Logical Relations for PCF

Peter Gammie

March 17, 2025

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

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 $\perp$ and is closed under countable chains:

definition *admS* :: *'a::pcpo set set* **where**
  $admS \equiv \{\ R :: 'a\ set.\ \perp \in R \wedge adm\ (\lambda x.\ x \in R)\ \}$

typedef ('a::pcpo) *admS* = $\{\ x::'a::pcpo\ set\ .\ x \in admS\ \}$
  morphisms *unlr mklr* ⟨*proof*⟩

These sets form a complete lattice.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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 $'d$ *lf-rep* $= 'd$ *admS dual* $\times$ $'d$ *admS* $\Rightarrow$ $'d$ *set*
type-synonym $'d$ *lf* $= 'd$ *admS dual* $\times$ $'d$ *admS* $\Rightarrow$ $'d$ *admS*

The predicate *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 $'i$ to the domain $'d$. Note that the minimal invariant is for the domain $'d$ only.

abbreviation
  eRSV :: $('d{::}pcpo \to 'd) \Rightarrow ('i{::}type \Rightarrow 'd)$ *admS dual* $\Rightarrow ('i \Rightarrow 'd)$ *admS* $\Rightarrow$ *bool*
where
  eRSV e R S $\equiv \forall\, d \in$ *unlr* (*undual R*). $(\lambda x.\ e{\cdot}(d\ x)) \in$ *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 *DomSol* =
  fixes F :: $'a{::}order$ *dual* $\times$ $'a{::}order \Rightarrow 'a$
  assumes *monoF*: *mono F*
begin

definition *sym-lr* :: $'a$ *dual* $\times$ $'a \Rightarrow 'a$ *dual* $\times$ $'a$
where
  sym-lr = (λ(*rm*, *rp*). (*dual* (*F* (*dual rp*, *undual rm*)), *F* (*rm*, *rp*)))

lemma *sym-lr-mono*:
  mono sym-lr
⟨*proof*⟩

end

locale *DomSolV* = *DomSol F* :: $('i{::}type \Rightarrow 'd{::}pcpo)$ *lf* **for** *F* +
  fixes δ :: $('d{::}pcpo \to 'd) \to 'd \to 'd$
  assumes *min-inv-ID*: *fix*$\cdot\delta$ = *ID*
  assumes *eRSV-deltaF*:
    $\bigwedge(e :: 'd \to 'd)\ (R :: ('i \Rightarrow 'd)$ *admS dual*$)\ (S :: ('i \Rightarrow 'd)$ *admS*$)$.
    eRSV e R S $\Longrightarrow$ *eRSV* $(\delta{\cdot}e)$ (*dual* (*F* (*dual S*, *undual R*))) (*F* (*R*, *S*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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* $\equiv$ *delta-pos*

lemma *delta-sol*: $delta = F\ (dual\ delta,\ delta)\langle proof\rangle$
lemma *delta-unique*:
  assumes r: $F\ (dual\ r,\ r) = r$
  shows $r = delta\langle proof\rangle$
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 $synlr :: ('d::pcpo \times 'a::type)\ set\ set$ **where**
  $synlr \equiv \{\ R :: ('d \times 'a)\ set.\ \forall\, a.\ \{\ d.\ (d,\ a) \in R\ \} \in admS\ \}$

typedef $('d::pcpo,\ 'a::type)\ synlr = \{\ x::('d \times 'a)\ set.\ x \in synlr\ \}$
  morphisms *unsynlr mksynlr* $\langle proof\rangle$

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.

$\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$
Again we define functors on $('d,\ 'a)\ synlr$.

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

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

abbreviation
  $eRSS :: ('d::pcpo \rightarrow 'd) \Rightarrow ('d,\ 'a::type)\ synlr\ dual \Rightarrow ('d,\ 'a)\ synlr \Rightarrow bool$
where
  $eRSS\ e\ R\ S \equiv \forall\, (d,\ a) \in unsynlr\ (undual\ R).\ (e\cdot d,\ a) \in unsynlr\ S$

locale $DomSolSyn = DomSol\ F :: ('d::pcpo,\ 'a::type)\ synlf$ **for** F +
  fixes $\delta :: ('d::pcpo \rightarrow 'd) \rightarrow 'd \rightarrow 'd$
  assumes *min-inv-ID*: $fix\cdot\delta = ID$
  assumes *min-inv-strict*: $\bigwedge r.\ \delta\cdot r\cdot\bot = \bot$
  assumes *eRS-deltaF*:
    $\bigwedge(e :: 'd \rightarrow 'd)\ (R :: ('d,\ 'a)\ synlr\ dual)\ (S :: ('d,\ 'a)\ synlr).$
      $[\![\ e\cdot\bot = \bot;\ eRSS\ e\ R\ S\ ]\!] \Longrightarrow eRSS\ (\delta\cdot e)\ (dual\ (F\ (dual\ S,\ undual\ R)))\ (F\ (R,$ $S))\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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 $('am, 'bm, 'av, 'bv)$ *lr-pair* $= ('am \times 'bm)$ *admS* $\times ('av \times 'bv)$ *admS*

type-synonym $('am, 'bm, 'av, 'bv)$ *lf-pair-rep* $=$
$('am, 'bm, 'av, 'bv)$ *lr-pair dual* $\times ('am, 'bm, 'av, 'bv)$ *lr-pair* $\Rightarrow (('am \times 'bm)$ *set* $\times ('av \times 'bv)$ *set*$)$

type-synonym $('am, 'bm, 'av, 'bv)$ *lf-pair* $=$
$('am, 'bm, 'av, 'bv)$ *lr-pair dual* $\times ('am, 'bm, 'av, 'bv)$ *lr-pair* $\Rightarrow (('am \times 'bm)$ *admS* $\times ('av \times 'bv)$ *admS*$)$

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

abbreviation
eRSP $:: (('am::pcpo \to 'am) \times ('av::pcpo \to 'av))$
$\Rightarrow (('bm::pcpo \to 'bm) \times ('bv::pcpo \to 'bv))$
$\Rightarrow (('am \times 'bm)$ *admS* $\times ('av \times 'bv)$ *admS*$)$ *dual*
$\Rightarrow ('am \times 'bm)$ *admS* $\times ('av \times 'bv)$ *admS*
$\Rightarrow$ *bool*
where
eRSP ea eb R S $\equiv$
$(\forall (am, bm) \in unlr\ (fst\ (undual\ R)).\ (fst\ ea{\cdot}am, fst\ eb{\cdot}bm) \in unlr\ (fst\ S))$
$\wedge (\forall (av, bv) \in unlr\ (snd\ (undual\ R)).\ (snd\ ea{\cdot}av, snd\ eb{\cdot}bv) \in unlr\ (snd\ S))$

