UNIVERSITY OF Southampton

Faculty of Physical and Applied Sciences
Electronics and Computer Science

CALCO = 2011

CALCO Young Researchers Workshop CALCO-jnr 2011

29 August 2011

Selected Papers

edited by

Corina Cîrstea, Monika Seisenberger, and Toby Wilkinson

Preface

CALCO brings together researchers and practitioners to exchange new results related to foundational aspects and both traditional and emerging uses of algebras and coalgebras in computer science.

This is a high-level, bi-annual conference formed by joining the forces and reputations of CMCS (the International Workshop on Coalgebraic Methods in Computer Science), and WADT (the Workshop on Algebraic Development Techniques). Previous very successful CALCO conferences were held 2005 in Swansea, Wales, 2007 in Bergen, Norway, and 2009 in Udine, Italy. The fourth event took place in 2011 in Winchester, UK.

The CALCO Young Researchers Workshop, CALCO-jnr, is a CALCO satellite event dedicated to presentations by PhD students and by those who completed their doctoral studies within the past few years. The workshop was open to all - many CALCO conference participants attended the CALCO-jnr workshop and vice versa. The workshop had 10 accepted contributions and 25 participants.

CALCO-jnr presentations were selected by the CALCO-jnr programme committee on the basis of submitted 2-page abstracts. After the workshop, the author(s) of each presentation were invited to submit a full 10-15 page paper on the same topic. They were also asked to write anonymous reviews of papers submitted by other authors on related topics. Additional reviewing and the final selection of papers was carried out by the programme committee. The selected papers from the workshop are published here as a University of Southampton technical report. Authors retain copyright, and are encouraged to disseminate the results reported at CALCO-jnr by subsequent publication elsewhere.

We would like to thank the workshop participants, the reviewers, and the CALCO 2011 local organisers for their great efforts and commitment which made this event successful. Special thanks go to John Power and Magne Haveraaen for their splendid work on the programme committee and their continuous support of CALCO-jnr. We are also grateful to the London Mathematical Society, the University of Southampton and the British Logic Colloquium for their support.

March 2012

Corina Cîrstea
Monika Seisenberger
Toby Wilkinson

Table of Contents

Domain-theoretic Modeling of a Language of Realizers

Tie Hou and Ulrich Berger

Department of Computer Science, Swansea University, UK
{cshou,u.berger}@swansea.ac.uk

Abstract. We study the domain-theoretic semantics of a Church-style typed λ-calculus with constructors, pattern matching and recursion, and show that it is closely related to the semantics of its untyped counterpart. The proof uses hybrid logical relations, which are related to Tait's computability method and Girard's method of reducibility candidates. The reason for studying this domain-theoretic semantics is that it allows for very simple and elegant proofs of computational adequacy, and hence for the correctness of program extraction.

1 Introduction

This paper is part of a research project aiming at a semantical foundation for program extraction from proofs [7]. It contributes to a soundness proof for a language of realizers of proofs involving inductive and coinductive definitions. The natural language of realizers for inductive and coinductive definitions is a typed lambda calculus with types modeling initial algebras and final coalgebras, and terms modeling structural recursion and corecursion. In this paper we study a more general calculus that allows fixed points of arbitrary type operators and definitions of functions by general recursion. The advantage of this generality is that our results will apply to all conceivable extensions of our theory of realizers of inductive and coinductive definitions.

We study the domain-theoretic semantics of a Church-style typed λ-calculus with constructors, pattern matching and recursion, and compare it with its untyped counterpart. We work with polymorphic types that allow fixed points of arbitrary type operators. A type ρ is interpreted as (the image of) a finitary projection $\langle\rho\rangle$, following [2]. The main result (Theorem 24) relates the semantics of a typed term M with its untyped variant M^-: if M has type ρ, where ρ is a regular type, that is, fixed-points are only taken of positive operators, then

$$\langle\rho\rangle[\![M^-]\!] = [\![M]\!].$$

The proof uses a hybrid version of logical relations. We do not know whether the result also holds if ρ is not regular.

The motivation for this study comes from program extraction from proofs via realizability (see e.g. [6,4,7,8] for applications in constructive analysis) where one has the choice of extracting typed or untyped terms from proofs. Our result shows that if the extracted type is regular, the choice does not matter. In [5] the soundness of a realizability interpretation based on a fragment of the untyped version of our calculus was proven, and the calculus was shown to be computationally adequate with respect to a domain-theoretic semantics (the same semantics we are considering here). In [9] it was shown that the extracted programs admit a Curry-style typing. In the present paper we provide the missing semantical link to Curry-style typing.

2 Types and terms

In this and the next section we study the syntax and semantics of types and typed terms. Untyped terms will be introduced in Section 4.

Definition 1 (Types) *The set of types is defined by the following grammar:*

$$\mathbf{Type} \ni \rho, \sigma, \tau ::= \alpha \mid \rho \to \sigma \mid \mathbf{1} \mid \rho \times \sigma \mid \rho + \sigma \mid \mathrm{fix}\ \alpha.\rho.$$

where α ranges over a set **TVar** *of type variables. A fixed-point construction,* fix $\alpha.\rho$, *binds all free occurrences of α in ρ.*

We work with a Church-style typed lambda-calculus with constructors, pattern matching and recursion which we call *Language of Realizers* (**LoR**) because its terms are intended to be used as extracted programs from proofs obtained by a realizability interpretation.

We consider only the constructors Nil (nullary), Pair (binary), and Left, Right, In (unary). The intention behind the first four constructors should be obvious. The constructor In is used to model type fixed points up to isomorphism. Many definitions and results could be extended to an arbitrary set of constructors.

Definition 2 (Terms) *The set of (Church-style typed) terms is defined by*

$$\mathbf{LoR} \ni M, N, R_i ::= x \mid \lambda x : \rho.M \mid MN \mid \mathrm{rec}\ x : \rho.M \mid C(M_1, \ldots, M_n) \mid \\ \mathrm{case}\ M\ \mathrm{of}\ \{C_i(\boldsymbol{x_i}) \to R_i\}_{i\in\{1,\ldots,n\}}.$$

where x ranges over a set of variables **Var**, *C is a constructor of arity n, and in* case M of $\{C_i(\boldsymbol{x_i}) \to R_i\}_{i\in\{1,\ldots,n\}}$ *all constructors C_i are distinct and each $\boldsymbol{x}_i$ is a vector of distinct variables whose length coincide with the arity of C_i. Lambda abstraction, $\lambda x : \rho.M$, and recursion,* rec $x : \rho.M$, *bind all free occurrences of x in M, and a pattern matching clause, $C_i(\boldsymbol{x_i}) \to R_i$, binds all free occurrences of $\boldsymbol{x_i}$ in R_i.*

We introduce typing rules for **LoR**-terms. A *type context* is a set of pairs $\Gamma := x_1 : \rho_1, \ldots, x_n : \rho_n$ (for notational convenience we omit the curly braces) where ρ_i are types and x_i are distinct variables. The set of variables $\{x_1, \ldots, x_n\}$ (which may be empty) is denoted by $\mathrm{dom}(\Gamma)$.

The relation $\Gamma \vdash M : \rho$ (M is a **LoR** term of type ρ in context Γ) is inductively defined in Table 1.

Table 1. Typing Rules for **LoR** terms

$$\Gamma \vdash \mathrm{Nil} : 1 \qquad \Gamma, x : \rho \vdash x : \rho$$

$$\frac{\Gamma, x : \rho \vdash M : \sigma}{\Gamma \vdash \lambda x : \rho . M : \rho \to \sigma} \qquad \frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mathrm{rec}\ \ x : \tau . M : \tau}$$

$$\frac{\Gamma \vdash M : \rho \to \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash M\ N : \sigma} \qquad \frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{Pair}(M, N) : \rho \times \sigma}$$

$$\frac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathrm{Left}(M) : \rho + \sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{Right}(M) : \rho + \sigma}$$

$$\frac{\Gamma \vdash M : \rho + \sigma \qquad \Gamma, x_1 : \rho \vdash L : \tau \qquad \Gamma, x_2 : \sigma \vdash R : \tau}{\Gamma \vdash \mathrm{case}\ M\ \ \mathrm{of}\ \{\mathrm{Left}(x_1) \to L; \mathrm{Right}(x_2) \to R\} : \tau}$$

$$\frac{\Gamma \vdash M : \rho \times \sigma \qquad \Gamma, x : \rho, y : \sigma \vdash N : \tau}{\Gamma \vdash \mathrm{case}\ M\ \ \mathrm{of}\ \{\mathrm{Pair}(x, y) \to N\} : \tau}$$

$$\frac{\Gamma \vdash M : \rho[\mathrm{fix}\ \alpha.\rho/\alpha]}{\Gamma \vdash \mathrm{In}(M) : \mathrm{fix}\ \alpha.\rho}$$

$$\frac{\Gamma \vdash M : \mathrm{fix}\ \alpha.\rho \qquad \Gamma, x : \rho[\mathrm{fix}\ \alpha.\rho/\alpha] \vdash N : \sigma}{\Gamma \vdash \mathrm{case}\ M\ \ \mathrm{of}\ \{\mathrm{In}(x) \to N\} : \sigma}$$

3 Domain-theoretic semantics

We assume familiarity with the basic theory of Scott domains and the method of defining domains by recursive domain equations [11,1,3]. We omit the proofs of the most basic results since they are rather elementary, or can be found in the above cited literature. The reason for working with Scott domains is that all the semantic constructions we need are readily available, e.g. cartesian closure, solutions to recursive domain equation, recursive definition of functions, interpretation of types, including recursive types, as finitary projections. All these constructions are very elementary and do not require a heavy category-theoretical machinery.

By a *Scott-domain*, or *domain* for short, we mean a bounded complete ω-algebraic dcpo with least element. We will denote the least element of a domain by $\perp$. By **1** we denote the sole-element domain {Nil}, and by $(D_1 + \ldots + D_n)_\perp$, $D \times E$, $[D \to E]$ the separated sum, cartesian product, and continuous function space of domains [1].

Due to ω-algebraicity, every element x of a domain D is the directed countable supremum of compact elements, where $y \in D$ is called *compact* if for every

[1] These domain operations should not be confused with the syntactic constructors for types which for simplicity we denoted by the same symbols.

directed $A \subseteq D$, A has a supremum $\sqcup A$ and $y \sqsubseteq \sqcup A$ s.t. $y \sqsubseteq z$ for some $z \in A$. By D_c we denote the set of compact elements of D.

Definition 3 (Subdomain) *$E \subseteq D$ is a* subdomain *of D if*

(i) $\bot_D \in E$.
(ii) If $A \subseteq E$ and $\sqcup_D A$ exists in D, then $\sqcup_D A \in E$ and $\sqcup_D A = \sqcup_E A$.
(iii) If x is compact in E, then x is compact in D.
(iv) $\forall y \in D_c \forall x \in E(y \sqsubseteq x \to \exists y' \in E_c(y \sqsubseteq y' \sqsubseteq x))$.

Lemma 4 *Let $E \subseteq D$ be a subdomain of D. Then E is a domain.*

Proof. By verifying clauses (directed complete, algebraic, bounded complete) of the definition of domain.

Following [2] we interpret types as finitary projections in D. Since the range of a finitary projection is a subdomain of D the semantics of types can be viewed as a domain. This approach provides an easy solution to the problem of defining the semantics of a fixed point type: one can simply take the least fixed point of a suitable continuous function on the domain [D → D].

Definition 5 (Finitary projection) *$f : D \to D$ is a* projection *if*

– *f is continuous,*
– $f \sqsubseteq \mathrm{id}$, *i.e.* $\forall x \in D. f(x) \sqsubseteq x$,
– $f \circ f = f$, *i.e.* $\forall x \in D. f(f(x)) = f(x)$.

A projection f is finitary *if the range of f, denoted by $f(D)$, is a subdomain of D.*

For $f : D \to D$ we set $\mathrm{Fix}(f) := \{x \in D \mid f(x) = x\}$. Obviously, if $f \circ f = f$, then $f(D) = \mathrm{Fix}(f)$.

In the following two lemmas we assume that $p : D \to D$ is a projection, and we set $p(D)_c := D_c \cap p(D)$. We omit their proofs since they are easy.

Lemma 6 *E is a subdomain of D if and only if there exists a finitary projection $p : D \to D$ such that $E = p(D)$.*

Lemma 7 *The following are equivalent:*

(a) p is finitary
(b) $\forall x \in \mathrm{D}(A_x := \{a \in p(\mathrm{D})_c \mid a \sqsubseteq x\})$ *is directed and* $p(x) = \sqcup A_x$.
(c) $\exists A \subseteq \mathrm{D}_c. \forall x \in \mathrm{D}(p(x) = \sqcup\{a \in A \mid a \sqsubseteq x\}$.

Now we define by a recursive domain equation a particular domain D which we will use to interprete types and terms.

Definition 8 *We define the Scott domain* D *by the recursive domain equation:*

$$\mathrm{D} \simeq (\mathbf{1} + \mathrm{D} + \mathrm{D} + \mathrm{D} + \mathrm{D} \times \mathrm{D} + [\mathrm{D} \to \mathrm{D}])_{\perp}$$

Using the constructors of **LoR** *as names for the injections into the sum, each element in* D *has exactly one of the following forms:* $\perp$, Nil, $\mathrm{Left}(a)$, $\mathrm{Right}(a)$, $\mathrm{In}(a)$, $\mathrm{Pair}(a, b)$, $\mathrm{Fun}(f)$*, where* a *and* b *range over* D*, and* f *ranges over continuous function from* D *to* D*.*

It will be convenient to use the following continuous functions: $\mathrm{case}_{C_1,\dots,C_n} : \mathrm{D} \to [\mathrm{D}^{\mathrm{arity}(C_1)} \to \mathrm{D}] \to \dots \to [\mathrm{D}^{\mathrm{arity}(C_n)} \to \mathrm{D}] \to \mathrm{D}$ defined by

$$\mathrm{case}_{C_1,\dots,C_n}\ a\ f_1 \dots f_n := \begin{cases} f_i(\boldsymbol{b_i}) & \text{if } a = C_i(\boldsymbol{b_i}), \\ \perp & \text{otherwise.} \end{cases}$$

We also use an informal lambda-notation $\lambda a.f(a)$ and composition $f \circ g$ to define continuous functions on D. We don't prove the continuity in each case since this follows from well-known fact about the category of Scott domains and continuous functions. We also let $\mathrm{LFP} : [\mathrm{D} \to \mathrm{D}] \to \mathrm{D}$ be the continuous least fixed point operator, which can be defined by $\mathrm{LFP}(f) = \bigsqcup_n f^n(\perp)$.

Definition 9 (Semantics of types) *For every type* ρ *we define* $\langle\rho\rangle : [[\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}} \to [\mathrm{D} \to \mathrm{D}]]$ [2]

$$\begin{aligned}
\langle\mathbf{1}\rangle\zeta(a) &= \mathrm{case}_{\mathrm{Nil}}\ a\ \mathrm{Nil} \qquad \left(= \begin{cases} \mathrm{Nil} & \textit{if } a = \mathrm{Nil} \\ \perp & \textit{otherwise} \end{cases}\right) \\
\langle\alpha\rangle\zeta(a) &= \zeta(\alpha)(a) \\
\langle\rho + \sigma\rangle\zeta(a) &= \mathrm{case}_{\mathrm{Left,Right}}\ a\ (\mathrm{Left} \circ \langle\rho\rangle\zeta)\ (\mathrm{Right} \circ \langle\sigma\rangle\zeta) \\
\langle\rho \times \sigma\rangle\zeta(a) &= \mathrm{case}_{\mathrm{Pair}}\ a\ (\lambda b_1 b_2.\mathrm{Pair}(\langle\rho\rangle\zeta(b_1), \langle\sigma\rangle\zeta(b_2))) \\
\langle\rho \to \sigma\rangle\zeta(a) &= \mathrm{case}_{\mathrm{Fun}}\ a\ (\mathrm{Fun}(\lambda f.\langle\sigma\rangle\zeta \circ f \circ \langle\rho\rangle\zeta)) \\
\langle\mathrm{fix}\ \alpha.\rho\rangle\zeta &= \mathrm{LFP}(\lambda p.\lambda a.\mathrm{case}_{\mathrm{In}}\ a\ (\lambda b.\mathrm{In}(\langle\rho\rangle\zeta[\alpha := p](b))))
\end{aligned}$$

We set $[\![\rho]\!]\zeta := (\langle\rho\rangle\zeta)(\mathrm{D})$.

Lemma 10 *If* $\zeta : [\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}}$ *is such that* $\zeta(\alpha)$ *is a finitary projection for all* $\alpha \in \mathbf{TVar}$*, then* $\langle\rho\rangle\zeta$ *is a finitary projection.*

Proof. By induction on ρ using Lemma 6 and 7.

Now we are ready to define the semantics of **LoR**-terms. The leading idea in the definition of the value of a typed lambda-abstraction $\lambda x : \rho.M$ is that the domain of the resulting function is (the semantics of) ρ. Therefore, the incoming argument a is first projected down to ρ.

[2] $[\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}}$ is the set of type environments, i.e., functions from **TVar** to $[\mathrm{D} \to \mathrm{D}]$.

Definition 11 (Semantics of terms) *For every environment* $\zeta : [\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}}$, $\eta : \mathbf{Var} \to \mathrm{D}$*, and every* **LoR** *term* M *we define the value* $[\![M]\!]^{\zeta}\eta \in \mathrm{D}$.

$$\begin{aligned}
[\![x]\!]^{\zeta}\eta &= \eta(x)\\
[\![C(M_1,\ldots,M_n)]\!]^{\zeta}\eta &= C([\![M_1]\!]^{\zeta}\eta,\ldots,[\![M_n]\!]^{\zeta}\eta)\\
[\![MN]\!]^{\zeta}\eta &= \mathrm{case}_{\mathrm{Fun}}\ ([\![M]\!]^{\zeta}\eta)\ (\lambda f.f([\![N]\!]^{\zeta}\eta))\\
[\![\lambda x:\rho.M]\!]^{\zeta}\eta &= \mathrm{Fun}(\lambda a.[\![M]\!]^{\zeta}\eta[x := \langle\rho\rangle\zeta(a)])\\
[\![\mathrm{rec}\ x:\tau.M]\!]^{\zeta}\eta &= \mathrm{LFP}(\lambda a.[\![M]\!]^{\zeta}\eta[x := \langle\tau\rangle\zeta(a)])\\
[\![\mathrm{case}\ M\ \mathrm{of}\ \{C_i(x_i) \to R_i\}_i]\!]^{\zeta}\eta &= \mathrm{case}_{C_1,\ldots,C_n}\ ([\![M]\!]^{\zeta}\eta)\ (\lambda \boldsymbol{a}.[\![R_i]\!]^{\zeta}\eta[\boldsymbol{x_i} := \boldsymbol{a}])_i
\end{aligned}$$

One can prove the following soundness theorem, stating that if from a context Γ we can derive **LoR** term M with type ρ, and for every variable x_i in the context Γ, $\eta(x_i)$ is an element of $[\![\rho_i]\!]\zeta$, then the value of term M is an element of the value of type ρ. We write $\eta \in [\![\Gamma]\!]\zeta$ as an abbreviation for $\Gamma(x_i) = \rho_i \wedge \eta(x_i) \in [\![\rho_i]\!]\zeta$.

Theorem 12 (Soundness For LoR terms) *If* $\Gamma \vdash M : \rho$ *and* $\eta \in [\![\Gamma]\!]\zeta$*, then* $[\![M]\!]^{\zeta}\eta \in [\![\rho]\!]\zeta$.

Proof. By induction on the structure of the relation $\Gamma \vdash M : \rho$.

4 Relating typed and untyped terms

We now relate the semantics of typed terms with the semantics of untyped terms which are defined exactly as typed terms except that the type annotations for abstraction and recursion are omitted:

Definition 13 (Untyped terms)

$$\begin{aligned}
\mathbf{LoR}^- \ni M, N, R_i ::= x \mid \lambda x.M \mid MN \mid \mathrm{rec}\ x.M \mid C(M_1,\ldots,M_n) \mid\\
\mathrm{case}\ M\ \mathrm{of}\ \{C_i(\boldsymbol{x_i}) \to R_i\}_{i\in\{1,\ldots,n\}}
\end{aligned}$$

The same provisions made in Definition 2 for typed terms apply here.

The semantics of untyped terms is straightforward. It can be defined exactly as in the typed case except that the type environment $\zeta : [\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}}$ and finitary projections involved in typed abstraction and recursion are omitted.

Definition 14 (Semantics of untyped terms) *For every environment* $\eta : \mathbf{Var} \to \mathrm{D}$ *and every* $\mathbf{LoR}^-$ *term* M *we define the value* $[\![M]\!]\eta \in \mathrm{D}$.

$$\begin{aligned}
[\![x]\!]\eta &= \eta(x)\\
[\![C(M_1,\ldots,M_n)]\!]\eta &= C([\![M_1]\!]\eta,\ldots,[\![M_n]\!]\eta)\\
[\![MN]\!]\eta &= \mathrm{case}_{\mathrm{Fun}}\ ([\![M]\!]\eta)\ (\lambda f.f([\![N]\!]\eta))\\
[\![\lambda x.M]\!]\eta &= \mathrm{Fun}(\lambda a.[\![M]\!]\eta[x := a])\\
[\![\mathrm{rec}\ x.M]\!]\eta &= \mathrm{LFP}(\lambda a.[\![M]\!]\eta[x := a])\\
[\![\mathrm{case}\ M\ \mathrm{of}\ \{C_i(x_i) \to R_i\}_i]\!]\eta &= \mathrm{case}_{C_1,\ldots,C_n}\ ([\![M]\!]\eta)\ (\lambda \boldsymbol{a}.[\![R_i]\!]\eta[\boldsymbol{x_i} := \boldsymbol{a}])_i
\end{aligned}$$

Our main result, the Coincidence Theorem 24, only applies to terms that are typed w.r.t. to a restricted notion of types where fixed point types fix $\alpha.\rho$ are allowed only if ρ is positive in α.

