An Operational Domain-theoretic Treatment of Recursive Types

Weng Kin Ho [1,2]

School of Computer Science
The University of Birmingham
Birmingham, United Kingdom

Abstract

We develop a domain theory for treating recursive types with respect to contextual equivalence. The principal approach taken here deviates from classical domain theory in that we do not produce the recursive types via the usual inverse limits constructions - we have it for free by working directly with the operational semantics. By extending type expressions to endofunctors on a 'syntactic' category, we establish algebraic compactness. To do this, we rely on an operational version of the minimal invariance property. In addition, we apply techniques developed herein to reason about FPC programs.

Key words: Operational domain theory, recursive types, FPC, realisable functor, algebraic compactness, generic approximation lemma, denotational semantics

1 Introduction

We develop a domain theory for treating recursive types with respect to contextual equivalence. The language we consider is sequential and has, in addition to recursive types, sum, product, function and lifed types. It is, by now, folklore that the domain-theoretic model of such a language is *computationally adequate* but fails to be *fully abstract*, i.e., the denotational equality of two terms implies their contextual equivalence but not the converse.

In order to cope with this phenomenon, we develop the operational counterpart of domain theory that deals with the solutions of recursive domain equations. Such an enterprise may be seen as an extension of a similar programme by M.H. Escardó and W.K. Ho in [6] for the language PCF. Indeed

[1] Ph.D. program sponsored by School of Computer Science International Research Studentships, University of Birmingham
[2] Email: W.K.Ho@cs.bham.ac.uk

many authors have already exported domain-theory into the study of the operational order. Amongst these are I.A. Mason, et al, [19], as well as L. Birkedal and R. Harper [5] who, in particular, gave a relational interpretation to recursive types in an operational setting. This paper complements with [5] in that we focus on the connection between an operational version of Freyd's algebraic compactness and the underlying domain structure of the contextual order. Our study also makes use of A.M. Pitts' operational tools (which can be traced back to works like [12,14,15,19]) developed in [22] which are recorded in Section 2.

We work with the diagonal category $\mathcal{A}$ built from the category of closed FPC types $\mathcal{C}$. Details of this are discussed in Section 3 in conjunction with the extension of formal type expressions to endofunctors on $\mathcal{A}$. Based on this, we introduce an operational notion of (parameterised) algebraic compactness.

The main result of this paper (in Section 4) asserts that $\mathcal{A}$ is (parameterised) algebraically compact operationally, the consequences of which are covered in the next two sections. In particular, every closed type has an SFP-structure from which one discovers an operational proof of the "generic approximation lemma" [16]. In Section 6, running examples taken from [22] demonstrate the versatility of this proof method as compared to others such as bisimulation techniques ([12,22]) and program fusion ([11]).

Throughout the discussion we assume certain pre-requisites from the reader which include the operational semantics of the call-by-name version of FPC ([17]), theory of recursive domain equations ([2,26]) and category theory ([18]).

2 Preliminaries

We choose to work with a call-by-name version of FPC[3] (Fixed Point Calculus). The most salient part of the operational semantics of FPC has to do with the following typing and evaluation rules:

$$\frac{\Gamma \vdash t : \sigma[\mu X.\sigma/X]}{\Gamma \vdash \mathrm{fold}(t) : \mu X.\sigma}\ (\mathrm{fold}) \qquad \frac{\Gamma \vdash t : \mu X.\sigma}{\Gamma \vdash \mathrm{unfold}(t) : \sigma[\mu X.\sigma/X]}\ (\mathrm{unfold})$$

$$\frac{}{\mathrm{fold}(t) \Downarrow \mathrm{fold}(t)}\ (\Downarrow \mathrm{fold}) \qquad \frac{t \Downarrow \mathrm{fold}(s) \quad s \Downarrow v}{\mathrm{unfold}(t) \Downarrow v}\ (\Downarrow \mathrm{unfold})$$

Note that sums of types are given an operational semantics corresponding to the separated sum in domain theory, namely:

$$\frac{}{\mathrm{inl}(t) \Downarrow \mathrm{inl}(t)}\ (\Downarrow \mathrm{inl}) \qquad \frac{}{\mathrm{inr}(t) \Downarrow \mathrm{inr}(t)}\ (\Downarrow \mathrm{inr})$$

[3] The call-by-value version of FPC was introduced by G.D. Plotkin [23] in his 1985 CSLI lectures.

$$\frac{s \Downarrow \mathrm{inl}(t) \quad s_1[t/x] \Downarrow v}{\mathrm{case}(s) \ \text{ of } \begin{cases} \mathrm{inl}(x).s_1 \\ \mathrm{inr}(y).s_2 \end{cases} \Downarrow v} \ (\Downarrow \mathrm{case1})$$

$$\frac{s \Downarrow \mathrm{inr}(t) \quad s_2[t/y] \Downarrow v}{\mathrm{case}(s) \ \text{ of } \begin{cases} \mathrm{inl}(x).s_1 \\ \mathrm{inr}(y).s_2 \end{cases} \Downarrow v} \ (\Downarrow \mathrm{case2})$$

The typing and evaluation rules concerning lifted types are given by:

$$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \mathrm{up}(x) : \sigma_\perp} \ (\mathrm{up}) \qquad \frac{\Gamma \vdash x : \sigma_\perp}{\Gamma \vdash \mathrm{down}(x) : \sigma} \ (\mathrm{down})$$

$$\frac{}{\mathrm{up}(x) \Downarrow \mathrm{up}(x)} \ (\Downarrow \mathrm{up}) \qquad \frac{x \Downarrow \mathrm{up}(t) \quad t \Downarrow v}{\mathrm{down}(x) \Downarrow v} \ (\Downarrow \mathrm{down})$$

In addition, we have:

$$\frac{\Gamma \vdash z : \sigma_\perp \quad \Gamma \vdash y : \tau}{\Gamma \vdash \mathrm{case}(z) \ \text{ of } \ \mathrm{up}(x).y : \tau} \ (\mathrm{caseup})$$

$$\frac{z \Downarrow \mathrm{up}(t) \quad y[t/x] \Downarrow v}{\mathrm{case}(z) \ \text{ of } \ \mathrm{up}(x).y \Downarrow v} \ (\Downarrow \mathrm{caseup})$$

By exploiting the recursive types, we define the fix operator as follows:

$$\mathrm{fix}_\sigma := \lambda f : \sigma \to \sigma . k(\mathrm{fold}^\tau(k))$$

where $\tau := \mu X.(X \to \sigma)$ and $k := \lambda x : \tau . f(\mathrm{unfold}^\tau(x)x)$. Thus we do not include an explicit fix operator.

Our theory relies heavily on three important facts.

(1) Every closed FPC type is *rational-chain complete.* We apply A.M. Pitts' operationally based theories of program equivalence [22] to FPC where contextual equivalence is taken with respect to the unit type $\Sigma := (\mu X.X)_\perp$. Most importantly, each closed type σ is (pre)ordered contextually (denoted by $\sqsubseteq_\sigma$) and is closed under the formation of rational chains:

$$\perp_\sigma \sqsubseteq_\sigma f(\perp_\sigma) \sqsubseteq_\sigma f^{(2)}(\perp_\sigma) \sqsubseteq_\sigma \ldots \sqsubseteq_\sigma f^{(n)}(\perp_\sigma) \sqsubseteq \ldots$$

where $\perp_\sigma := \mathrm{fix}(\lambda x : \sigma . x)$ and $f : \sigma \to \sigma$. Moreover, $\bigsqcup_n f^{(n)}(\perp) = \mathrm{fix}(f)$. The crucial point here is that rational chain-completeness is proven by purely operational means and is *independent* of the properties of recursive type expressions reported herein.

(2) Every program of function type is *rationally continuous*, i.e., for any $h : \sigma \to \tau$ and $f : \sigma \to \sigma$, it holds that

$$h(\textstyle\bigsqcup_n f^{(n)}(\perp_\sigma)) = \bigsqcup_n h \circ f^{(n)}(\perp_\sigma).$$

(3) fold and unfold are mutually inverse (modulo contextual equivalence). This can be justified by the η- and β-rules enjoyed by the contextual equivalence: fold $\circ$ unfold $=$ id (η-rule) and unfold $\circ$ fold $=$ id (β-rule).

Proofs of (1), (2) and (3) are simply a reworking of those developed by A.M. Pitts for the language PCFL in [22] and are thus omitted in this paper.

3 The categorical setting

This section gives an account of the categorical framework within which our theory is organised. Our approach, largely adapted from [1], turns out to be the appropriate option among others. We carefully explain this in two stages: (i) understand the basic type expressions (i.e., without the function types) as functors, and then (ii) consider those built from all possible type constructors.

The objects of $\mathcal{C}$ are the closed FPC types (i.e., type expressions with no free variables) and the morphisms are terms-in-context of the form $x : \sigma \vdash t : \tau$ for some type variable x and open term t such that $fv(t) \subseteq \{x\}$. Here $fv(t)$ denotes the set of free variables occurring in the term t. Given closed type σ, the identity morphism id_σ is just the $x : \sigma \vdash x : \sigma$ and the composition of two morphisms $x : \rho \vdash s : \sigma$ and $y : \sigma \vdash t : \tau$, for instance, is defined as $x : \rho \vdash t[s/y] : \tau$. Moreover, we identify morphisms $x : \sigma \vdash t : \tau$ and $x : \sigma \vdash t' : \tau$ as the same if t and t' are contextually equivalent. Here the contextual equivalence of open terms is defined using closed instantiations of the free variables.