locale *DomSolP* $=$ *DomSol* $F :: ('am::pcpo, 'bm::pcpo, 'av::pcpo, 'bv::pcpo)$ *lf-pair* **for** F +
fixes $ad :: (('am \to 'am) \times ('av \to 'av)) \to (('am \to 'am) \times ('av \to 'av))$
fixes $bd :: (('bm \to 'bm) \times ('bv \to 'bv)) \to (('bm \to 'bm) \times ('bv \to 'bv))$
assumes *ad-ID*: $fix{\cdot}ad = (ID, ID)$
assumes *bd-ID*: $fix{\cdot}bd = (ID, ID)$
assumes *ad-strict*: $\bigwedge r.\ fst\ (ad{\cdot}r){\cdot}\bot = \bot\ \bigwedge r.\ snd\ (ad{\cdot}r){\cdot}\bot = \bot$
assumes *bd-strict*: $\bigwedge r.\ fst\ (bd{\cdot}r){\cdot}\bot = \bot\ \bigwedge r.\ snd\ (bd{\cdot}r){\cdot}\bot = \bot$
assumes *eRSP-deltaF*:
$[\![\ eRSP\ ea\ eb\ R\ S;\ fst\ ea{\cdot}\bot = \bot;\ snd\ ea{\cdot}\bot = \bot;\ fst\ eb{\cdot}\bot = \bot;\ snd\ ea{\cdot}\bot = \bot\ ]\!]$
$\Longrightarrow eRSP\ (ad{\cdot}ea)\ (bd{\cdot}eb)\ (dual\ (F\ (dual\ S, undual\ R)))\ (F\ (R, S))\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof$

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.

domain *ValD* =
  ValF (**lazy** *appF* :: *ValD* → *ValD*)
 | *ValTT* | *ValFF*
 | *ValN* (**lazy** *nat*)

The **lazy** keyword means that the *ValF* constructor is lifted, i.e. $ValF \cdot \bot \neq \bot$, which further means that $ValF \cdot (\Lambda\ x.\ \bot) \neq \bot$.

The naturals are discretely ordered.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The minimal invariant for *ValD* is straightforward; the function *cfun-map·f·g·h* denotes *g oo h oo f*.

fixrec
  ValD-copy-rec :: (*ValD* → *ValD*) → (*ValD* → *ValD*)
where
  ValD-copy-rec·r·(*ValF·f*) = *ValF·*(*cfun-map·r·r·f*)
| *ValD-copy-rec·r·*(*ValTT*) = *ValTT*
| *ValD-copy-rec·r·*(*ValFF*) = *ValFF*
| *ValD-copy-rec·r·*(*ValN·n*) = *ValN·n*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

definition *cond* :: *ValD* → *ValD* → *ValD* → *ValD* **where**
  cond ≡ Λ *i t e. case i of ValF·f* ⇒ ⊥ | *ValTT* ⇒ *t* | *ValFF* ⇒ *e* | *ValN·n* ⇒ ⊥

definition *succ* :: *ValD* → *ValD* **where**
  succ ≡ Λ (*ValN·n*). *ValN·*(*n* + *1*)

definition *pred* :: *ValD* → *ValD* **where**
  pred ≡ Λ (*ValN·n*). *case n of 0* ⇒ ⊥ | *Suc n* ⇒ *ValN·n*

definition *isZero* :: *ValD* → *ValD* **where**
  isZero ≡ Λ (*ValN·n*). *if n* = *0 then ValTT else ValFF*

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

type-synonym *Var* = *var*

type-synonym $'a\ Env = Var \to 'a$

definition $env\text{-}empty :: 'a\ Env$ **where**
$env\text{-}empty \equiv \bot$

definition $env\text{-}ext :: Var \to 'a \to 'a\ Env \to 'a\ Env$ **where**
$env\text{-}ext \equiv \Lambda\ v\ x\ \varrho\ v'.\ if\ v = v'\ then\ x\ else\ \varrho{\cdot}v'$⟨proof⟩⟨proof⟩

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

type-synonym $EnvD = ValD\ Env$

primrec
$evalD :: expr \Rightarrow EnvD \to ValD$
where
$evalD\ (Var\ v) = (\Lambda\ \varrho.\ \varrho{\cdot}v)$
$|\ evalD\ (App\ f\ x) = (\Lambda\ \varrho.\ appF{\cdot}(evalD\ f{\cdot}\varrho){\cdot}(evalD\ x{\cdot}\varrho))$
$|\ evalD\ (AbsN\ v\ e) = (\Lambda\ \varrho.\ ValF{\cdot}(\Lambda\ x.\ evalD\ e{\cdot}(env\text{-}ext{\cdot}v{\cdot}x{\cdot}\varrho)))$
$|\ evalD\ (AbsV\ v\ e) = (\Lambda\ \varrho.\ ValF{\cdot}(strictify{\cdot}(\Lambda\ x.\ evalD\ e{\cdot}(env\text{-}ext{\cdot}v{\cdot}x{\cdot}\varrho))))$
$|\ evalD\ (Diverge) = (\Lambda\ \varrho.\ \bot)$
$|\ evalD\ (Fix\ v\ e) = (\Lambda\ \varrho.\ \mu\ x.\ evalD\ e{\cdot}(env\text{-}ext{\cdot}v{\cdot}x{\cdot}\varrho))$
$|\ evalD\ (tt) = (\Lambda\ \varrho.\ ValTT)$
$|\ evalD\ (ff) = (\Lambda\ \varrho.\ ValFF)$
$|\ evalD\ (Cond\ i\ t\ e) = (\Lambda\ \varrho.\ cond{\cdot}(evalD\ i{\cdot}\varrho){\cdot}(evalD\ t{\cdot}\varrho){\cdot}(evalD\ e{\cdot}\varrho))$
$|\ evalD\ (Num\ n) = (\Lambda\ \varrho.\ ValN{\cdot}n)$
$|\ evalD\ (Succ\ e) = (\Lambda\ \varrho.\ succ{\cdot}(evalD\ e{\cdot}\varrho))$
$|\ evalD\ (Pred\ e) = (\Lambda\ \varrho.\ pred{\cdot}(evalD\ e{\cdot}\varrho))$
$|\ evalD\ (IsZero\ e) = (\Lambda\ \varrho.\ isZero{\cdot}(evalD\ e{\cdot}\varrho))$

abbreviation $eval' :: expr \Rightarrow ValD\ Env \Rightarrow ValD$ (‹⟦-⟧-› [0,1000] 60) **where**
$eval'\ M\ \varrho \equiv evalD\ M{\cdot}\varrho$

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 $Y\text{-}delta :: expr$ **where**
$Y\text{-}delta \equiv AbsN\ 0\ (AbsN\ 1\ (App\ (Var\ 0)\ (App\ (Var\ 1)\ (Var\ 1))))$

definition $Ycomb :: expr$ **where**
$Ycomb \equiv AbsN\ 0\ (App\ (App\ Y\text{-}delta\ (Var\ 0))\ (App\ Y\text{-}delta\ (Var\ 0)))$

definition $fixD :: ValD \to ValD$ **where**
$fixD \equiv \Lambda\ (ValF{\cdot}f).\ fix{\cdot}f$

lemma Y: $[\![Ycomb]\!]\varrho = ValF{\cdot}fixD$⟨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* → *ValD*. This is complicated by the *ValF* tag and having strict function abstraction.

