Logical Relations for PCF

Peter Gammie

March 12, 2013

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 Concluding remarks 23
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:

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

\[
\text{typedef} \ ('a::pcpo) \text{ admS } = \{ \ x :: 'a::pcpo set . \ x \in \text{admS} \} \\
\text{morphisms unlr mklr} \langle \text{proof} \rangle \\
\]

These sets form a complete lattice.

\[
\langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \\
\]

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 ‘d. Here the type constructor ‘a dual yields a type
with the same elements as ’a but with the reverse order. The functions dual and undual mediate the isomorphism.

**type-synonym**

\[
\text{lf-rep} = \text{admS dual} \times \text{admS} \Rightarrow \text{set}
\]

**type-synonym**

\[
\text{lf} = \text{admS dual} \times \text{admS} \Rightarrow \text{admS}
\]

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

**abbreviation**

\[
eRSV :: (\text{lf-rep} \Rightarrow \text{admS dual}) \Rightarrow (\text{lf} \Rightarrow \text{admS}) \Rightarrow \text{bool}
\]

where

\[
eRSV e R S \equiv \forall d \in \text{unlr (undual R)}. (\lambda x. e\cdot(d\cdot x)) \in \text{unlr S}
\]

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 δ 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.

**locale**

\[
\text{DomSolV} =
\]

\[
\text{fixes} \; \delta :: (\text{lf-rep} \Rightarrow \text{admS dual}) \Rightarrow \text{admS} \Rightarrow \text{admS}
\]

\[
\text{fixes} \; F :: (\text{lf} \Rightarrow \text{lf-rep}) \Rightarrow \text{admS dual} \Rightarrow (\text{lf} \Rightarrow \text{admS})
\]

\[
\text{assumes} \; \text{min-inv-ID}: \delta \cdot \text{ID} = \text{ID}
\]

\[
\text{assumes} \; \text{monoF}: \text{mono } F
\]

\[
\text{assumes} \; \text{eRSV-deltaF}:
\]

\[
\forall (e :: \text{lf-rep} \Rightarrow \text{admS dual}) (R :: (\text{lf} \Rightarrow \text{admS}) (S :: (\text{lf} \Rightarrow \text{admS})).
\]

\[
eRSV e R S \Rightarrow eRSV (\delta\cdot e) (\text{dual } F (\text{dual } S, \text{undual } R)) (F (R, S))
\]

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

**definition**

\[
\delta \equiv \text{delta-pos}
\]

**lemma**

\[
\text{delta-sol}: \delta = F (\text{dual } \delta, \delta)
\]

**lemma**

\[
\text{delta-unique}:
\]

\[
\text{assumes } r : F (\text{dual } r, r) = r
\]

\[
\text{shows } r = \delta
\]

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**

\[
\text{synlr} :: (\text{lf-rep} \times \text{a::type}) \Rightarrow \text{set}
\]

where
sylnr \equiv \{ R \subseteq (d \times a) \text{ set. } \forall a. \{ d. (d, a) \in R \} \in \text{admS} \}

definition {'d':pcpo, 'a::type} sylnr = \{ x:(d \times a) \text{ set. } x \in \text{sylnr} \}

morphism unsylnr mksynlr \langle \text{proof} \rangle

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

\text{proof} \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

Again we define functors on ('d, 'a) sylnr.

type-synonym ('d, 'a) synlf-rep = ('d, 'a) sylnr dual \times ('d, 'a) sylnr \Rightarrow ('d \times 'a) \text{ set}

type-synonym ('d, 'a) synlf = ('d, 'a) sylnr dual \times ('d, 'a) sylnr \Rightarrow ('d, 'a) sylnr

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

abbreviation

eRSS \subseteq ('d::pcpo \rightarrow 'd) \Rightarrow ('d, 'a::type) sylnr dual \Rightarrow ('d, 'a) sylnr \Rightarrow \text{bool}

where

eRSS e R S \equiv \forall (d, a) \in unsylnr (undual R). (e\cdot d, a) \in unsylnr S

locale DomSolSyn =

fixes \delta :: ('d::pcpo \rightarrow 'd) \Rightarrow 'd \rightarrow 'd

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

assumes min-inv-ID: \text{fix}\,\delta = \text{ID}

assumes min-inv-strict: \forall r. \delta \cdot r \cdot \bot = \bot

assumes monoF: \text{mono} F

assumes eRS-deltaF:

(\forall (e :: 'd \rightarrow 'd) \in ('d :: ('d, 'a) sylnr dual) \in ('d :: ('d, 'a) sylnr). [ e \cdot \bot = \bot ; eRSS e R S ] \Rightarrow eRSS (\delta \cdot e) (\text{undual} (F (\text{undual} S, \text{undual} R))) (F (R, S)) \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)

