logical relation *S*k+1 as in equation ([5](#_bookmark42)), page [27](#_bookmark42). Also, it is worth mentioning that both proofs are crucially based on the so-called *fundamental lemma of logical relations* (Lemma [1.29](#_bookmark25), page [17](#_bookmark25)).

Let us suppose that for any RPCF-type *σ*, *D*σ is the interpretation of type

*σ* in the interval domain model (see Definition [1.35](#_bookmark32), page [21](#_bookmark32)). Now for any

*k* ≥ 1, we define a relation *S*o of arity *k* +1 over the elements of *D*o, where

k+1

*o* is a ground type, i.e bool, nat or *I* :

(*x*1*,... , x*k*, x*k+1) ∈ *S*o

k+1

e E*j* ≤ *k* : (6*i* ≤ *k* + 1 : *x*j ± *x*i) (5)

We build up logical relations *S*k+1 for each *k* ≥ 1 over these basic cases, as in Definition [1.28](#_bookmark24), page [16](#_bookmark24). It is pretty easy to show that each *S*k+1 is in fact a *w* YCA-logical relation. If *c* is a unary constant in *w* YCA, then the *S*k+1’s are preserved as a result of YA(*c*) being monotone (in fact continuous). If *c* ∈ {ifbool*,* ifnat}, then it can be verified by case analysis over the boolean argument.

Remark 2.6 It is also worth mentioning that the so-called parallel operators do not preserve all *S*k+1’s. As an example, take *por* (see Definition [1.19](#_bookmark18), page [11](#_bookmark18)) and *k* = 2. The following figure can give a better picture of why this is the case. The first two left columns are elements of *S*bool whereas the rightmost

ˆ

3

column, which is the result of applying *por* over the elements of the first two, is not in *S*bool:

ˆ

3

*p*d*or*

T ⊥ → T

*p*d*or*

⊥ T → T

*p*d*or*

⊥ ⊥ → ⊥

The only non-trivial case may be the so-called *ﬁx-point operators* Yσ. Remember (Remark [1.36](#_bookmark33), page [21](#_bookmark33)) that for each wRPCF-type *σ* there is a wRPCF-constant Yσ : (*σ* → *σ*) → *σ*, with the denotation Yσ) ∈ *D*(σ→σ)→σ

defined by:

6*f* ∈ *D* : Yσ)(*f* ) = H{*f* (⊥D*σ* ) | *n* ≥ 0} (6)

σ→σ (n)

So, to prove that Yσ preserves all *S*k+1’s, we first show that the set of *S*k+1 invariant elements of any *D*σ forms a so-called *inclusive predicate*. That is, we have to verify two properties at each type *σ*:

1. (⊥D*σ ,... ,* ⊥D*σ*

) ∈ *S*σ , where *D*σ

is the denotation of the type *σ* in

1 *k*+1

k+1 i

the model Mi, for each *i* ∈ {1*,... ,k* + 1}.

1. If {*x*i = (*x*i *,... , x*i

) ∈ *S*σ

| *i* ∈ N} is an ascending chain in *S*σ ,

1 k+1

then H{*x*i}∈ *S*σ .

k+1

k+1

k+1

We prove this two-fold fact by induction over the type *σ*:

* If *σ* is a ground type then ([i](#_bookmark44)) holds by the definition of *S*σ

k+1

at ground

types. To prove ([ii](#_bookmark45)), suppose the sequence {*x*i} is given. For each *i* ≥ 0,

there exists an index *j*i ≤ *k*, such that 6*m* ≤ *k* + 1 : *x*i ± *x*i , because

j*i*

m

i σ

*x* ∈ *S*

k+1

and *σ* is ground. So, in particular, there must be an index *l* ≤ *k*

for which there are infinitely many *i*’s — e.g. elements of an infinite set

*A* ⊆ N — with 6*m* ≤ *k* + 1 : *x*i ± *x*i . Hence we have:

l m

6*m* ≤ *k*+1 : H{*x*i | *i* ∈ N} = H{*x*i | *i* ∈ *A*}± H{*x*i | *i* ∈ *A*} = H{*x*i

| *i* ∈ N}

l l m m

which means H{*x*i | *i* ∈ N}∈ *S*σ .

k+1

* If *σ* = *σ*1 → *σ*2 then both ([i](#_bookmark44)) and ([ii](#_bookmark45)) hold by induction hypothesis on *σ*2.