Formal FPC type expressions are called *basic* if they are generated by the following fragment of the grammar:

$$\sigma := X \mid \sigma + \sigma \mid \sigma \times \sigma \mid \mu X.\sigma \mid \sigma_\perp$$

where X, Y range over type variables and σ, τ range over type expressions. In other words, these are type expressions which do not include the function-type constructor $\to$. For convenience of notation, we write:

$\vec{\sigma}$	for a sequence of closed types $\sigma_1, \dots, \sigma_n$
$\vec{t}$	for a sequence of closed terms $t_1, \dots, t_n$
$\vec{X}$	for a sequence of type variables $X_1, \dots, X_n$
$\vec{x}$	for a sequence of term variables $x_1, \dots, x_n$
$\vec{\sigma}/\vec{X}$	for the substitutions $\sigma_1/X_1, \dots, \sigma_n/X_n$
$\vec{t}/\vec{x}$	for the substitutions $t_1/x_1, \dots, t_n/x_n$.

A functor $T : \mathcal{C}^n \to \mathcal{C}$ is *basic* if

(1) there exists a basic type expression τ in context $\vec{X}$ such that for every sequence $\vec{\sigma}$ of closed types, it holds that $T(\vec{\sigma}) := \tau[\vec{\sigma}/\vec{X}]$, and

(2) there exist a sequence of term variables $\vec{f}$ and a term t with $fv(t) \subseteq \{f_1, \dots, f_n\}$ such that for all morphisms $\vec{u} : \vec{\sigma} \to \vec{\rho}$ (meaning $u_i : \sigma_i \to \rho_i$

for $i = 1, \ldots, n$) we have the term-in-context

$$f_i : (\sigma_i \to \rho_i)|_{i=1}^{n} \vdash t : \tau[\vec{\sigma}/\vec{X}] \to \tau[\vec{\rho}/\vec{X}]$$

with $T(\vec{u}) = t[\vec{u}/\vec{f}]$. We say that t *realises* T.

Of course, it comes as no surprise that

Proposition 3.1 *Basic type expressions define basic functors.*

We first present the construction and then prove functoriality. For each type expression σ in context $\Theta \equiv \vec{X}$, we define by induction on the structure of types the associated functor $S_{\Theta\vdash\sigma} : \mathcal{C}^n \to \mathcal{C}$ (or simply S) as follows:

(1) Type variable
Let $\Theta \vdash X_i$ ($i \in \{1, \ldots, n\}$). For object $\vec{\sigma}$, define $S(\vec{\sigma}) = \sigma_i$ and for morphism $\vec{u} : \vec{\sigma} \to \vec{\rho}$, define $S(\vec{u}) = u_i$.
Let $\Theta \vdash \tau_1, \tau_2$ and suppose that T_j ($j = 1, 2$) is the basic functor associated with $\Theta \vdash \tau_j$ and whose morphism part is realised by t_j.

(2) Sum type
For object $\vec{\sigma}$, define $S(\vec{\sigma}) = T_1(\vec{\sigma}) + T_2(\vec{\sigma})$. For morphism $\vec{u} : \vec{\sigma} \to \vec{\rho}$, define $S(\vec{u})$ to be the term-in-context

$$\vec{f} : (\vec{\sigma} \to \vec{\rho}) \vdash \lambda z.\mathrm{case}(z) \text{ of } \begin{cases} \mathrm{inl}(x).\mathrm{inl}(t_1(\vec{f})(x)) \\ \mathrm{inr}(y).\mathrm{inr}(t_2(\vec{f})(y)) \end{cases}.$$

(3) Product type
For object $\vec{\sigma}$, define $S(\vec{\sigma}) = T_1(\vec{\sigma}) \times T_2(\vec{\sigma})$. For morphism $\vec{u} : \vec{\sigma} \to \vec{\rho}$, define $S(\vec{u})$ to be the unique morphism h such that $\pi_j \circ h = T_j(\vec{u}) \circ \pi_j$ ($j = 1, 2$), i.e., it is the term-in-context

$$\vec{f} : (\vec{\sigma} \to \vec{\rho}) \vdash \lambda z.(t_1(\vec{f})(\pi_1 z), t_2(\vec{f})(\pi_2 z)).$$

(4) Recursive type
Let $\Theta, X \vdash \tau$ and $X \notin \{X_1, \ldots, X_n\}$. For object $\vec{\sigma}$, define $S(\vec{\sigma}) = \mu X.T(\vec{\sigma}, X)$. We write $T(\vec{\sigma}, S(\vec{\sigma}))$ for $\tau[\vec{\sigma}/\vec{X}, S(\vec{\sigma})/X]$. For the morphism $\vec{u} : \vec{\sigma} \to \vec{\rho}$, define $S(\vec{u})$ to be the (contextually) least map h that makes the following diagram

$$\begin{tikzcd}
S(\vec{\sigma}) \arrow[r, "\mathrm{unfold}^{S(\vec{\sigma})}"] \arrow[d, "h"'] & T(\vec{\sigma}, S(\vec{\sigma})) \arrow[d, "{T(\vec{u}, h)}"] \\
S(\vec{\rho}) \arrow[r, "\mathrm{unfold}^{S(\vec{\rho})}"'] & T(\vec{\rho}, S(\vec{\rho}))
\end{tikzcd}$$

commute, i.e., it is the term-in-context $\vec{f} : (\vec{\sigma} \to \vec{\rho}) \vdash g : S(\vec{\sigma}) \to S(\vec{\rho})$ with g being the least solution to the recursive equation

$$g = \mathrm{fold}^{S(\vec{\rho})} \circ t(g, \vec{f}) \circ \mathrm{unfold}^{S(\vec{\sigma})}$$

where t is the basic term realising T.

(5) Lifted type
Let $\Theta \vdash \tau$ and T the associated functor whose morphism part is realised by the term t. For object $\vec{\sigma}$, define $S(\vec{\sigma}) = (T(\vec{\sigma}))_\perp$. For morphism $\vec{u} : \vec{\sigma} \to \vec{\rho}$, define $S(\vec{u})$ to be the term-in-context $\vec{f} : (\vec{\sigma} \to \vec{\rho}) \vdash \mathrm{up} \circ T(\vec{f}) \circ \mathrm{down}$ where t is the basic term realising T.

Functoriality relies on the following two key lemmas.

Lemma 3.2 (Plotkin's uniformity principle) *Let $f : \sigma \to \sigma$, $g : \tau \to \tau$ be FPC programs and $h : \sigma \to \tau$ a strict program, i.e., $h(\perp_\sigma) = \perp_\tau$ such that $h \circ f = g \circ h$. Then $\mathrm{fix}(g) = h(\mathrm{fix}(f))$.*

Proof. Using rational-chain completeness, rational continuity, $h \circ f = g \circ h$ in turn, it follows that

$$\begin{aligned} h(\mathrm{fix}(f)) &= h(\textstyle\bigsqcup_n f^{(n)}(\perp_\sigma)) \\ &= \textstyle\bigsqcup_n h \circ f^{(n)}(\perp_\sigma) \\ &= \textstyle\bigsqcup_n g^{(n)} \circ h(\perp_\sigma) \\ &= \textstyle\bigsqcup_n g^{(n)}(\perp_\tau) \\ &= \mathrm{fix}(g). \end{aligned}$$

□

Lemma 3.3 (Operational Minimal Invariance) *Let $T : \mathcal{C}^{n+1} \to \mathcal{C}$ be a basic functor and $\vec{\sigma}$ a given sequence of closed types. Write $S(\vec{\sigma})$ for $\mu X.T(\vec{\sigma}, X)$. Then the least endomorphism $e : S(\vec{\sigma}) \to S(\vec{\sigma})$ for which*

$$e = \mathrm{fold}^{S(\vec{\sigma})} \circ T(\vec{id}, e) \circ \mathrm{unfold}^{S(\vec{\sigma})}$$

must be $\mathrm{id}_{S(\vec{\sigma})}$.

