Logical Relations for PCF

Peter Gammie

August 16, 2018

Abstract

We apply Andy Pitts’s methods of defining relations over domains to several classical results in the literature. We show that the Y combinator coincides with the domain-theoretic fixpoint operator, that parallel-or and the Plotkin existential are not definable in PCF, that the continuation semantics for PCF coincides with the direct semantics, and that our domain-theoretic semantics for PCF is adequate for reasoning about contextual equivalence in an operational semantics. Our version of PCF is untyped and has both strict and non-strict function abstractions. The development is carried out in HOLCF.

Contents

1 Introduction 2

2 Pitts’s method for solving recursive domain predicates 2
  2.1 Sets of vectors 2
  2.2 Relations between domains and syntax 3
  2.3 Relations between pairs of domains 4

3 Logical relations for definability in PCF 5
  3.1 Direct denotational semantics 6
  3.2 The Y Combinator 7
  3.3 Logical relations for definability 7
  3.4 Parallel OR is not definable 9
  3.5 Plotkin’s existential quantifier 11
  3.6 Concluding remarks 12

4 Logical relations for computational adequacy 12
  4.1 Direct semantics using de Bruijn notation 12
  4.2 Operational Semantics 15
  4.3 Computational Adequacy 16
    4.3.1 Contextual Equivalence 18

5 Relating direct and continuation semantics 19
  5.1 Logical relation 21
  5.2 A retraction between the two definitions 22

6 A small-step (reduction) operational semantics for PCF 23
  6.0.1 Reduction is consistent with evaluation 24
1 Introduction

Showing the existence of relations on domains has historically been an involved process. This is due to the presence of the contravariant function space domain constructor that defeats familiar inductive constructions; in particular we wish to define “logical” relations, where related functions take related arguments to related results, and the corresponding relation transformers are not monotonic. Before Pitts (1996) such demonstrations involved laborious appeals to the details of the domain constructions themselves. (See Mulmuley (1987); Stoy (1977) for historical perspective.)

Here we develop some standard results about PCF using Pitts’s technique for showing the existence of particular recursively-defined relations on domains. By doing so we demonstrate that HOLCF (Müller et al. 1999; Huffman 2012b) is useful for reasoning about programming language semantics and not just particular programs.

We treat a variant of the PCF language due to Plotkin (1977). It contains both call-by-name and call-by-value abstractions and is untyped. We show the breadth of Pitts’s technique by compiling several results, some of which have only been shown in simply-typed settings where the existence of the logical relations is straightforward to demonstrate.

2 Pitts’s method for solving recursive domain predicates

We adopt the general theory of Pitts (1996) for solving recursive domain predicates. This is based on the idea of minimal invariants that Pitts (1993, Def 2) ascribes “essentially to D. Scott”.

Ideally we would like to do the proofs once and use Pitts’s relational structures. Unfortunately it seems we need higher-order polymorphism (type functions) to make this work (but see Huffman (2012a)). Here we develop three versions, one for each of our applications. The proofs are similar (but not quite identical) in all cases.

We begin by defining an admissible set (aka an inclusive predicate) to be one that contains ⊥ and is closed under countable chains:

\[
\begin{align*}
definition \text{admS} :: 'a::pcpo set set \text{ where} \\
\text{admS} \equiv \{ R :: 'a set. \bot \in R \land \text{adm} (\lambda x. x \in R) \}
\end{align*}
\]

\[
\begin{align*}
\text{typedef} \ ('a::pcpo) \text{ admS} = \{ x :: 'a::pcpo set . x \in \text{admS} \}
\end{align*}
\]

These sets form a complete lattice.

\[
\begin{align*}
\langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle
\end{align*}
\]

2.1 Sets of vectors

The simplest case involves the recursive definition of a set of vectors over a single domain. This involves taking the fixed point of a functor where the positive (covariant) occurrences of the recursion variable are separated from the negative (contravariant) ones. (See §3.4 etc. for examples.)
By dually ordering the negative uses of the recursion variable the functor is made monotonic with respect to the order on the domain \( \text{d}' \). Here the type constructor \( \text{d}' \text{ dual} \) yields a type with the same elements as \( \text{d}' \) but with the reverse order. The functions \text{dual} \ and \text{undual} \ mediate the isomorphism.

\text{type-synonym} \quad \text{d}' \text{ lf-rep} = \text{d}' \text{ admS dual} \times \text{d}' \text{ admS} \Rightarrow \text{d}' \text{ set} \\
\text{type-synonym} \quad \text{d}' \text{ lf} = \text{d}' \text{ admS dual} \times \text{d}' \text{ admS} \Rightarrow \text{d}' \text{ admS} \\

The predicate \text{eRSV} \ encodes our notion of relation. (This is Pitts’s \( e : R \subset S \).) We model a vector as a function from some index type \( \text{i}' \) to the domain \( \text{d}' \). Note that the minimal invariant is for the domain \( \text{d}' \) only.

\text{abbreviation} \quad \text{eRSV} :: (\text{d}' :: \text{pcpo} \rightarrow \text{d}') \Rightarrow (\text{d}' :: \text{admS dual}) \Rightarrow (\text{d} \Rightarrow \text{admS}) \Rightarrow \text{bool} \\
\text{where} \quad \text{eRSV} \ e \ R \ S \equiv \forall d \in \text{unlr} (\text{undual} R). (\lambda x. e \cdot (d x)) \in \text{unlr} S \\

In general we can also assume that \( e \) here is strict, but we do not need to do so for our examples.

Our locale captures the key ingredients in Pitts’s scheme:

- that the function \( \delta \) is a minimal invariant;
- that the functor defining the relation is suitably monotonic; and
- that the functor is closed with respect to the minimal invariant.

\text{locale} \quad \text{DomSolV} = \\
\text{fixes} \quad \delta :: (\text{d}' :: \text{pcpo} \rightarrow \text{d}') \rightarrow \text{d}' \rightarrow \text{d}' \\
\text{fixes} \quad F :: (\text{i}' :: \text{type} \Rightarrow \text{d}' :: \text{pcpo}) \text{ lf} \\
\text{assumes} \quad \text{min-inv-ID}: \text{fix} \cdot \delta = \text{ID} \\
\text{assumes} \quad \text{monoF}: \text{mono} \ F \\
\text{assumes} \quad \text{eRSV-deltaF}: \\
\quad \lambda (e :: \text{d} \rightarrow \text{d}'). (R :: (\text{i}' \Rightarrow \text{admS dual}) (S :: (\text{i}' \Rightarrow \text{admS})). \\
\quad \text{eRSV} \ e \ R \ S \Rightarrow \text{eRSV} \ (\delta \cdot e) (\text{dual} (F (\text{dual} S, \text{undual} R))) (F (R, S)) \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \text{proof} \\

From these assumptions we can show that there is a unique object that is a solution to the recursive equation specified by \( F \).

\text{definition} \quad \text{delta} \equiv \text{delta-pos} \\
\text{lemma} \quad \text{delta-sol}: \text{delta} = F (\text{dual delta}, \text{delta}) \text{proof} \\
\text{lemma} \quad \text{delta-unique}: \\
\quad \text{assumes} \quad r: F (\text{dual} r, r) = r \\
\quad \text{shows} \quad r = \text{delta} \text{proof} \\
\text{end}

We use this to show certain functions are not PCF-definable in §3.3.

### 2.2 Relations between domains and syntax

To show computational adequacy (§4.3) we need to relate elements of a domain to their syntactic counterparts. An advantage of Pitts’s technique is that this is straightforward to do.
definition synlr :: (′d::pcpo × ′a::type) set set where
synlr \equiv \{ R :: (′d × ′a) set. \forall a. \{ d, (d, a) \in R \} \in admS \}  

typedef (′d::pcpo, ′a::type) synlr = \{ x:(′d × ′a) set. x \in synlr \}  
morphisms unsynlr mksynlr ⟨proof⟩

An alternative representation (suggested by Brian Huffman) is to directly use the type ′a \Rightarrow ′b admS as this is automatically a complete lattice. However we end up fighting the automatic methods a lot.

 ⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩  

Again we define functors on (′d, ′a) synlr.

type-synonym (′d, ′a) synlf-rep = (′d, ′a) synlr dual × (′d, ′a) synlr ⇒ (′d × ′a) set

type-synonym (′d, ′a) synlf = (′d, ′a) synlr dual × (′d, ′a) synlr ⇒ (′d, ′a) synlr

We capture our relations as before. Note we need the inclusion e to be strict for our example.

abbreviation eRSS :: (′d::pcpo → ′d) ⇒ (′d, ′a::type) synlr dual ⇒ (′d, ′a) synlr ⇒ bool where

eRSS e R S \equiv \forall (d, a) \in unsynlr (undual R). (e-d, a) \in unsynlr S

locale DomSolSyn =

  fixes δ :: (′d::pcpo → ′d) → ′d → ′d

  fixes F :: (′d::pcpo, ′a::type) synlf

  assumes min-inv-ID: fix-δ = ID

  assumes min-inv-strict: \forall r. δ-r-⊥ = ⊥

  assumes monoF: mono F

  assumes eRS-deltaF:

  \forall (e :: ′d → ′d) (R :: (′d, ′a) synlr dual) (S :: (′d, ′a) synlr).  

  \[ e-⊥ = ⊥; eRSS e R S \] \implies eRSS (δ-e) (undual (F (undual S, undual R))) (F (R, S))⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

Again, from these assumptions we can construct the unique solution to the recursive equation specified by F.