definition
logical-relation :: (*'i*::*type* ⇒ *ValD*) *set* ⇒ *bool*
where
logical-relation R ≡
(∀ *fs* ∈ *R*. ∀ *xs* ∈ *R*. (λ*j*. *appF*·(*fs j*)·(*xs j*)) ∈ *R*)
∧ (∀ *fs* ∈ *R*. ∀ *xs* ∈ *R*. (λ*j*. *strictify*·(*appF*·(*fs j*))·(*xs j*)) ∈ *R*)
∧ (∀ *fs*. (∀ *xs* ∈ *R*. (λ*j*. (*fs j*)·(*xs j*)) ∈ *R*) ⟶ (λ*j*. *ValF*·(*fs j*)) ∈ *R*)
∧ (∀ *fs*. (∀ *xs* ∈ *R*. (λ*j*. *strictify*·(*fs j*)·(*xs j*)) ∈ *R*) ⟶ (λ*j*. *ValF*·(*strictify*·(*fs j*))) ∈ *R*)
∧ (∀ *xs* ∈ *R*. (λ*j*. *fixD*·(*xs j*)) ∈ *R*)
∧ (∀ *cs* ∈ *R*. ∀ *ts* ∈ *R*. ∀ *es* ∈ *R*. (λ*j*. *cond*·(*cs j*)·(*ts j*)·(*es j*)) ∈ *R*)
∧ (∀ *xs* ∈ *R*. (λ*j*. *succ*·(*xs j*)) ∈ *R*)
∧ (∀ *xs* ∈ *R*. (λ*j*. *pred*·(*xs j*)) ∈ *R*)
∧ (∀ *xs* ∈ *R*. (λ*j*. *isZero*·(*xs j*)) ∈ *R*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

definition
PCF-consts-rel :: (*'i*::*type* ⇒ *ValD*) *set* ⇒ *bool*
where
PCF-consts-rel R ≡
⊥ ∈ *R*
∧ (λ*i*. *ValTT*) ∈ *R*
∧ (λ*i*. *ValFF*) ∈ *R*
∧ (∀ *n*. (λ*i*. *ValN*·*n*) ∈ *R*)⟨*proof*⟩⟨*proof*⟩

abbreviation
PCF-lr R ≡ *adm* (λ*x*. *x* ∈ *R*) ∧ *logical-relation R* ∧ *PCF-consts-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 *ϱ*: ∀ *v*. (λ*i*. *ϱ i*·*v*) ∈ *R*
shows (λ*i*. ⟦*M*⟧(*ϱ i*)) ∈ *R*⟨*proof*⟩

We can use this result to show that there is no PCF term that maps the vector $args \in R$ to $result \notin R$ 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
  $appFLv\ f\ args \equiv (\lambda i.\ foldl\ (\lambda f\ x.\ appF{\cdot}f{\cdot}(x\ i))\ f\ args)$

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 :: $('i{::}type \Rightarrow ValD)\ set$
  fixes $args$ :: $('i \Rightarrow ValD)\ list$
  fixes $result$ :: $'i \Rightarrow ValD$
  assumes *lr*: *PCF-lr R*
  assumes *args*: $set\ args \subseteq R$
  assumes *result*: $result \notin R$
  shows $\neg(\exists (f{::}ValD).\ definable\ f \wedge appFLv\ f\ args = result)\langle proof\rangle$

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-circuting or except that if the first argument is $\perp$ and the second one is *ValTT*, we get *ValTT* (and not $\perp$). It is continuous and then have included in the *ValD* domain.

definition *por* :: $ValD \Rightarrow ValD \Rightarrow ValD$ (‹- *por* -› [31,30] 30) **where**
  $x\ por\ y \equiv$
    $if\ x = ValTT\ then\ ValTT$
      $else\ if\ y = ValTT\ then\ ValTT$
        $else\ if\ (x = ValFF \wedge y = ValFF)\ then\ ValFF\ else\ \perp$

The defining properties of parallel-or.

lemma *POR-simps* [*simp*]:
  $(ValTT\ por\ y) = ValTT$
  $(x\ por\ ValTT) = ValTT$
  $(ValFF\ por\ ValFF) = ValFF$
  $(ValFF\ por\ \perp) = \perp$
  $(ValFF\ por\ ValN{\cdot}n) = \perp$
  $(ValFF\ por\ ValF{\cdot}f) = \perp$
  $(\perp\ por\ ValFF) = \perp$
  $(ValN{\cdot}n\ por\ ValFF) = \perp$
  $(ValF{\cdot}f\ por\ ValFF) = \perp$
  $(\perp\ por\ \perp) = \perp$
  $(\perp\ por\ ValN{\cdot}n) = \perp$
  $(\perp\ por\ ValF{\cdot}f) = \perp$
  $(ValN{\cdot}n\ por\ \perp) = \perp$
  $(ValF{\cdot}f\ por\ \perp) = \perp$

$(ValN \cdot m\ por\ ValN \cdot n) = \bot$
$(ValN \cdot n\ por\ ValF \cdot f) = \bot$
$(ValF \cdot f\ por\ ValN \cdot n) = \bot$
$(ValF \cdot f\ por\ ValF \cdot g) = \bot$
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

We need three-element vectors.

datatype *Three* = *One* | *Two* | *Three*

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

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

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
  POR-base-lf-rep :: (*Three* ⇒ *ValD*) *lf-rep*
where
  POR-base-lf-rep ≡ λ(*mR*, *pR*).
    { (λ*i*. *ValTT*) } ∪ { (λ*i*. *ValFF*) } — $x = y = z$ for bools
  ∪ (⋃ *n*. { (λ*i*. *ValN*·*n*) }) — $x = y = z$ for numerals
  ∪ { *f* . *f One* = ⊥ } — $x = \bot$
  ∪ { *f* . *f Two* = ⊥ } — $y = \bot$

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

definition
  fn-lf-rep :: ('*i*::*type* ⇒ *ValD*) *lf-rep*
where
  fn-lf-rep ≡ λ(*mR*, *pR*). { λ*i*. *ValF*·(*fs i*) |*fs*. ∀ *xs* ∈ *unlr* (*undual mR*). (λ*j*. (*fs j*)·(*xs j*)) ∈ *unlr pR*
}⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

definition *POR-lf-rep* :: (*Three* ⇒ *ValD*) *lf-rep* **where**
  POR-lf-rep R ≡ *POR-base-lf-rep R* ∪ *fn-lf-rep R*

abbreviation *POR-lf* ≡ λ*r*. *mklr* (*POR-lf-rep r*)⟨proof⟩⟨proof⟩

Again it yields an admissible relation and is monotonic.