Proposition 2.7 *For any k* ≥ 1*, S*k+1 *(as deﬁned in (*[*5*](#_bookmark42)*), page* [*27*](#_bookmark42)*) is a C- logical relation, where C is the set of wRPCF-constants w* YCA *(Deﬁnition* [*2.1*](#_bookmark37)*, page* [*25*](#_bookmark37)*).*

Proof. It remains to show the proof for the constants Yσ : (*σ* → *σ*) → *σ*.

To prove that Yσ preserves *S*σ

k+1

, by ([6](#_bookmark46)) (page [28](#_bookmark46)) it suffices to show three

properties at each type *σ*:

1. (⊥D*σ ,... ,* ⊥D*σ*

) ∈ *S*σ , where *D*σ

is the denotation of the type *σ* in

1 *k*+1

k+1 i

the model Mi, for each *i* ∈ {1*,... ,k* + 1}.

1. If (*x*1*,... , x*k+1) ∈ *S*σ and (*F*1*,... , F*k+1) ∈ *S*σ→σ, then (*F*1*x*1*,... , F*k+1*x*k+1) ∈

k+1

k+1

σ

k+1.

*S*

1. If {*x*i = (*x*i *,... , x*i

) ∈ *S*σ

| *i* ∈ N} is an ascending chain in *S*σ ,

1 k+1

then H{*x*i}∈ *S*σ .

k+1

k+1

k+1

We have already shown that ([i](#_bookmark48)) and ([iii](#_bookmark50)) hold at each type *σ*, and ([ii](#_bookmark49)) holds by definition of logical relations.

It is possible to embark on proving the sequentiality of first order wRPCF- definable function of the interval domain model right now. But perhaps pre-

senting the relation between *S*k+1’s and sequentiality as a separate result would give a better understanding of why *S*k+1’s were chosen in the first place:

Lemma 2.8 *Let*

*where*

*f* : *D*1 × ··· × *D*n → *D*0

6*j* ∈ {0*,... , n*} : *D*j ∈{ ⊥*,* N⊥*,* f}

*be a ﬁrst order function in the interval domain model of RPCF. Then f is Vuillemin-sequential if and only if it preserves all S*k+1*’s,* (*k* ≥ 1)*.*

Proof.

(⇐) : Suppose *f* is as in the statement of the theorem, and preserves all

*S*k+1’s, (*k* ≥ 1). We prove that *f* is Vuillemin-sequential by contradiction:

Suppose *f* is *not Vuillemin-sequential* at a point *x* = (*x*1*,... , x*n) in its domain. Define:

*A* = {*j* | 1 ≤ *j* ≤ *n, x*j is not maximal}

As *A* /= ∅ [8](#_bookmark52) , without loss of generality let us suppose *A* = {1*,... , k*}, *k* ≤ *n*. For any *j* ∈ *A*, there must exist an *x*j = (*x*j *,... , x*j ) such that:

1 n

* 1. *x*j = *x*j

j

* 1. *x*j = *x*i, for all *i > k*, if any [9](#_bookmark53) .

i

* 1. *x*j N *x*
  2. *f* (*x*j) N *f* (*x*)

Now consider the (*k* + 1) × *n* matrix *X* defined by:

*X*i,j

⎧⎨ *x*i

*if i* ≤ *k*

⎩ *x*j *if i* = *k* +1

j

=

⎡*x*1 *x*1 *... x*1 *x*1 *... x*1 ⎤

2 2 2 2

2

1

2

k

k+1

n

⎢*x x ... x x ... x* ⎥

1 2 k k+1 n

⎢ . . . . . ⎥

.

.

k k k k

k

⎢*x x ... x x ... x* ⎥

1

2

k

k+1

n

⎣*x*1 *x*2 *... x*k *x*k+1 *... x*n⎦

8 In fact *A* has at least two elements, otherwise *f* would be vacuously sequential at *x*.

9 Notice that these are the maximal elements.

It is easy to see that for any *j* ≤ *n*, the *j*-th column of *X* is an element of

σ*j*

*S*

k+1

, because:

* + If *j* ≤ *k* then 6*m* ≤ *k* + 1 : *X*j,j = *x*j = *x*j ± *x*m = *X*m,j

j j

* + If *j > k*, then *X*1,j = *X*2,j = ··· = *X*k+1,j

Applying *f* to all the rows of *X* results in the vector :

⎡ *f* (*x*1) ⎤

⎢ . ⎥

k

⎢⎣ *f* (*x* ) ⎥⎦

*f* (*x*)

As *f* is supposed to preserve *S*k+1, for an index *i*0 ≤ *k*, *f* (*x*i0 ) is the min- imum element of the above vector (under ±). In particular *f* (*x*i0 ) ± *f* (*x*). On the other hand, by (iv) above, we have *f* (*x*i0 ) N *f* (*x*), a contradiction.