Proof. The interpretation of e in the Scott model, denoted by $[\![e]\!]$, is the least endomorphism $f : [\![S(\vec{\sigma})]\!] \to [\![S(\vec{\sigma})]\!]$ such that the corresponding equation

$$f = [\![\mathrm{fold}^{S(\vec{\sigma})}]\!] \circ [\![T]\!]([\![\vec{\mathrm{id}}]\!], f) \circ [\![\mathrm{unfold}^{S(\vec{\sigma})}]\!]$$

holds. By classical domain theory, the least endomorphism $[\![e]\!]$ on $[\![S(\vec{\sigma})]\!]$ must be $\mathrm{id}_{[\![S(\vec{\sigma})]\!]}$. But Lemma 3.4 below asserts that $e(x) = x$ for all $x : S(\vec{\sigma})$. Thus by the Context Lemma ([20]), $h = \mathrm{id}_{S(\vec{\sigma})}$. □

Lemma 3.4 *Let $e : \tau \to \tau$ be a closed term such that $[\![e]\!] = \mathrm{id}_{[\![\tau]\!]}$ in the Scott-model. Then for all $x : \tau$, $e(x) = x$.*

Proof. For all $x : \tau$, it holds that $[\![e(x)]\!] = [\![e]\!]([\![x]\!]) = \mathrm{id}_{[\![\tau]\!]}([\![x]\!]) = [\![x]\!]$. The desired result then follows from the computational adequacy of the Scott-model which in turn can be proven by adapting the proof of Theorem 7.14 of [10] to our call-by-name setting. □

Remark 3.5 Although a semantic trick has been employed in the above proof, the result is purely operational. It is interesting to note that there is indeed a purely operational proof of Lemma 3.3.in [5] by L. Birkedal and R. Harper for a simply-typed fragment of ML[4] with one top-level recursive type. As our principal objective is to understand the connections between operational algebraic compactness and the underlying domain structure of the contextual order, we choose not to present a modification of their operational proof here. Notice that the operational proof by L. Birkedal and R. Harper cannot be easily replaced by a direct proof by induction on types. That would deem to fail because the least fixed point of a type expression is not in any way built from those of its constituents.

Remark 3.6 Theorem 5.2 of [25] seems to suggest that the Plotkin's uniformity can be used to prove that the morphism $\mathrm{fold} : \sigma[\mu X.\sigma/X] \to \mu X.\sigma$ is a special invariant (which is equivalent to our operational minimal invariance property). However, a closer inspection reveals that this implication holds provided $\mathrm{fold} : \sigma[\mu X.\sigma/X] \to \mu X.\sigma$ is an initial algebra. But this, in our case, cannot be established without invoking the operational minimal invariance property itself. Thus we do not know for sure if the uniformity principle alone implies Lemma 3.3.

We are now ready to prove functoriality.

Proof. First we prove the preservation of composition of morphisms. Consider the following composition of morphisms:

$$(\vec{u} : \vec{\sigma} \to \vec{\rho}) \circ (\vec{v} : \vec{\rho} \to \vec{\tau})$$

Avoiding routine details, we just verify the case for constructors of the form $\mu X.T(\vec{X}, X)$, i.e., $S(\vec{v} \circ \vec{u}) = S(\vec{v}) \circ S(\vec{u})$ where $\mu X.T(\vec{\sigma}, X)$ is abbreviated by $S(\vec{\sigma})$. For that purpose, let $\Phi = \lambda h.\mathrm{fold}^{S(\vec{\tau})} \circ T(\vec{v}, h) \circ \mathrm{unfold}^{S(\vec{\rho})}$ and $\Psi = \lambda f.\mathrm{fold}^{S(\vec{\tau})} \circ T(\vec{v} \circ \vec{u}, f) \circ \mathrm{unfold}^{S(\vec{\sigma})}$. Because for any $h : S(\vec{\rho}) \to S(\vec{\tau})$,

$$\begin{aligned}\Psi(h \circ S(\vec{v})) &= \mathrm{fold}^{S(\vec{\tau})} \circ T(\vec{v} \circ \vec{u}, h \circ S(\vec{u})) \circ \mathrm{unfold}^{S(\vec{\sigma})} \\ &= \mathrm{fold}^{S(\vec{\tau})} \circ T(\vec{v}, h) \circ \mathrm{unfold}^{S(\vec{\rho})} \circ \mathrm{fold}^{S(\vec{\rho})} \circ T(\vec{u}, S(\vec{u})) \circ \mathrm{unfold}^{S(\vec{\sigma})} \\ &= \Phi(h) \circ S(\vec{u})\end{aligned}$$

the following diagram commutes:

$$\begin{tikzcd}
(S(\vec{\rho}) \to S(\vec{\tau})) \arrow[r, "- \circ S(\vec{u})"] \arrow[d, "\Phi"'] & (S(\vec{\sigma}) \to S(\vec{\tau})) \arrow[d, "\Psi"] \\
(S(\vec{\rho}) \to S(\vec{\tau})) \arrow[r, "- \circ S(\vec{u})"'] & (S(\vec{\sigma}) \to S(\vec{\tau}))
\end{tikzcd}$$

Since $- \circ S(\vec{u})$ is a strict program, by Lemma 3.2 it follows that

$$S(\vec{v} \circ \vec{u}) = \mathrm{fix}(\Psi) = \mathrm{fix}(\Phi) \circ S(\vec{u}) = S(\vec{v}) \circ S(\vec{u}).$$

[4] This language is almost the same as ours except that it has a call-by-value evaluation strategy and an explicit fix operator.

Finally we establish the preservation of identity morphisms. By definition $S(\mathrm{id}_{\vec{\sigma}})$ is the least solution e of the equation $e = \mathrm{fold}^{S(\vec{\sigma})} \circ T(\mathrm{id}_{\vec{\sigma}}, e) \circ \mathrm{unfold}^{S(\vec{\sigma})}$. But Lemma 3.3 already asserts that $e = \mathrm{id}_{S(\vec{\sigma})}$. Hence $S(\mathrm{id}_{\vec{\sigma}}) = \mathrm{id}_{S(\vec{\sigma})}$, which concludes the proof of functoriality. $\square$

An unrestricted FPC type expression is more problematic.

(i) Once the function-type constructor $\rightarrow$ is involved, one needs to separate the covariant and the contravariant variables (e.g. X is covariant and Y is contravariant in $X \rightarrow Y$).

(ii) A particular type variable may be covariant and contravariant (e.g. X in $X \rightarrow X$).

The usual solution to this problem, following the work of Freyd [9] is to work with the category $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$. However, as discussed in Section 4, this is not possible for FPC-definable functors. We work instead with a full subcategory $\mathcal{C}^*$ of this. Define $\mathcal{C}^*$, the diagonal category, to be the full subcategory of $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ whose objects are those of $\mathcal{C}$ and morphisms being pairs of $\mathcal{C}$-morphisms $\langle v, u\rangle$ of the form: $u : \sigma \leftrightarrows \tau : v$. In this category, composition of morphisms $\langle t, s\rangle \circ \langle v, u\rangle$ is defined as the pair of compositions $\langle v \circ t, s \circ u\rangle$.

A functor $T : (\mathcal{C}^*)^n \rightarrow \mathcal{C}^*$ is said to be *realisable* if

(1) there is an FPC type expression τ in context $\vec{X}$ such that for each sequence $\vec{\sigma}$ of closed terms, $T(\vec{\sigma}) = \tau[\vec{\sigma}/\vec{X}]$, and

(2) there is a sequence of term variables $\vec{f}$ and a term t with $fv(t) \subseteq \{f_1, \ldots, f_n\}$ such that for all morphism pairs $\vec{u} : \vec{\sigma} \rightleftarrows \vec{\rho} : \vec{v}$ (i.e., the morphism pairs $\sigma_i \underset{v_i}{\overset{u_i}{\rightleftarrows}} \rho_i$ for $i = 1, \ldots, n$) the terms-in-context

$$f_i : (\sigma_i \rightarrow \rho_i), g_i : (\rho_i \rightarrow \sigma_i)|_{i=1}^{n} \vdash t : \tau[\vec{\sigma}/\vec{X}] \rightleftarrows \tau[\vec{\rho}/\vec{X}] : s$$

are such that $T(v_1, u_i, \ldots, v_n, u_n) = \langle s, t\rangle[\vec{u}/\vec{f}, \vec{v}/\vec{g}]$.

Notation. We write $\overrightarrow{v;u}$ for $v_1, u_1, \ldots, v_n, u_n$.

A type expression is *functional* if it is of the form $\tau_1 \rightarrow \tau_2$ for some types-in-context $\Theta \vdash \tau_1, \tau_2$.

Theorem 3.7 *All FPC type expressions define realisable functors.*

Proof. Again we proceed by induction on the structure of types.

(1) Functional type expressions. Suppose $\vec{X} \vdash \tau_1 \rightarrow \tau_2$ for some types-in-context $\vec{X} \vdash \tau_1, \tau_2$. Let $T_j : (\mathcal{C}^*)^n \rightarrow \mathcal{C}^*$ $(j = 1, 2)$ be functors that realise $\vec{X} \vdash \tau_j$. Define the action of S on
   (i) Objects: Given $\vec{\sigma}$, define $S(\vec{\sigma}) = T_1(\vec{\sigma}) \rightarrow T_2(\vec{\sigma})$.
   (ii) Morphisms: Let $\vec{u} : \vec{\sigma} \rightleftarrows \vec{\rho} : \vec{v}$ be given. Since T_j's are realisable, there are term variables f_i, g_i and terms t_j, s_j with $fv(t_j), fv(s_j) \subseteq \{f_1, \ldots, f_n, g_1, \ldots, g_n\}$ such that for all morphism pairs $\vec{u} : \vec{\sigma} \rightleftarrows \vec{\rho} : \vec{v}$, the term-in-context

$$f_i : (\sigma_i \to \rho_i), g_i : (\rho_i \to \sigma_i)|_{i=1}^{n} \vdash t_j : \tau_j[\vec{\sigma}/\vec{X}] \rightleftarrows \tau_j[\vec{\rho}/\vec{X}] : s_j$$

such that $T_j(\overrightarrow{v;u}) = \langle s_j, t_j \rangle[\vec{u}/\vec{f}, \vec{v}/\vec{g}]$ (for $j = 1, 2$). We can thus define the morphism part of S by $u : S(\vec{\sigma}) \leftrightarrows S(\vec{\rho}) : v$ where

$$\vec{u} = \lambda h : S(\vec{\sigma}).\lambda g : S(\vec{\rho}).t_2(\overrightarrow{v;u}) \circ h \circ s_1(\overrightarrow{v;u})$$
$$\vec{v} = \lambda g : S(\vec{\rho}).\lambda h : S(\vec{\sigma}).s_2(\overrightarrow{v;u}) \circ g \circ t_1(\overrightarrow{v;u})$$

Notice that S is realisable because T_j's are.