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) br-pair = ('am \times 'bm) \text{admS} \times ('av \times 'bv) \text{admS}

type-synonym ('am, 'bm, 'av, 'bv) if-pair-rep =

('am, 'bm, 'av, 'bv) \text{br-pair dual} \times ('am, 'bm, 'av, 'bv) \text{br-pair} \Rightarrow (('am \times 'bm) \text{set} \times ('av \times 'bv) \text{set})

type-synonym ('am, 'bm, 'av, 'bv) if-pair =

('am, 'bm, 'av, 'bv) \text{br-pair dual} \times ('am, 'bm, 'av, 'bv) \text{br-pair} \Rightarrow (('am \times 'bm) \text{admS} \times ('av \times 'bv) \text{admS})

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

abbreviation

eRSF :: (('am::pcpo \rightarrow 'am) \times ('av::pcpo \rightarrow 'av))
⇒ ((\'bm::pcpo \to \'bm) \times (\'bv::pcpo \to \'bv))
⇒ ((\'am \times \'bm) admS \times (\'av \times \'bv) admS) dual
⇒ (\'am \times \'bm) admS \times (\'av \times \'bv) admS
⇒ bool

where

eRSP ea eb R S ≡
(\forall (am, bm) \in unlr (fist (undual R)). (fist ea-am, fist eb-bm) \in unlr (fist S))
\land (\forall (av, bv) \in unlr (snd (undual R)). (snd ea-av, snd eb-bv) \in unlr (snd S))

locale DomSolP =
frees ad :: ((\'am::pcpo \to \'am) \times (\'av::pcpo \to \'av)) \to ((\'am \to \'am) \times (\'av \to \'av))
fixes bd :: ((\'bm::pcpo \to \'bm) \times (\'bv::pcpo \to \'bv)) \to ((\'bm \to \'bm) \times (\'bv \to \'bv))
fixes F :: (\'am, \'bm, \'av, \'bv) \pfunpair

assumes monoF: mono F
assumes ad-ID: fix-ad = (ID, ID)
assumes bd-ID: fix-bd = (ID, ID)
assumes ad-strict: \land r. fist (ad-r) \bot = \bot \land r. snd (ad-r) \bot = \bot
assumes bd-strict: \land r. fist (bd-r) \bot = \bot \land r. snd (bd-r) \bot = \bot
assumes eRSP-deltaF:
[ eRSP ea eb R S; fist ea \bot = \bot; snd ea \bot = \bot; fist eb \bot = \bot; snd eb \bot = \bot ]
\implies eRSP (ad-ea) (bd-eb) (dual (F (dual S, undual R))) (F (R, S))⟨proof⟩⟨proof⟩⟨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:

\textbf{type-synonym} \textit{var} = \textit{nat}

\textbf{datatype} \textit{expr} =
Var \textit{var}
| \textit{App} \textit{expr} \textit{expr}
| \textit{AbsN} \textit{var} \textit{expr}
| \textit{AbsV} \textit{var} \textit{expr}
| \textit{Diverge} (\Omega)
| \textit{Fix} \textit{var} \textit{expr}
| \textit{tt}
| \textit{ff}
| \textit{Cond} \textit{expr} \textit{expr} \textit{expr}
| \textit{Num} \textit{nat}
| \textit{Succ} \textit{expr}
| \textit{Pred} \textit{expr}
| \textit{IsZero} \textit{expr}
3.1 Direct denotational semantics

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

\[
\text{domain } \text{ValD} =
\begin{align*}
\text{ValF} (\text{lazy } \text{appF} :: \text{ValD} \to \text{ValD}) \\
\text{ValTT} \mid \text{ValFF} \\
\text{ValN} (\text{lazy } \text{nat})
\end{align*}
\]

The \texttt{lazy} keyword means that the \texttt{ValF} constructor is lifted, i.e. \texttt{ValF}.:\perp \neq \perp, which further means that \texttt{ValF}(\Lambda x. \perp) \neq \perp.

The naturals are discretely ordered.

\[
\text{The minimal invariant for } \text{ValD} \text{ is straightforward; the function } cfun-map \cdot f \cdot g \cdot h \text{ denotes } g \circ h \circ f.
\]

\[
\text{fixrec}
\begin{align*}
\text{ValD-copy-rec} :: (\text{ValD} \to \text{ValD}) \to (\text{ValD} \to \text{ValD}) \\
\text{where}
\text{ValD-copy-rec} \cdot r \cdot (\text{ValF} \cdot f) = \text{ValF} \cdot (\text{cfun-map} \cdot r \cdot r \cdot f) \\
\text{ValD-copy-rec} \cdot r \cdot (\text{ValTT}) = \text{ValTT} \\
\text{ValD-copy-rec} \cdot r \cdot (\text{ValFF}) = \text{ValFF} \\
\text{ValD-copy-rec} \cdot r \cdot (\text{ValN} \cdot n) = \text{ValN} \cdot n(\text{proof}) \cdot (\text{proof}) \cdot (\text{proof})
\end{align*}
\]