Definition 15 (ρ positive/negative in α) *We give the following definitions.*
α is positive in α.
1 *is positive/negative in α.*
$\rho \to \sigma$ is positive in α if ρ is negative in α and σ is positive in α.
$\rho \to \sigma$ is negative in α if ρ is positive in α and σ is negative in α.
$\rho + \sigma$ and $\rho \times \sigma$ are positive in α if ρ and σ are positive in α.
$\rho + \sigma$ and $\rho \times \sigma$ are negative in α if ρ and σ are negative in α.
fix *$\beta.\rho$ is positive in α if $\alpha = \beta$ or ρ is positive in α.*
fix *$\beta.\rho$ is negative in α if $\alpha = \beta$ or ρ is negative in α.*

Definition 16 (Regular types) *We define* regular *types ρ as follows.*
1 *is regular.*
α is regular.
$\rho + \sigma$, $\rho \times \sigma$, $\rho \to \sigma$ are regular if ρ and σ are regular.
fix *$\alpha.\rho$ is regular if ρ is regular and ρ is positive in α.*

In the following all types are assumed to be regular.

To prove our main result we define a hybrid logical relation $\sim_\rho^{\mathrm{R},\zeta} \subseteq \mathrm{D} \times \mathrm{D}$ which can intuitively be understood as a notion of equivalence of elements of a *regular* type ρ. We use the informal (second-order) lambda abstraction $\Lambda r \subseteq \mathrm{D} \times \mathrm{D}$. to define functions on the set $\mathcal{P}(\mathrm{D}^2)$ of binary relations on D.

Definition 17 and Lemma 18 below should be considered simultaneously, since in the clause of fix $\alpha.\rho$, the clause is well-defined only if ρ is positive in α.

Definition 17 (Hybrid Logical Relation) *In the following definition it assumed that* $\mathrm{R} : \mathbf{TVar} \to \mathcal{P}(\mathrm{D}^2)$ *and* $\zeta : [\mathrm{D} \to \mathrm{D}]^{\mathbf{TVar}}$.

$$\begin{aligned}
\sim_{\mathbf{1}}^{\mathrm{R},\zeta} &:= \{(\bot,\bot),(\mathrm{Nil},\mathrm{Nil})\}\\
\sim_{\alpha}^{\mathrm{R},\zeta} &:= \mathrm{R}(\alpha)\\
\sim_{\rho_1\times\rho_2}^{\mathrm{R},\zeta} &:= \{(\bot,\bot)\} \cup \{(\mathrm{Pair}(a_1,a_2),\mathrm{Pair}(b_1,b_2)) \mid a_1 \sim_{\rho_1}^{\mathrm{R},\zeta} b_1, a_2 \sim_{\rho_2}^{\mathrm{R},\zeta} b_2\}\\
\sim_{\rho_1+\rho_2}^{\mathrm{R},\zeta} &:= \{(\bot,\bot)\} \cup \{(\mathrm{Left}(a_1),\mathrm{Left}(b_1)) \mid a_1 \sim_{\rho_1}^{\mathrm{R},\zeta} b_1\}\\
&\qquad\qquad \cup \{(\mathrm{Right}(a_2),\mathrm{Right}(b_2)) \mid a_2 \sim_{\rho_2}^{\mathrm{R},\zeta} b_2\}\\
\sim_{\rho\to\sigma}^{\mathrm{R},\zeta} &:= \{(\bot,\bot)\} \cup \{(\mathrm{Fun}(f),\mathrm{Fun}(g)) \mid\\
&\qquad\qquad \forall a,b \in \mathrm{D}(a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow f(a) \sim_{\sigma}^{\mathrm{R},\zeta} g(b))\\
&\qquad\qquad \wedge \langle\sigma\rangle\zeta \circ f \circ \langle\rho\rangle\zeta = \langle\sigma\rangle\zeta \circ g \circ \langle\rho\rangle\zeta\}\\
\sim_{\mathrm{fix}\ \alpha.\rho}^{\mathrm{R},\zeta} &:= \mathrm{LFP}(\Lambda r \subseteq \mathrm{D} \times \mathrm{D}.\{(\mathrm{In}(a),\mathrm{In}(b)) \mid\\
&\qquad\qquad a \sim_{\rho}^{\mathrm{R}[\alpha:=r],\zeta[\alpha:=\mathrm{LFP}(\lambda p\in[\mathrm{D}\to\mathrm{D}].\langle\rho\rangle\zeta[\alpha:=p])]} b\})
\end{aligned}$$

Remark 1. Logical relations have been used successfully to prove properties of typed systems. Famous examples are the strong normalization proofs by Tait and Girard using logical relations called computability predicates or reducibility candidates. The crucial feature of a logical relation is that it is a family of relations indexed by types and defined by induction on types such that all type constructors are interpreted by their logical interpretations, e.g. $\rightarrow$ is interpreted as logical implication $\Rightarrow$. Our logical relation is hybrid because of the added component $\langle\sigma\rangle\zeta \circ f \circ \langle\rho\rangle\zeta = \langle\sigma\rangle\zeta \circ g \circ \langle\rho\rangle\zeta$ in the definition of $\sim^{\mathrm{R},\zeta}_{\rho\to\sigma}$.

Lemma 18

(1) If ρ is positive in α, then $\Lambda r \subseteq \mathrm{D}\times\mathrm{D}.\ \sim^{\mathrm{R}[\alpha:=r],\zeta}_{\rho}$ is monotone.
(2) If ρ is negative in α, then $\Lambda r \subseteq \mathrm{D}\times\mathrm{D}.\ \sim^{\mathrm{R}[\alpha:=r],\zeta}_{\rho}$ is anti-monotone.

Proof. By induction on ρ.

The notion of *admissibility* has been used in [1] and generalized in [10], where is used to prove properties of least fixed points. An admissible relation holds for the least upper bound of a chain, if it contains $(\perp,\perp)$ and it holds for every element of the chain.

Definition 19 (Admissible relation) *A relation* $\mathrm{R} \subseteq \mathrm{D}^2$ *on* D *is called* admissible *if it satisfies*

1. $(\perp,\perp) \in \mathrm{R}$.
2. *If* $(d_n, d'_n) \in \mathrm{R}$ *and* $(d_n, d'_n) \sqsubseteq (d_{n+1}, d'_{n+1})$ *for all* n*, then* $\sqcup_{n\in N}(d_n, d'_n) \in \mathrm{R}$.

Note that a finite relation $R \subseteq \mathrm{D}^2$ with $(\perp,\perp) \in R$ is always admissible.

Let $\mathrm{Ad} := \{\mathrm{R} \subseteq \mathrm{D}^2 \mid \mathrm{R} \text{ is admissible}\}$.

Lemma 20 Ad *is a complete lattice with* $\sqcap_{\mathrm{Ad}} = \sqcap_{\mathcal{P}(\mathrm{D}^2)} = \cap$.

Proof. Easy.

Lemma 21 *If* $\mathrm{R}(\alpha)$ *is admissible for all* $\alpha \in \mathbf{TVar}$*, then* $\sim^{\mathrm{R},\zeta}_{\rho}$ *is admissible.*

Proof. By induction on ρ.

We only look at the most interesting and difficult case, which is fix $\alpha.\rho$.

We have $\sim^{\mathrm{R},\zeta}_{\mathrm{fix}\ \alpha.\rho} = \mathrm{LFP}_{\mathcal{P}(\mathrm{D}^2)}(\Phi)$ where

$$\begin{aligned}&\Phi : \mathcal{P}(\mathrm{D}^2) \to \mathcal{P}(\mathrm{D}^2)\\ &\Phi(r) = \sim^{\mathrm{R}[\alpha:=r],\zeta[\alpha:=\mathrm{LFP}(\lambda p\in[\mathrm{D}\to\mathrm{D}].\langle\rho\rangle\zeta[\alpha:=p])]}_{\rho}\ .\end{aligned}$$

We have

$$\begin{aligned}\mathrm{LFP}_{\mathcal{P}(\mathrm{D}^2)}(\Phi) &= \sqcap_{\mathcal{P}(\mathrm{D}^2)}\{r \in \mathcal{P}(\mathrm{D}^2) \mid \Phi(r) \subseteq r\}\\ &= \sqcap_{\mathrm{Ad}}\{r \in \mathrm{Ad} \mid \Phi(r) \subseteq r\}\\ &= \mathrm{LFP}_{\mathrm{Ad}}(\Phi)\end{aligned}$$

By I.H. if $r \in \mathrm{Ad}$, then $\Phi(r) \in \mathrm{Ad}$, i.e. $\Phi : \mathrm{Ad} \to \mathrm{Ad}$.

By I.H. we get $\sim^{\mathrm{R}[\alpha:=r],\zeta[\alpha:=\mathrm{LFP}(\lambda p\in[\mathrm{D}\to\mathrm{D}].\langle\rho\rangle\zeta[\alpha:=p])]}_{\rho}$ is admissible.

By Lemma 18, we get Φ is monotone. Applying Theorem 20 and the Knaster-Tarski theorem, we get $\mathrm{LFP}(\Phi)$ is admissible.

Lemma 22

(1) $a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow \langle\rho\rangle\zeta(a) = \langle\rho\rangle\zeta(b)$.
(2) $a, b \in \langle\rho\rangle\zeta(\mathrm{D}) \Rightarrow (a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow a = b)$.
(3) $a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow \langle\rho\rangle\zeta(a) \sim_{\rho}^{\mathrm{R},\zeta} b$.

Proof. By induction on ρ.

Let M be a Church-style term, M^- the corresponding untyped term and ρ a regular recursive type.

Let $\eta \sim_{\Gamma}^{\mathrm{R},\zeta} \eta'$ denote the following: for all $x \in \mathrm{dom}(\Gamma)$, if $\Gamma(x) = \sigma$, then $\eta(x) \sim_{\sigma}^{\mathrm{R},\zeta} \eta'(x)$.

Let $\Gamma \vdash^{\mathrm{r}} M : \rho$ mean that $\Gamma \vdash M : \rho$ has been derived using regular types only.

The following lemma is the core of the proof of the Coincidence Theorem.

Lemma 23

$$\Gamma \vdash^{\mathrm{r}} M : \rho, \eta \sim_{\Gamma}^{\mathrm{R},\zeta} \eta' \Rightarrow [\![M]\!]^{\zeta}\eta \sim_{\rho}^{\mathrm{R},\zeta} [\![M^-]\!]\eta'.$$

Proof. By induction on the structure of the relation $\Gamma \vdash^{\mathrm{r}} M : \rho$.

The interesting cases are lambda abstraction and recursion.

1. $\dfrac{\Gamma, x : \rho \vdash^{\mathrm{r}} M : \sigma}{\Gamma \vdash^{\mathrm{r}} \lambda x : \rho.M : \rho \to \sigma}$.
   To show $[\![\lambda x : \rho.M]\!]^{\zeta}\eta \sim_{\rho\to\sigma}^{\mathrm{R},\zeta} [\![\lambda x.M^-]\!]\eta'$.
   By Definition 11 and 14, we have

$$[\![\lambda x : \rho.M]\!]^{\zeta}\eta = \mathrm{Fun}(f) \text{ where } f(a) = [\![M]\!]^{\zeta}\eta[x := \langle\rho\rangle\zeta(a)]$$
$$[\![\lambda x.M^-]\!]\eta' = \mathrm{Fun}(g) \text{ where } g(b) = [\![M^-]\!]\eta'[x := b] \qquad (\diamond)$$

Then it is to show $\mathrm{Fun}(f) \sim_{\rho\to\sigma}^{\mathrm{R},\zeta} \mathrm{Fun}(g)$. By definition of hybrid logical relation (Definition 17), it is to show

$$\forall a, b \in \mathrm{D}(a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow f(a) \sim_{\sigma}^{\mathrm{R},\zeta} g(b)) \qquad \text{(i)}$$

and

$$\langle\sigma\rangle\zeta \circ f \circ \langle\rho\rangle\zeta = \langle\sigma\rangle\zeta \circ g \circ \langle\rho\rangle\zeta \qquad \text{(ii)}$$

We have the following

$$\forall a, b \in \mathrm{D}(a \sim_{\rho}^{\mathrm{R},\zeta} b \Rightarrow [\![M]\!]^{\zeta}\eta[x := \langle\rho\rangle\zeta(a)] \sim_{\sigma}^{\mathrm{R},\zeta} [\![M^-]\!]\eta'[x := b]) \qquad \text{(IH1)}$$

Since we have the assumption $\eta \sim_{\Gamma}^{\mathrm{R},\zeta} \eta'$, in order to apply IH1, we need to show $\eta[x := \langle\rho\rangle\zeta(a)] \sim_{\Gamma,x:\rho}^{\mathrm{R},\zeta} \eta'[x := b]$.

case1 If $y \in \mathrm{dom}(\Gamma)$. Then $(\Gamma, x : \rho)(y) = \Gamma(y)$, $\eta[x := \langle\rho\rangle\zeta(a)](y) = \eta(y)$ and $\eta'[x := b](y) = \eta'(y)$. By assumption, we get $\eta(y) \sim_{\Gamma(y)}^{\mathrm{R},\zeta} \eta'(y)$.

case2 If $y = x$. Then $(\Gamma, x : \rho)(y) = \rho$, $\eta[x := \langle\rho\rangle\zeta(a)](y) = \langle\rho\rangle\zeta(a)$ and $\eta'[x := b](y) = b$. We get $\langle\rho\rangle\zeta(a) \sim_\rho^{\mathrm{R},\zeta} b$ by Lemma 22(3) since $a \sim_\rho^{\mathrm{R},\zeta} b$.

For (i), it follows by applying IH1.

For (ii), it is to show $\forall a \in \mathrm{D}.\langle\sigma\rangle\zeta(f(\langle\rho\rangle\zeta(a))) = \langle\sigma\rangle\zeta(g(\langle\rho\rangle\zeta(a)))$. By equations ($\diamond$), it is to show $\langle\sigma\rangle\zeta([\![M]\!]^\zeta\eta[x := \langle\rho\rangle\zeta(\langle\rho\rangle\zeta(a))]) = \langle\sigma\rangle\zeta([\![M^-]\!]\eta'[x := \langle\rho\rangle\zeta(a)])$.

By IH1, we get $[\![M]\!]^\zeta\eta[x := \langle\rho\rangle\zeta(\langle\rho\rangle\zeta(a))] \sim_\sigma^{\mathrm{R},\zeta} [\![M^-]\!]\eta'[x := \langle\rho\rangle\zeta(a)]$.

Then applying Lemma 22(1), proved.

2. $\dfrac{\Gamma, x : \tau \vdash^{\mathrm{r}} M : \tau}{\Gamma \vdash^{\mathrm{r}} \mathrm{rec}\ \ x : \tau.M : \tau}$.

To show $[\![\mathrm{rec}\ x : \tau.M]\!]^\zeta\eta \sim_\tau^{\mathrm{R},\zeta} [\![\mathrm{rec}\ x.M^-]\!]\eta'$.

By Definition 11 and 14, we have

$$\begin{aligned}[\![\mathrm{rec}\ x : \tau.M]\!]^\zeta\eta &= \mathrm{LFP}(f) \text{ where } f(a) = [\![M]\!]^\zeta\eta[x := \langle\tau\rangle\zeta(a)]\\ [\![\mathrm{rec}\ x.M^-]\!]\eta' &= \mathrm{LFP}(g) \text{ where } g(b) = [\![M^-]\!]\eta'[x := b]\end{aligned}$$

Now to show $\mathrm{LFP}(f) \sim_\tau^{\mathrm{R},\zeta} \mathrm{LFP}(g)$.

By definition it is to show $\sqcup_n f^n(\bot) \sim_\tau^{\mathrm{R},\zeta} \sqcup_n g^n(\bot)$. Then it is to show the following two statements.

- $\forall n.f^n(\bot) \sim_\tau^{\mathrm{R},\zeta} g^n(\bot)$.

  By induction on n.

  <u>Base: $n \equiv 0$</u>. To show $f^0(\bot) \sim_\tau^{\mathrm{R},\zeta} g^0(\bot)$, i.e. $\bot \sim_\tau^{\mathrm{R},\zeta} \bot$. Trivial.

  <u>Step: $n \equiv n + 1$</u>. Assume $f^n(\bot) \sim_\tau^{\mathrm{R},\zeta} g^n(\bot)$, to show $f^{(n+1)}(\bot) \sim_\tau^{\mathrm{R},\zeta} g^{(n+1)}(\bot)$.

  We have

  $$\begin{aligned}f^{(n+1)}(\bot) &= f(f^n(\bot)) = [\![M]\!]^\zeta\eta[x := \langle\tau\rangle\zeta(f^n(\bot))],\\ g^{(n+1)}(\bot) &= g(g^n(\bot)) = [\![M^-]\!]\eta'[x := g^n(\bot)].\end{aligned}$$

  Then it is to show $[\![M]\!]^\zeta\eta[x := \langle\tau\rangle\zeta(f^n(\bot))] \sim_\tau^{\mathrm{R},\zeta} [\![M^-]\!]\eta'[x := g^n(\bot)]$.

  By I.H. we get $f^n(\bot) \sim_\tau^{\mathrm{R},\zeta} g^n(\bot) \Rightarrow [\![M]\!]^\zeta\eta[x := \langle\tau\rangle\zeta(f^n(\bot))] \sim_\tau^{\mathrm{R},\zeta} [\![M^-]\!]\eta'[x := g^n(\bot)]$.

- $\sim_\tau^{\mathrm{R},\zeta}$ is admissible.

  It follows by Lemma 21.

Let $\eta \in [\![\Gamma]\!]\zeta$ be $\Gamma(x_i) = \rho_i \wedge \eta(x_i) \in [\![\rho_i]\!]\zeta$.

The following theorem states that if from a context Γ we can derive a **LoR** term M with regular type ρ, and for every variable x_i in the context Γ, $\eta(x_i)$ is an element of $[\![\rho_i]\!]\zeta$, then the value of M and its corresponding untyped term M^- coincide up to the finitary projection $\langle\rho\rangle\zeta$.

Theorem 24 (Coincidence) *If $\Gamma \vdash^{\mathrm{r}} M : \rho$ and $\eta \in [\![\Gamma]\!]\zeta$, then $[\![M]\!]^\zeta\eta = \langle\rho\rangle\zeta([\![M^-]\!]\eta)$.*

Proof. By Lemma 23, we have $[\![M]\!]^\zeta\eta \sim_\rho^{\mathrm{R},\zeta} [\![M^-]\!]\eta$. By Lemma 22(1), we get $\langle\rho\rangle\zeta([\![M]\!]^\zeta\eta) = \langle\rho\rangle\zeta([\![M^-]\!]\eta)$.

Then by Soundness Theorem 12 and the definition of $\langle\rho\rangle\zeta(\mathrm{D})$, we have $[\![M]\!]^\zeta\eta = \langle\rho\rangle\zeta([\![M]\!]^\zeta\eta)$.

Thus, $[\![M]\!]^\zeta\eta = \langle\rho\rangle\zeta([\![M^-]\!]\eta)$.

5 Conclusion

We have studied a domain-theoretic semantics for Church-style system **LoR** of typed lambda terms and proved that, when restricted to regular types, it is closely related to its untyped counterpart. The proof uses hybrid logical relations. The reason for studying this domain-theoretic semantics is that it allows for very simply and elegant proofs of computational adequacy, and hence the correctness of program extraction.

As future work we intend to investigate whether the requirement of regularity is indeed necessary for our result to hold. Furthermore, we plan to compare the Church-style system with a corresponding Curry-style system.

References

1. Abramsky, S., and Jung, A.: Domain theory. In Handbook of Logic in Computer Science (1994), S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds., Oxford University Press, 1-168.
2. Amadio, R. M., Bruce, K. B., and Longo, G.: The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In First Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1986) 122–130.
3. Amadio, R. M., and Curien, P.-L.: Domains and Lambda-Calculi, vol. 46 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
4. Berger, U.: ¿From coinductive proofs to exact real arithmetic. LNCS **5771** (2009) 132–146.
5. Berger, U.: Realisability for Induction and Coinduction with Applications to Constructive Analysis. Jour. Universal Comput. Sci. **16(18)** (2010), 2535–2555.
6. Berger, U., Hou, T.: Coinduction for Exact Real Number Computation. Theory Comput. Sys. **43** (2008) 394–409.
7. Berger, U., Seisenberger, M.: Proofs, programs, processes. In (Ferreira, F., Löwe, B., Mayordomo, E., Gomes, L. M., eds.) Programs, Proofs, Processes, CiE 2010, Ponta Delgada, Azores, Portugal LNCS **6158** (2010) 39–48.
8. Berger, U.: ¿From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Comput. Sci. **7(1)** (2011) 1–24.
9. Berger, U. Seisenberger, M.: Program extraction via typed realisability for induction and coinduction. In (Schindler, R., ed.) Ways of Proof Theory. Ontos Series in Mathematical Logic, Ontos Verlag, Frankfurt. (2011) 157-182.
10. Pitts, A. M. : Relational properties of recursively defined domains. In Proceedings of the Eighth Annual IEEE Symp. on Logic in Computer Science, LICS **1993** (1993), M. Vardi, Ed., IEEE Computer Society Press, 86-97.
11. Stoltenberg-Hansen, V., Lindström, I., and Griffor, E.R.: Mathematical Theory of Domains, vol. 22 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

A probabilistic monad with nondeterminism for coalgebraic trace semantics

Tetsuya Sato

Research Institute for Mathematical Sciences, Kyoto University, Japan
satoutet@kurims.kyoto-u.ac.jp

Abstract. We construct a set monad which captures probabilistic systems with nondeterminism based on the indexed valuation monad by Varacca et al. [1]. By simple translation, a probabilistic automaton by Segala [2] is captured as a coalgebra in the Kleisli category of the monad. The finite trace semantics of a probabilistic automaton is captured as the largest coalgebra morphism to a suitable weakly final coalgebra in the Kleisli category of the monad.

1 Introduction

1.1 Overview

In recent studies on trace semantics of coalgebras, it has been shown that trace semantics can be captured as coalgebra morphisms in Kleisli categories of monads [3]. The key idea is to regard the functor TF of coalgebra $c\colon X \to TFX$ as a composite of two parts: a monad T and a transition type functor F. For example, in the case of a nondeterministic LTS $c\colon X \to \mathcal{P}(1 + A \times X)$, the functor $\mathcal{P}(1 + A \times X)$ can be decomposed into a powerset monad $T = \mathcal{P}$ and a functor $F = 1 + (A \times id)$. Intuitively, the unit of the monad stands for *zero-times* transition and the multiplication stands for *merging* multiple transitions.

This perspective works well in the studies of trace semantics. The study of trace semantics by Hasuo et al. [3] shows that the finite trace semantics of nondeterministic LTSs is captured by a final coalgebra in Kleisli category of $\mathcal{P}$. By replacing $\mathcal{P}$ by a subdistribution monad $\mathcal{D}$, they also obtain the finite trace semantics of probabilistic LTSs. Jacobs' work [4] shows that an infinite trace semantics of nondeterministic LTSs is captured as the greatest coalgebra morphism to a suitable weakly final coalgebra.