We need to show the functor respects the minimal invariant.

lemma *min-inv-POR-lf*:
  assumes *eRSV e R′ S′*
  shows *eRSV* (*ValD-copy-rec*·*e*) (*dual* (*POR-lf* (*dual S′*, *undual R′*))) (*POR-lf* (*R′*, *S′*))⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨p

We can show that the solution satisfies the expectations of the fundamental theorem *lr-fundamental*.

lemma *PCF-lr-POR-delta*: *PCF-lr* (*unlr POR.delta*)⟨proof⟩

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 *POR-arg1-rel* **where**

POR-arg1-rel ≡ λi. *case i of One* ⇒ *ValTT* | *Two* ⇒ ⊥ | *Three* ⇒ *ValFF*

definition *POR-arg2-rel* **where**
POR-arg2-rel ≡ λi. *case i of One* ⇒ ⊥ | *Two* ⇒ *ValTT* | *Three* ⇒ *ValFF*

definition *POR-result-rel* **where**
POR-result-rel ≡ λi. *case i of One* ⇒ *ValTT* | *Two* ⇒ *ValTT* | *Three* ⇒ *ValFF*

lemma *lr-POR-arg1-rel*: *POR-arg1-rel* ∈ *unlr POR.delta*
⟨*proof*⟩

lemma *lr-POR-arg2-rel*: *POR-arg2-rel* ∈ *unlr POR.delta*
⟨*proof*⟩

lemma *lr-POR-result-rel*: *POR-result-rel* ∉ *unlr POR.delta*⟨*proof*⟩

Parallel-or satisfies these tests:

theorem *POR-sat*:
appFLv (*ValF*·(Λ *x*. *ValF*·(Λ *y*. *x por y*))) [*POR-arg1-rel*, *POR-arg2-rel*] = *POR-result-rel*
⟨*proof*⟩

... but is not PCF-definable:

theorem *POR-is-not-definable*:
shows ¬(∃ *f*. *definable f* ∧ *appFLv f* [*POR-arg1-rel*, *POR-arg2-rel*] = *POR-result-rel*)
⟨*proof*⟩

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* ⇒ *ValD* **where**
plotkin-exists f ≡
  if (*appF*·*f*·⊥ = *ValFF*)
    then ValFF
    else if (∃ *n*. *appF*·*f*·*n* = *ValTT*) *then ValTT else* ⊥⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

We can show this function is continuous.

lemma *cont-pe* [*cont2cont*, *simp*]: *cont plotkin-exists*⟨*proof*⟩⟨*proof*⟩

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 ≡ λi. *ValF*·(*case i of*
    0 ⇒ (Λ -. *ValFF*)
  | *Suc n* ⇒ (Λ (*ValN*·*x*). *if x* = *Suc n then ValTT else* ⊥))

definition *PE-result-rel* **where**
PE-result-rel ≡ λi. *case i of 0* ⇒ *ValFF* | *Suc n* ⇒ *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*:
  $appFLv\ (ValF{\cdot}(\Lambda\ x.\ plotkin\text{-}exists\ x))\ [PE\text{-}arg\text{-}rel] = PE\text{-}result\text{-}rel$
  ⟨*proof*⟩

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* ⇒ *ValD*) *lf-rep* **where**
  $PE\text{-}base\text{-}lf\text{-}rep \equiv \lambda(mR,\ pR).$
    $\{ \bot \}$
  $\cup \{ (\lambda i.\ ValTT) \} \cup \{ (\lambda i.\ ValFF) \}$ — $x = y = z$ for bools
  $\cup (\bigcup n.\ \{ (\lambda i.\ ValN{\cdot}n) \})$ — $x = y = z$ for numerals
  $\cup \{ f\ .\ f\ 1 = \bot \vee f\ 2 = \bot \}$ — Vectors that diverge on one or two.⟨*proof*⟩⟨*proof*⟩

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

definition *PE-lf-rep* :: (*nat* ⇒ *ValD*) *lf-rep* **where**
  $PE\text{-}lf\text{-}rep\ R \equiv PE\text{-}base\text{-}lf\text{-}rep\ R \cup fn\text{-}lf\text{-}rep\ R$

abbreviation $PE\text{-}lf \equiv \lambda r.\ mklr\ (PE\text{-}lf\text{-}rep\ r)$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The solution satisfies the expectations of the fundamental theorem:

lemma *PCF-lr-PE-delta*: *PCF-lr* (*unlr PE.delta*)⟨*proof*⟩
lemma *lr-PE-arg-rel*: $PE\text{-}arg\text{-}rel \in unlr\ PE.delta$⟨*proof*⟩
lemma *lr-PE-result-rel*: $PE\text{-}result\text{-}rel \notin unlr\ PE.delta$⟨*proof*⟩
theorem *PE-is-not-definable*: $\neg(\exists f.\ definable\ f \wedge appFLv\ f\ [PE\text{-}arg\text{-}rel] = PE\text{-}result\text{-}rel)$⟨*proof*⟩