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

\[
\text{definition cond} :: \text{ValD} \to \text{ValD} \to \text{ValD} \to \text{ValD} \text{ where}
\begin{align*}
\text{cond} \equiv \Lambda \text{ ite}. \text{ case } i \text{ of ValF} \cdot f \Rightarrow \perp \mid \text{ValTT} \Rightarrow t \mid \text{ValFF} \Rightarrow e \mid \text{ValN} \cdot n \Rightarrow \perp
\end{align*}
\]

\[
\text{definition succ} :: \text{ValD} \to \text{ValD} \text{ where}
\begin{align*}
\text{succ} \equiv \Lambda (\text{ValN} \cdot n). \text{ ValN} \cdot (n + 1)
\end{align*}
\]

\[
\text{definition pred} :: \text{ValD} \to \text{ValD} \text{ where}
\begin{align*}
\text{pred} \equiv \Lambda (\text{ValN} \cdot n). \text{ case } n \text{ of } 0 \Rightarrow \perp \mid \text{Suc } n \Rightarrow \text{ValN} \cdot n
\end{align*}
\]

\[
\text{definition isZero} :: \text{ValD} \to \text{ValD} \text{ where}
\begin{align*}
\text{isZero} \equiv \Lambda (\text{ValN} \cdot n). \text{ if } n = 0 \text{ then } \text{ValTT} \text{ else } \text{ValFF}
\end{align*}
\]

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

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

\[
\text{definition env-empty} :: 'a \text{ Env} \text{ where}
\begin{align*}
\text{env-empty} \equiv \perp
\end{align*}
\]

\[
\text{definition env-ext} :: \text{Var} \to 'a \to 'a \text{ Env} \to 'a \text{ Env} \text{ where}
\begin{align*}
\text{env-ext} \equiv \Lambda v x. g v'. \text{ if } v = v' \text{ then } x \text{ else } g \cdot v'(\text{proof}) \cdot (\text{proof})\cdot (\text{proof})
\end{align*}
\]

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

\[
\text{type-synonym } \text{EnvD} = \text{ValD} \text{ Env}
\]
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))

abbreviation 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 ∆ ≡ λf x. f (x x) and Y ≡ λf. (∆ f) (∆ 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 ≡ Λ (ValF · f). fix f

lemma Y: [Ycomb]ϱ = 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
For this purpose we define a logical relation to be a set of vectors over $ValD$ that is closed under continuous functions of type $ValD \rightarrow ValD$. This is complicated by the $ValF$ tag and having strict function abstraction.

**definition**

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

**where**

$logical-relation R \equiv$

$(\forall fs \in R. \forall xs \in R. (\lambda j. \text{appF} \cdot (fs j) \cdot (xs j)) \in R)$

$\land (\forall fs. \forall xs \in R. (\lambda j. \text{strictify} \cdot (\text{appF} \cdot (fs j)) \cdot (xs j)) \in R)$

$\land (\forall fs. (\forall xs \in R. (\lambda j. \text{strictify} \cdot (fs j) \cdot (xs j)) \in R) \implies (\lambda j. \text{ValF} \cdot (fs j)) \in R)$

$\land (\forall xs \in R. (\lambda j. \text{fixD} \cdot (xs j)) \in R)$

$\land (\forall xs \in R. (\lambda j. \text{cond} \cdot (cs j) \cdot (ts j) \cdot (es j)) \in R)$

$\land (\forall xs \in R. (\lambda j. \text{succ} \cdot (xs j)) \in R)$

$\land (\forall xs \in R. (\lambda j. \text{pred} \cdot (xs j)) \in R)$

$\land (\forall xs \in R. (\lambda j. \text{isZero} \cdot (xs j)) \in R) \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle$

**abbreviation**

$PCF-consts-rel :: (\tau :: type \Rightarrow ValD) set \Rightarrow bool$

**where**

$PCF-consts-rel R \equiv$

$\bot \in R$

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

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

$\land (\forall n. (\lambda i. ValN \cdot n) \in R) \langle \text{proof} \rangle \langle \text{proof} \rangle$

**lemma** $lr$-fundamental:

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

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

**shows** $(\lambda i. \llbracket M \rrbracket (g \cdot i)) \in R \langle \text{proof} \rangle$

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

**abbreviation**

$\text{appFL}_v :: ValD \Rightarrow (\tau :: type \Rightarrow ValD) \ \text{list} \Rightarrow (\tau \Rightarrow ValD)$

**where**

$\text{appFL}_v \ f \ \text{args} \equiv (\lambda i. \text{foldl} (\lambda f \cdot x. \text{appF} \cdot f \cdot (x \cdot i)) \ f \ \text{args})$

**lemma** $lr$-appFL$_v$:

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

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

**assumes** $\text{args} : \text{set} \ \text{args} \subseteq R$

**shows** $\text{appFL}_v \ f \ \text{args} \in R \langle \text{proof} \rangle$
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)

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 lazy-or except that if the first argument is ⊥ and the second one is ValTT, we get ValTT (and not ⊥). It is continuous and hence 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 ⊥) = ⊥
  (ValN·m por ValN·n) = ⊥
  (ValN·n por ValF·f) = ⊥
  (ValF·f por ValN·n) = ⊥
  (ValF·f por ValF·g) = ⊥