(⇒) : Assume *f* is Vuillemin-sequential, and for any *i* ∈ {1*,... , n*}, a *k* + 1-

dimensional vector *x*i = (*x*i *,... , x*i

) is given such that:

1 k+1

E*j* ≤ *k* : 6*m* ≤ *k* + 1 : *x*i ± *x*i

j m

We denote the least such *j* as *j*(*i*). Take *i*0 ∈ {1*,... , n*} to be an index of

sequentiality for *f* at (*x*1

j(1)

j(n)

*,... , x*n

). Then we have:

because

j(n)

j(i0 )

1

j(1)

*f* (*x*

*,... , x*i0

0

j(i )

*,... , x*n

1

j(i0 )

) = *f* (*x*

*,... , x*i0

0

j(i )

*,... , x*n )

1

(*x*

j(1)

*,... , x*i0

0

j(i )

*,... , x*n

1

j(i0 )

) ± (*x*

*,... , x*i0

0

j(i )

*,... , x*n )

and the two vectors agree on their *i*0-th components. On the other hand for any *i* /= *i*0 we have:

j(n)

j(i0 )

*f* (*x*1*,... , x*n) ± *f* (*x*1 *,... , x*n )

because

i i j(1)

j(n)

(*x*1*,... , x*n) ± (*x*1 *,... , x*n )

i i

therefore, for all *i* ≤ *k* + 1:

j(1)

j(n)

*f* (*x*1 *,... , x*n ) ± *f* (*x*1*,... , x*n)

j(i0) j(i0 ) i i

which shows that *f* preserves the logical relation *S*k+1 (with *j*(*i*0) being the required index of the minimum element).

Combining proposition [2.7](#_bookmark47) (page [28](#_bookmark47)) and Lemma [2.8](#_bookmark51) (page [29](#_bookmark51)), we obtain the main result of the paper:

Theorem 2.9 *Let where*

*f* : *D*1 × ··· × *D*n → *D*0

6*j* ∈ {0*,... , n*} : *D*j ∈{ ⊥*,* N⊥*,* f}

*bea ﬁrst order wRPCF-definable function in the interval domain model of* *RPCF. Then f is Vuillemin-sequential.*

Therefore, by virtue of this theorem, functions like *parallel-or* are ruled out from being definable in wRPCF (see Remark [2.6](#_bookmark43), page [27](#_bookmark43)).

Although we will show that not all unary first order functions of the interval domain model are definable in wRPCF (see Lemma [3.9](#_bookmark65), page [36](#_bookmark65) and Part [3.1](#_bookmark66), page [38](#_bookmark66)), they are Vuillemin-sequential (see Remark [2.5](#_bookmark40), page [26](#_bookmark40)) and adding them to the language does not affect the sequentiality:

Corollary 2.10 *Let* Γ = *w* YCA + *C, where* 6*c* ∈ *C* : ( *c*): *D*1 → *D*2 *is a computable unary ﬁrst order function, i.e : D*1*, D*2 ∈{ ⊥*,* N⊥*,* f})*, and denote the segment of RPCF built upon the constants in* Γ *by wRPCF* Γ*. Then any ﬁrst order wRPCF* Γ*-deﬁnable function is Vuillemin-sequential.*

Proof. [(Sketch)] For any *c* ∈ *C,* *c*) is monotone (because it is computable), hence preserves *S*k+1 for any *k*. Now Lemma [2.8](#_bookmark51) (page [29](#_bookmark51)) is applicable.