(2) Non-functional types expressions. We define these functors in a way similar to those of the basic case, except that these are upgraded to functors typed $(\mathcal{C}^*)^n \to C^*$ by adding the dual arrow when defining its morphism part.

Functoriality of FPC type expressions can be established in a manner similar to that of basic type expressions (c.f. Proposition 3.1). □

4 Algebraic compactness operationally

In [8], P.J. Freyd introduced the notion of algebraic compactness to capture the bifree nature of the canonical solution to the domain equation $X = FX$ in that "every endofunctor (on cpo-enriched categories, for example, $\mathcal{D}_\perp$, the category of pointed cpos and strict maps [5]) has an initial algebra and a final co-algebra and they are canonically isomorphic". Freyd's *Product Theorem* asserts that the algebraically compactness is closed under finite products. Crucially this implies that $\mathcal{D}_\perp^{\mathrm{op}} \times \mathcal{D}_\perp$ is algebraically compact (since its components are) and thus allows one to cope well with the mixed-variant functors - making the study of domain equations complete. Now proving that $\mathcal{D}_\perp$ is algebraically compact is no easy feat as one inevitably has to switch to the category of embeddings and projections, together with a bilimit construction. As opposed to the classical case, operational algebraic compactness (with respect to the class of realisable functors) is relatively simple.

The first step is to restrict our attention to a subcategory of $\mathcal{C}^*$, namely $\mathcal{A}$, whose objects are those of $\mathcal{C}^*$ but whose morphisms are strict morphism pairs. Despite this restriction, there is no need to contract on our class of realisable functors, owing to the following lemma.

Lemma 4.1 (Freyd, [7]) *If $T : (\mathcal{C}^*)^{n+1} \to \mathcal{C}^*$ is a locally monotone functor, then it preserves strict morphism-pairs. In particular, so does every functor that realises a type expression.*

Proof. Essentially the same as Lemma 1 of [7]. □

Notation. We define $\chi = \mathcal{A}^n$ where n is a fixed natural number.

[5] If non-strict maps are considered then the identity functor does not have an initial algebra.

We are now ready for one of the main theorems in this paper.

Theorem 4.2 (Operational parameterised algebraic compactness)
Let $X, X_1, \ldots, X_n \vdash \tau$ be a type-in-context and $T : \chi \times \mathcal{A} \to \mathcal{A}$ its realising functor. Then there exists a realisable functor $S : \chi \to \mathcal{A}$ together with a natural isomorphism $i : T(-, S(-)) \to S$ such that for any parameter P of χ, the $\mathcal{A}$-morphism $i_P : T(P, S(P)) \rightleftarrows S(P) : i_P^{-1}$ satisfies the following universal property: For any realisable functor $S' : \chi \to \mathcal{A}$ and two natural transformations $k_P : T(P, S'(P)) \rightleftarrows S'(P) : j_P$, there is a unique pair of natural transformations α and β such that the following diagrams commute:

$$\begin{tikzcd}
T(P,S(P)) & S(P) \arrow[l, "i_P^{-1}"'] \\
T(P,S'(P)) \arrow[u, "T(\mathrm{id}_P,\alpha_P)"] & S'(P) \arrow[l, "j_P"] \arrow[u, "\alpha_P"']
\end{tikzcd}
\qquad
\begin{tikzcd}
T(P,S(P)) \arrow[r, "i_P"] \arrow[d, "T(\mathrm{id}_P,\beta_P)"'] & S(P) \arrow[d, "\beta_P"] \\
T(P,S'(P)) \arrow[r, "k_P"'] & S'(P)
\end{tikzcd}$$

We say that $\mathcal{A}$ is (operationally) parameterised algebraically compact (with respect to the class of realisable functors). In the special case when $\chi = \mathcal{A}^0$, we drop the word "parameterised".

Proof. Given P in χ, let $S(P) = \mu X.T(P, X)$ and $i_P = \mathrm{fold}^{S(P)}$. Recall that (i) $i_P^{-1} = \mathrm{unfold}^{S(P)}$ and (ii) S is functorial by virtue of Theorem 3.7. It remains to establish the universal property. For that purpose, define α_P and β_P to be the least solutions of the recursive equations:

$$\alpha_P = i_P \circ T(\mathrm{id}_P, \alpha_P) \circ j_P$$

$$\beta_P = k_P \circ T(\mathrm{id}_P, \beta_P) \circ i_P^{-1}.$$

Of course, α_P and β_P make the required diagrams commute. That there are no others is shown by an application of Lemma 3.2 to the following diagram:

$$\begin{tikzcd}
(S(P) \to S(P)) \times (S(P) \to S(P)) \arrow[r, "G"] \arrow[d, "\Phi"'] & (S'(P) \to S(P)) \times (S(P) \to S'(P)) \arrow[d, "\Psi"] \\
(S(P) \to S(P)) \times (S(P) \to S(P)) \arrow[r, "G"'] & (S'(P) \to S(P)) \times (S(P) \to S'(P))
\end{tikzcd}$$

where $\Phi = \lambda(g, h).(i_p \circ T(\mathrm{id}_P, g) \circ i_P^{-1}, i_P \circ T(\mathrm{id}_P, h) \circ i_P^{-1})$
$\Psi = \lambda(a, b).(i_P \circ T(\mathrm{id}_P, a) \circ j_P, k_P \circ T(\mathrm{id}_P, b) \circ i_P^{-1})$
and $G = \lambda(g, h).(g \circ \alpha'_P, \beta'_P \circ h)$ with α'_P and β'_P being "potential alternatives" to α_P and β_P. □

We close this section with a justification of our use of $\mathcal{A}$ rather than $\mathcal{C}_\perp^{\mathrm{op}} \times \mathcal{C}_\perp$. Classical theory of recursive domain equations centres around functors of the form $F : (\mathcal{D}_\perp^{\mathrm{op}} \times \mathcal{D}_\perp)^{n+1} \to (\mathcal{D}_\perp^{\mathrm{op}} \times \mathcal{D}_\perp)$ where $\mathcal{D}_\perp$ is the category of pointed cpos and strict morphisms. As noted before, $\mathcal{D}_\perp^{\mathrm{op}} \times \mathcal{D}_\perp$ is algebraically compact. But even more generally $\mathcal{D}_\perp^{\mathrm{op}} \times \mathcal{D}_\perp$ is parameterised algebraically compact - a result proven in [10].

Let $\mathcal{C}^\S$ denote the product category $\mathcal{C}_\perp^{\mathrm{op}} \times \mathcal{C}_\perp$. The question is whether we can define the type expressions as functors (belonging to a certain class $\mathcal{F}$) of the form $T : (\mathcal{C}^\S)^n \to (\mathcal{C}^\S)$. Assume for the moment that we can do so. Let $\vec{X}^\mp$ denote $X_1^-, X_1^+, \ldots, X_n^-, X_n^+$ and consider a type-in-context $\vec{X}^\mp, X \vdash \tau$. Then by assumption there is an $\mathcal{F}$-functor $T : (\mathcal{C}^\S)^{n+1} \to (\mathcal{C}^\S)$ that realises σ. We expect that there exist an $\mathcal{F}$-functor $H := (H^-, H^+) : (\mathcal{C}^\S)^n \to (\mathcal{C}^\S)$ and a natural isomorphism i such that for every sequence of closed types $\vec{\sigma}^\mp := \sigma_1^-, \sigma_1^+, \ldots, \sigma_n^-, \sigma_n^+$, the pair $(H(\vec{\sigma}^\mp), i_{\vec{\sigma}^\mp})$ is a bifree algreba of the endofunctor $T(\vec{\sigma}^\mp, -, +) : C^\S \to \mathcal{C}^\S$. Moreover, we anticipate that

$$H^+(\vec{\sigma}^\mp) := \mu X.\tau[\vec{\sigma}^\mp/\vec{X}^\mp] \tag{†}$$

and in this way $\mu X.\tau$ is realised by H.

However, it turns out that our anticipation (†) is wrong as we shall explain in the following stages:

1. Define the class of functors $\mathcal{F}$ based on the category $\mathcal{C}^\S$.
2. Modify Freyd's argument used in the Product Theorem (see Section 4 of [9]) to prove that $\mathcal{C}^\S$ is parametrised algebraically compact with respect to the class of $\mathcal{F}$-functors.
3. Conclude that our anticipation (†) is wrong.