It is natural to consider a trace semantics of systems whose transitions are both probabilistic and nondeterministic, like probabilistic automata [2]. One might expect that they are captured as $\mathcal{PD}F$-coalgebra, where F is some polynomial functor (e.g. $1 + (A \times id)$ or $A \times id \times id$). However, $\mathcal{PD}$ is not a monad in natural way (for detail, see [1]). The monad that captures a probabilistic choice with nondeterminism is needed. In this work, we construct such a monad $\mathcal{P}_{+}ID$ based on *indexed valuations* by Varacca et al. [1]. We translate probabilistic automata to coalgebras in the Kleisli category, and construct trace semantics as coalgebra morphism. More examples are found in [5].

1.2 Notations

We introduce the following notations:

- inverse image: $f^{-1}[A] = \{x \in X | f(x) \in A\}$ and $f^{-1}(y) = f^{-1}[\{y\}]$.
- coproduct: $[f_i]_{i\in I} \colon \coprod_{i\in I} X_i \to Y$ is the unique arrow such that $[f_i]_{i\in I} \circ \kappa_i(j) = f_i(j)$ where, $\kappa_i \colon X_i \to \coprod_{i\in I} X_i$ is a injection.
- $X \subseteq_\omega Y$ if and only if $X \subseteq Y$ and X is finite set.
- $X \ni x \mapsto f(x)$ stands for a function $f \colon X \to Y$. The range Y is determined by the context where it is used.

The infinite sum of nonnegative numbers is defined by the least upper bound of the set of finite partial sums: $\sum_{i\in I} a_i = \sup_{J\subseteq_\omega I} \sum_{j\in J} a_j$.

2 Probabilistic automata

2.1 Structures of probabilistic automata

A *probabilistic automaton* [2] is defined to be a tuple $(X, A, \textbf{start}, \textbf{trans})$

- X: states, A: labels/actions, $\textbf{start} \subseteq X$: initial states
- $\textbf{trans} \colon X \to \mathcal{PD}(\{\surd\} + (A \times X))$: transition function

where, $\mathcal{P}$ is the powerset functor and $\mathcal{D}$ is the subdistribution functor that is defined by $\mathcal{D}X = \{d \colon X \to [0,1] | \sum_{x\in X} d(x) \le 1\}$ and $\mathcal{D}f(y) = \sum_{x\in f^{-1}(y)} f(x)$.

Transitions of a probabilistic automaton are defined as alternating steps of a nondeterministic transition and a probabilistic transition. For each state x, a distribution $d \in \textbf{trans}(x)$ is chosen. For each distribution d, termination $\surd$ or a pair (a, y) of an output character a and a next state y is selected stochastically.

Note that transitions of a probabilistic automaton are also *nondeterministic.*

We sometimes consider an automaton such that $\sum_{\xi\in(\{\surd\}+(A\times X))} d(\xi) < 1$ for some $d \in \textbf{trans}(x)$. Probabilistic transitions can select not to move stochastically in this case.

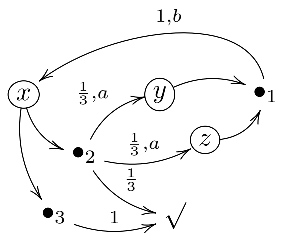
2.2 The finite trace of probabilistic automata

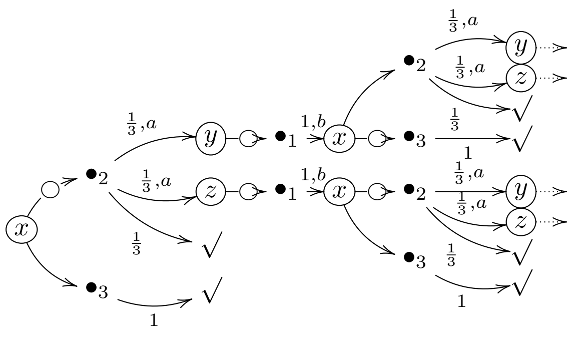
We sketch the finite trace semantics of probabilistic automata.

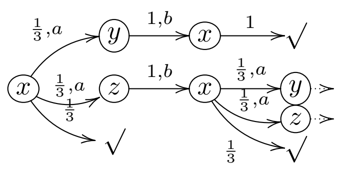
1. First, we draw a probabilistic automaton as a graph. Nodes of the graph are states in $x \in X$, distributions $d \in \textbf{trans}(x)$ and terminations $\surd$. Note that if $d = 0$, there is no transition *from* d.
2. Consider an initial state x. By unfolding transitions, we obtain the *transition tree* starting from x.
   Transitions of the probabilistic automaton is regarded as a two-player alternating game on the transition tree. The first player chooses a nondeterministic transition and the second player selects a transition stochastically. When there is no branch to go or the second player select not to move stochastically, the game is stopped.

We call the first player *scheduler* and the second player *adversary.* When a termination $\surd$ is selected and the game is terminated, Scheduler wins. Otherwise, Adversary wins.
We call a strategy of Scheduler *a scheduler* again. A scheduler is a function that is defined on position of the tree which stands for states and returns choices of distributions. Note that schedulers are not always history-free.
3. We fix a scheduler f. By pruning branches that are not chosen by the scheduler and composing one-step transitions, we obtain a probabilistic automaton *without* nondeterminism.
We call the probabilistic automaton *execution fragment* of f.
4. By making finite sequences of labels correspond to its probability, we define finite trace of f.
We denote the finite trace of f by $\mathbf{tdistr}(f)$.
5. The finite trace semantics of M is defined by collecting the finite traces of M for all scheduler: $\mathsf{Trace}^M(x) = \{\mathbf{tdistr}(f) | f \text{ is a scheduler starting from} x\}$.

The following figure is an example of above construction $1 \to 2 \to 3 \to 4$ for an automaton M and a scheduler f. The structure of M is drawn in 1. of the following figure; x, y and z are states and $\bullet_1$, $\bullet_2$ and $\bullet_3$ stand for distributions. We draw the choices of the scheduler f by $\bigcirc$:

1. 

2. 

3. 

4. $$\begin{cases} \langle\rangle & \mapsto \frac{1}{3} \\ ab & \mapsto \frac{1}{3} + \frac{1}{9} \\ abab & \mapsto \cdots \\ \cdots & \end{cases}$$

3 A probabilistic-nondeterministic monad

3.1 Indexed valuation

Definition 1 *An* indexed valuation *[1] on a set X is a pair* $(\mathsf{Ind}, \mathsf{v})$ *of two functions:*

– *a* indexing function $\mathsf{Ind} \colon I \to X$

– *a* valuation function $\mathsf{v}\colon I \to [0,\infty]$.

where I is some index set.

We also denote $(\mathsf{Ind},\mathsf{v})$ by $(x_i,p_i)_{i\in I}$ with $x_i = \mathsf{Ind}(i)$ and $p_i = \mathsf{v}(i)$. If functions are constant, we can drop the subscript. For example, $(x,p)_I$ stands for $x_i = x$ and $p_i = p$ for all $i \in I$.

We define the following notations:

– *the* support *:* $\mathrm{Spt}(\mathsf{v}) = \{i \in I | p_i \neq 0\}$
– *the* total *:* $\sum(\mathsf{v}) = \sum_{i\in I} \mathsf{v}(i)$
– $\bigoplus$ *operation :* $\bigoplus_{j\in J}(\mathsf{Ind}^j,\mathsf{v}^j) = ([\mathsf{Ind}^j]_{j\in J}, [\mathsf{v}^j]_{j\in J})$.

A zero element *is an indexed valuation $(\mathsf{Ind},\mathsf{v})$ such that $\mathsf{v} = 0$ or $I = \emptyset$.*

We introduce an equivalence relation $\sim$ on indexed valuations [1] and a partial order $\sqsubseteq$ on indexed valuations.

Definition 2 *An equivalence relation $\sim$ is defined as follows: $(\mathsf{Ind},\mathsf{v}) \sim (\mathsf{Ind}',\mathsf{v}')$ if there is an* bijection $h\colon \mathrm{Spt}(\mathsf{v}) \to \mathrm{Spt}(\mathsf{v}')$ *such that $(\mathsf{Ind}' \circ f(i), \mathsf{v}' \circ f(i)) = (\mathsf{Ind}(i), \mathsf{v}(i))$ for any $i \in \mathrm{Spt}(\mathsf{v})$.*

We remark that an equivalent class of zero elements is unique with respect to $\sim$. We denote the equivalent class of zero elements by $\underline{0}$. We identify indexed valuations with respect to $\sim$. Equivalent classes with respect to $\sim$ are called indexed valuations (on X) again.

Definition 3 *A partial order $\sqsubseteq$ is defined as follows: $(\mathsf{Ind},\mathsf{v}) \sqsubseteq (\mathsf{Ind}',\mathsf{v}')$ if there is an* injection $h\colon \mathrm{Spt}(\mathsf{v}) \to \mathrm{Spt}(\mathsf{v}')$ *such that $(\mathsf{Ind}' \circ h(i), \mathsf{v}' \circ h(i)) = (\mathsf{Ind}(i), \mathsf{v}(i))$ for any $i \in \mathrm{Spt}(\mathsf{v})$.*

Lemma 1 *$\sqsubseteq$ is indeed a partial order.*

Definition 4 *Fix a cardinal number α. We define $\mathit{IV}_\alpha(X)$ by*

$$\mathit{IV}_\alpha(X) = \left\{(\mathsf{Ind}\colon I \to X, \mathsf{v}\colon I \to [0,\infty]) \middle| |I| \leq \alpha\right\} / \sim_X .$$

It is easy to realise that $\mathit{IV}_\alpha(X)$ is a set. Note that if $\alpha < \beta$, $\mathit{IV}_\alpha(X) \subsetneq \mathit{IV}_\beta(X)$.

3.2 Indexed distribution

Lemma 2 *For each indexed valuation $(\mathsf{Ind},\mathsf{v})$, the total $\sum \mathsf{v}$ is well-defined.*

We then define the set of *indexed distributions*.

Definition 5 *Fix a cardinal number α. We define $\mathit{ID}_\alpha(X)$ by*

$$\mathit{ID}_\alpha(X) = \left\{(\mathsf{Ind},\mathsf{v}) \in \mathit{IV}_\alpha(X) \middle| \textstyle\sum \mathsf{v} \leq 1\right\}.$$

Remark 1 *Each indexed distribution has* finite *or* countable *support because given an indexed distribution $(\mathsf{Ind},\mathsf{v})$, $\mathsf{v}^{-1}((\frac{1}{n+1}, \frac{1}{n}]) \subseteq \mathrm{Spt}(\mathsf{v})$ is finite set for each natural number n. By definition of $\sim$, if $\aleph_0 \leq \beta$, $\mathit{ID}_{\aleph_0}(X) = \mathit{ID}_\beta(X)$.*

Thus, it suffices to fix $\mathit{ID}(X) = \mathit{ID}_{\aleph_0}(X)$.

3.3 An ω-CPO structure on the set of indexed distributions

We then define an partial order on $I\!D(X)$ by $\sqsubseteq_{I\!D(X)} = (\sqsubseteq \cap (I\!D(X) \times I\!D(X)))$. We will use ω-completeness of $\sqsubseteq$ in section 4.

Henceforth, we denote $\sqsubseteq_{I\!D(X)}$ by $\sqsubseteq$.

Proposition 1 *For any set X, $(I\!D(X), \sqsubseteq)$ is an ω-complete partial order with the least element $\underline{0}$.*

Proof. It is easy to see that the zero element $\underline{0}$ is the least element in $(I\!D(X), \sqsubseteq)$.

It suffices to prove the ω-completeness. We split the proof into three steps.
Step 1. Consider an ω-chain $(\mathsf{Ind}^0, \mathsf{v}^0) \sqsubseteq (\mathsf{Ind}^1, \mathsf{v}^1) \sqsubseteq \cdots \sqsubseteq (\mathsf{Ind}^n, \mathsf{v}^n) \sqsubseteq \cdots$.

Fix n and $(x,p) \in X \times (0, \infty]$. Since $\sum \mathsf{v}^n \leq 1$, $(\mathsf{Ind}^n, \mathsf{v}^n)^{-1}(x,p)$ is a finite set. Thus, $m_n(x,p) = \left|(\mathsf{Ind}^n, \mathsf{v}^n)^{-1}(x,p)\right|$ is a natural number. Note that it is easy to prove well-definedness of $m_n(x,p)$. Since $\{(\mathsf{Ind}^n, \mathsf{v}^n)\}_n$ is an ω-chain, $m_n(x,p) \leq m_{n+1}(x,p)$. Also, it is obvious to see $m_n(x,p) \leq \frac{1}{p}$.

Therefore, $\sup_n m_n(x,p)$ is indeed a natural number for any $(x,p) \in X \times (0, \infty]$. We define the following notations:

$$\begin{aligned} m(x,p) &= \sup_n m_n(x,p) = \lim_{n \to \infty} m_n(x,p) \\ J(x,p) &= \left\{m\text{: natural number } \middle| 0 < m \leq m(x,p)\right\} \end{aligned}$$

Step 2. We define an indexed valuation $(\mathsf{Ind}, \mathsf{v}) = \bigoplus_{(x_i,p_i) \in X \times (0,\infty]} (x,p)_{J(x,p)}$. We prove $\sum \mathsf{v} \leq 1$.

First, by the definition of infinite sum,

$$\sum \mathsf{v} = \sum_{(x,p) \in X \times (0,\infty]} \sum_{i \in J(x,p)} \mathsf{v}(i) = \sum_{(x,p) \in X \times (0,\infty]} p \cdot m(x,p) = \sum_{(x,p) \in X \times (0,\infty]} \lim_{n \to \infty} p \cdot m_n(x,p).$$

Next, by Lebesgue monotone convergence theorem,

$$\sum_{(x,p) \in X \times (0,\infty]} \lim_{n \to \infty} p \cdot m_n(x,p) = \lim_{n \to \infty} \sum_{(x,p) \in X \times (0,\infty]} p \cdot m_n(x,p) = \lim_{n \to \infty} \sum \mathsf{v}^n \leq 1.$$

Step 3. We prove that $(\mathsf{Ind}, \mathsf{v})$ is the least upper bound of $\{(\mathsf{Ind}^n, \mathsf{v}^n)\}_n$. By the definition of $(\mathsf{Ind}, \mathsf{v})$, it is obviously an upper bound.

Consider another upper bound $(\mathsf{Ind}', \mathsf{v}')$ of $\{(\mathsf{Ind}^n, \mathsf{v}^n)\}_n$. Since $(\mathsf{Ind}', \mathsf{v}')$ is an upper bound of $\{(\mathsf{Ind}^n, \mathsf{v}^n)\}_n$, $m_n(x,p) \leq \left|(\mathsf{Ind}', \mathsf{v}')^{-1}(x,p)\right|$ for any n. Thus, $m(x,p) \leq \left|(\mathsf{Ind}', \mathsf{v}')^{-1}(x,p)\right|$. Hence, there is an injection $h_{(x,p)} \colon J(x,p) \to (\mathsf{Ind}', \mathsf{v}')^{-1}(x,p)$ such that $(\mathsf{Ind}(i), \mathsf{v}(i)) = (\mathsf{Ind}'(h_{(x,p)}(i)), \mathsf{v}'(h_{(x,p)}(i)))$ for any $i \in J(x,p)$.

Since $\mathrm{Spt}(\mathsf{v}) = \coprod_{(x,p) \in X \times (0,\infty]} J(x,p)$, we obtain an injection $f \colon \mathrm{Spt}(\mathsf{v}) \to \mathrm{Spt}(\mathsf{v}')$ such that $(\mathsf{Ind}(i), \mathsf{v}(i)) = (\mathsf{Ind}' \circ f(i), \mathsf{v}' \circ f(i))$ for any $i \in \mathrm{Spt}(\mathsf{v})$. Therefore, $(\mathsf{Ind}, \mathsf{v}) \sqsubseteq (\mathsf{Ind}', \mathsf{v}')$. □

3.4 The indexed distribution monad

Based on the indexed valuation monad by Varacca et al. [1], we define the monad $I\!D$captures that probabilistic transition with nondeterminism.

Lemma 3 *$I\!D(X)$ extends to a functor on* **Set** *defined by*

$$I\!D(f)((\mathsf{Ind},\mathsf{v})) = (f \circ \mathsf{Ind},\mathsf{v})$$

Lemma 4 *Functor $I\!D$ extends to a monad $(I\!D, \eta^{I\!D}, \mu^{I\!D})$.*

$$\eta^{I\!D}_X(x) = (x,1)_{\{*\}}$$
$$\mu^{I\!D}_X(((\mathsf{Ind}^i_j,\mathsf{v}^i_j)_{j\in J_i},p_i)_{i\in I}) = \bigoplus_{i\in I}(\mathsf{Ind}_j, p_i\cdot \mathsf{v}_j)_{j\in J_i}$$

Lemma 5 *The monad $I\!D$ is commutative on $(\mathbf{Set},\times,\{*\})$. The tensor strength $st^{I\!D}_{A,B}\colon A\times I\!DB \to I\!D(A\times B)$ is defined by $st^{I\!D}_{A,B}(a,(x_i,p_i)_{i\in I}) = ((a,x_i),p_i)_{i\in I}$. The* double strength *$dst^{I\!D}_{A,B}\colon I\!DA\times I\!DB \to I\!D(A\times B)$ is defined by $dst^{I\!D}_{A,B} = \mu\circ I\!D(st'^{I\!D}_{A,I\!DB})\circ st^{I\!D}_{I\!DA,B}$.*

Proof (sketch). It is easy to prove the commutativity: $\mu_{(A\times B)}\circ I\!D(st'^{I\!D}_{A,B})\circ st^{I\!D}_{I\!DA,B} = \mu_{(A\times B)}\circ I\!D(st^{I\!D}_{I\!DA,B})\circ st'^{I\!D}_{A,I\!DB}$ where $\gamma_{A,B}\to A\times B\Rightarrow B\times A$ with $\gamma_{A,B}(a,b)=(b,a)$ and $st'^{I\!D}_{A,B} = I\!D(\gamma_{B,A})\circ st^{I\!D}_{B,A}\circ\gamma_{I\!DA,B}$. It is straightforward. □

3.5 Composition with a nondeterministic monad

The nonempty powerset monad $(\mathcal{P}_+,\eta^{\mathcal{P}_+},\mu^{\mathcal{P}_+})$ is defined by

- $\mathcal{P}_+(X) = \{Y\subseteq X | Y\neq\emptyset\}$
- $\eta^{\mathcal{P}_+}(x)=\{x\}$ and $\mu^{\mathcal{P}_+}(\Xi) = \bigcup\{A|A\in\Xi\}$

Note that $\mathcal{P}_+(\emptyset)=\emptyset$. It is easy to prove $\mathcal{P}_+$ is commutative; the double strength $dst^{\mathcal{P}_+}_{A,B}\colon \mathcal{P}_+A\times\mathcal{P}_+B\to\mathcal{P}_+(A\times B)$ is defined by $dst^{\mathcal{P}_+}(X,Y)=X\times Y$.

The indexed valuation monad is composed with a *nonempty powerset* monad using a distributive law between monads [1]. We now compose monads $I\!D$ and $\mathcal{P}_+$ using a *distributive law* defined in the following lemma:

Lemma 6 *The following gives a distributive law $d\colon I\!D\mathcal{P}_+\Rightarrow\mathcal{P}_+I\!D$:*

$$d_X((\Xi_i,p_i)_{i\in I}) = \{(h_i,p_i)_{i\in I} | h_i\in\Xi_i\}$$

Lemma 7 *The composite monad $\mathcal{P}_+I\!D$ is commutative.*

Proof (sketch). It is easy to prove that the distributive law $d\colon I\!D\mathcal{P}_+\Rightarrow\mathcal{P}_+I\!D$ which is defined in lemma 6 is *commutative* [6], that is, $\mathcal{P}_+(st'^{I\!D}_{A,B})\circ st^{\mathcal{P}_+}_{I\!DA,B} = d_{(A\times B)}\circ I\!D(st^{\mathcal{P}_+}_{A,B})\circ st'^{I\!D}_{A,\mathcal{P}_+B}$. Therefore, the composite monad $\mathcal{P}_+I\!D$ is commutative(For detail, see [6]). □

An endofunctor F has a lifting $\overline{F}$ in the Kleisli category $\mathbf{Set}_T$ if and only if there is a distributive law $\lambda^F : FT \Rightarrow TF$. For detail, see [7].

By the next lemma, any set functor F which is defined by BNF $F ::= id \mid A(\text{const.}) \mid F \times F \mid \coprod_i F_i$ has a lifting $\overline{F}$ in the Kleisli category $\mathbf{Set}_{\mathcal{P}_+ I\!D}$ of commutative monad.

Lemma 8 ([3]) *Given a commutative monad (T, η, μ) on $(\mathbf{Set}, \times, \{*\})$ and a set functor F which is defined by BNF $F ::= id \mid A(\text{const.}) \mid F \times F \mid \coprod_i F_i$, the canonical distributive law $\lambda \colon FT \Rightarrow TF$ is inductively defined as follows:*

- *If $F = id$, the λ is the identity natural transformation $id_T \colon T \Rightarrow T$.*
- *If $F = A$, $X \mapsto A$, the λ is the unit $\eta_A \colon A \Rightarrow TA$.*
- *If $F = F_1 \times F_2$, distributive laws $\lambda^{F_j} : F_j T \Rightarrow TF_j$ for $j \in \{1, 2\}$ are defined. We then form the composite $\lambda^F_X = dst^T_{F_1X, F_2X} \circ (\lambda^{F_1}_X \times \lambda^{F_2}_X)$.*
- *If $\coprod_{j\in J} F_j$, distributive laws $\lambda^{F_j} : F_j T \Rightarrow TF_j$ for $j \in J$ are defined. We then form the composite $\lambda^F_X = [T(\kappa_j) \circ \lambda^{F_j}]_{j\in J}$.*

4 Coalgebraic trace semantics

We this section, we define schedulers on probabilistic automata as an $\overline{F}$-coalgebra formally in $\mathbf{Set}_{\mathcal{P}_+ I\!D}$, construct trace semantics in $\mathbf{Set}_{\mathcal{P}_+ I\!D}$ and see the trace semantics is a maximum coalgebra morphism under suitable order of morphisms.