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) ∈ R iff x = y = z ∨ (x = ⊥ ∨ y = ⊥)

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

**Definition**

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

**Where**

\[
\text{POR-lf-rep} \equiv \lambda (mR, pR). \{ (\lambda i. \text{ValTT}) \cup \{ (\lambda i. \text{ValFF}) \} (* x = y = z \text{ for bools } *) \}
\]

\[
\cup (\bigcup n. \{ (\lambda i. \text{ValN}\cdot n) \}) (* x = y = z \text{ for numerals } *)
\]

\[
\cup \{ f \cdot f \cdot \text{One} = \bot \} (* x = \bot *)
\]

\[
\cup \{ f \cdot f \cdot \text{Two} = \bot \} (* y = \bot *)
\]

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

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

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

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

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

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

\[
\text{POR-result-rel} \in \text{unlr POR} \cdot \delta \text{eta}
\]

Parallel-or satisfies these tests:
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 $ValTT$ then $plotkin-exists$ yields $ValTT$. It may be more plausible to test $f$ on numerals only.

definition plotkin-exists :: $ValD \Rightarrow ValD$ where
plotkin-exists $f \equiv$
if $(appF \cdot f \cdot \bot = ValFF)$
then $ValFF$
else if $(\exists n. appF \cdot f \cdot n = ValTT)$ then $ValTT$ else $\bot$

We can show this function is continuous.

lemma cont-pe [cont2cont, simp]: cont plotkin-exists
Again we construct argument and result test vectors such that $plotkin-exists$ satisfies these tests but no PCF-definable term does.

definition PE-arg-rel where
PE-arg-rel $\equiv \lambda i. ValF \cdot (case i of$
$0 \Rightarrow (\lambda -. ValFF)$
$| Suc n \Rightarrow (\lambda (ValN \cdot x). if x = Suc n then ValTT else \bot))$

definition PE-result-rel where
PE-result-rel $\equiv \lambda i. case i of 0 \Rightarrow ValFF | Suc n \Rightarrow ValTT$

Note that unlike the POR case the argument relation does not characterise PE: we don’t treat functions that return $ValTT$s and $ValFF$s.

The Plotkin existential satisfies these tests:

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

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

definition PE-base-lf-rep :: $(nat \Rightarrow ValD)$ lf-rep where
PE-base-lf-rep $\equiv \lambda (mR, pR).$
\{ $\bot$ \}
$\cup \{ (\lambda i. ValTT) \} \cup \{ (\lambda i. ValFF) \} (\ast x = y = z \text{ for bools} \ast)$
$\cup (\bigcup n. \{ (\lambda i. ValN \cdot n) \}) (\ast x = y = z \text{ for numerals} \ast)$
$\cup \{ f . f 1 = \bot \lor f 2 = \bot \} (\ast \text{Vectors that diverge on one or two.} \ast)$
Again we close this under the function space, and show that it is admissible, monotonic and respects the minimal invariant.

**Definition**

\[ \text{PE-lf-rep} :: (\text{nat} \Rightarrow \text{ValD}) \ \text{l}f\text{-rep} \]

where

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

**Abbreviation**

\[ \text{PE-lf} \equiv \lambda r. \text{mklr} (\text{PE-lf-rep} r) \]

The solution satisfies the expectations of the fundamental theorem:

**Lemma**

\[ \text{PCF-lr-PE-delta} : \text{PCF-lr} (\text{unlr PE}.\text{delta}) \]

**Lemma**

\[ \text{lr-PE-arg-rel} \in \text{unlr PE}.\text{delta} \]

**Lemma**

\[ \text{lr-PE-result-rel} / \in \text{unlr PE}.\text{delta} \]

**Theorem**

\[ \text{PE-is-not-definable} : \neg (\exists f. \text{definable} f \wedge \text{appFLv} f [\text{PE-arg-rel}] = \text{PE-result-rel}) \]

### 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 α-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**

\[ \text{db} = \]

\[ \text{DBVar} \ \text{var} \]

\[ \text{DBApp} \ \text{db} \ \text{db} \]

\[ \text{DBAbsN} \ \text{db} \]

\[ \text{DBAbsV} \ \text{db} \]

\[ \text{DBDiverge} \]

\[ \text{DBFix} \ \text{db} \]

\[ \text{DBtt} \]

\[ \text{DBff} \]

\[ \text{DBCond} \ \text{db} \ \text{db} \ \text{db} \]

\[ \text{DBNum} \ \text{nat} \]

\[ \text{DBSucc} \ \text{db} \]

\[ \text{DBPred} \ \text{db} \]

\[ \text{DBIsZero} \ \text{db} \]

12
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 $\theta::'a$ may be free, and while we could develop a simpler account, we retain the traditional one.