Stage 1

The class $\mathcal{F}$ of functors which we define below is called the class of *syntactic* functors (originally used by A. Rohr in his Ph.D. thesis [24]). A functor $T : (\mathcal{C}^\S)^n \to \mathcal{C}_\perp$ is *syntactic* if

(i) there are types in context $\vec{X}^\mp := X_1^-, X_1^+, \ldots, X_n^-, X_n^+ \vdash \tau$ such that for every sequence of closed types $\vec{\sigma}^\mp := \sigma_1^-, \sigma_1^+, \ldots, \sigma_n^-, \sigma_n^+$, it holds that $T(\vec{\sigma}^\mp) = \tau[\vec{\sigma}^\mp/\vec{X}^\mp]$.

(ii) there are term variables $\vec{f}^\mp$ and a term t with $fv(t) \subseteq \{f_1^-, f_1^+, \ldots, f_n^-, f_n^+\}$ such that for all morphisms $\vec{u}^+ : \vec{\sigma}^+ \to \vec{\rho}^+$ (i.e., $u_i^+ : \sigma_i^+ \to \rho_i^+$ for $i = 1, \ldots, n$) and $\vec{u}^- : \vec{\rho}^- \to \vec{\sigma}^-$ (i.e., $u_i^- : \rho_i^- \to \sigma_i^-$ for $i = 1, \ldots, n$), we have a term-in-context
$$f_i^- : (\rho_i^- \to \sigma_i^-), f_i^+ : (\sigma_i^+ \to \rho_i^+) \vdash t : \tau[\vec{\sigma}^\mp] \to \tau[\vec{\rho}^\mp]$$
with $T(\vec{u}^\mp) = t[\vec{u}^\mp/\vec{f}^\mp]$. As usual, we say that t realises T.

We can generalise this a little further by defining a functor of the form $T : (\mathcal{C}^\S)^n \to \mathcal{C}^\S$ to be syntactic if its projections onto $\mathcal{C}_\perp$ are syntactic.

Stage 2
For the sake of clarity, we restrict to $n = 2$.

Proposition 4.3 *Let $F : (\mathcal{C}^\S)^2 \to \mathcal{C}_\perp$ be a syntactic functor. Upgrade F to $F^\S : (\mathcal{C}^\S)^2 \to (\mathcal{C}^\S)$ by defining*

$$F^\S(X^-, X^+, Y^-, Y^+) := (F(X^+, X^-, Y^+, Y^-), F(X^-, X^+, Y^-, Y^+)).$$

Then there exists a functor $H : \mathcal{C}^\S \to \mathcal{C}^\S$ and a pair of natural isomorphisms $i = (i^-, i^+)$ such that for all pairs of closed types $P = (P^-, P^+)$ in $\mathcal{C}^\S$ we have

$$i^-_P : H^-(P) \cong F(P^+, P^-, H^+(P), H^-(P)) \text{ and}$$
$$i^+_P : F(P, H^-(P), H^+(P)) \cong H^+(P)$$

where $H(P) := (H^-(P), H^+(P))$. Moreover, $(H(P), i_P)$ is a bifree algebra for the endofunctor $F^\S(P, -) : \mathcal{C}^\S \to \mathcal{C}^\S$. In other words, the category $\mathcal{C}^\S$ is (operationally) parametrised algebraically compact.

Proof. (Sketch) Let $P = (P^-, P^+)$ be given. Then $F^\S(P, -)$ defines a syntactic endofunctor on $\mathcal{C}^\S$. To such a P, we assign the pair of closed types $H(P)$ in the following way.

(1) Resolve $F^\S(P, -)$ into its components:

$$T' : \mathcal{C}^\S \to \mathcal{C}^{\mathrm{op}}_\perp \text{ and } T : \mathcal{C}^\S \to \mathcal{C}_\perp$$

i.e., $T'(A^\mp) := F(P^+, P^-, A^+, A^-)$ and $T(A^\mp) := F(P, A^-, A^+)$.

(2) Given a closed type A, $T'(-, A)$ defines an endofunctor on $\mathcal{C}^{\mathrm{op}}_\perp$. This induces a functor $F' : \mathcal{C}_\perp \to \mathcal{C}^{\mathrm{op}}_\perp$ defined by $F'(A) := \mu X.T'(X, A)$. Let $u : A \to B$ in $\mathcal{C}_\perp$. The morphism $F'(u) : F'(B) \to F'(A)$ is the least map which fits into the following commutative diagram:

$$\begin{tikzcd}
F(P^+, P^-, A, F'(A)) & & F(P^+, P^-, B, F'(B)) \arrow[ll, "F(\mathrm{id}_{P^+}, \mathrm{id}_{P^-}, u, F'(u))"'] \\
F'(A) \arrow[u, "\mathrm{unfold}^{F'(A)}"] & & F'(B) \arrow[ll, "F'(u)"] \arrow[u, "\mathrm{unfold}^{F'(B)}"']
\end{tikzcd}$$

Again the functoriality of F' can be proven using methods similar to those outlined in Proposition 3.1.

(3) Define the endofunctor G on $\mathcal{C}_\perp$ by $G(A) := T(F'(A), A)$. Notice that $G(A) = F(P, F'(A), A)$.

(4) Define $H^+(P) := \mu Y.G(Y)$ and $H^-(P) := F'(H^+(P))$, i.e.,

$$H^+(P) = \mu Y.F(P, \mu X.F(P^+, P^-, Y, X), Y) \text{ and}$$
$$H^-(P) = \mu X.T'(X, H^+(P)).$$

This defines the object part of H. Then define the natural isomorphisms i^-_P and i^+_P via the obvious folding and unfolding so that

$$i^-_P : H^-(P) \cong F(P^+, P^-, H^+(P), H^-(P))$$
$$i^+_P : F(P^-, P^+, H^-(P), H^+(P)) \cong H^+(P).$$

(5) The morphism part of H is defined as follows. Given $f : P \to Q$ in $\mathcal{C}^\S$, we

define $H(f) : H(P) \to H(Q)$ as the least morphism $g : H(P) \to H(Q)$ that makes the diagram below commute.

$$\begin{tikzcd}
F^{\S}(P, H(P)) \arrow[r, "i_P"] \arrow[d, "F^{\S}(f,g)"'] & H(P) \arrow[d, "g"] \\
F^{\S}(Q, H(Q)) \arrow[r, "i_Q"'] & H(Q)
\end{tikzcd}$$

(6) Then one routinely verifies that $H(P) := (H^-(P), H^+(P))$ together with the pair of isomorphisms $i_P := (i_P^-, i_P^+)$ is indeed a bifree algebra for the endofunctor $F^{\S}(P, -)$.

□

Stage 3
From (4), we already know that $H^+(P)$ is not given by $\mu X.F(P^-, P^+, X, X)$. Thus we have demonstrated that if we use the category $\mathcal{C}^{\S}$ to develop our theory, the recursive type constructor cannot be defined as a functor H which satisfies the bifree condition stated in Proposition 4.3. Note that our justification echoes with a remark[6] made by M. Abadi and M. Fiore in [1]:

"Interestingly, the syntax seems to steer us away from the approach based on $\mathcal{C}^{\S}$, and towards that based on $\mathcal{A}$."

5 SFP-structure and the generic approximation lemma

In this section, we establish that every FPC closed type is rationally SFP. This is similar to the one proven in [6] regarding PCF types. The most vital consequence of this is a rigorous proof of the generic approximation lemma[7] first proposed by G. Hutton and J. Gibbons in [16] - in which the lemma was proven via denotational semantics for *polynomial* types (i.e., types built only from unit, sums and products) and claimed to "generalise to mutually recursive, parameterised, exponential and nested datatypes" (page 4 of [16]). The fresh insight here is that this useful lemma is a direct result of the underlying operational domain structure of types.

A *deflation* on a type σ is an element of type $(\sigma \to \sigma)$ that (i) is below the identity and (ii) has a finite image modulo contextual equivalence. A *rational SFP structure* on a type σ is a rational chain id_n^σ of idempotent deflations with $\bigsqcup_n \mathrm{id}_n^\sigma = \mathrm{id}_\sigma$. We say that a type is *rationally SFP* if it has an SFP structure.

[6] In [1], the category $\mathcal{C}^{\S}$ is denoted by $\breve{\mathcal{C}}$ and its diagonal subcategory $\mathcal{A}$ is denoted by $\mathcal{C}^{\mathcal{D}}$.
[7] This is a generalisation of R. Bird's approximation lemma [3], which in turn generalises the well-known take lemma [4].

Here we need to introduce an auxiliary type - the vertical natural numbers $\overline{\omega}$, defined by $\overline{\omega} = \mu X.X_\perp$. For each $x : \overline{\omega}$, its successor is defined by $\mathrm{succ}(x) = \mathrm{fold} \circ \mathrm{up}(x)$. We define $0 := \perp$, $n := \mathrm{succ}^n(0)$, $n - 1 := \mathrm{fold} \circ \mathrm{down} \circ \mathrm{unfold}(n)$ and $\infty := \mathrm{fix}(\lambda x : \overline{\omega}.\mathrm{succ}(x))$. Note: $\infty = \infty - 1$. Given $x : \overline{\omega}$, the Σ-valued convergence test $(x > 0) := \mathrm{case}(\mathrm{unfold}(x))$ of $\mathrm{up}(y).\top$ returns $\top$ iff $x = \mathrm{fold}(\mathrm{up}(y))$ for some $y : \overline{\omega}$.