First, we translate probabilistic automata into an $\overline{F}$-coalgebras. Since any probabilistic automata is a $\mathcal{P}\mathcal{D}(\{\surd\} + (A \times id))$-coalgebra, we fix $F = 1 + (A \times id)$.

Definition 6 *We translate each probabilistic automaton $c \colon X \to \mathcal{P}\mathcal{D}FX$ to the following automaton $c' \colon X \to \mathcal{P}_+\mathcal{D}FX$:*

$$c'(x) = \begin{cases} \{\underline{0}\} & \text{if } c(x) = \emptyset \\ c(x) & \text{if } c(x) \neq \emptyset \end{cases}.$$

Recalling section 2.2, this translation will not change the finite trace semantics. Each $f \in c'(x) \colon \mathcal{D}(X)$ is regarded as an indexed distribution that is equivalent to $(id_X, f) \in I\!D(X)$. Therefore we have an arrow $c'' \colon X \to \mathcal{P}_+ I\!DFX$.

Proposition 2 *For any $x \in X$, there is at least one representative of each element of $c''(x)$.*

Proof (sketch). First, define Δ_x to be $c(x)$. We have a representative of each $(\mathsf{Ind}^\delta, \mathsf{v}^\delta) \in c(x)$ that is defined as follows:

1. Provide indexed distributions $s(x, p, n) = \bigoplus_{k=0}^{n-1} (x, p)$.
2. For any (x, p), $m(x, p) = \sup \left\{ n \middle| s(x, p, n) \sqsubseteq (\mathsf{Ind}^\delta, \mathsf{v}^\delta) \right\}$ is a natural number.
3. We have $\bigoplus_{(x,p) \text{ such that } m(x,p) \neq 0} s(x, p, m(x, p)) \sim (\mathsf{Ind}^\delta, \mathsf{v}^\delta)$.

Taking care of $\bigoplus$ operation, we construct the representative directly. Note that there are at most countably many (x, p) such that $m(x, p) \neq 0$. □

Henceforth, we consider an $\overline{F}$-coalgebra c and fix its representative $c(x) = \left\{ (\mathsf{Ind}^\delta, \mathsf{v}^\delta) \middle| \delta \in \Delta_x \right\}$ for each $x \in X$ and $I_\delta = \mathrm{Dom}(\mathsf{Ind}^\delta, \mathsf{v}^\delta)$.

4.1 Schedulers of $\overline{F}$-coalgebras

To define a trace semantics, we have to define the notion of schedulers on $\overline{F}$-coalgebra. Transitions of a translated probabilistic automaton are defined as same as original probabilistic automata.

Scheduler and adversary. Recalling section 2.2, we define the notion of schedulers formally. Given an $\overline{F}$-coalgebra c and an initial state $x \in X$, a one-step transition may be regarded as a two-player alternating game between *scheduler* and *adversary*:

1. Scheduler chooses $(\mathsf{Ind}^{\delta}, \mathsf{v}^{\delta}) \in c(x)$. This is the same as choosing $\delta \in \Delta_x$.
2. Adversary selects $i \in \mathrm{Spt}(\mathsf{v})$ stochastically (with probability $\mathsf{v}(i)$). Note that adversary can select no transition i when the total $\sum \mathsf{v}$ is less than 1. When $\mathsf{Ind}(i) = \surd$, the transition will halt *successfully.* When $\mathsf{Ind}(i) = (a, x')$, the transition will continue, output an character a and the next state will be x'. Otherwise, the transition will halt *unsuccessfully.*

Scheduler wins if and only if the state of the game halts successfully. In simple words, we define a scheduler as a scheduler's strategy and we construct the finite trace map by the set of probabilistic distributions of output strings of winning plays that is determined by a scheduler.

We regard a finite-step play of the game as a sequence which can be written $(\delta_0, i_{\delta_0})(\delta_1, i_{\delta_1}) \ldots (\delta_{n-1}, i_{\delta_{n-1}})\delta_{x_n}$ or $(\delta_0, i_{\delta_0})(\delta_1, i_{\delta_1}) \ldots (\delta_n, i_{\delta_n})$. All plays stand for a position of the tree that obtained by unfolding transitions of c starting from x_0.

Definition 7 *Consider an $\overline{F}$-coalgebra c and an initial state $x_0 \in X$. A* play *of c starting from x_0 with length $n+1$ is a sequence $(\delta_0, i_0)(\delta_1, i_1) \cdots (\delta_{n-1}, i_{n-1})\delta_n$ or a sequence $(\delta_0, i_0)(\delta_1, i_1) \cdots (\delta_n, i_n)$, such that $\delta_k \in \Delta_{x_k}$, $i_k \in \mathrm{Spt}(\mathsf{v}^{\delta_k})$ and $\mathsf{Ind}^{\delta_k}(i_k) = (a_k, x_{k+1})$ for all $0 \leq k < n$. An empty sequence $(())$ is defined as a play with length 0. If $x \neq y$, we assume that $(())$ starting from x and $(())$ starting from y are different. We introduce the following notations:*

- *We denote the length of φ by $|\varphi|$.*
- *If $\varphi = (\delta_0, i_0)(\delta_1, i_1) \cdots (\delta_n, i_n)$,*
  we define $\mathsf{Trace}(\varphi) = a_0 a_1 \cdots a_n$ *and* $\mathsf{Prob}(\varphi) = 1 \cdot \mathsf{v}^{\delta_0}(i_0) \cdot \mathsf{v}^{\delta_1}(i_1) \cdots \mathsf{v}^{\delta_n}(i_n)$.
  Note that if $\varphi = (())$, $\mathsf{Trace}(\varphi)$ *is empty string and* $\mathsf{Prob}(\varphi) = 1$.
- *If $\varphi = (\delta_0, i_0)(\delta_1, i_1) \cdots (\delta_n, i_n)$ and $\mathsf{Ind}^{\delta_n}(i_n) = (a_n, x_{n+1})$,*
  we define the next state $\mathsf{Next}(\varphi) = x_{n+1}$.
  If $\varphi = (())$ starting from x, we define $\mathsf{Next}(\varphi) = x$.

Specifically, given a representative of c, each play φ of c is an element

$$\varphi \in (\coprod_{x \in X} \Delta_x \times \coprod_{x \in X} \coprod_{\delta \in \Delta_x} I_\delta)^* \times \coprod_{x \in X} \Delta_x + (\coprod_{x \in X} \Delta_x \times \coprod_{x \in X} \coprod_{\delta \in \Delta_x} I_\delta)^*$$

Definition 8 *We divide plays into the following three cases:*

- *A play φ in c starting from x* halts unsuccessfully *if $\varphi \in (\coprod_{x\in X} \Delta_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} I_\delta)^* \times \coprod_{x\in X} \Delta_x$.*
- *A play φ in c starting from x* halts successfully *if $\varphi \in (\coprod_{x\in X} \Delta_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} I_\delta)^*$, $|\varphi| > 0$ and $\mathsf{Ind}^{\delta_{|\varphi|}}(i_{|\varphi|}) = \surd$.*
- *A play φ in c starting from x* is extensible *if $\varphi \in (\coprod_{x\in X} \Delta_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} I_\delta)^*$, and $|\varphi| = 0$ or $|\varphi| > 0$ and $\mathsf{Ind}^{\delta_n}(i_n) = (a_n, x_{n+1}) = (a_{|\varphi|}, x_{|\varphi|+1})$ for some $a_{|\varphi|} \in A$ and $x_{|\varphi|+1} \in X$. In other words, the next state $\mathsf{Next}(\varphi)$ is defined.*

We now define the notion of schedulers formally. Note that we first fix a representative and define the notion of scheduler and trace semantics.

Definition 9 *A* scheduler *Q of c starting from x is a partial function*

$$Q\colon (\coprod_{x\in X} \Delta_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} I_\delta)^* \times \coprod_{x\in X} \Delta_x + (\coprod_{x\in X} \Delta_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} I_\delta)^* \rightharpoonup \coprod_{x\in X} \Delta_x$$

which is defined at least on $\{\varphi | \varphi$ is extensible and starting from $x\}$ and satisfies $Q(\varphi) \in \Delta_{\mathsf{Next}(\varphi)}$ for any φ that is extensible and and starting from x.

*Similarly, an n-*scheduler *Q_n is a partial function such that $Q_n(\varphi) \in \Delta_{\mathsf{Next}(\varphi)}$ and it is defined only on $\{\varphi | \varphi$, is extensible and starting from $x, |\varphi| \leq n\}$.*

We remark that Q is a scheduler if and only if $Q = \bigcup_n Q_n$ where Q_n is a n-scheduler for each n and $Q_n \subseteq Q_{n+1}$ as binary relation for each n.

Definition 10 *A play φ of c starting from x is* compatible with *Q if $\varphi = (\delta_0, i_0)(\delta_1, i_1)\cdots(\delta_{n-1}, i_{n-1})\delta_n$ or $\varphi = (\delta_0, i_0)(\delta_1, i_1)\cdots(\delta_n, i_n)$, $\delta_0 = Q(())$ and $\delta_{k+1} = Q((\delta_0, i_0)(\delta_1, i_1)\cdots(\delta_k, i_k))$ for any $0 \leq k < n$.*

We define $\Phi_Q = \{\varphi | \varphi \text{ is compatible with } Q\}$ for scheduler Q. Similarly, we define $\Phi_{Q_n} = \{\varphi | \varphi \text{ is compatible with } Q_n \text{ and } |\varphi| \leq n+1\}$ for n-scheduler Q_n.

4.2 Trace semantics as arrows

In this section, we define formally trace semantics of a $\overline{F}$-coalgebra as an arrow in the Kleisli category $\mathbf{Set}_{\mathcal{P}_+ ID}$. First, we define trace for a scheduler.

Definition 11 *If Q is a scheduler, the* trace of Q *is a indexed valuation $(\mathsf{Ind}^Q, \mathsf{v}^Q) = (\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\Phi_Q}$ that is defined by*

$$(\boldsymbol{a}^Q_\varphi, p^Q_\varphi) = \begin{cases} (\mathsf{Trace}(\varphi'), \mathsf{Prob}(\varphi)) & \varphi = \varphi'(\delta, i) \textit{ halts successfully} \\ ((), 0) & \textit{otherwise} \end{cases}$$

We also define the trace $(\mathsf{Ind}^{Q_n}, \mathsf{v}^{Q_n}) = (\boldsymbol{a}^{Q_n}_\varphi, p^{Q_n}_\varphi)_{\varphi\in\Phi_{Q_n}}$ of n-scheduler Q_n.

Proposition 3 *For each scheduler Q, $(\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\Phi_Q}$ is indeed an indexed distribution.*

Proof. Let $\Phi_Q^n = \{\varphi \in \Phi_Q \big| |\varphi| \le n\}$. By the ω-completeness of $I\!D(A^*)$, It suffices to prove that $\{(\boldsymbol{a}_\varphi^Q, p_\varphi^Q)_{\varphi \in \Phi_Q^n}\}_n$ is an ω-chain on $I\!D(A^*)$ with respect to $\sqsubseteq$.

First, we provide functions $q^m \colon \Phi_Q \to [0, \infty]$ that are defined by

$$q^m(\varphi) = \begin{cases} \mathsf{Prob}(\varphi) & |\varphi| = m \text{ and } \varphi \text{ is extensible} \\ 0 & \text{otherwise} \end{cases}$$

Next, we show $\sum_{\varphi \in \Phi_Q^n} p_\varphi^Q + q^n(\varphi) \le 1$ by induction on n.

When $n = 0$, $\sum_{\varphi \in \Phi_Q^0} p_\varphi^Q + q^0(\varphi) \le 1$ because $(p_\varphi^Q, p_\varphi^Q)_{\varphi \in \Phi_Q^0} = \underline{0}$. When $n = m + 1$, for any $\varphi \in \Phi_Q^{m+1} \setminus \Phi_Q^m$, there is an extensible play $\varphi' \in \Phi_Q^m \setminus \Phi_Q^{m-1}$ such that $\varphi = \varphi'(\delta_{m+1}, i_{m+1})$ or $\varphi = \varphi'\delta_{m+1}$. Hence, we have,

- If φ halts unsuccessfully, $q^{m+1}(\varphi) = 0$ and $p_\varphi^Q = 0$.
- If φ halts successfully, $q^{m+1}(\varphi) = 0$ and $p_\varphi^Q = q^m(\varphi') \cdot \mathsf{v}^{\delta_{m+1}}(i_{m+1})$.
- If φ is extensible, $p_\varphi^Q = 0$ and $q^{m+1}(\varphi) = q^m(\varphi') \cdot \mathsf{v}^{\delta_{m+1}}(i_{m+1})$.

By simple calculations, we obtain,

$$\sum_{\varphi \in \Phi_Q^{m+1} \setminus \Phi_Q^m} p_\varphi^Q + q^{m+1}(\varphi) \le \sum_{\varphi' \in \Phi_Q^m \setminus \Phi_Q^{m-1}} q^m(\varphi') = \sum_{\varphi' \in \Phi_Q^m} q^m(\varphi')$$

By the induction hypothesis, we obtain,

$$\sum_{\varphi \in \Phi_Q^{m+1}} p_\varphi^Q + q^{m+1}(\varphi) \le \sum_{\varphi \in \Phi_Q^m} p_\varphi^Q + \sum_{\varphi' \in \Phi_Q^m} q^m(\varphi') \le 1.$$

This completes the induction.

This gives $\sum_{\varphi \in \Phi_Q^n} p_\varphi^Q \le 1$ for any n. It is obvious that $\{(\boldsymbol{a}_\varphi^Q, p_\varphi^Q)_{\varphi \in \Phi_Q^n}\}_n$ is an ω-chain whose least upper bound is $(\boldsymbol{a}_\varphi^Q, p_\varphi^Q)_{\varphi \in \Phi_Q}$. This completes the proof. $\square$

We now define trace semantics as an arrow in the Kleisli category $\mathbf{Set}_{\mathcal{P}_+ I\!D}$.

Definition 12 *The* trace semantics *of $\overline{F}$-coalgebra c is defined by*

$$\mathsf{Trace}^c(x) = \left\{(\mathsf{Ind}^Q, \mathsf{v}^Q) \big| Q \textit{ is a scheduler starting from } x\right\}.$$

Proposition 4 *The arrow $\mathsf{Trace}^c \colon X \to A^*$ in $\mathbf{Set}_{\mathcal{P}_+ I\!D}$ is well-defined.*

Informal outline of the proof. Consider two representatives $\mathbf{A}$ and $\mathbf{B}$ of c. Using bijections that gives $c(x)(\text{under } \mathbf{A}) = c(x)(\text{under } \mathbf{B})$ with respect to $\sim$, we construct a bijection that gives $\mathsf{Trace}^c(x)(\text{under } \mathbf{A}) = \mathsf{Trace}^c(x)(\text{under } \mathbf{B})$ with respect to $\sim$.

Proof. First, we denote two representatives $\mathbf{A}$ and $\mathbf{B}$ of c by

$$\mathbf{A}\!: c(x) = \left\{(\mathsf{Ind}^\delta, \mathsf{v}^\delta) \big| \delta \in \Delta_x\right\}, \mathbf{B}\!: c(x) = \left\{(\mathsf{Ind}'_\lambda, \mathsf{v}'_\lambda) \big| \lambda \in \Lambda_x\right\}$$

Assume each of $\{\Delta_x\}_x$ and $\{\Lambda_x\}_x$ is disjoint collection without loss of generality. For any $x \in X$, there is a bijection $f_x \colon \Delta_x \to \Lambda_x$ such that $(\mathsf{Ind}^\delta, \mathsf{v}^\delta) \sim (\mathsf{Ind}'_{f_x(\delta)}, \mathsf{v}'_{f_x(\delta)})$, that is, for each $\delta \in \Delta_x$, there is a bijection $h_{x,\delta} \colon \mathrm{Spt}(\mathsf{v}^\delta) \to \mathrm{Spt}(\mathsf{v}^{f_x(\delta)})$ such that $(\mathsf{Ind}^\delta(i), \mathsf{v}^\delta(i)) = (\mathsf{Ind}'_{f_x(\delta)}(h_{x,\delta}(i)), \mathsf{v}'_{f_x(\delta)})(h_{x,\delta}(i))$ for any $i \in \mathrm{Spt}(\mathsf{v}^\delta)$. We fix such a collection $(\{f_x\}_x, \{h_{x,\delta}\}_{x,\delta})$.

Now, we define the following construction Ω by

$$\Omega(\{g_x\}_x, \{k_{x,\delta}\}_{x,\delta}) = (\coprod_{x\in X} g_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} k_{x,\delta})^* + (\coprod_{x\in X} g_x \times \coprod_{x\in X} \coprod_{\delta\in\Delta_x} k_{x,\delta})^* \coprod_{x\in X} g_x.$$

It is straightforward to prove that functions $L = \Omega(\{f_x\}_x, \{h_{x,\delta}\}_{x,\delta})$ and $R = \Omega(\{{f_x}^{-1}\}_x, \{{h_{x,f_x^{-1}(\lambda)}}^{-1}\}_{x,\lambda})$ satisfy the following properties:

- $L^{-1} = R$. Hence, L and R are bijections.
- For any play φ of c under $\mathbf{A}$, $L(\varphi)$ is a play of c under $\mathbf{B}$. Furthermore, $|\varphi| = |L(\varphi)|$ and the following properties hold:
  - If φ halts unsuccessfully, so does $L(\varphi)$.
  - If $\varphi = \varphi'(\delta_{n+1}, i_{n+1})$ halts successfully, $L(\varphi)$ also halts successfully and $\mathsf{Prob}(\varphi) = \mathsf{Prob}(L(\varphi))$ and $\mathsf{Trace}(\varphi') = \mathsf{Trace}(L(\varphi'))$.
  - If φ is extensible, $L(\varphi)$ is also extensible with $\mathsf{Prob}(\varphi) = \mathsf{Prob}(L(\varphi))$ and $\mathsf{Trace}(\varphi) = \mathsf{Trace}(L(\varphi))$.

We define transformations g_R and g_L as follows: $g_R(Q) = (\coprod_{x\in X} f_x) \circ Q \circ R$ for any scheduler Q under $\mathbf{A}$ $g_L(Q') = (\coprod_{x\in X} {f_x}^{-1}) \circ Q' \circ L$ for any scheduler Q' under $\mathbf{B}$. It is easy to prove that functions g_R and g_L satisfy the following properties:

- ${g_R}^{-1} = g_L$. Thus, g_R and g_L are bijection.
- $g_R(Q)$ is a scheduler under $\mathbf{B}$ for any scheduler Q under $\mathbf{A}$.
- $g_L(Q')$ is a scheduler under $\mathbf{A}$ for any scheduler Q' under $\mathbf{B}$.
- $L(\varphi) \in \Phi_{g_R(Q)}$ for any $\varphi \in \Phi_Q$.
- $R(\varphi') \in \Phi_{g_L(Q')}$ for any $\varphi' \in \Phi_{Q'}$.

Finally, since $(\boldsymbol{a}^{g_R(Q)}_{L(\varphi)}, p^{g_R(Q)}_{L(\varphi)}) = (\boldsymbol{a}^Q_\varphi, p^Q_\varphi)$ for any play $\varphi \in \Phi_Q$ under $\mathbf{A}$, bijection L gives $(\mathsf{Ind}^Q, \mathsf{v}^Q) \sim (\mathsf{Ind}^{g_R(Q)}, \mathsf{v}^{g_R(Q)})$. Therefore, Trace^c is well-defined. □

4.3 Trace semantics as a coalgebra morphisms

Theorem 1 *For any $\overline{F}$-coalgebra $c \colon X \to \overline{F}X$ in Kleisli category $\mathbf{Set}_{\mathcal{P}_+ \mathcal{ID}}$, the trace semantics $\mathsf{Trace}^c \colon X \to A^*$ is an $\overline{F}$-coalgebra morphisms $c \to \eta^{\mathcal{P}_+ \mathcal{ID}} \circ [\mathsf{Nil}, \mathsf{Cons}]^{-1}$ in $\mathbf{Set}_{\mathcal{P}_+ \mathcal{ID}}$.*

Informal outline of the proof.

Proof. We take representatives c and Trace^c which are denoted by

$$c(x) = \{(x^\delta_i, p^\delta_i)_{i\in I_\delta} | \delta \in \Delta_x\}$$
$$\mathsf{Trace}^c(x) = \{(\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\Phi_Q} | Q \text{ is a scheduler starting from } x\}$$

By lemma 8, The distributive law $\lambda\colon F\mathcal{P}_{+}I\!D \Rightarrow \mathcal{P}_{+}I\!DF$ is defined is as follows:

$$\lambda_X(\surd) = (\surd, 1)_{\{*\}}$$
$$\lambda_X(a, \left\{(x_i^y, p_i^y)_{i\in I_y} \middle| y \in Y\right\}) = \left\{((a, x_i^y), p_i^y)_{i\in I_y} \middle| y \in Y\right\}$$

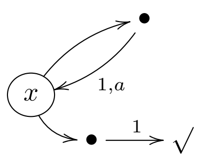
A scheduler Q can be decomposed into the first choice $\delta = Q(())$ and a collection of schedulers $\left\{Q_i \middle| (\delta, i) \text{ is extensible}\right\}$ that are defined by $Q_i(\varphi') = Q((\delta, i)\varphi')$. Each Q_i is a scheduler starting from $\mathsf{Next}((\delta, i))$.