**fun**

```plaintext
lift :: db ⇒ nat ⇒ 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 (DBCnd c t e) k = DBCnd (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**

```plaintext
subst :: db ⇒ db ⇒ var ⇒ db (¬<·>/⇒ [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<s/k> / k+1>)
  subst-AbsV: (DBAbsV t)<s/k> = DBAbsV (t<s/k> / k+1>)
  subst-App: (DBApp t u)<s/k> = DBApp (t<s/k> / u<s/k>)
  subst-Fix e<s/k> = DBFix e<s/k> (e<s/k>/k+1>)
  subst-Cond e t e<s/k> = DBCnd (e<s/k>) (t<s/k>) (e<s/k>)
  subst-Succ e<s/k> = DBSucc e<s/k>
  subst-Pred e<s/k> = DBPred e<s/k>
  subst-Zero e<s/k> = DBIsZero e<s/k>
  subst-Consts: x<s/k> = x(proof)<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>
```

We elide the standard lemmas about these operations.

A variable is free in a de Bruijn term in the standard way.

**fun**

```plaintext
freedb :: db ⇒ var ⇒ bool
where
  freedb (DBVar j) k = (j = k)
  freedb (DBAbsN s) k = freedb s (k + 1)
  freedb (DBAbsV s) k = freedb s (k + 1)
  freedb (DBApp s t) k = (freedb s k ∨ freedb t k)
  freedb (DBFix e) k = freedb e (Suc k)
  freedb (DBCnd c t e) k = (freedb c e ∨ freedb t k ∨ freedb e k)
  freedb (DBSucc e) k = freedb e k
  freedb (DBPred e) k = freedb e k
  freedb (DBIsZero e) k = freedb e k
  freedb - - = False<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>
```

Programs are closed expressions.

**definition**