Remark 2.11 Generally logical relations are helpful for studying the be- haviour of (first order) functions definable in extensions of *λ*-calculus. As a simple example, one can show that excluding ifbool and ifnat from wRPCF,

leaves us with a language in which all functions are essentially unary. Let

∆ := *w* YCA \ {ifbool*,* ifnat}, and *wRPCF* ∆ be the segment of wRPCF built upon the set of constants ∆. For any *k* ≥ 1, let *T*k+1 be the *k* + 1-ary logical relation defined at the ground types *o* ∈ {bool*,* nat*, I* } by:

o k+1

*T*

= {(*x*1*,... , x*k+1) ∈ (*D*o)k+1 | E*j* ≤ *k* : *x*j = *x*k+1}

then similar to the proof of the main theorem, it can be shown that *wRPCF* ∆- definable functions preserve *T*k+1 for any *k* ≥ 1. The following counter- example shows how ifnat does not preserve *T*3, where the first three columns

on the left are elements of *T*3, while the rightmost column — the result of

applying ifnat to the elements of the first three — is not:

*tt* 0 1 i→fnat 0

*ff* 1 0 i→fnat 0

*tt* 1 0 i→fnat 1

Now using this, one can get a model theoretic proof of the following simple fact:

Let *f* : *D*1 × ··· × *D*n → *D*0 (where *D*j ∈ { ⊥*,* N⊥*,* f} for all 0 ≤ *j* ≤ *n*) be a first order *wRPCF* ∆-definable function. Then for some *i* ∈ {1*,... , n*},

there exists a *wRPCF* ∆-definable *f*i : *D*i → *D*0 such that :

*f* = *f*i ◦ *π*i (*π*i : *D*1 × ··· × *D*n → *D*i projection)

In words, first order *wRPCF* ∆-definable functions are essentially functions of one argument.

# Piece-wise affinity

In this section we derive another non-definability result in a segment of RPCF which contains wRPCF (Definition [2.2](#_bookmark38), page [26](#_bookmark38)) as a sub-segment. For that we need to clarify some terms and definitions:

Definition 3.1 [piece-wise affine] Let −∞ ≤ *p* ≤ *q* ≤ +∞ and *f* : [*p, q*] → R. We say that *f* is affine if and only if:

E*m, n* ∈ R: 6*x* ∈ [*p, q*]: *f* (*x*) = *mx* + *n* (7)

A continuous function *f* is said to be piece-wise affine if and only if for some:

such that:

{*p*0*,... , p*i*, p*i+1*,... , p*n}⊆ [*p, q*]

*p* = *p*0 ≤ ··· ≤ *p*i ≤ *p*i+1 ≤ ··· ≤ *p*n = *q*

*f* T [*p*i*, p*i+1], i.e. the restriction of *f* to [*p*i*, p*i+1], is affine for all 0 ≤ *i* ≤ *n* − 1.

Now take *a* = [*r, s*] ∈ f and consider consa) acting on the maximal ele- ments of f, i.e. [0*,* 1]. By equation ([2](#_bookmark33)) (page [22](#_bookmark33)) we have:

6*x* ∈ f : consa)(*x*) = (*s* − *r*)*x* + *r*

therefore by taking:

*m* := *s* − *r n* := *r*

in equation ([7](#_bookmark58)), we observe that consa) acts as an affine (hence trivially piece- wise affine) function over the maximal elements of f.

Assuming *r* /= *s* let us take a look at how taila) acts over [0*,* 1]. According to equation ([3](#_bookmark33)) (page [22](#_bookmark33)), and by taking:

(*p*0 := 0) ≤ (*p*1 := *r*) ≤ (*p*2 := *s*) ≤ (*p*3 := 1)

we see that substituting: (i)

*m* := 0 *n* := 0

in equation ([7](#_bookmark58)) makes taila) affine over [*p*o*, p*1] = [0*, r*].

(ii)

*m* := 1*/*(*s* — *r*) *n* := 0

in equation ([7](#_bookmark58)) makes taila) affine over [*p*1*, p*2] = [*r, s*].

(iii)

*m* := 0 *n* := 1

in equation ([7](#_bookmark58)) makes taila) affine over [*p*2*, p*3] = [*s,* 1].

hence taila) is piece-wise affine over [0*,* 1].

Observing the constants consa and taila being interpreted as piece-wise

affine functions over [0*,* 1], one might guess this property can be preserved by all wRPCF constructions on the basis that wRPCF is in fact weak when it comes to defining total functions over real types using definition by cases. With a suitable choice of logical relations, we can prove this guess for a lan- guage slightly more powerful than wRPCF.