We denote composition of two allows in $\mathbf{Set}_{\mathcal{P}_{+}I\!D}$ by $\bullet$. We then have,

$$\begin{aligned}
&(\eta^{\mathcal{P}_{+}I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]^{-1}) \bullet \mathsf{Trace}^c(x)\\
&= \mathcal{P}_{+}I\!D([\mathsf{Nil},\mathsf{Cons}]^{-1}) \circ \left\{(\boldsymbol{a}_\varphi^Q, p_\varphi^Q)_{\varphi\in\Phi_Q} \middle| Q = (\delta, \left\{Q_i \middle| (\delta, i)\text{: extensible}\right\}), \delta \in \Delta_x\right\}\\
&= \left\{\bigoplus\nolimits_{(i,\delta)\text{: ext.}}(\pi_1(x_i^\delta)\cdot \boldsymbol{a}_\varphi^{Q_i}, p_i^\delta \cdot p_\varphi^{Q_i})_{\varphi\in\Phi_{Q_i}} \middle| (\delta, \left\{Q_i \middle| (\delta, i)\text{: extensible}\right\}), \delta \in \Delta_x\right\}\\
&= \mu^{\mathcal{P}_{+}}_{I\!DFA^*} \circ {\mathcal{P}_{+}}^2(\mu^{I\!D}_{FA^*}) \circ \left\{\left\{(h_i^\delta, p_i^\delta)_{i\in I_\delta} \middle| (\delta, \left\{Q_i \middle| (\delta, i) \text{ is extensible}\right\})\right\} \middle| \delta \in \Delta_x\right\}\\
&\quad \text{where,}\\
&\quad h_i^\delta = \begin{cases} (\surd, i)_{\{*\}} & ((\delta, i) \text{ halts successfully })\\ ((\pi_1(x_i^\delta), \boldsymbol{a}_\varphi^{Q_i}), p_\varphi^{Q_i})_{\varphi\in\Phi^{Q_i}} & ((\delta, i) \text{ is extensible}) \end{cases}\\
&= \mu^{\mathcal{P}_{+}}_{I\!DFA^*} \circ {\mathcal{P}_{+}}^2(\mu^{I\!D}_{FA^*}) \circ \left\{d_{I\!DFA^*}(\lambda_{A^*} \circ F(\mathsf{Trace}^c) \circ x_i^\delta, p_i^\delta)_{i\in I_\delta} \middle| \delta \in \Delta_x\right\}\\
&= \overline{F}(\mathsf{Trace}^c) \bullet c(x)
\end{aligned}$$

Thus the trace semantics Trace^c is a coalgebra morphism. □

Failure of finality. The $\overline{F}$-coalgebra $\eta^{\mathcal{P}_{+}I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]^{-1}$ is *weakly final* however *not final* $\overline{F}$-coalgebra. For example, consider the following automaton:



The automaton $c\colon \{x\} \to \overline{F}\{x\}$ is defined by $c(x) = \left\{(\surd, 1)_{\{*\}}, ((a, x), 1)_{\{*\}}\right\}$

There are two different coalgebra morphisms $f_1, f_2\colon c \to \eta^T_{FA^*} \circ [Nil, Cons]^{-1}$,

$$f_1(x) = \left\{\underline{0}, (\langle\rangle, 1), (a, 1), (aa, 1), \ldots, (a^k, 1), \ldots\right\}$$
$$f_2(x) = \left\{(\langle\rangle, 1), (a, 1), (aa, 1), \ldots, (a^k, 1), \ldots\right\}$$

We see that $\eta^{\mathcal{P}_{+}I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]^{-1}$ is not a final $\overline{F}$-coalgebra.

The category $\mathbf{Set}_{\mathcal{P}_{+}I\!D}$ is not **CPPO**-enriched with respect to the pointwise set inclusion order because the composition of morphisms is not continuous. We also consider an order that is defined by lifting the order on indexed distributions $\sqsubseteq$. The order includes the set inclusion order and the composition of morphisms is also not continuous. We cannot apply the methods in [3].

However, since the trace map Trace^c captures all traces of scheduler, we obtain Trace^c is the greatest coalgebra morphism under the inclusion order.

For instance, $f_2(x) \subseteq f_1(x) = \mathsf{Trace}^c(x)$ holds in above example. We prove that such an inclusion holds in general case.

Theorem 2 *For any $\overline{F}$-coalgebra morphism $f : (c\colon X \to \overline{F}X) \to (\eta^{\mathcal{P}_+ I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]^{-1}\colon A^* \to \overline{F}A^*)$, $f(x) \subseteq \mathsf{Trace}^c(x)$ for any $x \in X$.*

Informal outline of the proof. Since f is an fixed point of $f \mapsto \eta^{\mathcal{P}_+ I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]) \bullet \overline{F}(f) \bullet c$, we decompose $(y_j, q_j)_{j\in J} \in f(x)$ into an element $\delta \in \varDelta$ and a collection of $(z_j, r_j)_{j\in J} \in f(x')$. Decomposing $(y_j, q_j)_{j\in J}$ n times, we construct an n-scheduler whose trace coincides with the part of $(y_j, q_j)_{j\in J}$ such that $|y_j| \leq n$. Finally, we construct a scheduler whose trace coincides $(y_j, q_j)_{j\in J}$.

Proof. We take a representative of c and Trace^c without loss of generality:

$$\begin{aligned} c(x) &= \left\{(x^\delta_i, p^\delta_i)_{i\in I_\delta} \middle| \delta \in \varDelta_x\right\} \\ \mathsf{Trace}^c(x) &= \left\{(\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\varPhi_Q} \middle| Q \text{ is a scheduler starting from } x\right\} \end{aligned}$$

We prove that for any $(y_j, q_j)_{j\in J} \in f(x)$, there is a scheduler Q starting from x such that $(\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\varPhi_Q} \sim (y_j, q_j)_{j\in J}$. We split the proof into three steps.
Step 1. Since $f(x)$ is an $\overline{F}$-coalgebra morphism, we have,

$$\begin{aligned} &f(x) \\ &= (\eta^{\mathcal{P}_+ I\!D} \circ [\mathsf{Nil},\mathsf{Cons}]) \bullet \overline{F}(f) \bullet c(x) \\ &= \mu^{\mathcal{P}_+}_{I\!D FA^*} \circ \mathcal{P}_+^2(\mu^{I\!D}_{FA^*}) \circ \\ &\quad \left\{\left\{((h^i_k, l^i_k)_{k\in K_i}, p^\delta_i)_{i\in I_\delta} \middle| (\delta, \left\{(y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} \middle| (\delta, i)\colon \text{extensible}\right\})\right\} \middle| \delta \in \varDelta_x\right\} \\ &\quad \text{where, } (y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} \in f(\mathsf{Next}((\delta, i))) \text{ for each extensible}(\delta, i)\text{, and} \\ &\quad (h^i_k, l^i_k)_{k\in K_i} = \begin{cases} ((), 1)_{\{*\}} & ((\delta, i) \text{ halts successfully }) \\ (\pi_1(x^\delta_i) \cdot y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} & ((\delta, i) \text{ is extensible}) \end{cases} \\ &= \left\{\textstyle\bigoplus_{i\in I_\delta}(h^i_k, p^\delta_i \cdot l^i_k)_{k\in K_i} \middle| \delta \in \varDelta_x, \left\{(y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} \middle| (\delta, i) \text{ is extensible}\right\}\right\} \end{aligned}$$

Therefore, any $(\mathsf{Ind}, \mathsf{v}) \in f(x)$ is decomposed into an element $\delta \in \varDelta_x$ and a collection$\left\{(y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} \middle| (\delta, i) \text{ is extensible}\right\}$ of indexed distributions such that $(y^{(\delta,i)}_j, q^{(\delta,i)}_j)_{j\in J_{(\delta,i)}} \in f(\mathsf{Next}((\delta, i)))$ and $(\mathsf{Ind}, \mathsf{v}) \sim \bigoplus_{i\in I_\delta}(h^i_k, p^\delta_i \cdot l^i_k)_{k\in K_i}$.
Step 2. We fix $(y_j, q_j)_{j\in J} \in f(x)$. Using the above decomposition, we define a 0-scheduler Q_0 starting from x such that Q_0 by $Q_0(()) = \delta$.

Let $J^{\leq n} = \left\{j \in J \middle| |y_j| \leq n\right\}$. By induction on n, we construct Q_n such that $(y_j, q_j)_{j\in J^{\leq n}} \sim (\boldsymbol{a}^{Q_n}_\varphi, p^{Q_n}_\varphi)_{\varphi\in\varPhi_{Q_n}}$.
When $n = 0$, since $\varPhi_{Q_0} = \left\{(\delta, i) \middle| \delta \in \varDelta_x, i \in \mathrm{Spt}(\mathsf{v}^\delta)\right\}$, we have,

$$(y_j, q_j)_{j\in J^{\leq 0}} \sim \bigoplus_{\substack{(\delta,i)\in\varPhi_{Q_0} \\ \text{halts successfully}}} ((), p^\delta_i)_{\{*\}} \sim (\boldsymbol{a}^{Q_0}_\varphi, p^{Q_0}_\varphi)_{\varphi\in\varPhi_{Q_0}}$$

When $n = m + 1$, we assume that we have a m-scheduler Q_m starting from x such that

$$(y_j, q_j)_{j\in J} \sim (\bigoplus_{\substack{\varphi\in\Phi_{Q_m}\\ \text{extensible}}} (\mathsf{Trace}(\varphi)y_j^{\varphi}, \mathsf{Prob}(\varphi)q_j^{\varphi})_{j\in J_\varphi}) \oplus (\boldsymbol{a}_{\varphi}^{Q_m}, p_{\varphi}^{Q_m})_{\varphi\in\Phi_{Q_m}}$$

where, $(y_j^{\varphi}, q_j^{\varphi})_{j\in J_\varphi} \in f(\mathsf{Next}(\varphi))$ for any extensible $\varphi \in \Phi_{Q_m}$.

First, we define $Q_{m+1}(\varphi) = Q_m(\varphi)$ for any extensible play φ such that $|\varphi| \le m$.

Next, consider an extensible play $\varphi \in \Phi_{Q_m}$ such that $|\varphi| = m + 1$. Since $(y_j^{\varphi}, q_j^{\varphi})_{j\in J_\varphi} \in f(\mathsf{Next}(\varphi))$,
we have $\delta \in \Delta_{\mathsf{Next}(\varphi)}$ and $\left\{(y_j^{\varphi(\delta,i)}, q_j^{\varphi(\delta,i)})_{j\in J_{\varphi(\delta,i)}} \middle| \varphi(\delta,i) \in \Phi_{Q_m}\right\}$ such that

$$\begin{aligned}&(y_j^{\varphi}, q_j^{\varphi})_{j\in J_\varphi}\\ &\sim \bigoplus_{i \text{ such that } p_i^{\delta}\neq 0} ((h_k^{\varphi i}, p_i^{\delta}\cdot l_k^i)_{k\in K_i}\end{aligned}$$

where, $(y_j^{\varphi(\delta,i)}, q_j^{\varphi(\delta,i)})_{j\in J_{\varphi(\delta,i)}} \in f(\mathsf{Next}(\varphi(\delta,i)))$ for each extensible(δ,i), and

$$(h_k^i, l_k^i)_{k\in K_i} = \begin{cases}((),1)_{\{*\}} & (\varphi(\delta,i) \text{ halts successfully })\\ (\pi_1(x_i^{\delta})\cdot y_j^{\varphi(\delta,i)}, q_j^{\varphi(\delta,i)})_{j\in J_{\varphi(\delta,i)}} & (\varphi(\delta,i) \text{ is extensible})\end{cases}$$

We then define $Q_{m+1}(\varphi) = \delta$. By the definition of h_k^i and l_k^i we obtain,

$$\begin{aligned}&(\mathsf{Trace}(\varphi)\cdot y_j^{\varphi}, \mathsf{Prob}(\varphi)\cdot q_j^{\varphi})_{j\in J_\varphi}\\ &\sim \bigoplus_{\substack{i \text{ such that } p_i^{\delta}\neq 0\\ \varphi(\delta,i):\text{extensible}}} (\mathsf{Trace}(\varphi(i,\delta))\cdot y_j^{\varphi(\delta,i)}, \mathsf{Prob}(\varphi(i,\delta))\cdot q_j^{\varphi(\delta,i)})_{j\in J_{\varphi(\delta,i)}}\\ &\oplus \bigoplus_{\substack{i \text{ such that } p_i^{\delta}\neq 0\\ \varphi(\delta,i):\text{successfully halts}}} (\mathsf{Trace}(\varphi), \mathsf{Prob}(\varphi(\delta,i)))_{\{*\}}\end{aligned}$$

We define $Q_{m+1}(\varphi)$ in this way when $\varphi \in \Phi_{Q_m}$ and φ is extensible. Otherwise, we define $Q_{m+1}(\varphi)$ arbitrary $\delta \in \Delta_{\mathsf{Next}(\varphi)}$. We then obtain a $m+1$-scheduler Q_{m+1} such that

$$(y_j, q_j)_{j\in J} \sim \left(\bigoplus_{\substack{\varphi(\delta,i)\in\Phi_{Q_{m+1}}\\ \text{extensible}}} (y_j^{\varphi(\delta,i)}, q_j^{\varphi(\delta,i)})_{j\in J_{\varphi(\delta,i)}}\right) \oplus (\boldsymbol{a}_{\varphi}^{Q_{m+1}}, p_{\varphi}^{Q_{m+1}})_{\varphi\in\Phi_{Q_{m+1}}}$$

It is easy to prove $(\boldsymbol{a}_{\varphi}^{Q_n}, p_{\varphi}^{Q_n})_{\varphi\in\Phi_{Q_n}} \sim (y_j, q_j)_{j\in J^{\le n}}$. Note that output strings with length n appear in the step n. This completes the induction.

Step 3. We define $Q = \bigcup_n Q_n$ since $Q_n \subseteq Q_{n+1}$ for any n under the set inclusion order. It is obvious that $\{(\boldsymbol{a}_{\varphi}^{Q_n}, p_{\varphi}^{Q_n})_{\varphi\in\Phi_{Q_n}}\}_n$ is an ω-chain such that $\sup_n(\boldsymbol{a}_{\varphi}^{Q_n}, p_{\varphi}^{Q_n})_{\varphi\in\Phi_{Q_n}} = (\boldsymbol{a}_{\varphi}^{Q}, p_{\varphi}^{Q})_{\varphi\in\Phi_Q}$.

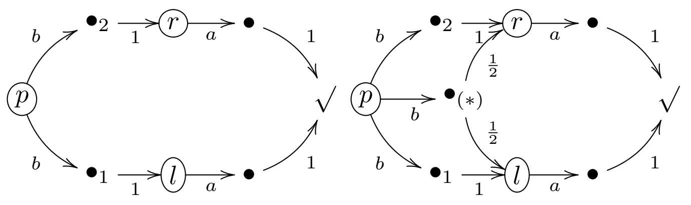
Since $\{(y_j, q_j)_{j\in J^{\leq n}}\}_n$ is an ω-chain such that $\sup_n (y_j, q_j)_{j\in J^{\leq n}} = (y_j, q_j)_{j\in J}$, we obtain $(\boldsymbol{a}^Q_\varphi, p^Q_\varphi)_{\varphi\in\Phi_Q} = (y_j, q_j)_{j\in J}$. This completes the proof. $\square$

By theorem 1, for any $\overline{F}$-coalgebra $c\colon X \to \overline{F}X$ in Kleisli category $\mathbf{Set}_{\mathcal{P}_+ ID}$, the trace semantics $\mathsf{Trace}^c\colon X \to A^*$ is *the greatest* coalgebra morphism $c \to \eta^{\mathcal{P}_+ ID} \circ [\mathsf{Nil}, \mathsf{Cons}]^{-1}$, when we focus on $\mathbf{Set}_{\mathcal{P}_+ ID}$ as **Poset**-enriched by pointwise order defined by set inclusion order (it is easy to prove the **Poset**-enrichment).

5 Conclusion

5.1 Future work

1. We expect to generalize this work to any polynomial functor F. In this paper, we have proved only a case of functor $F = 1 + (A \times id)$.
2. We expect to compare this work and Jacobs' work [8], which is based on the monad $\mathcal{CM}$ of convex subsets of distributions. We expect that our trace semantics is more detailed than trace semantics which is based on convex subsets.



For example, the above two automata have different trace semantics in our construction. On the other hand, the method that is based on convex subsets of distributions cannot distinguish them, because the distribution $\bullet_{(*)}$ is the *midpoint* of distributions $\bullet_1$ and $\bullet_2$ in set $\mathcal{CM}F\{x, y, z\}$.
3. We want to study whether there is the final $\overline{F}$-coalgebra or not. We guess there is no final $\overline{F}$-coalgebra. Perhaps our monad $\mathcal{P}_+ ID$ and trace semantics is too detailed to obtain a final coalgebra.

Acknowledgements. The author thanks Masahito Hasegawa and Shin-ya Katsumata for valuable comments, corrections and discussions and Ichiro Hasuo for suggestions about trace semantics.

References

1. Varacca, D., Winskel, G.: Distributing probability over non-determinism. Mathematical Structures in Computer Science **16**(1) (2006) 87–113
2. Segala, R.: A compositional trace-based semantics for probabilistic automata. In Lee, I., Smolka, S., eds.: CONCUR '95: Concurrency Theory. Volume 962 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1995) 234–248 10.1007/3-540-60218-6_17.

3. Hasuo, I., Jacobs, B., Sokolova, A.: Generic trace semantics via coinduction. Logical Methods in Computer Science **3**(4) (2007)
4. Bart, Jacobs: Trace semantics for coalgebras. Electronic Notes in Theoretical Computer Science **106**(0) (2004) 167 – 184 Proceedings of the Workshop on Coalgebraic Methods in Computer Science (CMCS).
5. Sato, T.: A nondeterministic probabilistic monad for deterministic schedulers on probabilistic automata. Master's thesis, Research Institute for Mathematical Sciences, Kyoto University (2011)
6. Wolff, H.: Commutative distributive laws. Journal of the Australian Mathematical Society (Series A) **19**(02) (1975) 180–195
7. Mulry, P.S.: Lifting theorems for kleisli categories. In Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D., eds.: Mathematical Foundations of Programming Semantics. Volume 802 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1994) 304–319 10.1007/3-540-58027-1_15.
8. Jacobs, B.: Coalgebraic trace semantics for combined possibilitistic and probabilistic systems. Electronic Notes in Theoretical Computer Science **203**(5) (2008) 131 – 152 Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

Coalgebraic Semantics of Recursion on Circular Data Structures

Baltasar Trancón y Widemann[1,2]

[1] Compiler Construction Group, Technical University of Berlin
[2] Ecological Modelling Group, University of Bayreuth
Dr. Hans-Frisch-Straße 1–3, 95445 Bayreuth, Germany
`Baltasar.Trancon@uni-bayreuth.de`

Abstract. In functional programming, the combination of recursive functions and circular data is traditionally regarded as ineffective, causing vicious circles and hence nontermination. We propose an alternative, coalgebraic perspective that encompasses everything from high-level semantics of data and functions to low-level implementations of evaluation strategies. Under this new perspective, circular data as represented by cycles of pointers among cells in memory can be processed with corecursive functions and predicates realized as search problems. The information required to escape the vicious circles is readily available, assuming a certain style of call-by-value conventions. The basic evaluation techniques, their current implementation and some example applications are described.

1 Introduction

1.1 Context and Motivation

This is a summary of the coalgebraic aspects of the author's PhD thesis [16], completed in 2006 at the Technical University of Berlin after preliminary experiments in 2002–2004 [13,14]. In that work, semantics and implementation techniques of functional programming are developed in a coalgebraic way for a scenario that has not been dealt with satisfactorily before: Referentially transparent recursive computations with strict, call-by-value semantics on circular data structures. The results are interesting primarily from a compiler construction perspective, because they entail fairly general and efficient coding schemes for circular computation problems, independent of a particular front-end language. But they are also worth a second thought from a coalgebraic, more theoretical perspective, because

1. it turns out that coalgebraic semantics of pointer graphs in computer memory is the adequate model of circular data for the purpose, and
2. many well-known algorithms have non-trivial and outright surprising generalizations to circular cases, simply by switching their definitions from (primitive) recursion to corecursion.

Only very basic concepts of universal coalgebra are needed: We consider a single, well-behaved functor as a universal signature of memory cells as the building blocks of finite data, whether circular or not. The final coalgebra of that functor is the semantics we assign to data. We use the technique of *coiteration* to define corecursive functions: A coalgebra that encodes a single step of the desired function gives rise to a unique homomorphism to the final coalgebra (anamorphism) that unfolds the computation. Compare the infamous lenses notation from [7].

1.2 Problem Definitions

Composite data as instances of recursive type definitions are usually organized as a collection of cells in memory, where substructures are referred to by pointers, and structurally recursive computation amounts to traversal of the pointer graph. For instance, the list $\ell_1 = [1, 2, 3, 5]$ might be represented by a cell containing the number 1 and a reference to a cell containing the number 2 and so forth, where the last cell contains the number 5 and a special reference signifying the empty list.

Structurally recursive computation works by computing partial result for substructures first, and finally combining them with local data of the root cell to a complete result. Hence the construction of data progresses bottom-up or, in the case of lists, right-to-left. In a strict, call-by-value setting, the effectiveness of the procedure depends crucially on the fact that structures are well-founded, that is, pointer paths are finite; either truly infinite or circular structures cause non-termination. The list $\ell_2 = [1, 2, 3, 5, 2, 3, 5, 2, 3, 5, \dots]$ or more precisely, using recursive equations and the *append* operator $\oplus$, $\ell_2 = [1] \oplus \ell_3$ where $\ell_3 = [2, 3, 5] \oplus \ell_3$, cannot be processed naïvely in the same way as ℓ_1.

Initial algebra models of data, as usual in strict purely functional programming, ensure that all structures are well-founded by construction. But there are severe downsides: On one hand, many interesting and useful data structures are inherently or accidentally circular. On the other hand, the technique of structural recursion (or while-loops for that matter) is also applied, and defeated accordingly, in impure contexts where cycles might arise. Immutable data produced by constructor calls is still safe in such a context. The most obvious way to obtain circular data is by destructive assignment, but we shall give an example of a cycle-inducing operation with purely functional semantics in section 2.2.

In the classical algorithmical literature, there is a dichotomy between two disparate worlds: One where referentially transparent, recursive computations are applied to data with initial algebra semantics, and hence without cycles; another where explicit pointer management and destructive assignment are applied to arbitrarily circular data with ad-hoc semantics defined by the CPU memory–pointer model, or by some no less ad-hoc object–reference model in more high-level languages. Our goal is to explore a region that has the best of both worlds: the ability do deal with circular data in a style that is as elegant and abstract as structural recursion.

2 Solutions

Our solution has been inspired by *The most unreliable technique in the world to compute* π [4], an article that appears hilarious at first glance, but actually reveals deep insights about the meaning of corecursive functions. It has prompted rational arithmetics as an example (see section 4.1), which has proven quite illuminating as to how preexisting non-circular algorithms can be extended to the circular case.

An obvious and well-established method of dealing with circular data is by lazy evaluation. In fact, laziness is sort of overkill for the job, because it supports potentially infinite data in general, subsuming circular data as the periodic case that neither receives nor requires special treatment. The severe downside of the lazy approach is that many problems that are decidable in principle on finite representations of circular data become at best semi-decidable on potentially infinite representations, see section 2.3 and 4 for a generic decision algorithm and its applications, respectively.