```plaintext
closed :: db ⇒ bool where
closed e ≡ ∀ i. ¬ freedb e i<proof><proof>
```
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 Env → 'a Env where
  env-ext-db ≡ λ x v. (case v of ∅ ⇒ x | Suc v′ ⇒ g·v′)(proof)(proof)

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

We show that our direct semantics using de Bruïjn 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 i 0)
  transdb (App t1 t2) Γ = DBApp (transdb t1 Γ) (transdb t2 Γ)
  transdb (AbsN v t) Γ = DBAbsN (transdb t (v # Γ))
  transdb (AbsV 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 = DBCond (transdb c 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)

This semantics corresponds with the direct semantics for named expressions.

⟨proof⟩⟨proof⟩lemma evalD-evalDdb:
assumes free e = []
shows [e]_Q = evaDb (transdb e [])_Q
⟨proof⟩

Conversely, all de Bruijn expressions have named equivalents.

primrec
  transdb-inv :: db ⇒ (var ⇒ var) ⇒ var ⇒ var ⇒ expr
where
  transdb-inv (DBVar i) Γ c k = Var (Γ i)
  transdb-inv (DBApp t1 t2) Γ c k = App (transdb-inv t1 Γ c k) (transdb-inv t2 Γ c k)
  transdb-inv (DBAbsN e) Γ c k = AbsN (e + k) (transdb-inv (evalOP-AbsN e) (nat-case (c + k) Γ) c (k + 1))
  transdb-inv (DBAbsV e) Γ c k = AbsV (e + k) (transdb-inv (evalOP-AbsV e) (nat-case (c + k) Γ) c (k + 1))
  transdb-inv (DBFix e) Γ c k = Diverge
  transdb-inv (DBtt e) Γ c k = Fix (c + k) (transdb-inv (evalOP-AppN e) (nat-case (c + k) Γ) c (k + 1))
  transdb-inv (DBff e) Γ c k = ff
  transdb-inv (DBCond i t e) Γ c k =
    Cond (transdb-inv i Γ c k) (transdb-inv t Γ c k) (transdb-inv e Γ c k)
  transdb-inv (DBNum n) Γ c k = (Num n)
  transdb-inv (DBSucc e) Γ c k = Succ (transdb-inv e Γ c k)
  transdb-inv (DBPred e) Γ c k = Pred (transdb-inv e Γ c k)
  transdb-inv (DBIsZero e) Γ c k = IsZero (transdb-inv e Γ c k) ⟨proof⟩

lemma transdb-inv:
  assumes closed e
  shows transdb (transdb-inv e Γ c k) Γ′ = e⟨proof⟩⟨proof⟩⟨proof⟩

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.

inductive
  val :: db ⇒ bool
where
  v-Num[intro]: val (DBNum n)
  v-TT[intro]: val DBtt
  v-FF[intro]: val DBff
  v-AbsN[intro]: closed (DBAbsN e) ⇒ val (DBAbsN e)
  v-AbsV[intro]: closed (DBAbsV e) ⇒ val (DBAbsV e)

inductive
  evalOP :: db ⇒ db ⇒ bool (- ↓ - [50,50] 50)
where
  evalOP-AppN[intro]: [ P ↓ DBAbsN M; M<Q/0> ↓ V ] ⇒ DBApp P Q ↓ V
  evalOP-AppV[intro]: [ P ↓ DBAbsV M; Q ↓ q; M<q/0> ↓ V ] ⇒ DBApp P Q ↓ V
  evalOP-AbsN[intro]: val (DBAbsN e) ⇒ DBAbsN e ↓ DBAbsN e
  evalOP-AbsV[intro]: val (DBAbsV e) ⇒ DBAbsV e ↓ DBAbsV e
  evalOP-Fix[intro]: P<DBFix P/0> ↓ V ⇒ DBFix P ↓ V
  evalOP-tt[intro]: DBtt ↓ DBtt
evalOP-ff[intro]: \( \text{DBff} \Downarrow \text{DBff} \)

evalOP-CondTT[intro]: [C \Downarrow \text{DBtt}; T \Downarrow V] \implies \text{DBCond} C T E \Downarrow V

evalOP-CondFF[intro]: [C \Downarrow \text{DBff}; E \Downarrow V] \implies \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 \implies \text{DBSucc} P \Downarrow \text{DBNum} (\text{Suc} n)

evalOP-Pred[intro]: P \Downarrow \text{DBNum} (\text{Suc} n) \implies \text{DBPred} P \Downarrow \text{DBNum} n

evalOP-IsZeroTT[intro]: [E \Downarrow \text{DBNum} 0] \implies \text{DBIsZero} E \Downarrow \text{DBtt}

evalOP-IsZeroFF[intro]: [E \Downarrow \text{DBNum} n; 0 < n] \implies \text{DBIsZero} E \Downarrow \text{DBff}

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

\textbf{lemma evalOP-sound:}

\begin{itemize}
    \item \textbf{assumes} P \Downarrow V
    \item \textbf{shows} evalDdb P \cdot \varrho = evalDdb V \cdot \varrho
\end{itemize}

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

definition 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}

\textbf{lemma POR-is-not-operationally-definable:} \neg \text{isPORdb e}

4.3 Computational Adequacy

The lemma \textit{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 §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 ValD domain and the set of programs (closed terms). These are termed ”formal approximation relations” by Plotkin. The machinery of §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 Prog = { P. \text{closed P} }

definitions unProg \text{mkProg} (\text{proof})

\textbf{where}

cap-lf-rep :: (ValD, Prog) symlf-rep
We can show it has the expected properties when all terms in $\Gamma$ are closed.

The key lemma is shown by induction over $|\Gamma|$.

fun

closing-subst :: $db \Rightarrow (var \Rightarrow db) \Rightarrow var \Rightarrow db$

where

closing-subst (DBVar i) $\Gamma$ $k$ = (if $k$ \leq i then $\Gamma$ (i - k) else DBVar i)

| closing-subst (DBApp t u) $\Gamma$ $k$ = DBApp (closing-subst t $\Gamma$ $k$) (closing-subst u $\Gamma$ $k$)