Definition 3.2 [Sierpinski space] We call the flat cpo ( *,* ±) where:

= {⊥*,* T}

6*x, y* ∈ : *x* ± *y* e (*x* = *y* V *x* = ⊥) the Sierpinski space. In picture:

T

⊥

Definition 3.3 [*w*^*por* ] *w*^*por* : × → is defined by:

6*a, b* ∈ : *w*^*por* (*a, b*) = ⎧⎨ T if a = T or b = T

⎩ ⊥ otherwise

Definition 3.4 [weakly-parallel RPCF (wPR)] The set of wPR-types are gen-

erated by the grammar:

*σ* ::= S | bool | nat | *I* | *σ* → *σ*

{*D*σ} is a collection of domains for wPR if:

*D*S =

*D*bool = ⊥ *D*nat = N⊥ *DI* = f

*D*σ→τ = [*D*σ → *D*τ ] The set *w* УҮСA of wPR-constants is defined as:

*w* УҮСA := *w* ҮСA ∪ {*\** : S*,* wpor : S → S → S}

where *w* ҮСA is the set of wRPCF-constants as in Definition [2.1](#_bookmark37), page [25](#_bookmark37).

By wPR we mean the extension of wRPCF built upon *w* УҮСA, the imme- diate reduction rules of which are those of wRPCF extended with the following

wpor *\* N* → T

⎩ wpor *M \** → T

rules: ⎧⎨

*M* → *M* '

wpor *M* → wpor *M* '

*N* → *N* '

wpor *M N* → wpor *M N* '

and whose denotational semantics is that of wRPCF extended with the fol-

lowing interpretations of the new constants: [10](#_bookmark62)

10 There are new fix-point operators due to the existence of types not-present in wRPCF, but the formula for their interpretation is the same, so we omit it!

wpor) = *w*^*por*

*\**) = T

where *w*^*por* is the function defined in Definition [3.3](#_bookmark60), page [34](#_bookmark60).

Note 3 *In the previous deﬁnition (i.e. Deﬁnition* [*3.4*](#_bookmark61)*) we mentioned the ba- sics of wPR, so we leave the exact deﬁnition to the reader, as we believe with the material presented here, a method similar to that of deﬁning RPCF (Part* [*1.3*](#_bookmark28)*) would lead to a complete deﬁnition for wPR in a straightforward manner.*

Remark 3.5 wPR as it is cannot be regarded as a segment of RPCF due to the presence of and the constants that come with it.

For each *k* ≥ 1, we define a *k*-ary logical relation *R*k which — to some extent — carries the meaning of piece-wise affinity on first order functions. When *o* ∈ {S*,* bool*,* nat}, *R*o is simply defined as:

k

6(*x*1*,... , x*k) ∈ (*D*o)k : (*x*1*,... , x*k) ∈ *R*o e E*i* ≤ *k* : 6*j* ≤ *k* : *x*i ± *x*j

k

To define *R*o when *o* = *I* , we should be more cautious as affinity is essen- tially a concept used for total functions over reals considering their effect on

k

total real numbers. Bearing that in mind we define *RI*

k

as follows:

Notation 3.6 *When x* ∈ f*, by x and x we mean the left and right end-points of x respectively. More concisely x* = [*x, x*]*.*

For any →—*x* = (*x*1*,... , x*k) ∈ fk :

→—*x* ∈ *RI* e *P*1(→—*x* ) V (*P*2a(→—*x* ) ∧ *P*2b(→—*x* )) where *P*1, *P*2a and *P*2b are defined as follows:

k

*P*1(*x*1*,... , x*k) e E*i* ∈ {1*,... , k*}: 6*j* ∈ {1*,... , k*}: *x*i ± *x*j *P*2a(*x*1*,... , x*k) e 6*i* ∈ {1*,... ,k* — 1}: *x*i = *x*i+1

*P*2b(*x*1*,... , x*k) e E→—*y* = (*y*1*,... , y*k): (→—*y* ± →—*x* )

∧ (6*i* ≤ *k* : *y*i is maximal in f)

∧ [E*d >* 0: (6*i* ∈ {2*,... ,k* — 2}:

If none of *x*i−1*, x*i*, x*i+1*, x*i+2 is maximal in f then

*y*i+1 — *y*i = *d*)]

Definition 3.7 [*R*k] *R*k is the logical relation generated by the above ground type cases.