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 db =
    DBVar var
  | DBApp db db
  | DBAbsN db
  | DBAbsV db
  | DBDiverge
  | DBFix db
  | DBtt
  | DBff
  | DBCond db db db
  | DBNum nat
  | DBSucc db
  | DBPred db
  | DBIsZero 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* may be free, and while we could develop a simpler account, we retain the traditional one.

```
fun
  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 (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 ⇒ 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<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 (c<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⟨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

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

Programs are closed expressions.

definition $\mathit{closed} :: \mathit{db} \Rightarrow \mathit{bool}$ **where**
$\mathit{closed}\ e \equiv \forall\, i.\ \neg\ \mathit{freedb}\ e\ i\langle\mathit{proof}\rangle\langle\mathit{proof}\rangle$

The direct denotational semantics is almost identical to that given in §3.1, apart from this change in the representation of environments.

definition $\mathit{env\text{-}empty\text{-}db} :: {'a}\ \mathit{Env}$ **where**
$\mathit{env\text{-}empty\text{-}db} \equiv \bot$

definition $\mathit{env\text{-}ext\text{-}db} :: {'a} \rightarrow {'a}\ \mathit{Env} \rightarrow {'a}\ \mathit{Env}$ **where**
$\mathit{env\text{-}ext\text{-}db} \equiv \Lambda\ x\ \varrho\ v.\ (\mathit{case}\ v\ \mathit{of}\ 0 \Rightarrow x \mid \mathit{Suc}\ v' \Rightarrow \varrho\cdot v')\langle\mathit{proof}\rangle\langle\mathit{proof}\rangle$

primrec
$\mathit{evalDdb} :: \mathit{db} \Rightarrow \mathit{ValD}\ \mathit{Env} \rightarrow \mathit{ValD}$
where
$\mathit{evalDdb}\ (\mathit{DBVar}\ i) = (\Lambda\ \varrho.\ \varrho\cdot i)$
$|\ \mathit{evalDdb}\ (\mathit{DBApp}\ f\ x) = (\Lambda\ \varrho.\ \mathit{appF}\cdot(\mathit{evalDdb}\ f\cdot\varrho)\cdot(\mathit{evalDdb}\ x\cdot\varrho))$
$|\ \mathit{evalDdb}\ (\mathit{DBAbsN}\ e) = (\Lambda\ \varrho.\ \mathit{ValF}\cdot(\Lambda\ x.\ \mathit{evalDdb}\ e\cdot(\mathit{env\text{-}ext\text{-}db}\cdot x\cdot\varrho)))$
$|\ \mathit{evalDdb}\ (\mathit{DBAbsV}\ e) = (\Lambda\ \varrho.\ \mathit{ValF}\cdot(\mathit{strictify}\cdot(\Lambda\ x.\ \mathit{evalDdb}\ e\cdot(\mathit{env\text{-}ext\text{-}db}\cdot x\cdot\varrho))))$
$|\ \mathit{evalDdb}\ (\mathit{DBDiverge}) = (\Lambda\ \varrho.\ \bot)$
$|\ \mathit{evalDdb}\ (\mathit{DBFix}\ e) = (\Lambda\ \varrho.\ \mu\ x.\ \mathit{evalDdb}\ e\cdot(\mathit{env\text{-}ext\text{-}db}\cdot x\cdot\varrho))$
$|\ \mathit{evalDdb}\ (\mathit{DBtt}) = (\Lambda\ \varrho.\ \mathit{ValTT})$
$|\ \mathit{evalDdb}\ (\mathit{DBff}) = (\Lambda\ \varrho.\ \mathit{ValFF})$
$|\ \mathit{evalDdb}\ (\mathit{DBCond}\ c\ t\ e) = (\Lambda\ \varrho.\ \mathit{cond}\cdot(\mathit{evalDdb}\ c\cdot\varrho)\cdot(\mathit{evalDdb}\ t\cdot\varrho)\cdot(\mathit{evalDdb}\ e\cdot\varrho))$
$|\ \mathit{evalDdb}\ (\mathit{DBNum}\ n) = (\Lambda\ \varrho.\ \mathit{ValN}\cdot n)$
$|\ \mathit{evalDdb}\ (\mathit{DBSucc}\ e) = (\Lambda\ \varrho.\ \mathit{succ}\cdot(\mathit{evalDdb}\ e\cdot\varrho))$
$|\ \mathit{evalDdb}\ (\mathit{DBPred}\ e) = (\Lambda\ \varrho.\ \mathit{pred}\cdot(\mathit{evalDdb}\ e\cdot\varrho))$
$|\ \mathit{evalDdb}\ (\mathit{DBIsZero}\ e) = (\Lambda\ \varrho.\ \mathit{isZero}\cdot(\mathit{evalDdb}\ e\cdot\varrho))\langle\mathit{proof}\rangle\langle\mathit{proof}\rangle$

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 $\mathit{index} :: \mathit{var}\ \mathit{list} \Rightarrow \mathit{var} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$ **where**
$\mathit{index}\ []\ v\ n = n$
$|\ \mathit{index}\ (h\ \#\ t)\ v\ n = (\mathit{if}\ v = h\ \mathit{then}\ n\ \mathit{else}\ \mathit{index}\ t\ v\ (\mathit{Suc}\ n))$

primrec

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

This semantics corresponds with the direct semantics for named expressions.

⟨*proof*⟩⟨*proof*⟩**lemma** *evalD-evalDdb*:
assumes $free\ e = []$
shows $[\![e]\!]\varrho = evalDdb\ (transdb\ e\ [])\cdot\varrho$
⟨*proof*⟩

Conversely, all de Bruijn expressions have named equivalents.

primrec
$transdb\text{-}inv :: db \Rightarrow (var \Rightarrow var) \Rightarrow var \Rightarrow var \Rightarrow expr$
where
$transdb\text{-}inv\ (DBVar\ i)\ \Gamma\ c\ k = Var\ (\Gamma\ i)$
$|\ transdb\text{-}inv\ (DBApp\ t1\ t2)\ \Gamma\ c\ k = App\ (transdb\text{-}inv\ t1\ \Gamma\ c\ k)\ (transdb\text{-}inv\ t2\ \Gamma\ c\ k)$
$|\ transdb\text{-}inv\ (DBAbsN\ e)\ \Gamma\ c\ k = AbsN\ (c + k)\ (transdb\text{-}inv\ e\ (case\text{-}nat\ (c + k)\ \Gamma)\ c\ (k + 1))$
$|\ transdb\text{-}inv\ (DBAbsV\ e)\ \Gamma\ c\ k = AbsV\ (c + k)\ (transdb\text{-}inv\ e\ (case\text{-}nat\ (c + k)\ \Gamma)\ c\ (k + 1))$
$|\ transdb\text{-}inv\ (DBDiverge)\ \Gamma\ c\ k = Diverge$
$|\ transdb\text{-}inv\ (DBFix\ e)\ \Gamma\ c\ k = Fix\ (c + k)\ (transdb\text{-}inv\ e\ (case\text{-}nat\ (c + k)\ \Gamma)\ c\ (k + 1))$
$|\ transdb\text{-}inv\ (DBtt)\ \Gamma\ c\ k = tt$
$|\ transdb\text{-}inv\ (DBff)\ \Gamma\ c\ k = ff$
$|\ transdb\text{-}inv\ (DBCond\ i\ t\ e)\ \Gamma\ c\ k =$
$Cond\ (transdb\text{-}inv\ i\ \Gamma\ c\ k)\ (transdb\text{-}inv\ t\ \Gamma\ c\ k)\ (transdb\text{-}inv\ e\ \Gamma\ c\ k)$
$|\ transdb\text{-}inv\ (DBNum\ n)\ \Gamma\ c\ k = (Num\ n)$
$|\ transdb\text{-}inv\ (DBSucc\ e)\ \Gamma\ c\ k = Succ\ (transdb\text{-}inv\ e\ \Gamma\ c\ k)$
$|\ transdb\text{-}inv\ (DBPred\ e)\ \Gamma\ c\ k = Pred\ (transdb\text{-}inv\ e\ \Gamma\ c\ k)$
$|\ transdb\text{-}inv\ (DBIsZero\ e)\ \Gamma\ c\ k = IsZero\ (transdb\text{-}inv\ e\ \Gamma\ c\ k)$⟨*proof*⟩

lemma *transdb-inv*:
assumes *closed e*
shows $transdb\ (transdb\text{-}inv\ e\ \Gamma\ c\ k)\ \Gamma' = e$⟨*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-FF[intro]:  val DBff
| v-TT[intro]:  val DBtt
| v-AbsN[intro]: val (DBAbsN e)
| v-AbsV[intro]: 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]:  DBff ⇓ DBff
| evalOP-CondTT[intro]: ⟦ C ⇓ DBtt; T ⇓ V ⟧ ⟹ DBCond C T E ⇓ V
| evalOP-CondFF[intro]: ⟦ C ⇓ DBff; E ⇓ V ⟧ ⟹ DBCond C T E  ⇓ V
| evalOP-Num[intro]:  DBNum n ⇓ DBNum n
| evalOP-Succ[intro]: P ⇓ DBNum n ⟹ DBSucc P ⇓ DBNum (Suc n)
| evalOP-Pred[intro]: P ⇓ DBNum (Suc n) ⟹ DBPred P ⇓ DBNum n
| evalOP-IsZeroTT[intro]: ⟦ E ⇓ DBNum 0 ⟧ ⟹ DBIsZero E ⇓ DBtt
| evalOP-IsZeroFF[intro]: ⟦ E ⇓ DBNum n; 0 < n ⟧ ⟹ DBIsZero E ⇓ DBff
```