| closing-subst (DBAbsN t) $\Gamma$ $k$ = DBAbsN (closing-subst t $\Gamma$ (k + 1))

| closing-subst (DBAbsV t) $\Gamma$ $k$ = DRAbsV (closing-subst t $\Gamma$ (k + 1))

| closing-subst (DBFix e) $\Gamma$ $k$ = DBFix (closing-subst e $\Gamma$ (k + 1))

| closing-subst (DBCond c t e) $\Gamma$ $k$ =

\[
DBCond \text{ (closing-subst e $\Gamma$ k) (closing-subst t $\Gamma$ k) (closing-subst e $\Gamma$ k)}
\]

| closing-subst (DBSucc e) $\Gamma$ $k$ = DBSucc (closing-subst e $\Gamma$ $k$)

| closing-subst (DBPred e) $\Gamma$ $k$ = DBPred (closing-subst e $\Gamma$ $k$)

| closing-subst (DBIsZero e) $\Gamma$ $k$ = DBIsZero (closing-subst e $\Gamma$ $k$)

| closing-subst x $\Gamma$ $k$ = x

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

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

lemma ca-open:

assumes $\forall v. \text{freedb } e \ v \Rightarrow \varnothing \ v \ \text{\leq } \Gamma \ v \ \text{closed (} \Gamma \ v \text{)}$

shows evalDdb $e \cdot \varnothing \ \text{\leq } \text{closing-subst } e \ \Gamma \ \theta$(proof)

lemma ca-closed:

assumes closed $e$

shows evalDdb $e \cdot \text{env-empty-db} \ \text{\leq } e$

(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 $\text{AbsNs}$. 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.

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.

A program $e_2$ refines the program $e_1$ if it converges in context at least as often. This is a preorder on programs.

Our ultimate theorem states that if two programs have the same denotation then they are contextually equivalent.

**Theorem computational-adequacy:**

assumes $1$: closed $e_1$
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 seman-
tics, 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 combinatory.

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 (′o ValK, ′o) K → (′o ValK, ′o) K as our
language includes call-by-name.

type-synonym (′a, ′o) K = (′a → ′o) → ′o

domain ′o ValK
   = ValKF (lazy appKF :: (′o ValK, ′o) K → (′o ValK, ′o) K)
     | ValKTT | ValKFF
     | ValKN (lazy nat)

type-synonym ′o ValKM = (′o ValK, ′o) K⟨proof⟩⟨proof⟩⟨proof⟩

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

definition unitK :: ′o ValK → ′o ValKM where
unitK ≡ Λ a. Λ c. c·a

definition bindK :: ′o ValKM → (′o ValK → ′o ValKM) → ′o ValKM where
bindK ≡ Λ m k. Λ c. m·(Λ a. k·a·c)
Following Reynolds (1974) and Filinski (2007, Remark 47) we use the following continuation:

\[
\text{appKM :: } \nu \text{ ValKM } \rightarrow \nu \text{ ValKM } \rightarrow \nu \text{ ValKM }
\]

\[
\text{appKM } \equiv \Lambda f K. \text{ bindKM } f K. (\Lambda (\text{ ValKF } f). f \cdot x K) (\text{ proof}) (\text{ proof}) (\text{ proof}) (\text{ proof}) (\text{ proof})
\]

The interpretations of the constants.

\[
\text{condK :: } \nu \text{ ValKM } \rightarrow \nu \text{ ValKM } \rightarrow \nu \text{ ValKM } \rightarrow \nu \text{ ValKM }
\]

\[
\text{condK } \equiv \Lambda i K. t K. e K. \text{ bindKM } - i K. (\Lambda i. \text{ case } i \text{ of } ValKF f \Rightarrow \bot | ValKTT \Rightarrow t K | ValKFF \Rightarrow e K | ValKN \cdot n \Rightarrow \bot)
\]

\[
\text{succK :: } \nu \text{ ValKM } \rightarrow \nu \text{ ValKM }
\]

\[
\text{succK } \equiv \Lambda n K. \text{ bindKM } n K. (\Lambda (\text{ ValKN } \cdot n). \text{ unitK } \cdot (\text{ ValKN } \cdot (n + 1)))
\]

\[
\text{predK :: } \nu \text{ ValKM } \rightarrow \nu \text{ ValKM }
\]

\[
\text{predK } \equiv \Lambda n K. \text{ bindKM } n K. (\Lambda (\text{ ValKN } \cdot n). \text{ case } n \cdot 0 \Rightarrow \bot | \text{ Succ } n \Rightarrow \text{ unitK } \cdot (\text{ ValKN } \cdot n))
\]

\[
\text{isZeroK :: } \nu \text{ ValKM } \rightarrow \nu \text{ ValKM }
\]

\[
\text{isZeroK } \equiv \Lambda n K. \text{ bindKM } n K. (\Lambda (\text{ ValKN } \cdot n). \text{ unitK } \cdot (\text{ if } n = 0 \text{ then ValKTT else ValKFF}))
\]

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

\[
\text{type-synonym } \nu \text{ EnvK } = \nu \text{ ValKM Env }
\]

\[
\text{primrec }
\text{evalK :: expr } \Rightarrow \nu \text{ EnvK } \rightarrow \nu \text{ ValKM }
\]

\[
\text{evalK } (\text{ Var } v) = (\Lambda g. g \cdot v)
\]

\[
\text{evalK } (\text{ App } f x) = (\Lambda g. \text{ appKM } (\text{ evalK } f \cdot g) \cdot (\text{ evalK } x \cdot g))
\]

\[
\text{evalK } (\text{ Abs } N v e) = (\Lambda g. \text{ unitK } \cdot (\text{ ValKF } (\Lambda x. \text{ evalK } e \cdot (\text{ env-ext } v \cdot x \cdot g))))
\]

\[
\text{evalK } (\text{ Abs } V v e) = (\Lambda g. \text{ unitK } \cdot (\text{ ValKF } (\Lambda x. x. (\Lambda x'. \text{ evalK } e \cdot (\text{ env-ext } v \cdot (\text{ unitK } \cdot x') \cdot g) \cdot c))))