The aim is to show that *R*k is *C*-logical, where *C* is the set *w* УҮСA. As before the tricky part is to show that the set of *R*k invariant elements of the ground type *I* forms an inclusive predicate.

Lemma 3.8 *If* {*x*i = (*x*i *,... , x*i ) ∈ *RI* | *i* ∈ N} *is an ascending chain then*

→— ↑ i *I*

1 k k

*x* = (H

i∈N

*x* ) ∈ *R*k*.*

Proof. There are two cases to consider, which might overlap but nevertheless are exhaustive:

case (a) For an infinite subset N1 ⊆ N of natural numbers, we have 6*i* ∈

N1 : *P*1(*x*i). In this case for all *i* ∈ N1 there is an index *j*(*i*) ∈ {1*,... , k*}

such that 6*m* ∈ {1*,... , k*}: *x*i ± *x*i . This implies that there is an index

j(i)

m

*l* ∈ {1*,... , k*} for which there are infinitely many *i*’s with *j*(*i*) = *l*. As

{*x*i | *i* ∈ N} is ascending we have:

6*m* ∈ {1*,... , k*}: H↑{*x*i | *i* ∈ N}± H↑{*x*i | *i* ∈ N}

l m

So *P*1(H↑{*x*i | *i* ∈ N}).

case (b) E*n*0 ∈ N: 6*i* ≥ *n*0 : *P*2a(*x*i) ∧ *P*2b(*x*i). In this case for any *i* ≥ *n*0, there is a vector *y*i = (*y*i *,... , y*i ) and a real number *d*i *>* 0 that make

i →—1 k i i

*P*2b(*x* ) true. Now take *x* = (*x*1*,... , x*k) := H{*x* }. As {*x* } is ascending

and for each *i* ≥ *n*o : *P*2a(*x*i), we have:

6*j* ∈ {2*,... ,k* — 1}*,* 6*i* ≥ *n*0 : *x*j = *x*i

j

therefore, for arbitrary *y*1 ∈ *x*1 and *y*k ∈ *x*k the vector:

(*y*1*, y*n0 *,... , y*n0

*, y*k)

2 k−1

and *d*n will make *P*2b(→—*x* ) true. To finish we notice that :

0

6*j* ≤ *k* — 1: *x*j = *x*n0 = *x*n0

= *x*j+1

hence *P*2a(→—*x* ).

j j+1

Lemma 3.9 *R*k *as deﬁned in Deﬁnition* [*3.7*](#_bookmark63) *is C-logical, where C is the set of wPR constants w* УҮСA*.*

Proof. We check out the more interesting constants:

1. *c* ∈ *C* is the constant taila : *I* → *I* for some non-maximal *a* ∈ f and (*x*1*,... , x*k) ∈ *RI* . There are two cases:

k

* 1. *P*1(*x*1*,... , x*k): then as taila is monotone we have

*P*1(taila(*x*1)*,... ,* taila(*x*k))

* 1. *P*2a(*x*1*,... , x*k) ∧ *P*2b(*x*1*,... , x*k) : Assume *a* = [*a, a*] and that *a* ∈ *x*i1 , *a* ∈ *x*i2 for some 1 ≤ *i*1 ≤ *i*2 ≤ *k* (other cases are more or less similar). In this case we have:

6*j < i*1 : taila(*x*j) = [0*,* 0]

6*j > i*2 : taila(*x*j) = [1*,* 1]

If →—*y* = (*y*1*,... , y*k) and *d* make *P*2b(*x*1*,... , x*k) true, then (0*,... ,* 0*,* taila(*y*i1+1)*,... ,* taila(*y*i2−1)*,* 1*,... ,* 1)

↑ ↑

*i*1 *i*2

and d will make *P*2b(taila(*x*1)*,... ,* taila(*x*k)) true (see equation ([3](#_bookmark33)), page [22](#_bookmark33)). *P*2a(taila(*x*1)*,... ,* taila(*x*k)) holds trivially.

a−a

1. *c* ∈ *C* is the constant headr : *I* → bool for some *r* ∈ Q ∩ (0*,* 1) :