2.3 Relations between pairs of domains

Following Reynolds (1974) and Filinski (2007), we want to relate two pairs of mutually-recursive domains. Each of the pairs represents a (monadic) computation and value space.

  type-synonym (′am, ′bm, ′av, ′bv) lr-pair = (′am × ′bm) admS × (′av × ′bv) admS

  type-synonym (′am, ′bm, ′av, ′bv) lf-pair-rep = 

  (′am, ′bm, ′av, ′bv) lr-pair dual × (′am, ′bm, ′av, ′bv) lr-pair ⇒ (′am × ′bm) set × (′av × ′bv) set

  type-synonym (′am, ′bm, ′av, ′bv) lf-pair = 

  (′am, ′bm, ′av, ′bv) lr-pair dual × (′am, ′bm, ′av, ′bv) lr-pair ⇒ (′am × ′bm) admS × (′av × ′bv) admS

The inclusions need to be strict to get our example through.

abbreviation
eRSP :: ((‘am::pcpo → ‘am) × (‘av::pcpo → ‘av))
⇒ ((‘bm::pcpo → ‘bm) × (‘bv::pcpo → ‘bv))
⇒ ((‘am × ‘bm) admS × (‘av × ‘bv) admS) dual
⇒ (‘am × ‘bm) admS × (‘av × ‘bv) admS
⇒ bool

where

eRSP ea eb R S ≡
(∀ (am, bm) ∈ unlr (fst (undual R)). (fst ea, fst eb-bm) ∈ unlr (fst S))
∧ (∀ (av, bv) ∈ unlr (snd (undual R)). (snd ea-av, snd eb-bv) ∈ unlr (snd S))

locale DomSolP =
fixes ad :: ((‘am::pcpo → ‘am) × (‘av::pcpo → ‘av)) → ((‘am → ‘am) × (‘av → ‘av))
fixes bd :: ((‘bm::pcpo → ‘bm) × (‘bv::pcpo → ‘bv)) → ((‘bm → ‘bm) × (‘bv → ‘bv))
fixes F :: (‘am, ‘bm, ‘av, ‘bv) lf-pair
assumes monoF : mono F
assumes ad-ID : fix ad = (ID, ID)
assumes bd-ID : fix bd = (ID, ID)
assumes ad-strict: ∃r. fst (ad·r)·⊥ = ⊥ ∧ r. snd (ad·r)·⊥ = ⊥
assumes bd-strict: ∃r. fst (bd·r)·⊥ = ⊥ ∧ r. snd (bd·r)·⊥ = ⊥
assumes eRSP-deltaF:
[ eRSP ea eb R S; fst ea·⊥ = ⊥; snd ea·⊥ = ⊥; fst eb·⊥ = ⊥; snd eb·⊥ = ⊥ ]
⇒ eRSP (ad·ea) (bd·eb) (dual (F (dual S, undual R))) (F (R, S))(proof)(proof)(proof)(proof)(proof)(proof)(proof)(proof)

We use this solution to relate the direct and continuation semantics for PCF in §5.

3 Logical relations for definability in PCF

Using this machinery we can demonstrate some classical results about PCF (Plotkin 1977).
We diverge from the traditional treatment by considering PCF as an untyped language
and including both call-by-name (CBN) and call-by-value (CBV) abstractions following
Reynolds (1974). We also adopt some of the presentation of Winskel (1993, Chapter 11), in particular
by making the fixed point operator a binding construct.

We model the syntax of PCF as a HOL datatype, where variables have names drawn from
the naturals:

type-synonym var = nat
datatype expr =
  Var var
  | App expr expr
  | AbsN var expr
  | AbsV var expr
  | Diverge (Ω)
  | Fix var expr
  | tt
  | ff
  | Cond expr expr expr
  | Num nat
  | Succ expr
  | Pred expr
  | IsZero expr
3.1 Direct denotational semantics

We give this language a direct denotational semantics by interpreting it into a domain of values.

\[ \text{domain } \text{ValD} = \]
\[ \text{ValF (lazy appF :: ValD } \rightarrow \text{ ValD)} \]
\[ \mid \text{ValTT} \mid \text{ValFF} \]
\[ \mid \text{ValN (lazy nat)} \]

The \textit{lazy} keyword means that the \text{ValF} constructor is lifted, i.e. \text{ValF}·⊥̸ = ⊥, which further means that \text{ValF}·(Λ \, x. ⊥) = ⊥.

The naturals are discretely ordered.

We interpret the PCF constants in the obvious ways. “Ill-typed” uses of these combinators are mapped to ⊥.

\[ \text{definition cond :: ValD } \rightarrow \text{ ValD } \rightarrow \text{ ValD } \rightarrow \text{ ValD where} \]
\[ \text{cond ≡ Λ i t e. case i of ValF·f ⇒ ⊥ | ValTT ⇒ t | ValFF ⇒ e | ValN·n ⇒ ⊥} \]

\[ \text{definition succ :: ValD } \rightarrow \text{ ValD where} \]
\[ \text{succ ≡ Λ (ValN·n). ValN·(n + 1)} \]

\[ \text{definition pred :: ValD } \rightarrow \text{ ValD where} \]
\[ \text{pred ≡ Λ (ValN·n). case n of 0 ⇒ ⊥ | Suc n ⇒ ValN·n} \]

\[ \text{definition isZero :: ValD } \rightarrow \text{ ValD where} \]
\[ \text{isZero ≡ Λ (ValN·n). if n = 0 then ValTT else ValFF} \]

We model environments simply as continuous functions from variable names to values.

\[ \text{type-synonym Var = var} \]
\[ \text{type-synonym } \text{'a Env = Var } \rightarrow \text{ 'a} \]

\[ \text{definition env-empty :: 'a Env where} \]
\[ \text{env-empty ≡ ⊥} \]

\[ \text{definition env-ext :: Var } \rightarrow \text{ 'a } \rightarrow \text{ 'a Env } \rightarrow \text{ 'a Env where} \]
\[ \text{env-ext ≡ Λ v x g v'. if v = v' then x else g·v'} \]

The semantics is given by a function defined by primitive recursion over the syntax.

\[ \text{type-synonym EnvD = ValD Env} \]
primrec
  evalD :: expr ⇒ EnvD → ValD
where
  evalD (Var v) = (Λ ϱ. ϱ·v)
  | evalD (App f x) = (Λ ϱ. appF·(evalD f·ϱ)·(evalD x·ϱ))
  | evalD (AbsN v e) = (Λ ϱ. ValF·(Λ x. evalD e·(env-ext·v·x·ϱ)))
  | evalD (AbsV v e) = (Λ ϱ. ValF·(strictify·(Λ x. evalD e·(env-ext·v·x·ϱ))))
  | evalD (Diverge) = (Λ ϱ. ⊥)
  | evalD (Fix v e) = (Λ ϱ. µ x. evalD e·(env-ext·v·x·ϱ))
  | evalD (tt) = (Λ ϱ. ValTT)
  | evalD (ff) = (Λ ϱ. ValFF)
  | evalD (Cond i t e) = (Λ ϱ. cond·(evalD i·ϱ)·(evalD t·ϱ)·(evalD e·ϱ))
  | evalD (Num n) = (Λ ϱ. ValN·n)
  | evalD (Succ e) = (Λ ϱ. succ·(evalD e·ϱ))
  | evalD (Pred e) = (Λ ϱ. pred·(evalD e·ϱ))
  | evalD (IsZero e) = (Λ ϱ. isZero·(evalD e·ϱ))

definition eval' :: expr ⇒ ValD Env ⇒ ValD [ ]- [0,1000] 60 where
  eval' M ϱ ≡ evalD M·ϱ

3.2 The Y Combinator

We can shown the Y combinator is the least fixed point operator using just the minimal
invariant. In other words, fix is definable in untyped PCF minus the Fix construct.
This is Example 3.6 from Pitts (1996). He attributes the proof to Plotkin.
These two functions are \( \Delta \equiv \lambda f x. f (x x) \) and \( Y \equiv \lambda f. (\Delta f) (\Delta f) \).
Note the numbers here are names, not de Bruijn indices.

definition Y-delta :: expr where
  Y-delta ≡ AbsN 0 (AbsN 1 (App (Var 0) (App (Var 1) (Var 1))))
definition Ycomb :: expr where
  Ycomb ≡ AbsN 0 (App (App Y-delta (Var 0)) (App Y-delta (Var 0)))
definition fixD :: ValD → ValD where
  fixD ≡ \L (ValF·f). fix f
lemma Y: [[Ycomb]]g = ValF·fixD(proof)

3.3 Logical relations for definability

An element of ValD is definable if there is an expression that denotes it.
definition definable :: ValD ⇒ bool where
  definable d ≡ ∃ M. [[M] env-empty = d

A classical result about PCF is that while the denotational semantics is adequate, as we show
in §4, it is not fully abstract, i.e. it contains undefinable values (junk).
One way of showing this is to reason operationally; see, for instance, Plotkin (1977, §4) and Gunter (1992, §6.1).
Another is to use logical relations, following Plotkin (1973), and also Mitchell (1996); Sieber (1992); Stoughton (1993).
For this purpose we define a logical relation to be a set of vectors over $ValD$ that is closed under continuous functions of type $ValD \to ValD$. This is complicated by the $ValF$ tag and having strict function abstraction.