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

```
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩theorem evalOP-sound:
  assumes P ⇓ V
  shows evalDdb P·ϱ = evalDdb V·ϱ⟨proof⟩
```

We can use soundness to conclude that POR is not definable operationally either. We rely on *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 ≡ closed e
    ∧ DBApp (DBApp e DBtt) DBDiverge ⇓ DBtt
    ∧ DBApp (DBApp e DBDiverge) DBtt ⇓ DBtt
    ∧ DBApp (DBApp e DBff) DBff ⇓ DBff
⟨proof⟩⟨proof⟩
lemma POR-is-not-operationally-definable: ¬isPORdb e⟨proof⟩
```

4.3 Computational Adequacy

The lemma *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 *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 $\{\bot\} \times \{P.\ closed\ P\}$. To that end we define the type of programs.

typedef *Prog* = { *P*. *closed P* }
  morphisms *unProg mkProg* ⟨*proof*⟩

definition
  ca-lf-rep :: (*ValD*, *Prog*) *synlf-rep*
where
  ca-lf-rep ≡ λ(*rm*, *rp*).
    ({⊥} × *UNIV*)
    ∪ { (*d*, *P*) |*d P*.
      (∃ *n*. *d* = *ValN*·*n* ∧ *unProg P* ⇓ *DBNum n*)
     ∨ (*d* = *ValTT* ∧ *unProg P* ⇓ *DBtt*)
     ∨ (*d* = *ValFF* ∧ *unProg P* ⇓ *DBff*)
     ∨ (∃ *f M*. *d* = *ValF*·*f* ∧ *unProg P* ⇓ *DBAbsN M*
          ∧ (∀ (*x*, *X*) ∈ *unsynlr* (*undual rm*). (*f*·*x*, *mkProg* (*M*<*unProg X*/*0*>)) ∈ *unsynlr rp*))
     ∨ (∃ *f M*. *d* = *ValF*·*f* ∧ *unProg P* ⇓ *DBAbsV M* ∧ *f*·⊥ = ⊥
          ∧ (∀ (*x*, *X*) ∈ *unsynlr* (*undual rm*). ∀ *V*. *unProg X* ⇓ *V*
               ⟶ (*f*·*x*, *mkProg* (*M*<*V*/*0*>)) ∈ *unsynlr rp*)) }

abbreviation *ca-lr* :: (*ValD*, *Prog*) *synlf* **where**
  ca-lr ≡ λ*r*. *mksynlr* (*ca-lf-rep r*)

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

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

interpretation *ca*: *DomSolSyn ca-lr ValD-copy-rec*
  ⟨*proof*⟩

definition *ca-lr-syn* :: *ValD* ⇒ *db* ⇒ *bool* (‹- ◃ -› [*80*,*80*] *80*) **where**
  d ◃ *P* ≡ (*d*, *P*) ∈ { (*x*, *unProg Y*) |*x Y*. (*x*, *Y*) ∈ *unsynlr ca.delta* }⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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
  closing-subst :: *db* ⇒ (*var* ⇒ *db*) ⇒ *var* ⇒ *db*
where
  closing-subst (*DBVar i*) Γ *k* = (*if k* ≤ *i then* Γ (*i* − *k*) *else DBVar i*)
| *closing-subst* (*DBApp t u*) Γ *k* = *DBApp* (*closing-subst t* Γ *k*) (*closing-subst u* Γ *k*)
| *closing-subst* (*DBAbsN t*) Γ *k* = *DBAbsN* (*closing-subst t* Γ (*k* + *1*))
| *closing-subst* (*DBAbsV t*) Γ *k* = *DBAbsV* (*closing-subst t* Γ (*k* + *1*))
| *closing-subst* (*DBFix e*) Γ *k* = *DBFix* (*closing-subst e* Γ (*k* + *1*))
| *closing-subst* (*DBCond c t e*) Γ *k* =
        DBCond (*closing-subst c* Γ *k*) (*closing-subst t* Γ *k*) (*closing-subst e* Γ *k*)

| *closing-subst* (*DBSucc* *e*) Γ *k* = *DBSucc* (*closing-subst* *e* Γ *k*)
| *closing-subst* (*DBPred* *e*) Γ *k* = *DBPred* (*closing-subst* *e* Γ *k*)
| *closing-subst* (*DBIsZero* *e*) Γ *k* = *DBIsZero* (*closing-subst* *e* Γ *k*)
| *closing-subst* *x* Γ *k* = *x*

We can show it has the expected properties when all terms in Γ are closed.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
The key lemma is shown by induction over *e* for arbitrary environments (Γ and ϱ):

lemma *ca-open*:
  assumes ∀ *v*. *freedb* *e* *v* ⟶ ϱ·*v* ◃ Γ *v* ∧ *closed* (Γ *v*)
  shows *evalDdb* *e*·ϱ ◃ *closing-subst* *e* Γ *0*⟨*proof*⟩

lemma *ca-closed*:
  assumes *closed* *e*
  shows *evalDdb* *e*·*env-empty-db* ◃ *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 *AbsN*s. 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*> ≡ *closing-subst* *C* (λ-. *e*) *0*⟨*proof*⟩⟨*proof*⟩

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*⟨*proof*⟩

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
$e1 \trianglelefteq e2 \equiv \forall C.\ \exists V1.\ C\text{<}e1\text{>} \Downarrow V1 \longrightarrow (\exists V2.\ C\text{<}e2\text{>} \Downarrow V2 \wedge V1 \sim V2)$

Contextually-equivalent programs refine each other.

definition
contextually-equivalent :: *db* ⇒ *db* ⇒ *bool* (‹- ≈ -›)
where
$e1 \approx e2 \equiv e1 \trianglelefteq e2 \wedge e2 \trianglelefteq e1$⟨*proof*⟩⟨*proof*⟩

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

theorem *computational-adequacy*:
assumes *1*: *closed e1*
assumes *2*: *closed e2*
assumes *D*: *evalDdb e1·env-empty-db* = *evalDdb e2·env-empty-db*
shows $e1 \approx e2$⟨*proof*⟩

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 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 $('o\ ValK,\ 'o)\ K \to ('o\ ValK,\ 'o)\ K$ as our language includes call-by-name.

type-synonym $('a,\ 'o)\ K = ('a \to 'o) \to 'o$

domain $'o\ ValK$
$= ValKF\ (\textbf{lazy}\ appKF :: ('o\ ValK,\ 'o)\ K \to ('o\ ValK,\ 'o)\ K)$
$\mid ValKTT \mid ValKFF$
$\mid ValKN\ (\textbf{lazy}\ nat)$

type-synonym $'o\ ValKM = ('o\ ValK,\ 'o)\ K\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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

definition $unitK :: 'o\ ValK \to 'o\ ValKM$ **where**
$unitK \equiv \Lambda\ a.\ \Lambda\ c.\ c \cdot a$

definition $bindK :: 'o\ ValKM \to ('o\ ValK \to 'o\ ValKM) \to 'o\ ValKM$ **where**
$bindK \equiv \Lambda\ m\ k.\ \Lambda\ c.\ m \cdot (\Lambda\ a.\ k \cdot a \cdot c)$

definition $appKM :: 'o\ ValKM \to 'o\ ValKM \to 'o\ ValKM$ **where**