\]

\[
\text{evalK } (\text{ Diverge }) = (\Lambda g. \bot)
\]

\[
\text{evalK } (\text{ Fix } v e) = (\Lambda g. \mu x. \text{ evalK } e \cdot (\text{ env-ext } v \cdot x \cdot g))
\]

\[
\text{evalK } (tt) = (\Lambda g. \text{ unitK } \cdot \text{ ValKTT})
\]

\[
\text{evalK } (ff) = (\Lambda g. \text{ unitK } \cdot \text{ ValKFF})
\]

\[
\text{evalK } (\text{ Cond } i t e) = (\Lambda g. \text{ condK } (\text{ evalK } i \cdot g) \cdot (\text{ evalK } t \cdot g) \cdot (\text{ evalK } e \cdot g))
\]

\[
\text{evalK } (\text{ Num } n) = (\Lambda g. \text{ unitK } \cdot (\text{ ValKN } \cdot n))
\]

\[
\text{evalK } (\text{ Succ } e) = (\Lambda g. \text{ succK } (\text{ evalK } e \cdot g))
\]

\[
\text{evalK } (\text{ Pred } e) = (\Lambda g. \text{ predK } (\text{ evalK } e \cdot g))
\]

\[
\text{evalK } (\text{ IsZero } e) = (\Lambda g. \text{ isZeroK } (\text{ evalK } e \cdot g))
\]

To establish the chain completeness (admissibility) of our logical relation, we need to show that \(\text{ unitK }\) is an order monic, i.e., if \(\text{ unitK } \cdot x \subseteq \text{ unitK } \cdot y\) then \(x \subseteq y\). 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:

\[
\text{locale at-least-two-elements =}
\]

\[
\text{fixes some-non-bottom-element :: } \nu::\text{domain}
\]

\[
\text{assumes some-non-bottom-element: some-non-bottom-element } \neq \bot
\]

Following Reynolds (1974) and Filinski (2007, Remark 47) we use the following continuation:
lemma cont-below [simp, cont2cont]:
  cont (λx::'a::pcpo. 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)

context at-least-two-elements

abbreviation lr-theta-rep where
  lr-theta-rep ≡ λr. lr-eta-rep-N ⊔ lr-eta-rep-F r

abbreviation lr where
  lr ≡ λr. (mklr (fst (lr-rep r)), mklr (snd (lr-rep r)))(proof)(proof)(proof)

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

(proof)
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 $\eta: - \mapsto \rightarrow \quad \langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle$
where
$\eta: x \mapsto \rightarrow (x, x') \in \text{unlr} (\text{fst } F. \text{delta})$

abbreviation vrel $\vartheta: - \mapsto \rightarrow \quad \langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle\langle \text{proof}\rangle$
where
$\vartheta: y \mapsto \rightarrow (y, y') \in \text{unlr} (\text{snd } F. \text{delta})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})$

Theorem 1 from Reynolds (1974).

lemma AbsV-aux:
assumes $\eta: \text{ValF} \cdot f \mapsto \rightarrow \text{unitK} \cdot (\text{ValKF} \cdot f')$
shows $\eta: \text{ValF} \cdot (\text{strictify} \cdot f) \mapsto \rightarrow \text{unitK} \cdot (\text{ValKF} \cdot (\Lambda x \ c. \ x \cdot (\Lambda x'. f' \cdot (\text{unitK} \cdot x'). c)))$;

theorem Theorem1:
assumes $\forall v. \eta: g \cdot v \mapsto \rightarrow g' \cdot v$
shows $\eta: \text{evalD} \cdot g \mapsto \rightarrow \text{evalK} \cdot g'$

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 \rightarrow 'o
 fixes OtoV :: 'o \rightarrow 'o ValK
 assumes OV: OtoV oo VtoO = ID

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

context value-retraction
begin

fun DtoKM-i :: nat \Rightarrow ValD \rightarrow 'o ValKM
and
KMSKtoD-i :: nat \Rightarrow 'o ValKM \rightarrow ValD
where
$\text{DtoKM-i } 0 = \bot$
$\| \text{DtoKM-i } (\text{Suc } n) = (\Lambda v. \text{case } v \text{ of }$
\begin{align*}
\text{ValF} \cdot f \Rightarrow \text{unitK} \cdot (\text{ValKF} \cdot (\text{cfun-map} \cdot (KMSKtoD-i n) \cdot (\text{DtoKM-i } n) \cdot f)) \\
\text{ValTT} \Rightarrow \text{unitK} \cdot \text{ValTT} \\
\text{ValFF} \Rightarrow \text{unitK} \cdot \text{ValFF} \\
\text{ValN} \cdot m \Rightarrow \text{unitK} \cdot (\text{ValKN} \cdot m)
\end{align*}

$\| \text{KMSKtoD-i } 0 = \bot$
$\| \text{KMSKtoD-i } (\text{Suc } n) = (\Lambda v. \text{case } OtoV \cdot (v \cdot VtoO) \text{ of }$
\begin{align*}
\text{ValKF} \cdot f \Rightarrow \text{ValF} \cdot (\text{cfun-map} \cdot (\text{DtoKM-i } n) \cdot (\text{KMSKtoD-i } n) \cdot f)
\end{align*}

22
ValKTT ⇒ ValTT
ValKFF ⇒ ValFF
ValKN·m ⇒ ValN·m)

abbreviation \( DtoKM \equiv (\bigsqcup_i DtoKM\_i \_i) \)
abbreviation \( KMtoD \equiv (\bigsqcup_i KMtoD\_i \_i) \)

Lemma 1 from Reynolds (1974).

**Lemma 1**: \( \eta : x \mapsto DtoKM \cdot x \)
\( \eta : x \mapsto x' \implies x = KMtoD \cdot x'(\text{proof}) \)

Theorem 2 from Reynolds (1974).

**Theorem 2**: \( evalD e \cdot \varrho = KMtoD \cdot (evalK e \cdot (DtoKM oo \varrho)) \)

(proof)

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 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


23