**definition**

$logical-relation :: ('i::type \Rightarrow ValD) \ set \Rightarrow bool$

**where**

$logical-relation \ R \equiv$

$(\forall \ fs \in R. \ \forall \ xs \in R. (\lambda j. \ appF.(fs \ j).(xs \ j)) \in R)$

$\wedge (\forall \ fs. \ (\forall \ xs \in R. (\lambda j. \ strictify.(appF.(fs j)).(xs j)) \in R) \wedge (\forall \ fs. \ (\forall \ xs \in R. (\lambda j. \ ValF.(fs j)) \in R) \wedge (\forall \ fs. \ (\forall \ xs \in R. (\lambda j. \ ValF.(strictify.(fs j))) \in R) \wedge (\forall \ fs \in R. (\lambda j. \ fixD.(xs \ j)) \in R)$

$\wedge (\forall \ cs \in R. \ \forall \ ts \in R. (\lambda j. \ cond.(cs \ j).(ts \ j).(es \ j)) \in R)$

$\wedge (\forall \ cs \in R. (\lambda j. \ succ.(xs \ j)) \in R)$

$\wedge (\forall \ cs \in R. (\lambda j. \ pred.(xs \ j)) \in R)$

$\wedge (\forall \ cs \in R. (\lambda j. \ isZero.(xs \ j)) \in R) \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

In the context of PCF these relations also need to respect the constants.

**definition**

$PCFconsts-rel :: ('i::type \Rightarrow ValD) \ set \Rightarrow bool$

**where**

$PCFconsts-rel \ R \equiv$

$\bot \in R$

$\wedge (\lambda i. \ ValTT) \in R$

$\wedge (\lambda i. \ ValFF) \in R$

$\wedge (\forall \ n. (\lambda i. \ ValN \ n) \in R) \langle proof \rangle \langle proof \rangle$

**abbreviation**

$PCF-lr \ R \equiv \ adm (\lambda x. \ x \in R) \wedge logical-relation \ R \wedge PCFconsts-rel \ R$

The fundamental property of logical relations states that all PCF expressions satisfy all PCF logical relations. This result is essentially due to Plotkin (1973). The proof is by a straightforward induction on the expression $M$.

**lemma** lr-fundamental:

**assumes** lr: $PCF-lr \ R$

**assumes** $g: \ \forall \ v. (\lambda i. \ g \ i \ v) \in R$

**shows** $(\lambda i. \ [M](g \ i)) \in R \langle proof \rangle$

We can use this result to show that there is no PCF term that maps the vector $\args \in R$ to $\result \not\in R$ for some logical relation $R$. If we further show that there is a function $f$ in $ValD$ such that $f \ \args = \ result$ then we can conclude that $f$ is not definable.

**abbreviation**

$appFLv :: ValD \Rightarrow ('i::type \Rightarrow ValD) \ list \Rightarrow ('i \Rightarrow ValD)$

**where**

$appFLv \ f \ args \equiv (\lambda i. \ foldl (\lambda f. \ appF.f.(x \ i)) \ f \ args)$

**lemma** lr-appFLv:

**assumes** lr: $logical-relation \ R$

**assumes** $f: (\lambda i::'i::type. \ f) \in R$

**assumes** $args: set \ args \subseteq R$

**shows** $appFLv \ f \ args \in R \langle proof \rangle$
Corollary not-definable:

fixes $R$ :: (′i::type ⇒ ValD) set
fixes $args$ :: (′i ⇒ ValD) list
fixes $result$ :: ′i ⇒ ValD
assumes $lr$: PCF-lr $R$
assumes $args$: set $args$ ⊆ $R$
assumes $result$: result /∈ $R$
shows ¬(∃(f::ValD). definable f ∧ appFLv f args = result)(proof)

3.4 Parallel OR is not definable

We show that parallel-or is not λ-definable following Sieber (1992) and Stoughton (1993).
Parallel-or is similar to the familiar short-circuiting or except that if the first argument is ⊥ and the second one is ValTT, we get ValTT (and not ⊥). It is continuous and then have included in the ValD domain.

definition por :: ValD ⇒ ValD ⇒ ValD (- por - [31,30] 30) where
  $x$ por $y$ ≡
  if $x$ = ValTT then ValTT
  else if $y$ = ValTT then ValTT
  else if ($x$ = ValFF ∧ $y$ = ValFF) then ValFF else ⊥

The defining properties of parallel-or.

lemma POR-simps [simp]:
  (ValTT por $y$) = ValTT
  ($x$ por ValTT) = ValTT
  (ValFF por ValFF) = ValFF
  (ValFF por ⊥) = ⊥
  (ValFF por ValN·n) = ⊥
  (ValFF por ValF·f) = ⊥
  (⊥ por ValFF) = ⊥
  (ValN·n por ValFF) = ⊥
  (ValF·f por ValFF) = ⊥
  (⊥ por ⊥) = ⊥
  (⊥ por ValN·n) = ⊥
  (⊥ por ValF·f) = ⊥
  (ValN·n por ⊥) = ⊥
  (ValF·f por ⊥) = ⊥
  (ValN·m por ValN·n) = ⊥
  (ValN·n por ValF·f) = ⊥
  (ValF·f por ValN·n) = ⊥
  (ValF·f por ValF·g) = ⊥
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩langle proof⟩

We need three-element vectors.

datatype Three = One | Two | Three

The standard logical relation $R$ that demonstrates POR is not definable is:

$$(x, y, z) \in R \text{ iff } x = y = z \lor (x = \bot \lor y = \bot)$$

That POR satisfies this relation can be seen from its truth table (see below).
Note we restrict the \( x = y = z \) clause to non-function values. Adding functions breaks the “logical relations” property.

**definition**

\[
\text{POR-base-lf-rep} :: (\text{Three} \Rightarrow \text{ValD}) \ \text{lif-rep}
\]

**where**

\[
\text{POR-base-lf-rep} \equiv \lambda (mR, pR).
\{ (\lambda i. \text{ValTT}) \} \cup \{ (\lambda i. \text{ValFF}) \} \quad x = y = \mathit{z} \text{ for bools}
\cup (\bigcup n. \{ (\lambda i. \text{ValN} \cdot n) \}) \quad x = y = \mathit{z} \text{ for numerals}
\cup \{ f . f \text{ One } = \bot \} \quad x = \bot
\cup \{ f . f \text{ Two } = \bot \} \quad y = \bot
\]

We close this relation with respect to continuous functions. This functor yields an admissible relation for all \( r \) and is monotonic.

**definition**

\[
\text{fn-lf-rep} :: (i::\text{type} \Rightarrow \text{ValD}) \ \text{lif-rep}
\]

**where**

\[
\text{fn-lf-rep} \equiv \lambda (mR, pR).
\{ (\lambda i. \text{ValF} \cdot (fs \ i)) \mid fs. \forall xs \in \text{unlr} \ (\text{undual } mR). \ (\lambda j. \ (fs \ j) \ (xs \ j)) \in \text{unlr } pR \}
\]

**definition**

\[
\text{POR-lf-rep} :: (\text{Three} \Rightarrow \text{ValD}) \ \text{lif-rep}
\]

**where**

\[
\text{POR-lf-rep} \equiv \text{POR-base-lf-rep} \cup \text{fn-lf-rep}
\]

**abbreviation**

\[
\text{POR-lf} \equiv \lambda r. \ mkr \ (\text{POR-lf-rep } r)
\]

Again it yields an admissible relation and is monotonic.

We need to show the functor respects the minimal invariant.

**lemma** \( \text{min-inv-POR-lf} \):

**assumes** \( \epsilon RSV \ e R \)
**shows** \( \epsilon RSV \ (\text{ValD-copy-rec} \cdot e) \ (\text{dual} \ (\text{POR-lf} \ (\text{dual } S', \text{undual } R'))) \ (\text{POR-lf} \ (R', S')) \)

We can show that the solution satisfies the expectations of the fundamental theorem \( \text{lr-fundamental} \).