$appKM \equiv \Lambda\ fK\ xK.\ bindK \cdot fK \cdot (\Lambda\ (ValKF \cdot f).\ f \cdot xK)\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

The interpretations of the constants.

definition
$condK :: 'o\ ValKM \to 'o\ ValKM \to 'o\ ValKM \to 'o\ ValKM$
where
$condK \equiv \Lambda\ iK\ tK\ eK.\ bindK \cdot iK \cdot (\Lambda\ i.\ case\ i\ of$
$ValKF \cdot f \Rightarrow \bot \mid ValKTT \Rightarrow tK \mid ValKFF \Rightarrow eK \mid ValKN \cdot n \Rightarrow \bot)$

definition $succK :: 'o\ ValKM \to 'o\ ValKM$ **where**
$succK \equiv \Lambda\ nK.\ bindK \cdot nK \cdot (\Lambda\ (ValKN \cdot n).\ unitK \cdot (ValKN \cdot (n + 1)))$

definition $predK :: 'o\ ValKM \to 'o\ ValKM$ **where**
$predK \equiv \Lambda\ nK.\ bindK \cdot nK \cdot (\Lambda\ (ValKN \cdot n).\ case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ n \Rightarrow unitK \cdot (ValKN \cdot n))$

definition $isZeroK :: 'o\ ValKM \to 'o\ ValKM$ **where**
$isZeroK \equiv \Lambda\ nK.\ bindK \cdot nK \cdot (\Lambda\ (ValKN \cdot n).\ unitK \cdot (if\ n = 0\ then\ ValKTT\ else\ ValKFF))$

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

type-synonym $'o\ EnvK = 'o\ ValKM\ Env$

primrec
$evalK :: expr \Rightarrow 'o\ EnvK \to 'o\ ValKM$
where
$evalK\ (Var\ v) = (\Lambda\ \varrho.\ \varrho \cdot v)$
$\mid evalK\ (App\ f\ x) = (\Lambda\ \varrho.\ appKM \cdot (evalK\ f \cdot \varrho) \cdot (evalK\ x \cdot \varrho))$
$\mid evalK\ (AbsN\ v\ e) = (\Lambda\ \varrho.\ unitK \cdot (ValKF \cdot (\Lambda\ x.\ evalK\ e \cdot (env\text{-}ext \cdot v \cdot x \cdot \varrho))))$
$\mid evalK\ (AbsV\ v\ e) = (\Lambda\ \varrho.\ unitK \cdot (ValKF \cdot (\Lambda\ x\ c.\ x \cdot (\Lambda\ x'.\ evalK\ e \cdot (env\text{-}ext \cdot v \cdot (unitK \cdot x') \cdot \varrho) \cdot c))))$
$\mid evalK\ (Diverge) = (\Lambda\ \varrho.\ \bot)$
$\mid evalK\ (Fix\ v\ e) = (\Lambda\ \varrho.\ \mu\ x.\ evalK\ e \cdot (env\text{-}ext \cdot v \cdot x \cdot \varrho))$
$\mid evalK\ (tt) = (\Lambda\ \varrho.\ unitK \cdot ValKTT)$

| *evalK* (*ff*) = ($\Lambda\ \varrho$. *unitK*·*ValKFF*)
| *evalK* (*Cond i t e*) = ($\Lambda\ \varrho$. *condK*·(*evalK i*·ϱ)·(*evalK t*·ϱ)·(*evalK e*·ϱ))
| *evalK* (*Num n*) = ($\Lambda\ \varrho$. *unitK*·(*ValKN*·*n*))
| *evalK* (*Succ e*) = ($\Lambda\ \varrho$. *succK*·(*evalK e*·ϱ))
| *evalK* (*Pred e*) = ($\Lambda\ \varrho$. *predK*·(*evalK e*·ϱ))
| *evalK* (*IsZero e*) = ($\Lambda\ \varrho$. *isZeroK*·(*evalK e*·ϱ))

To establish the chain completeness (admissibility) of our logical relation, we need to show that *unitK* is an *order monic*, i.e., if *unitK*·$x \sqsubseteq$ *unitK*·y then $x \sqsubseteq y$. 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:

locale *at-least-two-elements* =
  fixes *some-non-bottom-element* :: *'o*::*domain*
  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* $x \sqsubseteq d$ *then* $\bot$ *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*) *lf-pair-rep*
type-synonym *'o lflf* = (*ValD*, *'o ValKM*, *ValD* → *ValD*, *'o ValKM* → *'o ValKM*) *lf-pair*

context *at-least-two-elements*
begin

abbreviation *lr-eta-rep-N* **where**
  lr-eta-rep-N ≡ { (*e*, *e'*) .
      ($e = \bot \wedge e' = \bot$)
    ∨ (*e* = *ValTT* ∧ *e'* = *unitK*·*ValKTT*)
    ∨ (*e* = *ValFF* ∧ *e'* = *unitK*·*ValKFF*)
    ∨ (∃ *n*. *e* = *ValN*·*n* ∧ *e'* = *unitK*·(*ValKN*·*n*)) }

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

definition *lr-eta-rep* **where**

lr-eta-rep $\equiv \lambda r.$ *lr-eta-rep-N* $\cup$ *lr-eta-rep-F r*

definition *lr-theta-rep* **where**
lr-theta-rep $\equiv \lambda(rm,\ rp).\ \{\ (f,\ f')\ .$
$(\forall\,(x,\ x') \in$ *unlr* (*fst* (*undual rm*)). $(f{\cdot}x,\ f'{\cdot}x') \in$ *unlr* (*fst rp*)) }

definition *lr-rep* :: $'o$ *lfr* **where**
lr-rep $\equiv \lambda r.$ (*lr-eta-rep r*, *lr-theta-rep r*)

abbreviation *lr* :: $'o$ *lflf* **where**
lr $\equiv \lambda r.$ (*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.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩**sub**
cale *at-least-two-elements* < *F*: *DomSolP lr ValD-copy-rec ValK-copy-rec*
⟨*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* (‹η: - $\mapsto$ -› [*50*, *51*] *50*) **where**
η: $x \mapsto x' \equiv (x,\ x') \in$ *unlr* (*fst F.delta*)

abbreviation *vrel* (‹ϑ: - $\mapsto$ -› [*50*, *51*] *50*) **where**
ϑ: $y \mapsto y' \equiv (y,\ y') \in$ *unlr* (*snd F.delta*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

Theorem 1 from Reynolds (1974).

lemma *AbsV-aux*:
assumes η: *ValF*$\cdot f \mapsto$ *unitK*$\cdot$(*ValKF*$\cdot f'$)
shows η: *ValF*$\cdot$(*strictify*$\cdot f$) $\mapsto$ *unitK*$\cdot$(*ValKF*$\cdot$(Λ *x c*. $x\cdot$(Λ x'. $f'\cdot$(*unitK*$\cdot x'$)$\cdot c$)))⟨*proof*⟩

theorem *Theorem1*:
assumes $\forall\, v.\ \eta$: $\varrho\cdot v \mapsto \varrho'\cdot v$
shows η: *evalD e*$\cdot\varrho \mapsto$ *evalK e*$\cdot\varrho'$⟨*proof*⟩
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* $VtoO \cdot (ValKN \cdot 0)$
⟨*proof*⟩

context *value-retraction*
begin

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

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

abbreviation $DtoKM \equiv (\bigsqcup i.\ DtoKM\text{-}i\ i)$
abbreviation $KMtoD \equiv (\bigsqcup i.\ KMtoD\text{-}i\ i)$⟨*proof*⟩

Lemma 1 from Reynolds (1974).

lemma *Lemma1*:
  $\eta: x \mapsto DtoKM \cdot x$