Let →—*x* = (*x*1*,... , x*k) ∈ *RI* and consider the two cases:

k

* 1. *P*1(*x*1*,... , x*k): then as headr is monotone we have

*P*1(headr(*x*1)*,... ,* headr(*x*k))

* 1. *P*2a(*x*1*,... , x*k) ∧ *P*2b(*x*1*,... , x*k) : There are three cases:
     + *x*k *< r*: in this case

(headr(*x*1)*,... ,* headr(*x*k)) = (*tt ,... , tt* ) ∈ *R*bool

k

* + - *r < x*1: we have

(headr(*x*1)*,... ,* headr(*x*k)) = (*ff ,... , ff* ) ∈ *R*bool

k

* + - E*i* ≤ *k* : *r* ∈ *x*i: in this case headr(*x*i) = ⊥ so 6*j* ≤ *k* : headr(*x*i) ±

headr(*x*j) which implies

(headr(*x*1)*,... ,* headr(*x*k)) ∈ *R*bool

k

1. *c* ∈ *C* is a fix point constant Yσ : Using Lemma [3.8](#_bookmark64), page [36](#_bookmark64) the proof in this case is by induction over *σ* as was done before in proposition [2.7](#_bookmark47), page [28](#_bookmark47) for *S*k+1’s.
2. *c* ∈ *C* is any other constant : These are straightforward and left to the reader.
   1. *Discussion*

There are certain issues to be addressed regarding Lemma [3.9](#_bookmark65). The logical relations *R*k do not by any means characterize piecewise affinity, as functions such as neg : f → f:

neg([*r, s*]) := [1 — *s,* 1 — *r*]

which are obviously affine do not preserve all *R*k’s. On the other hand, non- affine functions are highly unlikely to preserve all *R*k’s. Take *f* : f → f defined by:

and consider: where

*f* ([*r, s*]) := [*r*2*, s*2]

→—*a* = (*a*1*,... , a*20) ∈ *RI*

20

61 ≤ *i* ≤ 20 : *a*i = [(*i* — 1)*/*20*, i/*20]

Under *f* , this element of *RI*

20

is sent to:

→—

*b* = (*b*1*,... , b*20)

where

61 ≤ *i* ≤ 20 : *b*i = [(*i* — 1)2*/*400*, i*2*/*400]

→—

Claim 3.10

= (*b ,... ,b*

) ∈*/ RI*

*b* 1 20 20

Proof. Take any arbitrary →—*y* = (*y*1*,... , y*20) such that:

1. *y*i’s are maximal in f (1 ≤ *i* ≤ 20)

→—

1. *b*

≤ →—*y*

Then we have:

1. *y*3 — *y*2 ≤ 8*/*400
2. max{*y*19 — *y*18*, y*18 — *y*17} *>* 17*/*400

therefore it is impossible to find any →—*y* and *d >* 0 such that *P*2b

→—

( *b* ).

Although we picked a special case, it suggests a uniform approach to show- ing non-affine functions not preserving some *R*k, which is part of the future work (see Part [4](#_bookmark68). page [39](#_bookmark68)).

Remark 3.11 None of the following constants:

por : bool × bool → bool

pifbool : bool × bool × bool → bool pifnat : bool × nat × nat → nat pif*I* : bool × *I* × *I* → *I*

preserves all the logical relations *R*k. Take pif*I* for example. In the following figure, the left three columns are elements of *R*3, whereas the rightmost column

— the result of applying pif*I* over the elements of the first three — is not:

*tt* [0*,* 1*/*3] [0*,* 1*/*4] pif*I*

→

⊥ [1*/*3*,* 2*/*3] [1*/*4*,* 3*/*4] pif*I*

→

*tt* [2*/*3*,* 1] [3*/*4*,* 1] pif*I*

→

[0*,* 1*/*3]

[1*/*4*,* 3*/*4]

[2*/*3*,* 1]

hence none of them is definable in wPR. As a minor result, we have another confirmation of the fact that wpor is strictly weaker than por.

Remark 3.12 Take wRPCF+, the extension of wRPCF with a constant

+: *I* → *I* → *I* , interpreted as the maximal continuous extension of the medi- ation operator (equation ([4](#_bookmark36)), page [25](#_bookmark36)). Escardo´, Hofmann and Streicher in [[6](#_bookmark76)] show that wpor is wRPCF+-definable. On the other hand, it is easy to show that wRPCF+-definable functions preserve all *R*k’s, which proves that none of the four *parallel* operations mentioned in Remark [3.11](#_bookmark67) is wRPCF+-definable.