**lemma** \( \text{PCF-lr-POR-delta} \):

\[ \text{PCF-lr} \ (\text{unlr } \text{POR-delta}) \]

This is the truth-table for POR rendered as a vector: we seek a function that simultaneously maps the two argument vectors to the result.

**definition** \( \text{POR-arg1-rel} \ **where**

\[
\text{POR-arg1-rel} \equiv \lambda i. \ \text{case } i \text{ of One } \Rightarrow \text{ValTT} \mid \text{Two } \Rightarrow \bot \mid \text{Three } \Rightarrow \text{ValFF}
\]

**definition** \( \text{POR-arg2-rel} \ **where**

\[
\text{POR-arg2-rel} \equiv \lambda i. \ \text{case } i \text{ of One } \Rightarrow \bot \mid \text{Two } \Rightarrow \text{ValTT} \mid \text{Three } \Rightarrow \text{ValFF}
\]

**definition** \( \text{POR-result-rel} \ **where**

\[
\text{POR-result-rel} \equiv \lambda i. \ \text{case } i \text{ of One } \Rightarrow \text{ValTT} \mid \text{Two } \Rightarrow \text{ValTT} \mid \text{Three } \Rightarrow \text{ValFF}
\]

**lemma** \( \text{lr-POR-arg1-rel} \ :

\[
\text{POR-arg1-rel} \ \in \ \text{unlr } \text{POR-delta}
\]

**lemma** \( \text{lr-POR-arg2-rel} \ :

\[
\text{POR-arg2-rel} \ \in \ \text{unlr } \text{POR-delta}
\]

**lemma** \( \text{lr-POR-result-rel} \ :

\[
\text{POR-result-rel} \ \notin \ \text{unlr } \text{POR-delta}
\]
Parallel-or satisfies these tests:

**theorem POR-sat:**
\[ \text{appFLv} (\text{ValF} \cdot (\Lambda x. \text{ValF} \cdot (\Lambda y. x \lor y))) \left[ \text{POR-arg1-rel, POR-arg2-rel} \right] = \text{POR-result-rel} \]

... but is not PCF-definable:

**theorem POR-is-not-definable:**
\[ \neg \exists f. \text{definable } f \land \text{appFLv} f \left[ \text{POR-arg1-rel, POR-arg2-rel} \right] = \text{POR-result-rel} \]

3.5 Plotkin’s existential quantifier

We can also show that the existential quantifier of Plotkin (1977, §5) is not PCF-definable using logical relations.

Our definition is quite loose; if the argument function \( f \) maps any value to \( \text{ValTT} \) then \( \text{plotkin-exists} \) yields \( \text{ValTT} \). It may be more plausible to test \( f \) on numerals only.

**definition plotkin-exists :: ValD \to ValD where**

\[
\text{plotkin-exists } f \equiv \\
\text{if } (\text{appF} \cdot f \cdot \bot = \text{ValFF}) \text{ then ValFF} \\
\text{else if } (\exists n. \text{appF} \cdot f \cdot n = \text{ValTT}) \text{ then ValTT else } \bot
\]

We can show this function is continuous.

**lemma cont-pe [cont2cont, simp]: cont plotkin-exists(\text{proof})(\text{proof})(\text{proof})**

Again we construct argument and result test vectors such that \( \text{plotkin-exists} \) satisfies these tests but no PCF-definable term does.

**definition PE-arg-rel where**

\[
\text{PE-arg-rel} \equiv \lambda i. \text{ValF} \cdot (\text{case } i \text{ of} \\
0 \Rightarrow (\Lambda -. \text{ValFF}) \\
\text{Suc } n \Rightarrow (\text{ValN} \cdot x. \text{if } x = \text{Suc } n \text{ then ValTT else } \bot)
\]

**definition PE-result-rel where**

\[
\text{PE-result-rel} \equiv \lambda i. \text{case } i \text{ of } 0 \Rightarrow \text{ValFF} \mid \text{Suc } n \Rightarrow \text{ValTT}
\]

Note that unlike the POR case the argument relation does not characterise PE: we don’t treat functions that return \( \text{ValTTs} \) and \( \text{ValFFs} \).

The Plotkin existential satisfies these tests:

**theorem pe-sat:**
\[ \text{appFLv} (\text{ValF} \cdot (\Lambda x. \text{plotkin-exists } x)) \left[ \text{PE-arg-rel} \right] = \text{PE-result-rel} \]

As for POR, the difference between the two vectors is that the argument can diverge but not the result.

**definition PE-base-lf-rep :: (\text{nat} \to \text{ValD}) \to \text{lf-rep} where**

\[
\text{PE-base-lf-rep} \equiv \lambda (mR, pR). \\
\{ \bot \} \\
\cup \{ (\lambda i. \text{ValTT}) \} \cup \{ (\lambda i. \text{ValFF}) \} \mid x = y = z \text{ for bools} \\
\cup (\bigcup n. \{ (\lambda i. \text{ValN} \cdot n) \}) \mid x = y = z \text{ for numerals}
\]
∪ { f₁, f₂ | f₁ ∨ f₂ = ⊥ } — Vectors that diverge on one or two.

Again we close this under the function space, and show that it is admissible, monotonic and respects the minimal invariant.

**definition** PE-if-rep :: (nat ⇒ ValD) → ValD

PE-if-rep R ≡ PE-base-if-rep R ∪ fn-if-rep R

**abbreviation** PE-if ≡ \( \lambda r. mklr (PE-if-rep r) \)

The solution satisfies the expectations of the fundamental theorem:

**lemma** PCF-lr-PE-delta: PCF-lr (unlr PE.delta)

**lemma** lr-PE-arg-rel: PE-arg-rel ∈ unlr PE.delta

**lemma** lr-PE-result-rel: PE-result-rel /∈ unlr PE.delta

**theorem** PE-is-not-definable: ~(∃ f. definable f ∧ appFLv f [PE-arg-rel] = PE-result-rel)

### 3.6 Concluding remarks

These techniques could be used to show that Haskell’s *seq* operation is not PCF-definable. (It is definable for each base “type” separately, and requires some care on function values.) If we added an (unlifted) product type then it should be provable that parallel evaluation is required to support *seq* on these objects (given *seq* on all other objects). (See Hudak et al. (2007, §5.4) and sundry posts to the internet by Lennart Augustsson.) This may be difficult to do plausibly without adding a type system.

### 4 Logical relations for computational adequacy

We relate the denotational semantics for PCF of §3.1 to a *big-step* (or *natural*) operational semantics. This follows Pitts (1993).

#### 4.1 Direct semantics using de Bruijn notation

In contrast to §3 we must be more careful in our treatment of \( \alpha \)-equivalent terms, as we would like our operational semantics to identify of all these. To that end we adopt de Bruijn notation, adapting the work of Nipkow (2001), and show that it is suitably equivalent to our original syntactic story.

**datatype** db =

- DBVar var
- DBApp db db
- DBAbsN db
- DBAbsV db
- DBDiverge
- DBFix db
- DBit
- DBff
- DBCond db db db
- DBNum nat
- DBSucc db
- DBPred db
Nipkow et al’s substitution operation is defined for arbitrary open terms. In our case we only substitute closed terms into terms where only the variable $0::a$ may be free, and while we could develop a simpler account, we retain the traditional one.

fun
lift :: $db \Rightarrow nat \Rightarrow db$
where
  lift (DBVar i) k = DBVar (if $i < k$ then $i$ else $(i + 1)$)
  lift (DBAbsN s) k = DBAbsN (lift s (k + 1))
  lift (DBAbsV s) k = DBAbsV (lift s (k + 1))
  lift (DBApp s t) k = DBApp (lift s k) (lift t k)
  lift (DBFix e) k = DBFix (lift e (k + 1))
  lift (DBCond c t e) k = DBCond (lift c k) (lift t k) (lift e k)
  lift (DBSucc e) k = DBSucc (lift e k)
  lift (DBPred e) k = DBPred (lift e k)
  lift (DBIsZero e) k = DBIsZero (lift e k)
  lift $x$ $k = x$

fun
  subst :: $db \Rightarrow db \Rightarrow var \Rightarrow db$ ::= \langle -'\rangle [300, 0, 0] 300
where
  subst-Var: (DBVar i)<s/k> =
    (if $k < i$ then DBVar (i - 1) else if $i = k$ then s else DBVar i)
  subst-AbsN: (DBAbsN t)<s/k> = DBAbsN (t<lift s 0 / k+1>)
  subst-AbsV: (DBAbsV t)<s/k> = DBAbsV (t<lift s 0 / k+1>)
  subst-App: (DBApp t u)<s/k> = DBApp (t<s/k>) (u<s/k>)
  (DBFix e)<s/k> = DBFix (e<lift s 0 / k+1>)
  (DBCond c t e)<s/k> = DBCond (e<s/k>) (t<s/k>) (e<s/k>)
  (DBSucc e)<s/k> = DBSucc (e<s/k>)
  (DBPred e)<s/k> = DBPred (e<s/k>)
  (DBIsZero e)<s/k> = DBIsZero (e<s/k>)
  subst-Consts: $x<s/k> = x\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