  $\eta: x \mapsto x' \Longrightarrow x = KMtoD \cdot x'$⟨*proof*⟩

Theorem 2 from Reynolds (1974).

theorem *Theorem2*: $evalD\ e \cdot \varrho = KMtoD \cdot (evalK\ e \cdot (DtoKM\ oo\ \varrho))$
⟨*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 β-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 :: $db \Rightarrow db \Rightarrow bool$ (‹- $\rightarrow_v$ -› [*50*, *50*] *50*)
where
  betaN: $DBApp\ (DBAbsN\ u)\ v \rightarrow_v u$<$v/0$>
| *betaV*: $val\ v \Longrightarrow DBApp\ (DBAbsV\ u)\ v \rightarrow_v u$<$v/0$>
| $f \rightarrow_v f' \Longrightarrow DBApp\ f\ x \rightarrow_v DBApp\ f'\ x$
| $[\![f = DBAbsV\ u;\ x \rightarrow_v x']\!] \Longrightarrow DBApp\ f\ x \rightarrow_v DBApp\ f\ x'$
| $DBFix\ f \rightarrow_v f$<$DBFix\ f/0$>
| $DBCond\ DBtt\ t\ e \rightarrow_v t$
| $DBCond\ DBff\ t\ e \rightarrow_v e$
| $DBSucc\ (DBNum\ n) \rightarrow_v DBNum\ (Suc\ n)$
| $DBPred\ (DBNum\ (Suc\ n)) \rightarrow_v DBNum\ n$
| $DBIsZero\ (DBNum\ 0) \rightarrow_v DBtt$
| $0 < n \Longrightarrow DBIsZero\ (DBNum\ n) \rightarrow_v DBff$

abbreviation — The transitive, reflexive closure of the reduction relation.
  reduction-trc :: $db \Rightarrow db \Rightarrow bool$ (‹- $\rightarrow_v{}^*$ -› [*100*, *100*] *100*)
where
  reduction-trc ≡ *rtranclp reduction*

declare *reduction.intros*[*intro*!]

inductive-cases *reduction-inv*:
  $DBVar\ v \rightarrow_v t'$
  $DBApp\ f\ x \rightarrow_v t'$
  $DBAbsN\ u \rightarrow_v t'$
  $DBAbsV\ u \rightarrow_v t'$
  $DBFix\ f \rightarrow_v t'$
  $DBCond\ i\ t\ e \rightarrow_v t'$
  $DBff \rightarrow_v t'$
  $DBtt \rightarrow_v t'$
  $DBNum\ n \rightarrow_v t'$
  $DBSucc\ n \rightarrow_v t'$
  $DBPred\ n \rightarrow_v t'$
  $DBIsZero\ n \rightarrow_v t'$

lemma *reduction-val*:
  assumes *val v*
  assumes $v \rightarrow_v v'$
  shows *False*
⟨*proof*⟩

lemma *reduction-deterministic*:
  assumes $t \rightarrow_v t'$
  assumes $t \rightarrow_v t''$
  shows $t'' = t'$
⟨*proof*⟩

6.0.1 Reduction is consistent with evaluation

lemma *reduction-eval*:
  assumes $t \rightarrow_v t'$
  assumes $t' \Downarrow v$
  shows $t \Downarrow v$
⟨*proof*⟩

lemma *reduction-trc-eval*:
  assumes $t \rightarrow_v{}^* t'$
  assumes $t' \Downarrow v$
  shows $t \Downarrow v$
⟨*proof*⟩

theorem *reduction-trc-val-eval*:
  assumes $t \rightarrow_v{}^* v$
  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

N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In M. Leuschel and A. Podelski, editors, *PPDP*, pages 87–96. ACM, 2007.

N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In A. Porto and F. J. López-Fraguas, editors, *PPDP*, pages 301–312. ACM, 2009a.

N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in coq. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 115–130. Springer, 2009b.

S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors. *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics (MFPS '94)*, volume 802 of *LNCS*, 1994. Springer.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In Morrisett and Jones (2006), pages 206–217.

A. Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375 (1-3):41–75, 2007.

C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, USA, 1992.

P. Hudak, J. Hughes, S. L. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In B. G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

B. Huffman. Formal verification of monad transformers. In *ICFP 2012*, 2012a.

B. Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012b.

V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In Morrisett and Jones (2006), pages 141–152.

J. C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, Cambridge, MA, 1996.

J. G. Morrisett and S. L. Peyton Jones, editors. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, 2006. ACM.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

K. Mulmuley. *Full Abstraction and Semantic Equivalence*. MIT Press, 1987.

T. Nipkow. More Church-Rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.

A. M. Pitts. Computational adequacy via "mixed" inductive definitions. In Brookes et al. (1994), pages 72–82.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

G. D. Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4, School of Artificial Intelligence, Unversity of Edinburgh, 1973.

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

J. C. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming (ICALP '74)*, volume 14 of *LNCS*, pages 141–156. Springer, 1974.

R. Sethi and A. Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, 1980.

K. Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in LMS Lecture Note Series. Cambridge University Press, 1992.

A. Stoughton. Mechanizing logical relations. In Brookes et al. (1994), pages 359–377.

J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

C. Urban and J. Narboux. Formal sos-proofs for the lambda-calculus. *Electronic Notes on Theoretical Computer Science*, 247:139–155, 2009.

P. Wadler. The essence of functional programming (invited talk). In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*, Albuquerque, New Mexico, January 1992.

M. Wand and D. Vaillancourt. Relating models of backtracking. In C. Okasaki and K. Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*, pages 54–65. ACM, 2004.

G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.