Additionally, we shall demonstrate in section 2.2 that classical implementation techniques for non-lazy evaluation already have precisely the kind of memory of the past that is needed to avoid repeating it, in the form of the call stack. By contrast, implementations of lazy evaluation based on *thunks* [3] desynchronize the creation of data, breaking the connection between (circular) calls; hence they do not normally have the information required to detect cycles.

2.1 Coalgebraic Model of Circular Data

Under mild assumptions about data sanity, the organization of memory into cells for the representation of signature-based data types, defined in terms of constructors, can be specified by the **Set** endofunctor

$$F(X) = \{0,1\}^* \times X^*$$

where the argument X is taken as a placeholder for references, that is, a cell is a sequence of bits and references. The main assumption here is that references are not confused with non-reference data. Note that F admits final coalgebras and preserves weak pullbacks, hence the usual machinery of universal coalgebra applies.

A state of memory can be specified by an F-coalgebra (A, α) where A is a finite set of live cell addresses and $\alpha : A \to F(A)$ is the dereferencing operation, provided that the following additional sanity assumptions hold:

1. A is closed under dereferencing: liveness is transitive.
2. α is total on A: no dangling or null pointers.

We may hence give an abstract meaning to a reference a in the context of a memory state (A, α) as the image of a under the unique homomorphism $\alpha! : A \to \Omega$ into the final F-coalgebra (Ω, ω) (*anamorphism*). This interpretation of data is called *final coalgebra semantics*. Note that not all elements of the final coalgebra are in fact denoted; in particular, infinite data proper can of course not be represented by finite memory states. We have a situation of three nested data domains:

1. Elements of the initial F-algebra (cycle-free data proper),
2. the larger set of elements of the final F-coalgebra with finite representations (also including circular data proper),
3. the even larger final F-coalgebra as a whole, regardless of the existence finite and/or cycle-free representations (also including infinite data proper).

We focus on a uniform treatment of the latter two classes, in contrast to lazy functional programming that deals with effective finite approximations of the former. Hence our approach excludes applications of potentially infinite data, but makes several notorious problems decidable, see below.

For functional programming, we assume a "classical" computation model: Besides the heap part of memory storing data in the representation detailed above, there is a call stack holding frames that represent the pending function incarnations. References from the stack to data cells form the *root set* of a state of computation: Cells reachable transitively from this set are live, unreachable cells are dead. That is, the F-coalgebra closure, or smallest F-coalgebra that contains the root set, is the semantically relevant data of each computation state. Formally, let (A, α) be a subcoalgebra of (B, β) if and only if inclusion is a homomorphism. Then, for a root set $R \subseteq A$, the closure (A_R, α_R) is the smallest subcoalgebra of (A, α) such that $R \subseteq A_R$.

We abstract from concrete machine instructions and consider only their effect in terms of memory state transitions. Of these transitions, we require that they are compatible with purely functional programming, in the sense that live cells remain unchanged, and new data is added by completely initializing either fresh or dead cells. Live cells may become dead when the root set shrinks because a stack frame is discarded. That is, for a valid transition from state (A, α) with root set R to (A', α') with root set R', we require that (A_S, α_S) be a subcoalgebra of (A'_S, α'_S) where $S = R \cap R'$. Automatic garbage collection can be integrated transparently with this specification, because it only affects the choice of cells for additional data.

These requirements allow us to resolve the evaluation procedure for circular functional programs into finer steps than we could, say, with a term graph rewriting approach. This is necessary because operational details such as the precise order of effects matter; see section 2.2 below. Note that only operational semantics of the execution engine are affected; user-level function definitions, which will be given in terms of coiteration of step coalgebras, are not restricted in any way.

Final coalgebra semantics is coarser than graph-based approaches, because differently shaped graphs may be identified: For instance, a circular list equivalent to $[0, 1, 0, 1, \dots]$ can be represented by a cycle of $[0, 1]$ or $[0, 1, 0, 1]$ or $[0, 1, 0, 1, 0, 1]$ etcetera, as well as by a linear prefix of $[0, 1, 0]$ followed by cycle of $[1, 0]$ and so forth. This abstraction is more appropriate than the actual pointer graph for a referentially transparent data model, because it considers

$$A = \{a_1, a_2, a_3, a_4\} \qquad \alpha = \left\{ \begin{array}{ll} a_1 \mapsto \bigl([0], [a_2]\bigr) & a_2 \mapsto \bigl([1], [a_3]\bigr) \\ a_3 \mapsto \bigl([0], [a_4]\bigr) & a_4 \mapsto \bigl([1], [a_1]\bigr) \end{array} \right\}$$

$$B = \{b_1, b_2, b_3, b_4, b_5\} \qquad \alpha = \left\{ \begin{array}{ll} b_1 \mapsto \bigl([0], [b_2]\bigr) & b_2 \mapsto \bigl([1], [b_3]\bigr) \\ b_3 \mapsto \bigl([0], [b_3]\bigr) & b_4 \mapsto \bigl([1], [b_5]\bigr) \\ b_5 \mapsto \bigl([0], [b_4]\bigr) & \end{array} \right\}$$

$$R = \underbrace{\{a_1, a_3\} \times \{b_1, b_3, b_5\}}_{\rightsquigarrow \ell_o} \cup \underbrace{\{a_2, a_4\} \times \{b_2, b_4\}}_{\rightsquigarrow \ell_e}$$

Fig. 1. Two F-coalgebras (A, α) and (B, β) both representing the circular lists $\ell_o = [0, 1, 0, 1, \dots]$ and $\ell_e = [1, 0, 1, 0, \dots]$ and the largest F-bisimulation equivalence $R \subseteq A \times B$.

only local bits and dereferencing in order to distinguish cells. In other words, data are fully abstract with respect to pattern matching. See Fig. 1 for examples. Recall the principle of *coinduction*: All elements that are *bisimilar* (related by any F-bisimulation) are mapped to the same element of the final F-coalgebra. Additionally, the internal degrees of freedom in memory representation enable powerful optimizations of cycle-handling; see section 2.4 below. All in all, pointer coalgebra can be considered an improvement over pointer algebra [9].

Referential Transparency Referential transparency is a property of data abstraction within a programming language. For the philosophically minded, the idea can be summarized as "references behave in a way that mathematicians can understand (and practical programmers cannot)". It has many aspects: unobservability of data layout, cell identity, write operations, memory management, and so forth.

Final coalgebraic semantics ensures *static* referential transparency, in the sense that data do not possess spurious identity or relations, as implied by pointer comparison and arithmetics. Monotonicity ensures *dynamic* referential transparency, in the sense that references never change their meaning.

2.2 Corecursive Functions

In order to deal with recursive computation on circular data, the deadlocking naïve bottom-up order of operations described above needs to be abandoned. Instead of passing results *out* when a function call returns, pass higher-order pointers to the desired location of results *in* when the function is called. By convention, these locations must be written to before the function returns. Many functions (at least all those that are primitively corecursive) can be operationalized such that the root cell of the output can be created, albeit with unitialized references, before any recursive calls are made, and stored at the desired location. Filling the gaps in the result is delegated to recursive calls in the same fashion, by passing on pointers.

This mode of operation for recursive function calls is not a new invention. It has been known for some time under the term *destination-passing style* [8]. Even before that, it has been studied by compiler constructors as *tail call modulo cons* [12]: an optimization that allows many function calls to be executed as jumps with reuse of stack frames, even if they do not appear conceptually in tail position. However, if this optimization is not performed (at least not always, see section 2.4) and the stack frames preserved, a generic technique for cycle handling can be devised.

If every function incarnation is recorded on the call stack as a pending frame until it completes, then we have enough information to detect cycles. Circular function incarnations manifest themselves as pairs of stack frames with identical input. This situation can be detected by inspecting the stack at call time.

Once the cycle is detected, means of escape can be devised. The outer incarnation of the function must have recorded its output at a location specified on the stack, before making a recursive call. Since we are dealing with pure functions, the inner incarnation must produce the same output as the outer. It seems obvious that the ouput should be copied from the outer to the inner frame, and the inner incarnation can then return immediately, effectively avoiding the looming vicious circle. This scheme performs a cycle-inducing operation with purely functional semantics, where circular input is translated to circular output.

Assume we wish to increase each element of ℓ_2 by one. In the first call, we produce an output cell with the number 2 and a hole, to be filled with the recursive computation on ℓ_3. The next three calls produce more output cells with the numbers $3, 4, 6$; the fifth call has the same input as the second and refers back to the output cell with 3. By virtue of the functional cycle-inducing operation we end up with the list $[2, 3, 4, 6, 3, 4, 6, 3, 4, 6, \dots]$. See also [13].

From this idea, a generic evaluation scheme for circular functions, complete for the class of corecursive functions, can be derived. A theorem of total correctness is given in the thesis. Operational details aside, it states the following:

- Let a circular function be given as a mapping † that takes memory states to other F-coalgebras over the same carrier to be used as coiteration steps, that is, to F-coalgebras that map inputs to "virtual cells" where arguments to recursive calls take the place of references.

  For the above example, and for each memory state (A, α), take the F-coalgebra $(A, \alpha^\dagger)$, where $\alpha(a) = (n, [a'])$ implies $\alpha^\dagger(a) = (n + 1, [a'])$.

  Such a step coalgebra can be thought of, and is in fact operationalized, as a sequence of state transitions that, when invoked on an argument a, first creates the root cell b of the output with the local content specified by $\alpha^\dagger(a)$, and then make a recursive call in destination-passing style, with input a' and output going into the uninitialized reference slot of the new cell b.
- Let the step be compatible with final coalgebra semantics of data, in the sense that it does not distinguish semantically equivalent cells:

$$\alpha!(a_1) = \alpha!(a_2) \implies F(\alpha!)\big(\alpha^\dagger(a_1)\big) = F(\alpha!)\big(\alpha^\dagger(a_2)\big)$$

– The evaluation algorithm performs for each initial memory state and correspondings step coalgebra a referentially transparent memory state transition, ending in a state (A', α'), such that the semantics of output data equal the image of the semantics of input data under the anamorphism of the step. Let a and b be the root cells of the function input and output, respectively. Note that a occurs in A and A' equivalently, but b in A' only. Then

$$\alpha^{\dagger}!(a) = \alpha'!(b)$$

Coming back to the principle of coinduction, we may state the same in other words: The evaluation algorithm modifies memory in a referentially transparent way, in order to make the output of the corecursive function under dereferencing bisimilar to the input under the step.

Hence we have given a low-level executable specification of the usual technique of coiterative function definition, where a step induces a function with final coalgebra range as its anamorphism. The proof has three main parts:

1. A graph coloring argument shows that incomplete outputs are not accidentally observed.
2. Cycle detection ensures termination.
3. The proposed semantic equivalence is constructed coinductively as an F-bisimulation.

The incomplete output cells that show up during evaluation, and are dealt with explicitly in the proof, are essential to the approach. They are created by an incarnation of the step, stored as outputs in destination-passing style (in other incomplete cells), and eventually completed by recursive function calls, before the stack frame associated with the incarnation is discarded, and before they become live.

Incomplete cells are patently incompatible with straightforward rewriting or reduction semantics, however, where all intermediate forms of the program are well-formed. That is the ultimate reason for the low-level approach to memory effects we have detailed above.

Fortunately, the low-level details need not be exposed to the user: A mapping † from memory states to coiteration steps can be specified in ordinary functional programming style by pattern equations.

Recalling our running example, the step that specifies how to increase each element of a list by one is specified for each memory state by:

$$\mathit{incrstep}([\,]) = [\,] \qquad\qquad \mathit{incrstep}([a] \oplus b) = [a+1] \oplus b$$

The base case (left) is only required if lists may be both finite and circular. It can be dropped (as above) if circular lists proper are the domain, and yet the function remains totally well-defined.

The associated anamorphism, and thus the desired corecursive function, is the greatest solution of the equations:

$$\mathit{incr}([\,]) = [\,] \qquad\qquad \mathit{incr}([a] + b) = [a+1] \oplus \mathit{incr}(b)$$

How to design a functional front-end language that allows to define circular functions in an elegant style, possibly similar to the last example, is still an open problem; see section 5.3.

2.3 Search Problems

The computation of coinductive functions on circular data can be emulated by lazy functional programming. But, since lazy computations are blind on the circular eye, the output is typically infinite even if the input is circular: Cycles are not detected and hence represented by unbounded spiral-shaped unrolling. We have already argued that certain problems are harder or impossible to solve when data are in this infinite form instead of a finite representation. In this section, we explore a large class of algorithmically challenging problems that remain tractable with corecursive functions implemented as specified above, but frustrate lazy evaluation.

Consider again the circular list $\ell_2 = [1, 2, 3, 5, 2, 3, 5, \dots]$ and the following questions:

1. Is there an even number in the list?
2. Is there a perfect number in the list?
3. Are all numbers in the list prime?
4. Are all numbers in the list Fibonacci?

All of these can be answered naturally by searching, but the truth value, the rationale behind it and the difficulty of deciding, respectively, vary:

1. Positive instance of an existential search problem; decided by the example 2 in the second cell.
2. Negative instance of an existential search problem; decided by the lack of an example (the smallest perfect number is 6).
3. Negative instance of a universal search problem; decided by the counterexample 1 in the first cell.
4. Positive instance of a universal search problem; decided by lack of a counterexample.

The positive cases are easy and terminate also in a lazy setting. The negative cases are harder; they require cycle detection and some escape strategy. For a search problem, it is obviously of no avail to search the same data twice. Hence, for an existential problem it is safe to assume that the search fails on the second leg through a cycle; if there is an example, it will be found on the first leg or another branch of computation. Conversely, for a universal problem it is safe to assume that the search succeeds on the second leg through a cycle; if there is a counterexample, it will be found on the first leg or another branch of computation.

This line of reasoning can be generalized. Consider an n-ary predicate on data specified as a system of Horn rules. The decision is a proper search problem if all variables occurring in the premises are bound to unique substructures of the conclusion arguments. As a first example, consider the definition of finite lists (there is an empty tail):

$$\mathit{finite}([\,]) \qquad\qquad \mathit{finite}([a] \oplus b) \Leftarrow \mathit{finite}(b)$$

Such recursive definitions are usually given fixpoint semantics. It can be shown that, for finite cycle-free data, there is a unique fixpoint. In the example, all cycle-free lists are finite, and fittingly the unique fixpoint encompasses all such lists. For circular data, however, fixpoints are no longer unique. By Tarski's theorem, there is a complete lattice of fixpoints for monotonic deduction systems such as Horn rules. The least fixpoint is the appropriate semantics for existential problems such as the above example. Dually, the greatest fixpoint is the appropriate semantics for universal problems. Consider for instance the complementary definition of infinite lists (all tails are nonempty):

$$\mathit{infinite}([a] \oplus b) \Leftarrow \mathit{infinite}(b)$$

For other classes of predicates, there may be an intuitive interpretation that suggests either of the extremal fixpoints, even if there is no immediate classification as either existential or universal. Contrast the binary predicates that define lists of the same length, implying greatest fixpoint semantics (one infinite list is as long as another),

$$\mathit{aslong}([\,], [\,]) \qquad\qquad \mathit{aslong}([a] \oplus b, [c] \oplus d) \Leftarrow \mathit{aslong}(b, d)$$

and longer lists, implying least fixpoint semantics (one infinite list is not longer than another):

$$\mathit{longer}([a] \oplus b, [\,]) \qquad\qquad \mathit{longer}([a] \oplus b, [c] \oplus d) \Leftarrow \mathit{longer}(b, d)$$

Using the same cycle-detection techniques as for coinductive functions, a generic evaluation scheme for predicates on circular data can be given. It combines regular depth-first search with the immediate return of a truth value for the inner incarnation of a detected cycle. The abstract collection of values to be returned in the latter case, for all predicates and arguments in the program, is called the *expectation*. Neither are all expectations compatible with final coalgebra semantics, nor do they all yield fixpoints. The complete classification is an unsolved, and apparently hard, problem. But the following proven rules are sufficient for a great number of practical examples:

1. The constantly *false* expectation is consistent and yields the least fixpoint.
2. The constantly *true* expectation is consistent and yields the greatest fixpoint.
3. Search success depends monotonically on expectation: Assuming expectations A and B are consistent and yield fixpoints F and G, respectively, then $F \subseteq G$ if $A(x) \Rightarrow B(x)$ pointwise for all x.
4. Expectations are properly stratified:
   (a) Consistent partial expectations for independent rule sets (no cross-references) combine unambiguously to consistent expectations for the combined rule set, via a coproduct construction.

(b) If rule set R refers to rule set S, but not vice versa, then the decision procedure for S can be used as a subprocedure for deciding R, with expectations chosen independently.

Semantic Equivalence Semantic equivalence, in the form of coalgebraic bisimilarity, is decidable within the system, thanks to finiteness of memory, as the greatest fixpoint of the generic recursive predicate "identical local bits and pairwise equivalent references"; that is, as the greatest F-bisimulation on memory under dereferencing. The notion of semantic equivalence is not only useful as an auxiliary predicate in higher-level algorithms; it can also be incorporated into corecursive evaluation to yield cycle detection up to semantic equivalence. See section 4.1 for an application of this powerful, albeit expensive, feature.

2.4 Efficiency Concerns

Searching the whole call stack on every function call is prohibitively expensive. Also, a tail call cannot be optimized when the caller's stack frame needs to be preserved for subsequent search. Both issues can be addressed in a very effective way by adding a single-bit mark to every reference, and postulating the invariant that every cycle must contain at least one marked edge. As long as the system is closed, and new cycles are only introduced by the cycle-handling action of corecursive functions, maintaining the invariant is extremely cheap: mark exactly the references obtained by the functional cycle-inducing operation; all other references in a purely functional setting are incorporated in fresh cells and hence cycle-free. For functions that proceed corecursively along references without shortcuts or detours (structural corecursion), invariants can be exploited in two ways:

1. A search of the call stack needs only be triggered if some argument of the current call is a marked reference.
2. A stack frame can only be found during search if some of the argument references points to a cell that is also pointed to by a marked reference. A mark or reference counter should be added to cells to keep track of those. In all other cases, the stack frame can be reclaimed for optimized tail calls.

Together, these optimizations imply that corecursive functions with cycle detection have only the overhead related to confirming the absence of marks, with regard to their cycle-ignorant counterparts, when invoked on cycle-free data. In other words: The cost of circular computations is only incurred when actually needed at runtime. The price for these benefits is that circular data is recognized only up to semantic equivalence, that is, a non-minimal representation may be chosen accidentally, depending on the placement of marks.

Additional optimization can be performed when the static call graph of the program is known at least approximately: A search of the call stack can be aborted when moving upwards from a frame of function f to a frame of its caller g, if f cannot call g transitively. This implies that the performance of a circular library function is not affected by how deeply its call is nested in an application context.

3 Technical Abstraction

Most existing back-end languages and runtime environments are ill-prepared for the precise kind of control over the call stack that is needed for our cycle techniques. Stack inspection, if available, is usually used for security purposes, and hence restricted to *input* parameters. Instead of waiting for a platform with full support for output parameters and stack inspection, we have chosen to implement our techniques in the form of a virtual machine, the *Malice* system, named for its paradigmatic opposition to an earlier generic platform for corecursive functions, the categorical language *Charity* [1].

Cycle detection is incorporated into the function calling conventions, whereas cycle handling is done by declaring an alternative function body for the circular case. An atomic cycle-inducing operation (called `ditto` in [14]) is provided. Functions both with cycle detection and without (and hence no overhead) are supported, as well as reference marking and the corresponding optimizations. Note that the consistency conditions for circular search still apply, but there is no analogous restriction to mixed recursive/corecursive function computation: Functions with and without cycle detection clauses may call each other freely.

The *Malice* machine is programmed in a bytecode assembly language, and comes with extensive support for program optimization, including a just-in-time compiler to threaded code and a dynamic partial evaluator. Since there is currently no generator for native machine code, the performance remains non-competitive with state-of-the-art functional language implementations; the optimizations have been implemented mainly to demonstrate that circular programming does not interfere unduly with compiler machinery.

4 Example Applications

4.1 Rational Arithmetics

The collection of algorithms described in this section has been used in [13] as the first working non-trivial example of effective circular computation. It has been inspired by [4], even though the approaches are quite different.

The standard algorithms for arithmetics and comparisons of numbers in decimal, or more generally b-adic representation as sequences of digits carry over to circular sequences which, endowed with a sign and decimal point (ignored here for simplicity), can represent all rational numbers. There are a few catches, but those are easily overcome with our cycle-handling techniques:

Addition cannot be computed naïvely by digitwise *full* addition, where each digit of the sum $s = x + y$ is computed from the corresponding digits of x, y and a *carry* digit that transports overflow from the right:

$$b \cdot c_i + s_i = x_i + y_i + c_{i+1}$$

This is effective for finite sequences of length n, where c_{n+1} can be assumed to be zero, thus starting right-to-left data flow. But for infinite sequences, a different approach is needed, namely *half* addition, where the carry digit is not immediately transported, but stored for later use:

$$b \cdot c'_i + s'_i = x_i + y_i$$

This operation is fully parallel on the digits, and can hence be executed as a corecursive *map/zip* function proceeding left-to-right. If c' is all zeroes then s' is the result. Otherwise, shift the two with respect to each other, either by discarding a leading zero from c' or by prepending a zero to s', and iterate the half addition with those in place of x, y. Since every digit can overflow at most once in the process, the iteration terminates in finitely many steps.

For division, the relation of corecursion and iteration is inverted: Each digit of the result can be computed by iterated trial subtraction. Division carries on with the remainder, and becomes circular when a remainder recurs. Hence the digit-producing iteration needs to be nested in a corecursive loop with cycle detection up to bisimilarity. Curiously, no direct circular multiplication algorithm is known; but multiplication can of course be emulated by double division.

4.2 Lists

Circular lists, both truly ring-shaped and with a linear prefix, have many applications. Some of the basic functional operations on lists carry over straightforwardly from the linear to the circular case, e.g., *map*, *insert*, *delete* and *append* as instances of corecursion; *any*, *all* and *sorted* as instances of circular search. Others require a bit more thought: The most interesting non-trivial case is the *filter* operation, which discards all elements that fail to satisfy a given predicate. It is infamous in lazy functional programming, because it fails when an infinite number of successive elements need to be discarded (which we call the *bust* situation). An effective circular *filter* can be constructed in several phases, in analogy to *mark&sweep* garbage collection.