# Summary of the results and possible future investi- gations

The results of this paper have a general non-definability flavour, in the sense that we have presented criteria against which functions can be tested to see if they are not definable in certain segments of RPCF. Theorem [2.9](#_bookmark54) (page [31](#_bookmark54))

assures us that pif*I* is the only source of parallelism in RPCF. The logical

relations *S*k presented in equation ([5](#_bookmark42)), page [27](#_bookmark42) might give an inspiration as to how to characterize the logical relations preserved by wRPCF terms, i.e. a result similar to that of Sieber’s for PCF (see [[1](#_bookmark72), Exercise 6.5.3, page 136]).

Also we have shown that wRPCF-definable functions are piecewise affine ( Definition [3.1](#_bookmark57), page [32](#_bookmark57)), though our result is slightly more powerful. Of course the language we studied is really weak so it does not come as a surprise that the functions definable in that language are so limited, one witness being their piecewise affinity. The logical relations *R*k we presented (Definition [3.7](#_bookmark63), page [36](#_bookmark63)) by themselves do not characterize piecewise affinity. But as we discussed in Part [3.1](#_bookmark66), page [38](#_bookmark66), the whole framework could give an inspiration for further studies, specially of the following problem:

Problem 4.1 *Having R*k*’s as deﬁned in Deﬁnition* [*3.7*](#_bookmark63)*, page* [*36*](#_bookmark63)*, how can we characterize piecewise aﬃnity?*

# Acknowledgement

Achim Jung — my supervisor — suggested the problem of sequentiality of wRPCF and the definition of the generalized Vuillemin-sequentiality, i.e. Def- inition [2.4](#_bookmark39), page [26](#_bookmark39). I like to thank him for his invaluable help throughout all the stages of preparing the paper. Also my thanks go to Mart´ın Escardo´ for his comments and suggestions, especially on Corollary [2.10](#_bookmark55), page [31](#_bookmark55).

# References

1. R. Amadio and P.-L. Curien. *Domains and Lambda Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
2. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.
3. H.-J. B¨ohm, R. Cartwright, M. J. O’Donnell, and M. Riggle. Exact real arithmetic: A case study in higher order programming. In *ACM Symposium on Lisp and Functional Programming*. Association of Computing Machinery, 1986.
4. G. Berry. Mod`eles Compl`ement Ad´equats et Stables des Lambda-calculs typ´es, 1979. Th`ese de Doctorat d’Etat, Universit´e Paris VII.
5. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.
6. M. H. Escard´o, M. Hofmann, and T. Streicher. Mediation is inherently parallel, 1998. University of Edinburgh, Laboratory for Foundations of Computer Science, EPSRC report for project GR/M64840.
7. Mart´ın Ho¨tzel Escardo´. Real pcf extended with ∃ is universal. In A. Edalat, S. Jourdan, and

G. McCusker, editors, *Advances in Theory and Formal Methods of Computing: Proceedings of the Third Imperial College Workshop*, pages 13–24, Christ Church, Oxford, April 1996. IC Press.

1. Mart´ın Ho¨tzel Escardo´. *PCF extended with real numbers: a domain-theoretic approach to higher-order exact real number computation*. PhD thesis, The University of Edinburgh, Department of Computer Science, November 1997.
2. Amin Farjudian. Conservativity of wrpcf over pcf. Presented in BCTCS 19, Leicester, England, April 7–9, 2003.
3. Pietro Di Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Universit`a di Pisa-Genova-Udine, 1993.
4. A. Jung and J. Tiuryn. A new characterization of lambda definability. Technical Report 1498, Technische Hochschule Darmstadt, 1992.
5. Val´erie M´enissier-Morain. Arbitrary precision real arithmetic: design and algorithms. Submitted to the Journal of Symbolic Computation, September 1996.
6. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
7. K. Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Proc. LMS Symposium on Applications of Categories in Computer Science, Durham 1991*, volume 177 of *LMS Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.
8. A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79:357–358, 1991.
9. J. Vuillemin. Syntaxe, s`emantique et axiomatique d’un langage de programmation simple, 1974. Th`ese de Doctorat d’Etat, Universit´e Paris VII.