Using $\overline{\omega}$, define $d^\sigma : \overline{\omega} \to (\sigma \to \sigma)$ by induction on σ as follows:

$$\begin{aligned}
d^{\sigma\times\tau}(n)(x,y) &= (d^\sigma(n)(x), d^\tau(n)(y))\\
d^{\sigma+\tau}(n)\mathrm{inl}(x) &= \mathrm{inl}(d^\sigma(n)(x))\\
d^{\sigma+\tau}(n)\mathrm{inr}(y) &= \mathrm{inr}(d^\tau(n)(y))\\
d^{\sigma_\perp}(n)(x) &= \mathrm{up} \circ d^\sigma(n) \circ \mathrm{down}(x)\\
d^{\sigma\to\tau}(n)(f) &= d^\tau(n) \circ f \circ d^\sigma(n)
\end{aligned}$$

and, most crucially, for the recursive type $\mu X.\sigma$, the program $d^{\mu X.\sigma}(n)$ is defined as follows. Recall that $\mathcal{A}$ is the subcategory of the diagonal category $\mathcal{C}^*$ whose morphisms are strict $\mathcal{C}^*$-morphisms. Let $S : \mathcal{A} \to \mathcal{A}$ be the realisable functor associated to the type-in-context $X \vdash \sigma$. Write $S(d^{\mu X.\sigma}(n), d^{\mu X.\sigma}(n)) = (e^{\mu X.\sigma}(n), e^{\mu X.\sigma}(n))$. Define

$$d^{\mu X.\sigma}(n)(x) := \text{if } (n > 0) \text{ then } \mathrm{fold} \circ e^{\mu X.\sigma}(n-1) \circ \mathrm{unfold}(x)$$

where if a then b is syntactic sugar for $\mathrm{case}(a)$ of $\mathrm{up}(x).b$. Then it is obvious that $d^{\mu X.\sigma}$ satisfies the following equations:

$$d^{\mu X.\sigma}(0) = \perp_{\mu X.\sigma \to \mu X.\sigma} \text{ and } d^{\mu X.\sigma}(n+1) = \mathrm{fold} \circ e^{\mu X.\sigma}(n) \circ \mathrm{unfold}.$$

In what follows, we abuse notations by writing $S(f)$ to denote one of the morphisms in the pair $S(f, f)$ where $S : \mathcal{A} \to \mathcal{A}$ is a realisable functor.

Lemma 5.1 *Let $S : \mathcal{A} \to \mathcal{A}$ be a realisable functor associated to the type-in-context $X \vdash \sigma$. Suppose $f : \tau \to \tau$ has a finite image modulo contextual equivalence. Then so does $S(f)$.*

Proof. The proof proceeds by induction on σ and only the most interesting part regarding recursive types is presented here. Suppose that $\sigma = \mu X.T(X, Y)$ for some realisable functor $T : \mathcal{A}^2 \to \mathcal{A}$. Recall that $S(f)$ makes the following diagram commute:

$$\begin{tikzcd}
S(\tau) \arrow[r, "S(f)"] \arrow[d, "\mathrm{unfold}"'] & S(\tau) \arrow[d, "\mathrm{unfold}"] \\
T(\tau, S(\tau)) \arrow[r, "T(f, S(f))"'] & T(\tau, S(\tau))
\end{tikzcd}$$

Let T' be the realisable functor $T(-, S(\tau)) : \mathcal{A} \to \mathcal{A}$. Then $T'(f) = T(f, \mathrm{id}_{S(\tau)})$ and thus by induction hypothesis has a finite image modulo contextual equivalence. Since $S(f) = \mathrm{fold} \circ T(\mathrm{id}_\tau, S(f)) \circ T(f, \mathrm{id}_{S(\tau)}) \circ \mathrm{unfold}$, it follows that $S(f)$ also has a finite image modulo contextual equivalence. □

Theorem 5.2 *The rational-chain* $\mathrm{id}_n^\sigma := d^\sigma(n)$ *is an SFP-structure on* σ *for every closed type* σ*, i.e., the following hold:*
(1) $\mathrm{id}_n^\sigma \circ \mathrm{id}_n^\sigma = \mathrm{id}_n^\sigma$.
(2) $\mathrm{id}_n^\sigma \sqsubseteq \mathrm{id}_\sigma$ *and* $\bigsqcup_n \mathrm{id}_n^\sigma = \mathrm{id}_\infty^\sigma = \mathrm{id}_\sigma$.
(3) id_n^σ *has finite image modulo contextual equivalence.*

Proof. By induction on σ as usual. Since (1) is obvious and (3) a direct result of Lemma 5.1, we shall only prove (2) by a further induction on n. Let S be the functor realising $X \vdash \sigma$. The base case is trivial and the inductive step is justified by the monotonicity of S in that:

$$\begin{aligned} \mathrm{id}_n^{\mu X.\sigma} &= \mathrm{fold} \circ S(\mathrm{id}_{n-1}^{\mu X.\sigma}) \circ \mathrm{unfold} \\ &\sqsubseteq \mathrm{fold} \circ S(\mathrm{id}_{\mu X.\sigma}) \circ \mathrm{unfold} \quad \text{(Induction hypothesis)} \\ &= \mathrm{id}_{\mu X.\sigma} \qquad\qquad (S \text{ is a functor, } \mathrm{unfold} = \mathrm{fold}^{-1}) \end{aligned}$$

Because $\infty = \infty - 1$, the morphism $k := \mathrm{id}_\infty^{\mu X.\sigma}$ satisfies the recursive equation $k = \mathrm{fold} \circ S(k) \circ \mathrm{unfold}$. By Lemma 3.3, $\mathrm{id}_{\mu X.\sigma}$ is the least solution and thus must be below $\mathrm{id}_\infty^{\mu X.\sigma}$. On the other hand, $\mathrm{id}_\infty^{\mu X.\sigma} = \bigsqcup_n \mathrm{id}_n^{\mu X.\sigma}$ so that $\mathrm{id}_\infty^{\mu X.\sigma} \sqsubseteq \mathrm{id}_{\mu X.\sigma}$. Hence $\mathrm{id}_\infty^{\mu X.\sigma} = \mathrm{id}_{\mu X.\sigma}$. □

Notation. We write $x =_n y$ for $\mathrm{id}_n(x) = \mathrm{id}_n(y)$.

Corollary 5.3 (The generic approximation lemma [16])
Let σ *be a closed type and* $x, y : \sigma$*. Then* $x = y$ *iff for all* $n \in \mathbb{N}$*,* $x =_n y$*.*

Proof. Immediate from Theorem 5.2. □

6 Sample applications

In this section, we demonstrate the versatility of the "generic approximation lemma" by using running examples of programs taken from [22]. For each example, we compare the use of Corollary 5.3 with an alternative method in proving program properties. Let τ be a closed type and recall that $1 := \mu X.X$ is the type which consists of $\bot$ alone. The type $[\tau] := \mu\alpha.1 + \tau \times \alpha$ is called the (lazy) *list type* associated to τ. An element of $[\tau]$ may be thought of as a (finite or infinite) list of elements in τ. Note that the elements in a list may include $\bot_\tau$. In the course of our discussion, we make use of the following:
(1) $[\,] := \mathrm{fold}(\mathrm{inl}(*))$
(2) $\mathrm{cons} : \tau \to [\tau] \to [\tau]$
$\mathrm{cons}\ x\ xs = \mathrm{fold}(\mathrm{inr}(x, xs)) \equiv (x : xs)$.
(3) Let σ be a closed type.
A program $f : [\tau] \to \sigma$ defined by cases, i.e.,

$$f(l) = \mathrm{case}(l)\ \text{of} \begin{cases} \mathrm{inl}(x).s_1 \\ \mathrm{inr}(y).s_2 \end{cases} \text{is written in } \texttt{Haskell} \text{ style:}$$

$$\begin{aligned} f\ [\,] \quad &= s_1 \\ f\ (x : xs) &= s_2 \end{aligned}$$

Case definitions producing divergence are omitted, e.g. $\mathrm{hd} : [\tau] \to \tau$ and tl : $[\tau] \to [\tau]$ are defined by: hd $(x : xs) = x$ and tl $(x : xs) = xs$

(4) For programs $f : (\tau \to \tau) \to (\tau \to \tau)$, we write $h := \mathrm{fix}(f)$ as:

$$h : \tau \to \tau$$
$$h = f\ h$$

Since all the examples covered here only involve the basic type constructors, one could have, for their sake, developed the machineries necessary for handling basic type expressions.

The following proves handy whenever Corollary 5.3 is applied to lists.