We elide the standard lemmas about these operations.
A variable is free in a de Bruijn term in the standard way.

fun
freenb :: $db \Rightarrow var \Rightarrow bool$
where
  freenb (DBVar j) $k = (j = k)$
  freenb (DBAbsN s) $k = freenb s (k + 1)$
  freenb (DBAbsV s) $k = freenb s (k + 1)$
  freenb (DBApp s t) $k = (\text{freenb s k } \lor \text{freenb t k})$
  freenb (DBFix e) $k = \text{freenb e} (\text{Suc k})$
  freenb (DBCond c t e) $k = (\text{freenb c k } \lor \text{freenb t k } \lor \text{freenb e k})$
  freenb (DBSucc e) $k = \text{freenb e k}$
  freenb (DBPred e) $k = \text{freenb e k}$
  freenb (DBIsZero e) $k = \text{freenb e k}$
  freenb $- - = \text{False\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle}$

Programs are closed expressions.

definition closed :: $db \Rightarrow bool$ where
The direct denotational semantics is almost identical to that given in §3.1, apart from this change in the representation of environments.

definition env-empty-db :: 'a Env where
  env-empty-db ≡ ⊥

definition env-ext-db :: 'a → 'a Env → 'a Env where
  env-ext-db ≡ Λ x. g. v. (case v of 0 ⇒ x | Suc v' ⇒ g.v')(proof)(proof)

primrec evalDdb :: db ⇒ ValD Env ⇒ ValD where
  evalDdb (DBVar i) = (Λ g. g i)
  | evalDdb (DBApp f x) = (Λ g. appF.(evalDdb f.g).(evalDdb x.g))
  | evalDdb (DBAbsN e) = (Λ g. ValF.(Λ x. evalDdb e.(env-ext-db x.g)))
  | evalDdb (DBAbsV e) = (Λ g. ValF.(strictify(Λ x. evalDdb e.(env-ext-db x.g))))
  | evalDdb (DBDiverge) = (Λ g. ⊥)
  | evalDdb (DBFix e) = (Λ g. μ x. evalDdb e.(env-ext-db x.g))
  | evalDdb (DBtt) = (Λ g. ValTT)
  | evalDdb (DBff) = (Λ g. ValFF)
  | evalDdb (DBCCond c t e) = (Λ g. cond.(evalDdb c.g).(evalDdb t.g).(evalDdb e.g))
  | evalDdb (DBNum n) = (Λ g. ValN·n)
  | evalDdb (DBSucc e) = (Λ g. succ.(evalDdb e.g))
  | evalDdb (DBPred e) = (Λ g. pred.(evalDdb e.g))
  | evalDdb (DBIsZero e) = (Λ g. isZero.(evalDdb e.g))(proof)(proof)

We show that our direct semantics using de Bruijn notation coincides with the evaluator of §3 by translating between the syntaxes and showing that the evaluators yield identical results. Firstly we show how to translate an expression using names into a nameless term. The following function finds the first mention of a variable in a list of variables.

primrec index :: var list ⇒ var ⇒ nat ⇒ nat where
  index [] v n = n
  | index (h # t) v n = (if v = h then n else index t v (Suc n))

primrec transdb :: expr ⇒ var list ⇒ db where
  transdb (Var i) Γ = DBVar (index Γ i 0)
  | transdb (App t1 t2) Γ = DBApp (transdb t1 Γ) (transdb t2 Γ)
  | transdb (AbsN n v t) Γ = DBAbsN (transdb t (v # Γ))
  | transdb (AbsV n v t) Γ = DBAbsV (transdb t (v # Γ))
  | transdb (Diverge) Γ = DBDiverge
  | transdb (Fix v e) Γ = DBFix (transdb e (v # Γ))
  | transdb (tt) Γ = DBtt
  | transdb (ff) Γ = DBff
  | transdb (Cond c t e) Γ = DBCCond (transdb c e Γ) (transdb t Γ) (transdb e Γ)
  | transdb (Num n) Γ = (DBNum n)
  | transdb (Succ e) Γ = DBSucc (transdb e Γ)
  | transdb (Pred e) Γ = DBPred (transdb e Γ)
  | transdb (IsZero e) Γ = DBIsZero (transdb e Γ)

\[ closed e ≡ ∀ i. \neg frees db e i(proof)(proof) \]
This semantics corresponds with the direct semantics for named expressions.

\[\text{lemma evalD-evalDdb:}\]
\[\begin{align*}
\text{assumes } & \text{free } e = [] \\
\text{shows } & [e]_\varphi = \text{evalDdb } (\text{transdb } e [] )_\varphi
\end{align*}\]

Conversely, all de Bruijn expressions have named equivalents.

\[\text{primrec transdb-inv :: db } \Rightarrow (\text{var } \Rightarrow \text{var }) \Rightarrow \text{var } \Rightarrow \text{var } \Rightarrow \text{expr}\]
\[\text{where}\]
\[\begin{align*}
\text{transdb-inv } (\text{DBVar } i) \Gamma c k &= \text{Var } (\Gamma i) \\
\text{transdb-inv } (\text{DBAbsN } e) \Gamma c k &= \text{AbsN } (c + k) \ (\text{transdb-inv } e (\text{case-nat } (c + k) \Gamma) c (k + 1)) \\
\text{transdb-inv } (\text{DBAbsV } e) \Gamma c k &= \text{AbsV } (c + k) \ (\text{transdb-inv } e (\text{case-nat } (c + k) \Gamma) c (k + 1)) \\
\text{transdb-inv } (\text{DBDiverge}) \Gamma c k &= \text{Diverge} \\
\text{transdb-inv } (\text{DBFix } e) \Gamma c k &= \text{Fix } (c + k) \ (\text{transdb-inv } e (\text{case-nat } (c + k) \Gamma) c (k + 1)) \\
\text{transdb-inv } (\text{DBtt}) \Gamma c k &= \text{tt} \\
\text{transdb-inv } (\text{DBff}) \Gamma c k &= \text{ff} \\
\text{transdb-inv } (\text{DBCond } i t e) \Gamma c k &= \text{Cond } (\text{transdb-inv } i \Gamma c k) \ (\text{transdb-inv } t \Gamma c k) \ (\text{transdb-inv } e \Gamma c k)
\end{align*}\]

\[\text{lemma transdb-inv:}\]
\[\begin{align*}
\text{assumes closed } e \\
\text{shows } & \text{transdb } (\text{transdb-inv } e \Gamma c k) \Gamma' = e \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle
\end{align*}\]

## 4.2 Operational Semantics

The evaluation relation (big-step, or natural operational semantics). This is similar to Gunter (1992, §6.2), Pitts (1993) and Winskel (1993, Chapter 11).

We firstly define the values that expressions can evaluate to: these are either constants or closed abstractions.

\[\text{inductive val :: db } \Rightarrow \text{bool}\]
\[\text{where}\]
\[\begin{align*}
\text{v-Num } \text{[intro]}: & \text{val } (\text{DBNum } n) \\
\text{v-FF } \text{[intro]}: & \text{val } \text{DBff} \\
\text{v-TT } \text{[intro]}: & \text{val } \text{DBtt} \\
\text{v-AbsN } \text{[intro]}: & \text{val } (\text{DBAbsN } e) \\
\text{v-AbsV } \text{[intro]}: & \text{val } (\text{DBAbsV } e)
\end{align*}\]

\[\text{inductive evalOP :: db } \Rightarrow \text{db } \Rightarrow \text{bool } (\rightarrow \downarrow) [50,50] 50\]
\[\text{where}\]
\[\begin{align*}
\text{evalOP-AppN } \text{[intro]}: & \ [P \downarrow \text{DBAbsN } M; M < Q / 0 > \downarrow V ] \Longrightarrow \text{DBApp } P Q \downarrow V \\
\text{evalOP-AppV } \text{[intro]}: & \ [P \downarrow \text{DBAbsV } M; Q \downarrow q; M < Q / 0 > \downarrow V ] \Longrightarrow \text{DBApp } P Q \downarrow V \\
\text{evalOP-AbsN } \text{[intro]}: & \text{val } (\text{DBAbsN } e) \Longrightarrow \text{DBAbsN } e \downarrow \text{DBAbsN } e \\
\text{evalOP-AbsV } \text{[intro]}: & \text{val } (\text{DBAbsV } e) \Longrightarrow \text{DBAbsV } e \downarrow \text{DBAbsV } e
\end{align*}\]
evalOP-Fix[intro]: $P < \text{DBFix } P/\emptyset > V \Rightarrow \text{DBFix } P \downarrow V$