1. Invalidate all elements to be discarded by replacement with a placeholder. (*mark*)
2. Test for *bust* case (not infinitely many valid elements, standard circular search problem) and return empty list immediately.
3. Otherwise, it is safe to discard placeholders by an iteration that proceeds to the next valid element nested inside a corecursive loop. (*sweep*)

With effective circular implementations of *map*, *append*, *filter* and *sorted*, we can even go ahead and extend the standard functional presentation of *quicksort* to circular lists:

$$\begin{aligned} \mathit{qsort}(\ell) = {} & \text{if } \mathit{sorted}(\ell) \text{ then } \ell \\ & \text{else let } [p] \oplus r = \ell \\ & \quad \text{in } \mathit{filter}(< p)(r) \oplus \mathit{filter}(= p)(r) \oplus \mathit{filter}(> p)(r) \end{aligned}$$

5 Conclusion

5.1 Summary

We have shown how to give final coalgebra semantics to the data type definitions of signature-based data types, where constructors correspond to classes of cells in memory. These, together with referentially transparent computations, form the basis of purely functional programming. We have pointed out that memory cells are prone to forming circular data structures, which constitute a subset of the final coalgebra, excluding infinite data proper, but a superset of the traditional initial algebra semantics which also excludes cycles. We have outlined generic evaluation strategies for corecursive functions and search procedures on circular data. Both separate the concerns of cycle detection and cycle handling, and deal with the former using information readily available on the call stack, assuming call-by-value and destination-passing style. The combination of functions and predicates on circular data leads to interesting and powerful algorithms, some of which are surprising generalizations of well-known cycle-free cases. There is a prototypic implementation in the *Malice* virtual machine, but the requirements of the techniques are modest enough to be adapted to other execution platforms.

5.2 Related Work

Several approaches address problems similar, but not quite equivalent, to ours. First of all, circular data with *static* shape can of course be created by recursive let-equations; the dynamic variant is apparently much harder. Then there are purely functional approaches to graph computations such as [2] that make cell identities visible to the programmer. In such a setting, both the dynamic creation and exhaustive search of circular structures is viable, but the graph semantics is too fine-grained for referentially transparent computations: it allows to assign different vertex identities to data with identical observable information (using local bits and dereferencing only). We are not aware of any graph-based approach that supports the level of abstraction provided by both initial algebra and final coalgebra semantics.

Lazy Evaluation Circular data structures are used a lot in lazy functional programming; sometimes consciously as in the *lazy virus* [5] and *credit card transform* [6] patterns, sometimes a fortiori as a subclass of infinite data. While these approaches provide reasonable abstraction, they abstract from a little too much and do not have support for effective circular searching.

Coinductive Logic Programming Independently of and concurrently to the work described here, virtually the same basic ideas and solutions have been proposed for coinductive logic programming (co-LP) [10]. The approaches inherit their respective strengths and weaknesses from the underlying programming paradigms: co-LP subsumes and considerably generalizes the circular search described above. On the other hand, it has no explicit notion of function computation and is essentially a first-order approach, whereas our corecursive functions extend straightforwardly to the higher-order case. Nevertheless, the similarities are striking at first glance, and it appears worthwhile to investigate them further.

5.3 Future Work

Front-End Language The current means for implementation of circular algorithms are very primitive: they have to programmed in the *Malice* byte-code assembly language. Even if memory management, static type checking and first-class functions are provided, the procedure remains tedious and error-prone, and the code hard to read, understand and maintain. A more user-friendly, circular functional programming language should be designed. The main difficulty of exposing enough of destination-passing style, cycle-handling capabilities and escape strategies for the sake of power, but not too much for the sake of abstraction, consistency and semantic tractability, is a yet unsolved problem.

Compiler to Machine Code Physical machines have support for stack-based function calling conventions anyway, so there is no theoretical reason not to have *Malice* code translated to native machine code. The main problems of a circular runtime system, in particular issues of garbage collection [11] and portable stack inspection [15] have been solved. Machine code could be generated using these techniques, either offline via C or online by use of some off-the-shelf just-in-time compiler library. In combination with state-of-the-art optimizing code generation techniques, a realistic evaluation of the performance of circular functional programming could be done.

Applications in Abstract Interpretation The arguments for the correctness of our circular search strategy rely heavily on the lattice structure of the Boolean data type. It should be possible to extend many or all of those arguments to other lattices. An obvious application would be the technique of abstract interpretation: Semantic properties of expressions of a formal language are approximated by evaluation in a coarse lattice-structured domain, reading and solving recursion as fixpoint equations. Since recursion in an expression is reified as cycles in the corresponding data, a generalization of circular search could be useful as an alternative to the classical, iterative solving techniques.

Acknowledgments

Peter Pepper supervised the PhD thesis underlying this article with patience and good judgement. Attendees and anonymous referees of the conference series IFL 2002–2004 provided excellent feedback, even though coalgebraic methods were apparently rather obscure to them. Attendees of the symposium CALCO-Jnr 2011 and anonymous referees provided even more up-to-date feedback, with additional potential for future work.

References

1. Cockett, R., Fukushima, T.: About charity. Tech. rep., University of Calgary (1992)
2. Erwig, M.: Inductive graphs and functional graph algorithms. Journal of Functional Programming 11(5), 467–492 (2001)
3. Ingerman, P.Z.: Thunks: A way of compiling procedure statements with some comments on procedure declarations. CACM 4(1), 55–58 (1961)
4. Karczmarczuk, J.: The most unreliable technique in the world to compute pi. In: 3rd International Summer School on Advanced Functional Programming (1998)
5. Kiselyov, O.: An exercise in lazy virology (1997), `http://okmij.org/ftp/Scheme/flatten-list.scm`
6. Kiselyov, O.: Transformations of cyclic graphs and the credit card transform (2002), `http://okmij.org/ftp/Haskell/CCard-transform-DFA.lhs`
7. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Proceedings 5th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1991). pp. 124–144. No. 523 in Lecture Notes in Computer Science, Springer (1991)
8. Minamide, Y.: A functional representation of data structures with a hole. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98). pp. 75–84. ACM (1998)
9. Möller, B.: Towards pointer algebra. Science of Computer Programming 21(1), 57–90 (1993)
10. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczynski, M. (eds.) Proceedings 22nd International Conference on Logic Programming (ICLP 2006). pp. 330–345. No. 4079 in Lecture Notes in Computer Science, Springer (2006)
11. Trancón y Widemann, B.: A reference-counting garbage collection algorithm for cyclical functional programming. In: Jones, R., Blackburn, S. (eds.) Proceedings 7th International Symposium on Memory Management (ISMM 2008). pp. 71–80. ACM (2008)
12. Warren, D.H.D.: DAI Research Report 141, University of Edinburgh (1980)
13. Trancón y Widemann, B.: Stacking cycles: Functional transformation of circular data. In: Arts, T., Peña, R. (eds.) Implementation of Functional Languages (IFL 2002), Revised Selected Papers. Lecture Notes in Computer Science, vol. 2670, pp. 150–164. Springer (2003)
14. Trancón y Widemann, B.: V→M: A virtual machine for strict evaluation of (co)recursive functions. In: Grelck, C., Huch, F., Michaelson, G., Trinder, P. (eds.) Implementation and Application of Functional Languages (IFL 2004), Revised Selected Papers. Lecture Notes in Computer Science, vol. 3474, pp. 90–107. Springer (2005)
15. Trancón y Widemann, B.: Stackless stack inspection – a portable escape route from vicious circles. In: Hanus, M., Fischer, S. (eds.) Programmiersprachen und Rechenkonzepte. pp. 188–196. No. 0811 in Technische Berichte, Institut für Informatik, Christian-Albrechts-Universität zu Kiel (2008)
16. Trancón y Widemann, B.: Strikte Verfahren zyklischer Berechnung. Dissertation, Technische Universität Berlin (2008)

Internal Models for Coalgebraic Modal Logics

Toby Wilkinson [⋆]

University of Southampton, UK
stw08r@ecs.soton.ac.uk

Abstract. We present ongoing work into the systematic study of the use of dual adjunctions in coalgebraic modal logic. We introduce a category of internal models for a modal logic. These are constructed from syntax, and yield a generalised notion of canonical model. Further, expressivity of a modal logic is seen to relate to properties of this category of internal models.

1 Introduction

The now standard approach to coalgebraic modal logic is through a so called logical connection - a dual adjunction between two base categories $\mathbb{X}$ and $\mathbb{A}$. The category $\mathbb{X}$ represents state spaces, or sets of processes, and the category $\mathbb{A}$ base logics, typically presented as algebras. The standard example is that of the categories **Set** and **BA**, where the latter is taken to represent classical propositional logics.

To these are added transition structures and modal operators. The modal operators, added to the base logics, aim to capture the dynamics of the transition structures. In choosing the modalities there is often a conflict between fully capturing the dynamics, and choosing modalities with an intuitive meaning, as logics with modalities that are hard to understand are unlikely to be adopted.

The transition structures are defined as coalgebras for an endofunctor T on $\mathbb{X}$, and the modal logics as algebras for an endofunctor L on $\mathbb{A}$. The semantics are then given by means of a natural transformation. Clearly this is a very general framework. Our work aims to explore the rich structure of this framework through the use of categorical techniques.

The first step is to make precise when a T-coalgebra is a model for an L-algebra, and this requires the notion of a valuation of an L-algebra in a T-coalgebra. A model then becomes a coalgebra, valuation pair. The models for an L-algebra form a category, and the structure of this category determines many of the properties of the modal logic that the L-algebra represents.

The main contribution of this paper is the observation that for each L-algebra, there is a full subcategory of its category of models that in many cases determines the logical properties of that L-algebra. These models we call the internal models, and as will be seen, they generalise the concept of canonical models found in Kripke semantics [2]. Like canonical models, they can be thought of as being constructed from the syntax of the modal logic.

[⋆] Research supported by an EPSRC Doctoral Training Account.

The most important property that an L-algebra can have, is that every model factors via an internal model. If an L-algebra has this property, then the information content of the category of models is contained entirely within the subcategory of internal models, and the other models need not be considered. This turns out to be very useful, since if $\mathbb{X}$ is wellpowered and certain morphisms are monomorphisms, then because the category of internal models is thin, its objects can be partitioned into a collection of equivalence classes that is a set (actually a poset). Moreover, under similar conditions, and if $\mathbb{X}$ has an appropriate factorisation system, the forgetful functor from the category of internal models to $\mathbb{X}$ detects colimits. So if $\mathbb{X}$ is cocomplete, wellpowered, and has an appropriate factorisation system, then the category of internal models is cocomplete, and a final internal model exists as the coproduct of a representative from each of the equivalence classes of internal models. This forms the basis of an adjoint functor theorem between the categories $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(T)$.

The factorisation of models via internal models is shown to follow from the existence of a factorisation system (E, M) in $\mathbb{X}$, and a condition that essentially amounts to T preserving M, and a particular natural transformation being componentwise in M. This is a restatement of [7, Theorem 4.2] and [4, Theorem 4].

In [7,4] this result is used to prove expressivity results for coalgebraic modal logics with respect to behavioural equivalence (bisimulation). We go beyond this, and show that by enriching over categories of what are known in [12] as the preordered sets, that similar expressivity results can be achieved for simulation. To demonstrate our approach we recover a well known result for simulation of image finite labelled transition systems.

A general outline of this paper is as follows. In section 2 we recall the category theoretic notion of a factorisation system. Then in section 3 we explain the dual adjunction framework in which we work. In section 4 we define what we mean by a model for a modal logic, and introduce the concept of an internal model. Then in section 5 we show when colimits of models and internal models exist. The proofs of these results are relatively straightforward, but long and tedious, so we restrict our presentation to an outline of the proofs. In section 6 an adjoint functor theorem is proved as a simple example of the utility of internal models. Then in section 7 internal models are used to explore a generalised notion of expressivity that encompasses both simulation and bisimulation.

2 Preliminaries

In what follows we will need to be able to factorise morphisms. The standard approach to this is via a factorisation system [1].

Definition 1. *In a category $\mathbb{C}$, a pair (E, M) of classes of morphisms is called a **factorisation system** for $\mathbb{C}$, if the following hold:*

1. *If $e \in E$, and h an isomorphism in $\mathbb{C}$, then if $h \circ e$ exists, $h \circ e \in E$.*
2. *If $m \in M$, and h an isomorphism in $\mathbb{C}$, then if $m \circ h$ exists, $m \circ h \in M$.*

*3. $\mathbb{C}$ has (E, M)-**factorisations**; i.e. every morphism f in $\mathbb{C}$ factors as $f = m \circ e$, with $m \in M$ and $e \in E$.*

*4. $\mathbb{C}$ has the **unique** (E, M)-**diagonalisation property**; i.e. every commuting square in $\mathbb{C}$ with $e \in E$ and $m \in M$, has a unique diagonal d such that the following commutes*

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[d, "f"'] & B \arrow[d, "g"] \arrow[dl, dashed, "d"'] \\ C \arrow[r, "m"'] & D \end{tikzcd}$$

Definition 2. *In a category $\mathbb{C}$ a factorisation system (E, M) is called **proper**, if E is a subclass of the epimorphisms of $\mathbb{C}$, and if M is a subclass of the monomorphisms of $\mathbb{C}$.*

Example 3.

1. In the category **Set** the obvious factorisation system (E, M), is to take E to be all the epimorphisms (surjective functions), and M all the monomorphisms (injective functions).
2. In the category **Top** of topological spaces, (Epi, Mono) is not a factorisation system, however (RegEpi, Mono) and (Epi, RegMono) are. Here, RegEpi is the class of regular epimorphisms (quotients), and RegMono is the class of regular monomorphisms (embeddings).

We shall also make use of the following proposition which is a statement of parts of [1, Propositions 14.6, 14.9].

Proposition 4. *Let $\mathbb{C}$ be a category with a factorisation system (E, M).*

1. Each of E and M is closed under composition.
2. If $f \circ g \in M$ and $f \in M$, then $g \in M$.
3. If $f \circ g \in E$ and $g \in E$, then $f \in E$.

A class of monomorphisms defines a notion of subobject in a category, and it is often important that for every object in a category its collection of subobjects is a set. The following definitions are standard [1].

Definition 5. *Given a class M of monomorphisms in a category $\mathbb{C}$ we define the following:*

*1. An M-**subobject** of an object A in $\mathbb{C}$ is a pair (S, m), where $m\colon S \to A$ is in M.*
*2. Two M-subobjects (S, m) and (S', m') of A are **isomorphic** if there exists an isomorphism $h\colon S \to S'$ such that $m = m' \circ h$.*
*3. $\mathbb{C}$ is M-**wellpowered** if no object in $\mathbb{C}$ has a proper class of pairwise non-isomorphic M-subobjects. Here by pairwise non-isomorphic we mean that any pair of distinct subobjects are non-isomorphic.*

*Dually, for a class E of epimorphisms we can define an E-**quotient object** of an object A as a pair (e, Q), where $e\colon A \to Q$ is in E. The obvious dual notion to $\mathbb{C}$ being M-wellpowered is that $\mathbb{C}$ is E-**cowellpowered**.*

Proposition 6. *Given a class M of monomorphisms in a category $\mathbb{C}$, the M-subobjects of an object A form a thin category $\mathbf{Sub}_M(A)$, the objects of which can be partitioned by isomorphisms into a collection of equivalence classes that carries a partial order, and if $\mathbb{C}$ is M-wellpowered this collection is a set.*

3 Dual-Adjunction Framework

Increasingly, the standard approach to coalgebraic modal logic is to formulate it in a dual-adjunction framework [9,7,4].

$$\begin{tikzcd} \mathbb{A} \arrow[loop left, "L"] \arrow[r, bend left, "S"] & \mathbb{X} \arrow[l, bend left, "P"] \arrow[loop right, "T"] \end{tikzcd}$$

Briefly this consists of two categories $\mathbb{A}$ and $\mathbb{X}$, and two contravariant functors P and S that form a dual adjunction i.e. there exists a natural isomorphism

$$\Phi\colon \mathbb{A}(-_1, P(-_2)) \Rightarrow \mathbb{X}(-_2, S(-_1))$$

Such a dual-adjunction is often referred to as a **logical connection** [13], and we denote the unit and counit by

$$\rho\colon id_{\mathbb{A}} \Rightarrow PS$$
$$\sigma\colon id_{\mathbb{X}} \Rightarrow SP$$

The category $\mathbb{X}$ represents a collection of state spaces, and a collection of generalised transition systems is defined on these state spaces as coalgebras for an endofunctor T. Similarly, the category $\mathbb{A}$ represents a collection of base logics to which modal operators are to be added. These are introduced via an endofunctor L, and the corresponding modal logics are the L-algebras. The semantics of these modal logics is given in two stages. First the dual adjunction gives a semantics for the base logics in terms of the state spaces, and then secondly, a natural transformation

$$\delta\colon LP \Rightarrow PT$$

gives the semantics of the modal operators in terms of the transition structures introduced by T [8,10].

Example 7. Many examples of logical connections have appeared in the literature. A small sample includes:

1. The logical connection arising from the contravariant powerset functor on **Set**, and the ultrafilter construction on the objects of **BA** [4].

2. Stone's Representation Theorem arising from taking the clopen sets of the objects of **Stone**, and an ultrafilter construction on the objects of **BA** [9].
3. The logical connection arising from the contravariant powerset functor on **Set**, and the filter construction on the objects of **MSL** (meet semilattices with top) [4].
4. The logical connection arising from taking the σ-algebra of the objects of **Meas** (measurable spaces), and a filter construction on the objects of **MSL** [4].

4 Models and Internal Models

The Kripke semantics for modal logic [2] introduces the concepts of frame, valuation, and model, where a model is a pair consisting of a frame and a valuation. There are obvious generalisations of these notions to coalgebraic modal logic.

Definition 8. *Given an L-algebra (A, α) and a T-coalgebra (X, γ), if there exists a morphism f (not necessarily unique) such that the diagram below commutes, then (X, γ) is called a **frame** for (A, α), and f is called a **valuation** of (A, α) in (X, γ), and the pair is called a **model** for (A, α).*

$$\begin{tikzcd}
L(A) \arrow[r, "L(f)"] \arrow[dd, "\alpha"'] & LP(X) \arrow[d, "\delta_X"] \\
 & PT(X) \arrow[d, "P(\gamma)"] \\
A \arrow[r, "f"'] & P(X)
\end{tikzcd}$$

Clearly, if (A, α) is the initial L-algebra then every T-coalgebra is a frame, but in general this is not the case.

Remark 9. If we were following the conventions of Kripke semantics for modal logic [2] we would call every T-coalgebra a frame irrespective of the choice of L-algebra. We do not do this as we already have a name for such entities - they are T-coalgebras. Therefore we reserve the name frame for only those T-coalgebras for which valuations exists, and this necessarily makes the concept of a frame one that is relative to a choice of L-algebra.

Now, as observed in [13], the logical connection allows every model diagram in $\mathbb{A}$ to be redrawn in $\mathbb{X}$ as

$$\begin{tikzcd} X \arrow[r, "f^\flat"] \arrow[dd, "\gamma"'] & S(A) \arrow[d, "S(\alpha)"] \\ & SL(A) \\ T(X) \arrow[r, "T(f^\flat)"'] & TS(A) \arrow[u, "\delta^*_A"'] \end{tikzcd}$$

where $f^\flat$ is the adjunct of f, and $\delta^* \colon TS \Rightarrow SL$ is defined as

$$\delta^* = SL\rho \circ \delta^\flat S$$

following [7]. Such an $f^\flat$ we will call a theory map.

Definition 10 (Models). *For an L-algebra (A, α) we define $\mathbf{Mod}(A, \alpha)$, the category of* ***models*** *for (A, α), with objects given by pairs*

$$((X, \gamma), f \colon X \to S(A))$$

where (X, γ) is a T-coalgebra, and f is a ***theory map*** *(as above), and morphisms*

$$g \colon ((X_1, \gamma_1), f_1) \to ((X_2, \gamma_2), f_2)$$

given by a T-coalgebra morphism $g \colon (X_1, \gamma_1) \to (X_2, \gamma_2)$ such that $f_1 = f_2 \circ g$.

In [3] a similar definition of a category of models for an L-algebra is made, however this is done in terms of diagrams in $\mathbb{A}$ i.e. pairs of T-coalgebras and valuations. In what follows next we prefer to work in $\mathbb{X}$, but as already noted above, and first observed in [13], the logical connection allows us to move freely backwards and forwards between the two definitions. We can make this precise with the following proposition.

Proposition 11. *The natural transformation $\delta \colon LP \Rightarrow PT$ defines a functor $\tilde{P} \colon \mathbf{CoAlg}(T) \to \mathbf{Alg}(L)$ given by*

$$\tilde{P}(X, \gamma) = (P(X), P(\gamma) \circ \delta_X)$$
$$\tilde{P}(f) = P(f) \colon \tilde{P}(Z, \xi) \to \tilde{P}(X, \gamma)$$

where $f \colon (X, \gamma) \to (Z, \xi)$, and for each L-algebra (A, α), $\mathbf{Mod}(A, \alpha)$ is dually isomorphic to the comma category $((A, \alpha) \downarrow \tilde{P})$.

We are now ready to introduce our key idea. Recall from Kripke semantics the notion of a canonical model [2]. This is a model of a modal logic constructed from the syntax itself. The idea is that when trying to prove completeness, by the way the canonical model is constructed from the syntax, for every formula that is not derivable, one can find a state that witnesses that the formula is not valid.

In such a canonical model the possible worlds are the theories of the logic. In our setup, $S(A)$ is the collection of all possible theories of (A, α), so an obvious question is when can we construct a model from $S(A)$ i.e. when can we put a T-coalgebra structure on $S(A)$ such that it becomes a model for (A, α)?

In general this cannot be done, but in [15] conditions are given in the case of the standard logical connection between **BA** and **Set** (Example 7 (1)) for the existence of a, not necessarily unique, model for the initial L-algebra with carrier set $S(A)$. From this they derive a strong completeness result.

To illustrate our approach consider a toy example, where the logical connection consists of functors P and S given by the contravariant powerset functor on **Set**, the functors L and T both map every object to the two element set $\mathbf{2}$, and $\delta_X(i) = \{i\}$ for $i \in \mathbf{2}$. Then the initial L-algebra is $(\mathbf{2}, id_{\mathbf{2}})$, and a final T-coalgebra is given by $(\{\{0\},\{1\}\}, \gamma)$, where $\gamma(\{i\}) = i$. Here it should be observed that the carrier set of the final coalgebra is clearly a proper subset of $S(\mathbf{2})$, so our approach is to consider not just models constructed from the whole of $S(A)$, but to consider models built from subobjects of $S(A)$. In other words, models where the theory map is a monomorphism.