Lemma 6.1 *Let $n > 0$ be a natural number and τ be a closed type. Then the program* $\mathrm{id}_n : [\tau] \to [\tau]$ *satisfies the following equations:*

$$\begin{aligned}\mathrm{id}_n[\ ] &= [\ ]\\ \mathrm{id}_n(x : xs) &= (x : \mathrm{id}_{n-1}(xs))\end{aligned}$$

Proof. Straightforward and thus omitted. □

We define two familiar functions map and iterate.

$$\begin{aligned}&\mathrm{map} : (\tau \to \tau) \to [\tau] \to [\tau] && \mathrm{iterate} : (\tau \to \tau) \to \tau \to [\tau]\\ &\mathrm{map}\ f\ [\ ] = [\ ] && \mathrm{iterate} f\ x = (x : \mathrm{iterate} f\ f(x))\\ &\mathrm{map}\ (x : xs) = (f(x) : \mathrm{map}\ f\ xs) &&\end{aligned}$$

Proposition 6.2 (The map-iterate property)
Let τ be a closed type. For any $f : (\tau \to \tau)$ and any $x : \tau$, it holds that

$$\mathrm{map} f\ (\mathrm{iterate} f\ x) = \mathrm{iterate} f\ f(x).$$

Program fusion has been used in [11] to prove the above. Define:

$$\mathrm{unfd} : (\sigma \to \mathrm{Bool}) \to (\sigma \to \tau) \to (\sigma \to \sigma) \to \sigma \to [\tau]$$
$$\mathrm{unfd}\ p\ h\ t\ x = \mathrm{if}\ p(x)\ \mathrm{then}\ [\ ]\ \mathrm{else}\ (h(x) : \mathrm{unfd}\ p\ h\ t\ tx)$$

The unfd function "encapsulates the natural basic pattern of co-recursive definition" (page 9 of [11]) and so several familiar co-recursive functions on lists can be defined in terms of unfd. For instance, if we have

$$\begin{aligned}&\mathrm{null} : [\tau] \to \mathrm{Bool} && \mathrm{false} : \sigma \to \mathrm{Bool}\\ &\mathrm{null}\ [\ ] = \mathrm{true}(:= \mathrm{inr}(\bot)) && \mathrm{false}\ x = \mathrm{false}\ (:= \mathrm{inl}(\bot))\\ &\mathrm{null}\ (x : xs) = \mathrm{false}(:= \mathrm{inl}(\bot)) &&\end{aligned}$$

then define map $f = \mathrm{unfd}\ \mathrm{null}\ (f \circ \mathrm{hd})\ \mathrm{tl}$ and iterate $f = \mathrm{unfd}\ \mathrm{false}\ \mathrm{id}\ f$. The proof method here relies on a universal property (described below) enjoyed by unfd. Define $q : \sigma \to 1 + \tau \times \sigma$ by

$$q(x) = \mathrm{if}\ p(x)\ \mathrm{then}\ [\ ]\ \mathrm{else}\ \mathrm{inr}(h(x), t(x))$$

and $k : \sigma \to [\tau]$ by $k = \mathrm{unfd}\ p\ h\ t$ respectively. Then the following commutes:

$$\begin{tikzcd}
\sigma \arrow[r, "q"] \arrow[d, "k"'] & 1 + \tau \times \sigma \arrow[d, "1 + \mathrm{id}_\tau \times k"] \\
{[\tau]} \arrow[r, "\mathrm{unfold}^{[\tau]}"'] & 1 + \tau \times [\tau]
\end{tikzcd}$$

Moreover $k = \text{unfd}\ p\ h\ t$ is the unique morphism making the above diagram commute since $\text{unfold}^{[\tau]} : [\tau] \longrightarrow 1 + \tau \times [\tau]$ is a final $(1 + \tau \times -)$-coalgebra. Further suppose that $p' : \sigma \longrightarrow \text{Bool}, h' : \sigma \longrightarrow \tau$ and $t' : \sigma \longrightarrow \sigma$ are programs such that $p' = p \circ g$, $h' = h \circ g$ and $g \circ t' = t \circ g$. By defining $q'(x) = \text{if}\ p(x)\ \text{then}\ [\ ]\ \text{else}\ \text{inr}(hx, t'x)$, the following diagram:

$$\begin{tikzcd}
\sigma \arrow[rrr, "q'"] \arrow[dr, "g"] \arrow[dd, dashed, "k \circ g"'] & & & 1 + \tau \times \sigma \arrow[dl, "1 + \tau + g"'] \arrow[dd, dashed, "1 + \tau \times (k \circ g)"] \\
& \sigma \arrow[r, "q"'] \arrow[dl, "k"] & 1 + \tau \times \sigma \arrow[dr, "1 \times \tau + k"] & \\
{[\tau]} \arrow[rrr, "\text{unfold}^{[\tau]}"'] & & & 1 + \tau \times [\tau]
\end{tikzcd}$$

commutes. Program fusion then guarantees that $k \circ g$ is the unique map of type $\sigma \longrightarrow [\tau]$ such that the above diagram commutes. In particular, we have

$$(\text{unfd}\ p\ h\ t) \circ t = \text{unfd}\ (p \circ t)\ (h \circ t)\ t \quad (1)$$

$$\text{map}\ f \circ (\text{unfd}\ p\ h\ t) = \text{unfd}\ p\ (f \circ h)\ t \qquad (2)$$

Proof. (of Proposition 6.2 using program fusion)

$$\begin{aligned}
(\text{iterate}\ f) \circ f &= (\text{unfd false id}\ f) \circ f && \text{(definition of iterate)} \\
&= \text{unfd}\ (\text{false} \circ f)\ (\text{id} \circ f)\ f && \text{(fusion (1))} \\
&= \text{unfd false}\ (f \circ \text{id})\ f && \text{(constant, composition)} \\
&= \text{map}\ f \circ \text{unfd false id}\ f && \text{(fusion (2))} \\
&= \text{map}\ f \circ \text{iterate}\ f && \text{(definition of iterate)} \quad \square
\end{aligned}$$

Remark 6.3 The above method is too specialised in that programs involved must be encoded using the unfd function.

Proof. (of Proposition 6.2 using Corollary 5.3)
We prove by induction on n that for any $x : \tau$ and any $f : (\tau \to \tau)$,

$$\text{map}\ f\ (\text{iterate}\ f\ x) =_n \text{iterate}\ f\ f(x).$$

The base case is trivial and the inductive step is justified by:

$$\begin{aligned}
& \text{id}_{n+1}(\text{map}\ f\ (\text{iterate}\ f\ x)) && \\
&= \text{id}_{n+1}(f(x) : \text{map}\ f\ (\text{iterate}\ f\ f(x))) && \text{(definition of iterate, map)} \\
&= f(x) : \text{id}_n(\text{map}\ f\ (\text{iterate}\ f\ f(x))) && \text{(Lemma 6.1)} \\
&= f(x) : \text{id}_n(\text{iterate}\ f\ f(f(x))) && \text{(induction hypothesis)} \\
&= \text{id}_{n+1}(f(x) : \text{iterate}\ f\ f(f(x))) && \text{(Lemma 6.1)} \\
&= \text{id}_{n+1}(\text{iterate}\ f\ f(x)) && \text{(definition of iterate)}
\end{aligned}$$

Thus the desired result holds by Corollary 5.3. □

The next sample application involves the filter function defined by:

filter : $(\tau \to \text{Bool}) \to ([\tau] \to [\tau])$
filter u $[\ ] = [\ ]$
filter u $(x : xs) =$ if $u(x)$ then $(x :$ filter u $xs)$ else filter u xs.

Proposition 6.4 *For any $u : (\tau \to \text{Bool})$, $v : (\tau \to \tau)$ and $l : [\tau]$, it holds:*

$$\text{filter } u \text{ map } v\ l = \text{map } v \text{ filter } (u \circ v)\ l.$$

In [22] the above was established using list-bisimulations. Given a typed-indexed family $\mathcal{R}$ of binary relations $\mathcal{R}_\sigma$ between closed terms of type σ, there is a known recipe of "expanding" $\mathcal{R}$ via the operation $\langle - \rangle$ [8] so that the new family $\langle \mathcal{R} \rangle$ respects the operational behaviour of evaluation at each type σ. For details, see page of 259 of [22]. A *bisimulation* is a typed-indexed family $\mathcal{R}$ of such binary relations $\mathcal{R}_\sigma$ satisfying the condition $\mathcal{R} \subseteq \langle \mathcal{R} \rangle$. The largest bisimulation (which is guaranteed to exist by the Tarski-Knaster fixed point theorem) is called *bisimilarity*. There are two deep insights, originally due to I.A. Mason, S.F. Smith and C.L. Talcott in [19] and re-iterated by A.M. Pitts in [22]:

(I1) Bisimilarity *coincides* with contextual equivalence.
(I2) To prove contextual equivalence of terms, it suffices to find a bisimulation of terms because such a bisimulation must be contained in the largest bisimulation. This is known as the *co-induction principle*.

A special bisimulation called *list-bisimulation* was developed in [22] to prove the contextual equivalence of lists.

Proposition 6.5 (List-bisimulation, Proposition 3.10 of [22])
For any type τ, a binary relation $\mathcal{R} \subseteq \tau \times \tau$ is called a $[\tau]$-bisimulation if whenever $l \mathcal{R} l'$
(1) $l \Downarrow [\ ] \Rightarrow l' \Downarrow [\ ]$
(2) $l' \Downarrow [\ ] \Rightarrow l \Downarrow [\ ]$
(3) $l \Downarrow (x : xs) \Rightarrow \exists x', xs'.(x =_\tau x' \ \&\ xs\ \mathcal{R}\ xs')$
(4) $l \Downarrow (x' : xs') \Rightarrow \exists x, xs.(x =_\tau x' \ \&\ xs\ \mathcal{R}\ xs')$
Then for any $l, l' : [\tau]$, $l = l'$ iff $l\ \mathcal{R}\ l'$ for some $[\tau]$-bisimulation $\mathcal{R}$.