evalOP-Itt[intro]: $\text{DBtt } \downarrow \text{DBtt}$

evalOP-pf[intro]: $\text{DBff } \downarrow \text{DBff}$

evalOP-CondTT[intro]: $[[ C \downarrow \text{DBtt}; T \downarrow V ]] \Rightarrow \text{DBCond } C T E \downarrow V$

evalOP-CondFF[intro]: $[[ C \downarrow \text{DBff}; E \downarrow V ]] \Rightarrow \text{DBCond } C T E \downarrow V$

evalOP-Num[intro]: $\text{DBNum } n \downarrow \text{DBNum } n$

evalOP-Succ[intro]: $P \downarrow \text{DBNum } n \Rightarrow \text{DBSucc } P \downarrow \text{DBNum } (\text{Suc } n)$

evalOP-Pred[intro]: $P \downarrow \text{DBNum } (\text{Suc } n) \Rightarrow \text{DBPred } P \downarrow \text{DBNum } n$

evalOP-IsZeroTT[intro]: $[[ E \downarrow \text{DBNum } 0 ]] \Rightarrow \text{DBIsZero } E \downarrow \text{DBtt}$

evalOP-IsZeroFF[intro]: $[[ E \downarrow \text{DBNum } n; 0 < n ]] \Rightarrow \text{DBIsZero } E \downarrow \text{DBff}$

It is straightforward to show that this relation is deterministic and sound with respect to the denotational semantics.

\begin{proof}
\begin{align*}
\text{assumes } & P \downarrow V \\
\text{shows } & \text{evalDdb } P \cdot \rho = \text{evalDdb } V \cdot \rho
\end{align*}
\end{proof}

We can use soundness to conclude that \text{POR} is not definable operationally either. We rely on \text{transdb-inv} to map our de Bruijn term into the syntactic universe of \S 3 and appeal to the results of \S 3.4. This takes some effort as \text{ValD} contains irrelevant junk that makes it hard to draw obvious conclusions; we use \text{DBCond} to restrict the arguments to the putative witness.

definition
\begin{align*}
isPORdb \ e & \equiv \text{closed } e \\
& \land \text{DBApp } (\text{DBApp } e \text{ DBtt}) \text{ DBDiverge } \downarrow \text{DBtt} \\
& \land \text{DBApp } (\text{DBApp } e \text{ DBDiverge}) \text{ DBtt } \downarrow \text{DBtt} \\
& \land \text{DBApp } (\text{DBApp } e \text{ DBff}) \text{ DBff } \downarrow \text{DBff}
\end{align*}

\begin{proof}
\end{proof}

\begin{lemma}
\text{POR-is-not-operationally-definable}: \lnot \text{isPORdb } e
\end{lemma}

### 4.3 Computational Adequacy

The lemma \text{evalOP-sound} tells us that the operational semantics preserves the denotational semantics. We might also hope that the two are somehow equivalent, but due to the junk in the domain-theoretic model (see \S 3.3) we cannot expect this to be entirely straightforward. Here we show that the denotational semantics is \textit{computationally adequate}, which means that it can be used to soundly reason about contextual equivalence.

We follow Pitts (1993, 1996) by defining a suitable logical relation between our \text{ValD} domain and the set of programs (closed terms). These are termed "formal approximation relations" by Plotkin. The machinery of \S 2.2 requires us to define a unique bottom element, which in this case is $\{ \bot \} \times \{ P. \text{ closed } P \}$. To that end we define the type of programs.

definition
\begin{align*}
\text{Prog} & = \{ P. \text{ closed } P \} \\
\text{morphisms} & \text{ unProg } \text{ mkProg }
\end{align*}

\begin{proof}
\end{proof}

\begin{proof}
\end{proof}
We can show it has the expected properties when all terms in $\Gamma$ are closed. 

**ca-closed lemma**

$\forall (\exists \, M. \, d = ValF \land \text{unProg } P \Downarrow DBAbsN M \land (\forall (x, \in) \in \text{unsynlr (undual } rm), \langle f \cdot x, \text{mkProg } (M\langle \text{unProg } X/\theta \rangle) \in \text{unsynlr } rp))$

$\forall (\exists \, M. \, d = ValF \land \text{unProg } P \Downarrow DBAbsN M \land (\forall (x, \in) \in \text{unsynlr (undual } rm), \forall V. \text{unProg } X \Downarrow V \rightarrow (f \cdot x, \text{mkProg } (M\langle V/\theta \rangle) \in \text{unsynlr } rp)))$

**abbreviation** $\text{ca-lr} :: (ValD, \text{Prog}) \Rightarrow \text{where}$

$\text{ca-lr} \equiv \lambda r. \text{mkSynlr } (\text{ca-lf-rep } r)$

Intuitively we relate domain-theoretic values to all programs that converge to the corresponding syntactic values. If a program has a non-$\perp$ denotation then we can use this relation to conclude something about the value it (operationally) converges to.

$\langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle$

**interpretation** $\text{ca}: \text{DomSolSyn ValD-copy-rec ca-lr}$

$\langle \text{proof} \rangle$

**definition** $\text{ca-lr-syn} :: ValD \Rightarrow db \Rightarrow \text{bool } (- \triangle - [80,80] 80)$

$d \triangle P \equiv (d, P) \in \{ (x, \text{unProg } Y) | x \cdot Y, (x, Y) \in \text{unsynlr ca.delta } \} \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle$

To establish this result we need a “closing substitution” operation. It seems easier to define it directly in this simple-minded way than reusing the standard substitution operation.

This is quite similar to a context-plugging (non-capturing) substitution operation, where the “holes” are free variables, and indeed we use it as such below.

**fun**

$\text{closing-subst} :: db \Rightarrow (\text{var} \Rightarrow db) \Rightarrow \text{var} \Rightarrow db$

**where**

$\text{closing-subst } (DBVar \cdot i) \Gamma k = (i + k \text{ else } DBVar \cdot i)$

$\text{closing-subst } (DBApp \cdot t \cdot u) \Gamma \kappa = DBApp \cdot (\text{closing-subst } t \Gamma \kappa) \cdot (\text{closing-subst } u \Gamma \kappa)$

$\text{closing-subst } (DBAbsN \cdot t) \Gamma \kappa = DBAbsN \cdot (\text{closing-subst } t \Gamma \kappa + 1)$

$\text{closing-subst } (DBAbsV \cdot t) \Gamma \kappa = DBAbsV \cdot (\text{closing-subst } t \Gamma \kappa + 1)$

$\text{closing-subst } (DBFix \cdot e) \Gamma \kappa = DBFix \cdot (\text{closing-subst } e \Gamma \kappa + 1)$

$\text{closing-subst } (DBCond \cdot e \cdot t \cdot e) \Gamma \kappa = DBCond \cdot (\text{closing-subst } e \Gamma \kappa) \cdot (\text{closing-subst } t \Gamma \kappa) \cdot (\text{closing-subst } e \Gamma \kappa)$

$\text{closing-subst } (DBSucc \cdot e) \Gamma \kappa = DBSucc \cdot (\text{closing-subst } e \Gamma \kappa)$

$\text{closing-subst } (DBPred \cdot e) \Gamma \kappa = DBPred \cdot (\text{closing-subst } e \Gamma \kappa)$

$\text{closing-subst } (DBIsZero \cdot e) \Gamma \kappa = DBIsZero \cdot (\text{closing-subst } e \Gamma \kappa)$

$\text{closing-subst } x \Gamma \kappa = x$

We can show it has the expected properties when all terms in $\Gamma$ are closed.

$\langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle$

The key lemma is shown by induction over $e$ for arbitrary environments ($\Gamma$ and $\theta$):

**lemma** $\text{ca-open}$:

**assumes** $\forall v. \text{freedb } e \cdot v \rightarrow (\theta \cdot v \triangle \Gamma \cdot v \land \text{closed } (\Gamma \cdot v))$

**shows** $\text{evalDdb } e \cdot \theta \triangle \text{closing-subst } e \Gamma \theta \langle \text{proof} \rangle$

**lemma** $\text{ca-closed}$:

**assumes** $\text{closed } e$

**shows** $\text{evalDdb } e \cdot \text{env-empty-db } \triangle e$
<proof>

**Theorem ca:**

- **Assumes** `nb: evalDdb e env-empty-db ≠ ⊥`
- **Assumes** `closed e`
- **Shows** `∃ V. e ↓ V`

<proof>

This last result justifies reasoning about contextual equivalence using the denotational semantics, as we now show.

### 4.3.1 Contextual Equivalence

As we are using an un(i)typed language, we take a context `C` to be an arbitrary term, where the free variables are the “holes”. We substitute a closed expression `e` uniformly for all of the free variables in `C`. If open, the term `e` can be closed using enough `AbsNs`. This seems to be a standard trick now, see e.g. Koutavas and Wand (2006). If we didn’t have CBN (only CBV) then it might be worth showing that this is an adequate treatment.

**Definition** `ctxt-sub :: db ⇒ db ⇒ db ([{-<->}] [300, 0] 300)` where

\[ C<e> ≡ \text{closing-subst } C (\lambda\cdot. e) 0 \]

Following Pitts (1996) we define a relation between values that “have the same form”. This is weak at functional values. We don’t distinguish between strict and non-strict abstractions.

**Inductive**

**have-the-same-form :: db ⇒ db ⇒ bool (- ~ [50,50] 50)**

where

- `DBAbsN e ~ DBAbsN e'`
- `DBAbsN e ~ DBAbsV e'`
- `DBAbsV e ~ DBAbsN e'`
- `DBAbsV e ~ DBAbsV e'`
- `DBFix e ~ DBFix e'`
- `DBtt ~ DBtt`
- `DBff ~ DBff`
- `DBNum n ~ DBNum n`

A program `e2` refines the program `e1` if it converges in context at least as often. This is a preorder on programs.

**Definition**

**refines :: db ⇒ db ⇒ bool (- ≤ - [50,50] 50)**

where

\[ e1 ≤ e2 ≡ \forall C. \exists V1. C<e1> ↓ V1 \implies (\exists V2. C<e2> ↓ V2 \land V1 ~ V2) \]

Contextually-equivalent programs refine each other.

**Definition**

**contextually-equivalent :: db ⇒ db ⇒ bool (- ≈ -)**

where

\[ e1 ≈ e2 ≡ e1 ≤ e2 \land e2 ≤ e1 \]

Our ultimate theorem states that if two programs have the same denotation then they are contextually equivalent.
Theorem computational-adequacy:

Assumes
1: closed e1
2: closed e2
D: evalDdb e1 e1 env-empty-db = evalDdb e2 e2 env-empty-db

Shows e1 \approx e2

This gives us a sound but incomplete method for demonstrating contextual equivalence. We expect this result is useful for showing contextual equivalence for typed programs as well, but leave it to future work to demonstrate this.

See Gunter (1992, §6.2) for further discussion of computational adequacy at higher types.

The reader may wonder why we did not use Nominal syntax to define our operational semantics, following Urban and Narboux (2009). The reason is that Nominal2 does not support the definition of continuous functions over Nominal syntax, which is required by the evaluators of §3 and §4.1. As observed above, in the setting of traditional programming language semantics one can get by with a much simpler notion of substitution than is needed for investigations into λ-calculi. Clearly this does not hold of languages that reduce “under binders”.

The “fast and loose reasoning is morally correct” work of Danielsson et al. (2006) can be seen as a kind of adequacy result.

Benton et al. (2009b) demonstrate a similar computational adequacy result in Coq. However their system is only geared up for this kind of metatheory, and not reasoning about particular programs; its term language is combinatorial.

Benton et al. (2007, 2009a) have shown that it is difficult to scale this domain-theoretic approach up to richer languages, such as those with dynamic allocation of mutable references, especially if these references can contain (arbitrary) functional values.

5 Relating direct and continuation semantics

This is a fairly literal version of Reynolds (1974), adapted to untyped PCF. A more abstract account has been given by Filinski (2007) in terms of a monadic meta language, which is difficult to model in Isabelle (but see Huffman (2012a)).

We begin by giving PCF a continuation semantics following the modern account of Wadler (1992). We use the symmetric function space (\(\text{prime ValK, 'o} K \rightarrow (\text{prime ValK, 'o} K\) K as our language includes call-by-name.

Type-synonym \( (\text{prime a, 'o} K = (\text{prime a} \rightarrow ) K \rightarrow \text{prime o} \)

Domain \( 'o \text{ValK} = \text{ValKF (lazy appKF :: (prime ValK, 'o) K \rightarrow (prime ValK, 'o) K}) \)
| ValKTT | ValKFF
| ValKN (lazy nat)

Type-synonym \( 'o \text{ValKM} = (\text{prime ValK, 'o} K (proof)(proof)(proof) \)

We use the standard continuation monad to ease the semantic definition.

Definition unitK :: 'o ValK \rightarrow 'o ValKM where
unitK \equiv \Lambda a. \Lambda c. c \cdot a

Definition bindK :: 'o ValKM \rightarrow ('o ValK \rightarrow 'o ValKM) \rightarrow 'o ValKM where
To establish the chain completeness (admissibility) of our logical relation, we need to show at least two elements:

locale at-least-two-elements =
  fixes some-non-bottom-element :: 'a::domain
  assumes some-non-bottom-element: some-non-bottom-element ≠ ⊥

The interpretations of the constants.

definition appKM :: 'o ValKM → 'o ValKM → 'o ValKM where
  appKM ≡ Λ fK xK. bindK·fK·(Λ (ValKF·f). f·xK)(proof)(proof)(proof)(proof)(proof)

The interpretations of the constants.

definition condK :: 'o ValKM → 'o ValKM → 'o ValKM where
  condK ≡ Λ iK tK eK. bindK·iK·(Λ i. case i of ValKF·f ⇒ ⊥ | ValKTF ⇒ tK | ValKFF ⇒ eK | ValKN·n ⇒ ⊥)

definition succK :: 'o ValKM → 'o ValKM where
  succK ≡ Λ nK. bindK·nK·(Λ (Val KN·n). unitK·(Val KN·(n + 1)))

definition predK :: 'o ValKM → 'o ValKM where
  predK ≡ Λ nK. bindK·nK·(Λ (Val KN·n). case n of 0 ⇒ ⊥ | Suc n ⇒ unitK·(Val KN·n))

definition isZeroK :: 'o ValKM → 'o ValKM where
  isZeroK ≡ Λ nK. bindK·nK·(Λ (Val KN·n). unitK·(if n = 0 then ValKTF else ValKFF))

A continuation semantics for PCF. If we had defined our direct semantics using a monad then the correspondence would be more syntactically obvious.

type-synonym 'o EnvK = 'o ValKM Env

primrec
evalK :: expr ⇒ 'o EnvK → 'o ValKM where
  evalK (Var v) = (Λ g. g·v)
  evalK (App f x) = (Λ g. appKM·(evalK f·g)·(evalK x·g))
  evalK (AbsN v e) = (Λ g. unitK·(ValKF·(Λ x. evalK e·(env-ext·v·x·g))))
  evalK (AbsV v e) = (Λ g. unitK·(ValKF·(Λ x. x·(Λ x'. evalK e·(env-ext·v·(unitK·x')·g)·c))))
  evalK (Diverge) = (Λ g. ⊥)
  evalK (Fix v e) = (Λ g. μ x. evalK e·(env-ext·v·x·g))
  evalK (tl) = (Λ g. unitK·ValKTF)
  evalK (ff) = (Λ g. unitK·ValKFF)
  evalK (Cond i t e) = (Λ g. condK·(evalK i·g)·(evalK t·g)·(evalK e·g))
  evalK (Num n) = (Λ g. unitK·(Val KN·n))
  evalK (Suc e) = (Λ g. succK·(evalK e·g))
  evalK (Pred e) = (Λ g. predK·(evalK e·g))
  evalK (IsZero e) = (Λ g. isZeroK·(evalK e·g))

To establish the chain completeness (admissibility) of our logical relation, we need to show that unitK is an order monic, i.e., if unitK·x ⊆ unitK·y then x ⊆ y. This is an order-theoretic version of injectivity.