In addition to these, one needs to use induction on the *depths of proofs of evaluation* to establish Proposition 6.4. Here we elaborate. Define the nth level evaluation relation $\Downarrow^n$ (written as $x \Downarrow^n v$) as follows. Replace in the axioms and rule regarding $\Downarrow$ each occurrence of $\Downarrow$ by $\Downarrow^n$ in an axiom or the premise of a rule and replacing $\Downarrow$ by $\Downarrow^{n+1}$ in the conclusion of each rule. Then we have:

$$x \Downarrow v \Leftrightarrow \exists n \in \mathbb{N}.(x \Downarrow^n v) \qquad (\dagger\dagger)$$

It suffices to show that there is a list bisimulation that relates filter u (map v l) and map v (filter $(u \circ v)$ l). Usually it is Hobson's choice:

[8] The original symbol used in [22] is $[-]$ but this confuses with the list operator in this paper.

Proof. (of Proposition 6.4 using list-bisimulation)
Define $\mathcal{R} := \{(\text{filter } u \text{ (map } v \ l), \text{map } v \text{ (filter } (u \circ v) \ l))|l : [\tau]\}$.
Instead of proving the four conditions (1)-(4) of a list bisimulation, we take advantage of (††) and prove by induction on n that:

(i) $\forall l.(\text{filter } u \ (\text{map } v \ l) \Downarrow^n [\ ] \Rightarrow \text{map } v \ (\text{filter } (u \circ v) \ l \Downarrow [\ ])$.
(ii) $\forall l.(\text{map } v \ (\text{filter } u \ (\text{map } v \ l) \Downarrow^n [\ ] \Rightarrow \text{filter } u \ (\text{map } v \ l) \Downarrow [\ ])$.
(iii) $\forall l, x, xs.(\text{filter } u \ (\text{map } v \ l) \Downarrow^n (x : xs)$
$\Rightarrow \exists xs'.(\text{map } v \ \text{filter } (u \circ v) \ l) \Downarrow (x : xs') \ \& \ xs\mathcal{R}xs')$.
(iv) $\forall l, x, xs'.(\text{map } v \ (\text{filter } (u \circ v) \ (x : xs')$
$\Rightarrow \exists xs.(\text{filter } u \ (\text{map } v \ l) \Downarrow (x : xs) \ \& \ xs\mathcal{R}xs')$.

The proofs of (i)-(iv) by induction are straightforward but tedious. □

Proof. (of Proposition 6.4 using Corollary 5.3)
This is trivially true when $l = [\ ]$. So it suffices to prove for $l = (x : xs)$. Again the argument is by induction on n that for every such u, v and l, it holds that

$$\text{filter } u \ \text{map } v \ l =_n \text{map } v \ \text{filter } (u \circ v) \ l.$$

The base case is trivially true. So we prove the inductive step:
Case 1: $u \circ v(x) = \text{true}$.

$$\begin{aligned}
\text{id}_{n+1}(\text{filter } u \ (\text{map } v \ l)) &= \text{id}_{n+1}(\text{filter } u \ (v(x) : \text{map } v \ xs)) \\
&= \text{id}_{n+1}(v(x) : \text{filter } u \ (\text{map } v \ xs)) \\
&= (v(x) : \text{id}_n(\text{filter } u \ (\text{map } v \ xs))) \\
&= (v(x) : \text{id}_n(\text{map } v \ \text{filter } (u \circ v) \ xs)) \\
&= \text{id}_{n+1}(v(x) : \text{map } v \ \text{filter } (u \circ v) \ xs) \\
&= \text{id}_{n+1}(\text{map } v \ (x : \text{filter } (u \circ v) \ xs))
\end{aligned}$$

Case 2: $u \circ v(x) = \text{false}$. Even more immediate than the previous case.
The proof is thus complete by appealing to Corollary 5.3. □

7 Conclusion and future work

The operational domain theory developed herein exploits the bifree algebra structure of the recursive construction offered by the syntax of FPC. It turns out that the chosen categorical framework facilitates a relatively clean theory from which a powerful proof principle emerges. In our future work, we shall explore the following possibilities.

(i) By adapting the methods in [5], construct a purely operational proof of the minimal invariance property (Lemma 3.3) in our setting.
(ii) Investigate the implications of Pitts' work ([21]) on relational properties of domains to our operational domain theory (c.f. similar to that of [5] but with emphasis on the functorial status of FPC type expressions).
(iii) Develop an operational domain-theory that caters for non-deterministic languages, such as [13].

Acknowledgement

My gratefulness goes to my supervisor, M.H. Escardó, whose deep insight and gentle advice I have greatly benefited from. I must also acknowledge T. Streicher for a series of discussions leading to the justification of using the diagonal category as our operational framework. I would also like to extend my gratitude to the three anonymous referees for their comments and suggestions.

References

[1] Abadi, M., and M.P. Fiore, *Syntactic considerations on recursive types*, In *Proceedings of the 11th Annual IEEE Symposium on Logic In Computer Science*, IEEE Computer Society Press. (1996) 242 – 252.

[2] Abramsky, S. and A. Jung, *Domain Theory*, In S. Abramsky, D. Gabbay, T. Maibaum, editors, *Handbook for Logic in Computer Science*, Oxford Science Publications, **3**, Clarendon Press, Oxford. (1994) 1 – 168.

[3] Bird, R., "Introduction to Functional Programming using Haskell (second edition), Prentice Hall. (1998)

[4] Bird, R. and P. Wadler, "Introduction to Functional Programming", Prentice-Hall, New York. (1988)

[5] Birkedal, L. and R. Harper, *Relational Interpretations of Recursive Types in an Operational Setting*, Information and Computation, **155**. (1999) 3 – 63.

[6] Escardó, M.H. and W.K. Ho, *Operational Domain Theory and topology of a sequential language*, In *Proceedings of the 20th Annual IEEE Symposium on Logic In Computer Science*, IEEE Computer Society Press. (2005) 427 – 436

[7] Freyd, P.J., *Recursive types reduced to inductive types*, In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press. (1990) 498 – 507.

[8] Freyd, P.J., *Algebraically complete categories*, In A. Carboni, M.C. Pedicchio, and G. Rosilini, editors, *Proceedings of the 1990 Como Category Conference*, Lecture Notes in Mathematics, **1488**. (1991) 95 – 104.

[9] Freyd, P.J., *Remarks on algebraically compact categories*, In *Applications of Categories in Computer Science*, **177**, Cambridge University Press. (1992) 95 – 106.

[10] Fiore, M.P. and G.D. Plotkin, *An Axiomatisation of Computationally Adequate Domain-Theoretic Models of FPC*, In *Proceedings of the 10th Annual IEEE Symposium on Logic In Computer Science*, IEEE Computer Society Press, (1994) 92 – 102.

[11] Gibbons, J. and G. Hutton, *Proof Methods for Corecursive Programs*, In *Fundamenta Informaticae XX*, IOS Press. (2005) 1 – 14.

[12] Gordon, A.D., *Bisimilarity as a theory of functional programming*, Theoretical Computer Science **228**(1-2). (1999) 5 – 47.

[13] Hennessy, M.C.B. and E.A. Ashcroft, *A mathematical semantics for a nondeterministic typed lambda-calculus*, Theoretical Computer Science, **11**(3). (1980) 227 – 245.

[14] Howe, D.J., *Equality in lazy computation systems*, In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press. (1989) 198 – 203.

[15] Howe, D.J., *Proving congruence of bisimulation in functional programming languages*, Information and Computation, **124**(2). (1996) 103 – 112.

[16] Hutton, G. and J. Gibbons, *The Generic Approximation Lemma*, Information Processing Letters, **79**(4). (2001) 197 – 201.

[17] McCusker, G., *Games and full abstraction for FPC*, Information and Computation, **160**. (2000) 1 – 61.

[18] Mac Lane, S., "Categories for the Working Mathematician", Springer. (1971)

[19] Mason, I.A., S.F. Smith and C.L. Talcott, *From Operational Semantics to Domain Theory*, Information and Computation, **128**(1). (1996) 26 – 47.

[20] Milner, R., *Fully abstract models of typed lambda-calculi*, Theoretical Computer Science, **4**. (1977) 1 – 22.

[21] Pitts, A.M., *Relational Properties of Domains*, Information and Computation, **127**(2). (1996) 66 – 90

[22] Pitts, A.M., *Operationally-based Theories of Program Equivalence*, In P. Dybjer and A.M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, Cambridge Press. (1997) 241 – 298.

[23] Plotkin, G.D., Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford. (1985)

[24] Rohr, A., "A Universal Realizability Model for Sequential Functional Computation", PhD. thesis, Technischen Universitat Darmstadt. (2002)

[25] Simpson, A.K., *Recursive Types in Kleisli Categories*, Unpublished note, Department of Computer Science, University of Edinburgh. (1992) URL: `http://homepages.inf.ed.ac.uk/als/Research`

[26] Streicher, T., *Mathematical Foundations of Functional Programming*, Lecture Notes, Department of Mathematics, Technischen Universitat Darmstadt. (2003) URL: `http://www.mathematik.tu-darmstadt.de/~streicher`.