In order to define a continuation that witnesses this, we need to be able to distinguish converging and diverging computations. We therefore require our observation domain to contain at least two elements:
Following Reynolds (1974) and Filinski (2007, Remark 47) we use the following continuation:

lemma cont-below [simp, cont2cont]:
cont (λx::'a::pipo. if x ⊑ d then ⊥ else c)(proof)

lemma (in at-least-two-elements) below-monic-unitK [intro, simp]:
below-monic-cfun (unitK :: 'o ValK → 'o ValKM)
(proof)

5.1 Logical relation

We follow Reynolds (1974) by simultaneously defining a pair of relations over values and functions. Both are bottom-reflecting, in contrast to the situation for computational adequacy in §4.3. Filinski (2007) differs by assuming that values are always defined, and relates values and monadic computations.

type-synonym 'o lfr = (ValD, 'o ValKM, ValD → ValD, 'o ValKM → 'o ValKM) if-pair-rep

context at-least-two-elements
begin
abbreviation lr-eta-rep-N where
lr-eta-rep-N ≡ {(e, e').
  (e = ⊥ ∧ e' = ⊥)
  ∨ (e = ValTT ∧ e' = unitK · ValTT)
  ∨ (e = ValFF ∧ e' = unitK · ValFF)
  ∨ (∃n. e = ValN · n ∧ e' = unitK · (ValKN · n))}

abbreviation lr-eta-rep-F where
lr-eta-rep-F ≡ λ(rm, rp). { (e, e').
  (e = ⊥ ∧ e' = ⊥)
  ∨ (∃f f'. e = ValF · f ∧ e' = unitK · (ValKF · f') ∧ (f, f') ∈ unlr (snd rp)) }

definition lr-eta-rep where
lr-eta-rep ≡ λr. lr-eta-rep-N ∪ lr-eta-rep-F r

definition lr-theta-rep where
lr-theta-rep ≡ λ(rm, rp). { (f, f').
  (∀(x, x'). ∈ unlr (fst (undual rm)). (f · x, f' · x') ∈ unlr (fst rp)) }

definition lr-rep :: 'o lfr where
lr-rep ≡ λr. (lr-eta-rep r, lr-theta-rep r)

abbreviation lr :: 'o lff where
lr ≡ λr. (mklr (fst (lr-rep r)), mklr (snd (lr-rep r)))(proof)(proof)(proof)end

It takes some effort to set up the minimal invariant relating the two pairs of domains. One might hope this would be easier using deflations (which might compose) rather than “copy” functions (which certainly don’t).
We elide these as they are tedious.


at-least-two-elements < F: DomSolP ValD-copy-rec ValK-copy-rec lr

21
5.2 A retraction between the two definitions

We can use the relation to establish a strong connection between the direct and continuation semantics. All results depend on the observation type being rich enough.

context at-least-two-elements
begin

abbreviation mrel (η: - ↦ - [50, 51] 50) where
η: x ↦ x' ≡ (x, x') ∈ unlr (fst F.delta)

abbreviation vrel (ϑ: - ↦ - [50, 51] 50) where
ϑ: y ↦ y' ≡ (y, y') ∈ unlr (snd F.delta)

Theorem 1 from Reynolds (1974).

lemma AbsV-aux:
assumes η: ValF · f ↦ unitK · (ValKF · f')
shows η: ValF · (strictify · f) ↦ unitK · (ValKF · (Λ x c. x · (Λ x' · f' · (unitK · x' · c))))

theorem Theorem1:
assumes ∀ v. η: ϱ · v ↦ ϱ' · v
shows η: evalD · ϱ ↦ evalK · ϱ'

end

The retraction between the two value and monadic value spaces.

Note we need to work with an observation type that can represent the “explicit values”, i.e. 'o ValK.

locale value-retraction
fixes VtoO :: 'o ValK → 'o
fixes OtoV :: 'o → 'o ValK
assumes OV: OtoV oo VtoO = ID

sublocale value-retraction < at-least-two-elements VtoO · (ValKN · 0)

context value-retraction
begin

fun DtoKM-i :: nat ⇒ ValD → 'o ValKM
and
KMtoD-i :: nat ⇒ 'o ValKM → ValD
where
DtoKM-i 0 = ⊥
| DtoKM-i (Suc n) = (Λ v. case v of
  ValF · f ⇒ unitK · (ValKF · (cfun-map · (KMtoD-i n · DtoKM-i n) · f))
  ValTT ⇒ unitK · ValTT
  ValFF ⇒ unitK · ValFF
  ValN · m ⇒ unitK · (ValKN · m))

22
\[ KMtoD-i \ 0 = \bot \]
\[ KMtoD-i \ (\text{Suc } n) = (\Lambda \ v. \ \text{case } OtoV \cdot (v \cdot VtoO) \ of \ ValKF \cdot f \Rightarrow ValF \cdot (\text{cfun-map} \cdot (DtoKM-i \ n) \cdot (KMtoD-i \ n) \cdot f) \]
\[ \text{ValKTT} \Rightarrow \text{ValTT} \]
\[ \text{ValKFF} \Rightarrow \text{ValFF} \]
\[ \text{ValKN} \cdot m \Rightarrow \text{ValN} \cdot m \]

abbreviation \( DtoKM \equiv (\bigcup i. \ DtoKM-i \ i) \)

abbreviation \( KMtoD \equiv (\bigcup i. \ KMtoD-i \ i) \)

Lemma 1 from Reynolds (1974).

\[ \eta: x \mapsto \ DtoKM \cdot x \]
\[ \eta: x \mapsto x' \implies x = KMtoD \cdot x'(\text{proof}) \]

Theorem 2 from Reynolds (1974).

\[ \text{theorem Theorem2: evalD e} \cdot \varrho = KMtoD \cdot (\text{evalK e} \cdot (DtoKM oo \varrho)) \]

\[ \text{(proof)} \]

end

Filinski (2007, Remark 48) observes that there will not be a retraction between direct and continuation semantics for languages with richer notions of effects.

It should be routine to extend the above approach to the higher-order backtracking language of Wand and Vaillancourt (2004).

I wonder if it is possible to construct continuation semantics from direct semantics as proposed by Sethi and Tang (1980). Roughly we might hope to lift a retraction between two value domains to a retraction at higher types by synthesising a suitable logical relation.

6 A small-step (reduction) operational semantics for PCF

A small-step semantics allows us to express more things, like the progress of well-typed programs.

FIXME adjust: This relation is non-deterministic, but only \( \beta \)-reduces terms where the argument is a value. Moreover if we start with a closed term then our values are also closed. So while in general (i.e., for open terms) our substitution operation is wrong and this relation is too big, we show that things work out if we start reducing from a closed term (i.e., a program).

FIXME following Tolmach https://www.cis.upenn.edu/~bcpierce/sf/current/Norm.html we make this relation deterministic. Eases the normalization proof.

inductive reduction :: \( db \Rightarrow db \Rightarrow \text{bool} \ (- \rightarrow \nu \cdot [50, 50] \ 50) \)

where

\[ \text{betaN: DApp (DBAbsN u) v} \rightarrow v \cdot u<\nu/0> \]
\[ \text{betaV: val v} \Rightarrow \text{DApp (DBAbsV u) v} \rightarrow v \cdot u<\nu/0> \]
\[ f \rightarrow v \ f' \Rightarrow \text{DApp f} \cdot x \rightarrow v \text{DApp f'} \cdot x \]
\[ [f = DBAbsV u; x \rightarrow v \ x'] \Rightarrow \text{DApp f} \cdot x \rightarrow v \text{DApp f} \cdot x' \]
\[ DBFix f \rightarrow v \ f<DBFix f/0> \]
\[ DBCond DBtt t c \rightarrow v \ t \]
| DBCond t e →_v e | DBff t e →_v e |
| DBSucc (DBNum n) →_v DBNum (Suc n) | DBPpred (DBNum (Suc n)) →_v DBNum n |
| DBIsZero (DBNum 0) →_v DBtt | 0 < n ⇒ DBIsZero (DBNum n) →_v DBff |

**abbreviation** — The transitive, reflexive closure of the reduction relation.

reduction-trc :: db ⇒ db ⇒ bool (- →_v* - [100, 100] 100)

**where**

reduction-trc ≡ rtranclp reduction

declare reduction.intros[intro!]

**inductive-cases** reduction-inv:

DBVar v →_v t'  
DBApp f x →_v t'  
DBAbsN u →_v t'  
DBAbsV u →_v t'  
DBFix f →_v t'  
DBCond i t e →_v t'  
DBff →_v t'  
DBtt →_v t'  
DBNum n →_v t'  
DBSucc n →_v t'  
DBPred n →_v t'  
DBIsZero n →_v t'

**lemma** reduction-val:

assumes val v  
assumes v →_v v'  
shows False
(proof)

**lemma** reduction-deterministic:

assumes t →_v t'  
assumes t →_v t''  
shows t'' = t'
(proof)

**6.0.1 Reduction is consistent with evaluation**

**lemma** reduction-eval:

assumes t →_v t'  
assumes t' ↓ v  
shows t ↓ v
(proof)

**lemma** reduction-trc-eval:

assumes t →_v* t'  
assumes t' ↓ v  
shows t ↓ v
(proof)
theorem reduction-trc-val-eval:
  assumes $t \rightarrow^* v$
  assumes val $v$
  shows $t \Downarrow v$
(proof)

We show the converse (of sorts) using the frame stack machinery of the next section.

7 Concluding remarks

We have seen that Pitts’s techniques for showing the existence of relations over domains is straightforward to mechanise and use in HOLCF.

One source of irritation in doing so is that Pitts’s technique is formulated in terms of minimal invariants, which presently must be written out by hand. (Earlier versions of HOLCF’s domain package provided these copy functions, though we would still need to provide our own in such cases as §5.) HOLCF ’11 provides us with take functions (approximations, deflations) on domains that compose, and so one might hope to adapt Pitts’s technique to use these instead. This has been investigated by Benton et al. (2009a, §6), but it is unclear that the deflations involved are those generated by HOLCF ’11.

References