Definition 12 (Internal Models). *Given a class M of monomorphisms in $\mathbb{X}$, we define the category $\mathbf{IntMod}_M(A, \alpha)$ to be the full subcategory of $\mathbf{Mod}(A, \alpha)$ where the theory maps are in M, and write*

$$G\colon \mathbf{IntMod}_M(A, \alpha) \to \mathbf{Mod}(A, \alpha)$$

for the corresponding inclusion functor.

We parameterise by the class M, as sometimes we require the morphisms of M to have additional properties, for example, that the members of M are preserved by T. In Example 21 (3) the Giry functor does not preserve all monomorphisms, but does preserve a particular subclass of them.

Proposition 13. *The category $\mathbf{IntMod}_M(A, \alpha)$ is thin, and if for all $m \in M$ $\delta^*_A \circ T(m)$ is a monomorphism, then the forgetful functor from $\mathbf{IntMod}_M(A, \alpha)$ to $\mathbf{Sub}_M(A)$ is full.*

Proof. Since the theory maps of internal models are monomorphisms the category $\mathbf{IntMod}_M(A, \alpha)$ is thin. For the second part, consider a pair of internal models I_1 and I_2, and a morphism g in $\mathbf{Sub}_M(A)$ between the theory maps of I_1 and I_2. We have to show that g is an internal model morphism, but this can be seen to easily follow if $\delta^*_A \circ T(m_2)$ is a monomorphism, where m_2 is the theory map of I_2. □

The utility of internal models arises from the observation that in many cases it is possible to take a model and "quotient by behavioural equivalence" i.e. produce a smaller model by identifying states that are behaviourally equivalent.

Such a quotiented model will be an internal model, and we say the model factors via the internal model.

The above is not very precise as we have not said what we mean by quotient and behavioural equivalence. This will be come clear in Section 7, but first we make the following definition.

Definition 14. *We say a model X in $\mathbf{Mod}(A,\alpha)$* ***factors*** *via the internal model I in $\mathbf{IntMod}_M(A,\alpha)$ if there exists a morphism $g\colon X \to G(I)$ in $\mathbf{Mod}(A,\alpha)$.*

It is possible to give very general conditions under which models factor via internal models. The following proposition is essentially a restatement of [7, Theorem 4.2] and [4, Theorem 4].

Proposition 15. *If the category $\mathbb{X}$ has a factorisation system (E,M) and*

$$m \in M \Rightarrow \delta_A^* \circ T(m) \in M$$

then every model in $\mathbf{Mod}(A,\alpha)$ factors via an internal model in $\mathbf{IntMod}_M(A,\alpha)$.

Proof. Consider a model $((X,\gamma),f)$ in $\mathbf{Mod}(A,\alpha)$. Then by the factorisation system there exists $e \in E$ and $m \in M$ such that $f = m \circ e$, and by the definition of a model, the perimeter of the following diagram commutes

$$\begin{tikzcd}
X \arrow[r, two heads, "e"] \arrow[d, "T(e)\circ\gamma"'] & I \arrow[d, "S(\alpha)\circ m"] \arrow[dl, dashed, "\zeta"'] \\
T(I) \arrow[r, hook, "\delta_A^*\circ T(m)"'] & SL(A)
\end{tikzcd}$$

and by assumption $\delta_A^* \circ T(m) \in M$, and so by the diagonalisation property of the factorisation system, there exists a unique $\zeta\colon I \to T(I)$ making the diagram commute.

Thus $((I,\zeta),m)$ is an internal model in $\mathbf{IntMod}_M(A,\alpha)$, and e is the morphism by which $((X,\gamma),f)$ factors via $((I,\zeta),m)$. □

If the category $\mathbb{X}$ has a proper factorisation system (E,M), factoring a model via an internal model, can be viewed as putting a T-coalgebra structure map on an E-quotient object of the state space of the model. As we will see in Section 7, this corresponds to quotienting with respect to behavioural equivalence.

5 Colimits in $\mathbf{Mod}(A,\alpha)$ and $\mathbf{IntMod}_M(A,\alpha)$

As we shall see in future sections, one of the most important aspects of the structure of the categories $\mathbf{Mod}(A,\alpha)$ and $\mathbf{IntMod}_M(A,\alpha)$ is the presence, or otherwise, of colimits. In this section we shall see that in the case of the category $\mathbf{Mod}(A,\alpha)$, the forgetful functor from $\mathbf{Mod}(A,\alpha)$ to $\mathbb{X}$ creates small colimits, but that for the category $\mathbf{IntMod}_M(A,\alpha)$, the corresponding forgetful functor does not. However, under certain additional conditions it does detect small colimits, but does not necessarily preserve them.

We define the following forgetful functors

$$\begin{aligned} U &\colon \mathbf{CoAlg}(T) \to \mathbb{X} \\ V &\colon \mathbf{Mod}(A,\alpha) \to \mathbb{X} \\ V^* &\colon \mathbf{Mod}(A,\alpha) \to \mathbf{CoAlg}(T) \\ W &\colon \mathbf{IntMod}_M(A,\alpha) \to \mathbb{X} \end{aligned}$$

where $V = UV^*$ and $W = VG$.

To start we state without proof the well known result (see for example [14] for the case in **Set**) that the forgetful functor $U\colon \mathbf{CoAlg}(T) \to \mathbb{X}$ creates small colimits.

Theorem 16. *The forgetful functor* $U\colon \mathbf{CoAlg}(T) \to \mathbb{X}$ *creates small colimits.*

The case for the forgetful functor $V\colon \mathbf{Mod}(A,\alpha) \to \mathbb{X}$ follows in a similar fashion, with the additional detail that a theory map must be constructed for the colimit.

Theorem 17. *The forgetful functor* $V\colon \mathbf{Mod}(A,\alpha) \to \mathbb{X}$ *creates small colimits.*

Proof. Consider a small category $\mathbb{J}$ and a functor $D\colon \mathbb{J} \to \mathbf{Mod}(A,\alpha)$, and suppose that $\mathbb{X}$ has colimits of shape $\mathbb{J}$. Then we have that VD has a colimit $(C, \phi_j\colon VD(j) \to C)_{j\in\mathbb{J}}$. We now proceed as follows (sketch):

1. Use the functor V^* and Theorem 16 to construct a colimiting T-coalgebra (C,χ).
2. Use the theory maps and the universal property of C to construct a morphism g from C to $S(A)$.
3. Use the universal property of C to show that there is a unique morphism from C to $SL(A)$ and that this makes g into a theory map, $((C,\chi),g)$ a model, and the ϕ_j model morphisms.
4. For another cocone of D use the functor V^* and the universal property of (C,χ) to construct a unique mediating morphism to the underlying T-coalgebra.
5. Use the uniqueness of g to show that the mediating morphism is a model morphism, and thus $((C,\chi),g)$ is the colimit of D.

It is clear that $(((C,\chi),g),\phi_j\colon D(j) \to ((C,\chi),g))_{j\in\mathbb{J}}$ is the unique cocone for D that is mapped by V to the colimit (C,ϕ) of VD. Thus we can conclude that V creates colimits of shape $\mathbb{J}$. $\square$

For the category $\mathbf{IntMod}_M(A,\alpha)$ the details are more complicated. The approach we take is that the colimit is constructed in $\mathbf{Mod}(A,\alpha)$, and then the resulting colimiting model is factored via an internal model using Proposition 15.

Theorem 18. *Given an L-algebra (A, α), if the following hold:*

1. *the category $\mathbb{X}$ has a factorisation system (E, M),*
2. *$m \in M \Rightarrow \delta^*_A \circ T(m) \in M$,*

then the forgetful functor $W \colon \mathbf{IntMod}_M(A, \alpha) \to \mathbb{X}$ detects small colimits.

Proof. Consider a small category $\mathbb{J}$ and a functor $D \colon \mathbb{J} \to \mathbf{IntMod}_M(A, \alpha)$, and suppose that $\mathbb{X}$ has colimits of shape $\mathbb{J}$. Then by Theorem 17, the functor GD has the colimit $(((C, \chi), g), \tau_j \colon GD(j) \to ((C, \chi), g))_{j \in \mathbb{J}}$ in $\mathbf{Mod}(A, \alpha)$. We now proceed as follows (sketch):

1. Use Proposition 15 to factor $((C, \chi), g)$ via an internal model $((I, \zeta), m)$ by $e \colon ((C, \chi), g) \to ((I, \zeta), m)$.
2. For another cocone $(((Z, \xi), h), \psi_j \colon D(j) \to ((Z, \xi), h))_{j \in \mathbb{J}}$ of D there is a unique mediating morphism $\mu \colon ((C, \chi), g) \to ((Z, \xi), h)$ in $\mathbf{Mod}(A, \alpha)$.
3. The uniqueness of g means $g = h \circ V(\mu)$, then use the diagonalisation property of the factorisation system to construct a unique $\eta \colon I \to Z$.
4. Use $\delta^*_A \circ T(h) \in M$ and thus a monomorphism to show that η is an internal model morphism, and $(((I, \zeta), m), e \circ \tau_j \colon D(j) \to ((I, \zeta), m))_{j \in \mathbb{J}}$ is the colimit of D.

□

6 An Adjoint Functor Theorem

As a simple example to show the utility of the categories $\mathbf{IntMod}_M(A, \alpha)$ we prove an adjoint functor theorem.

To find a functor $\tilde{S} \colon \mathbf{Alg}(L) \to \mathbf{CoAlg}(T)$ that together with $\tilde{P}$ forms a dual adjunction between $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(T)$ we must show that for every L-algebra there is a universal morphism to $\tilde{P}$. But this is the same as requiring that each comma category $((A, \alpha) \downarrow \tilde{P})$ has an initial object.

Theorem 19. *If for all L-algebras (A, α) the following hold:*

1. *every model in $\mathbf{Mod}(A, \alpha)$ factors via some model in $\mathbf{IntMod}_M(A, \alpha)$,*
2. *$\mathbf{IntMod}_M(A, \alpha)$ has a final object,*

then there exists a dual adjunction between $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(T)$.

Proof. We are required to show that for any L-algebra (A, α) that $((A, \alpha) \downarrow \tilde{P})$ has an initial object. But by Proposition 11 this is the same as requiring that each $\mathbf{Mod}(A, \alpha)$ has a final object.

Consider such a $\mathbf{Mod}(A, \alpha)$. By the two premises above, every model in $\mathbf{Mod}(A, \alpha)$ factors via the final object in $\mathbf{IntMod}_M(A, \alpha)$, and since the theory map of the final internal model is a monomorphism, the morphism from a model in $\mathbf{Mod}(A, \alpha)$ to the final internal model is unique. Thus the final object in $\mathbf{IntMod}_M(A, \alpha)$ is the final object in $\mathbf{Mod}(A, \alpha)$. □

In the above proof, no explicit use was made of the class M, only that $\mathbf{IntMod}_M(A,\alpha)$ is a special subcategory of $\mathbf{Mod}(A,\alpha)$ - it has a final object, and for all objects in $\mathbf{Mod}(A,\alpha)$ there exists a unique morphism to an object in $\mathbf{IntMod}_M(A,\alpha)$. Therefore for each L-algebra a different class M of monomorphisms could in principle be chosen, but typically the same class would be used for all L-algebras. Indeed, the choice of M is likely to be driven by the properties of the base category $\mathbb{X}$ and the functor T, as in the following corollary.

Corollary 20. *If the following hold:*

1. *$\mathbb{X}$ has a factorisation system (E,M), is M-wellpowered, and has small coproducts,*
2. *for all L-algebras (A,α) we have $m \in M \Rightarrow \delta^*_A \circ T(m) \in M$,*

then there is a dual adjunction between $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(T)$.

Proof. By Proposition 15, for every L-algebra, every model in $\mathbf{Mod}(A,\alpha)$ factors via an internal model in $\mathbf{IntMod}_M(A,\alpha)$.

Now since $\mathbb{X}$ is M-wellpowered, by Proposition 6, the objects of $\mathbf{Sub}_M(A)$ can be partitioned into a set of equivalence classes. Also by Proposition 13, the forgetful functor from $\mathbf{IntMod}_M(A,\alpha)$ to $\mathbf{Sub}_M(A)$ is full and $\mathbf{IntMod}_M(A,\alpha)$ is thin. Therefore the objects of $\mathbf{IntMod}_M(A,\alpha)$ can also be partitioned into a set of equivalence classes, and since $\mathbb{X}$ has small coproducts, by Theorem 18, the coproduct of a representative from each equivalence class exists. Further, since $\mathbf{IntMod}_M(A,\alpha)$ is thin, and between each pair of representatives of an equivalence class there exists an isomorphism, the injections into the coproduct yield a unique morphism from each object of $\mathbf{IntMod}_M(A,\alpha)$ to the coproduct. Thus the coproduct is the final object in $\mathbf{IntMod}_M(A,\alpha)$.

Finally by Theorem 19 we have the result. $\square$

In most cases, for a particular choice of $\mathbb{X}$, the existence of a factorisation system, wellpoweredness, and the existence of small coproducts, is well known. Further, it is often straightforward to show that T preserves M, thus by Proposition 4, what is left to show is that δ^* is componentwise in M, and this is typically where the bulk of the work lies.

Example 21.

1. Example 7 (1): **Set** clearly satisfies the premises of Corollary 20 with the usual factorisation system given by surjective and injective functions. Then if we take T to be the finite powerset functor $\mathcal{P}_f$, and L to be the functor that adds a finite meet preserving operator $\Box$ to a Boolean algebra, it is shown in [4, Theorem 9] that for a natural choice of δ, that δ^* is componentwise injective. From this, and that $\mathcal{P}_f$ preserves injections, Corollary 20 yields a dual adjunction between $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(\mathcal{P}_f)$.
2. Example 7 (3): Again take **Set** with the usual factorisation system. This time take T to be the valuation functor $\mathcal{V}_O$ of [4, Section 3.1]. This is a generalisation of the finite powerset, finitely supported discrete subdistribution,

and finitely supported multiset functors. O is a downward-closed subset of a partially ordered commutative cancellative monoid M, and M also has the property $x \leq x + y$ for all $x, y \in M$. Then the valuation functor sends a set to the set of its valuations in O

$$\mathcal{V}_O(X) = \{\phi \colon X \to O \mid \mathrm{supp}(\phi) \text{ is finite and } \textstyle\sum_{x \in X} \phi(x) \in O\}$$

The analogous generalisation of L in the previous example is $K_{\widehat{O}}$, where $\widehat{O}$ is a dense subset of O, and this functor adds to a meet semilattice an order preserving modality $\Box_o$ for each $o \in \widehat{O}$. It is shown in [4, Theorem 13] that for a natural choice of δ, that the resulting δ^* is componentwise injective. From this, and that $\mathcal{V}_O$ preserves injections, Corollary 20 yields a dual adjunction between $\mathbf{Alg}(K_{\widehat{O}})$ and $\mathbf{CoAlg}(\mathcal{V}_O)$.

3. Example 7 (4): Since σ-algebras are closed under intersection, **Meas** is topological over **Set**, and since **Meas** is fibre-small, by [1, Theorem 21.16], **Meas** is wellpowered and cocomplete. Also in [4, Section 3.1] it is observed that morphisms with surjective underlying functions, and morphisms with injective underlying functions and surjective inverse image functions, form a factorisation system (E, M). Moreover, the Giry functor (or monad) G is observed to preserve M. For L take K, an instance of $K_{\widehat{O}}$ above, for $\widehat{O} = \mathbb{Q} \cap [0, 1]$. Then for a natural choice of δ, it is shown in [4, Theorem 17] that δ^* is componentwise in M. From this, Corollary 20 yields a dual adjunction between $\mathbf{Alg}(K)$ and $\mathbf{CoAlg}(G)$.

7 Expressivity

In this section we develop a generalised notion of expressivity for coalgebraic modal logics. This unifies the work on expressivity of coalgebraic modal logic over posets of [5,6] with the previous work of [7,4].

The historical notion of expressivity for a coalgebraic modal logic states that two states are logically equivalent if and only if they are behaviourally equivalent. Here logically equivalent means "have the same theory", and behaviourally equivalent means "can be identified in a model", where the identification is by means of coalgebra homomorphisms. Thus there is a reliance on the use of an implicit equality relation associated with each object in the category $\mathbb{X}$.

In the work on coalgebraic modal logic over posets of [5,6] the equality relation in the definitions of logical and behavioural equivalence is replaced with the partial order relation carried by each poset. We wish to unify both approaches in a common framework.

Firstly, we recall the category **Preord** of preordered sets and monotone functions, the objects of which are pairs consisting of a set and a preorder relation on that set. Similarly, the categories **Pos** (partially ordered sets), **Setoid** (setoids), and **DiscSetoid** (discrete setoids) have for objects pairs consisting of a set and respectively, a partial order, equivalence relation, or the equality relation, on that set. In [12] these examples are collectively known as the preordered sets, and they have significance for the coalgebraic analysis of simulation, which we shall come back to later.

We can consider these examples together by means of the following definition, where by a relation of "type R" we mean either a *preorder*, *partial order*, *equivalence relation*, or *equality.* The type is fixed, and every object in the category $\mathbf{Set}_R$ must have a relation of that type.

Definition 22. *The category* $\mathbf{Set}_R$ *has for objects pairs* (X, R_X) *consisting of a set* X*, and* R_X *a binary relation of type* R *on* X*. The morphisms are the* R*-preserving functions i.e.* $f\colon (X, R_X) \to (Y, R_Y)$ *is a morphism if and only if for all* $x, x' \in X$

$$xR_X x' \Rightarrow f(x)R_Y f(x')$$

To be explicit we have the following four cases:

1. If R is the type preorder, then $\mathbf{Set}_R$ is **Preord**.
2. If R is the type partial order, then $\mathbf{Set}_R$ is **Pos**.
3. If R is the type equivalence relation, then $\mathbf{Set}_R$ is **Setoid**.
4. If R is the type equality, then $\mathbf{Set}_R$ is **DiscSetoid** (which is obviously isomorphic to **Set**).

Now it turns out, though we won't go into the details, that the framework of a logical connection can be extended to the enriched setting [11], where the enrichment is over a symmetric monoidal closed category that is complete and cocomplete.

It is easy to verify that the forgetful from $\mathbf{Set}_R$ to **Set** creates limits and colimits - the product of (X, R_X) and (Y, R_Y) is given by $(X \times Y, R_{X\times Y})$, where $(x, y)R_{X\times Y}(x', y') \Leftrightarrow xR_X x'$ and $yR_Y y'$, and the final object is $(\mathbf{1}, R_{\mathbf{1}})$, where $\mathbf{1}$ is the singleton set, and $R_{\mathbf{1}} = \mathbf{1} \times \mathbf{1}$.

It is also easy to verify that binary product and the final object form the tensor and unit of a symmetric monoidal category. To make $\mathbf{Set}_R$ also closed we need internal hom objects $[(X, R_X), (Y, R_Y)]$ such that $[(Y, R_Y), -]$ is right adjoint to $- \times (Y, R_Y)$. The obvious definition for $[(X, R_X), (Y, R_Y)]$ is the set of all R-preserving functions from X to Y carrying the relation

$$fR_{[(X,R_X),(Y,R_Y)]}g \Leftrightarrow \forall x \in X\ f(x)R_Y g(x)$$

Further, we can define the unit $\eta_{(X,R_X)}\colon (X, R_X) \to [(Y, R_Y), (X, R_X)\times(Y, R_Y)]$ by $\eta_{(X,R_X)}(x) = f_x\colon (Y, R_Y) \to (X, R_X)\times(Y, R_Y)$, where $f_x(y) = (x, y)$, and the counit $\varepsilon_{(Z,R_Z)}\colon [(Y, R_Y), (Z, R_Z)] \times (Y, R_Y) \to (Z, R_Z)$, by $\varepsilon_{(Z,R_Z)}(g, y) = g(y)$. Thus we have the following proposition.

Proposition 23. *The category* $\mathbf{Set}_R$ *is cartesian closed.*

For the rest of this section we make the following assumptions.

Assumption 1. *The categories* $\mathbb{A}$ *and* $\mathbb{X}$ *are enriched over* $\mathbf{Set}_R$ *for some fixed type* R *of relations. Further, the categories* $\mathbb{A}$ *and* $\mathbb{X}$ *are concrete categories i.e. the objects are sets with some additional structure and carry a relation of type* R*, and the morphisms have underlying* R*-preserving functions.*

We do this for two reasons. Firstly enriching over $\mathbf{Set}_R$ is an obvious generalisation of the approach of [5,6], and secondly to investigate expressivity we need to access individual states of objects in $\mathbb{X}$.

Example 24. As expected from the motivating paragraphs we have the following examples:

1. In the case where R is the type equality, then enrichment over $\mathbf{Set}_R$ is just ordinary category theory, and so we have all the examples of logical connections from Example 7.
2. Example 7 (3) can be generalised to a logical connection between $\mathbf{MSL}$ and $\mathbf{Set}_R$. To do this we need to observe that the objects of $\mathbf{MSL}$ come with two built-in preorders. The first is the well known partial order defined by $x \leq y \Leftrightarrow x = x \wedge y$, and the second is equality. In what follows, if type R represents preorders or partial orders, then objects of $\mathbf{MSL}$ should be considered as having the standard partial order, and if R represents equivalence relations or equality, then they should be considered as carrying the equality relation. The functor $P\colon \mathbf{Set}_R \to \mathbf{MSL}$ sends an object (X, R_X) to the meet semilattice of its right R-closed subsets. A subset $U \subseteq X$ is right R-closed if $x \in U$ and xR_Xy implies $y \in U$. $P(X, R_X)$ is either ordered by inclusion or equality, depending upon the type R. The functor $S\colon \mathbf{MSL} \to \mathbf{Set}_R$ sends a meet semilattice A to the set of its filters, again either ordered by inclusion or equality depending upon the type R.

To develop a generalised notion of expressivity, we first need to extend the notions of logical equivalence and behavioural equivalence.

Definition 25. *Given two models X_1, X_2 in $\mathbf{Mod}(A, \alpha)$, and states $x_1 \in X_1$, $x_2 \in X_2$, we say x_1 and x_2 are **logically R-related** if*

$$f_1(x_1)\, R_{S(A)}\, f_2(x_2)$$

where f_1 and f_2 are the theory maps of X_1 and X_2 respectively.

Definition 26. *Given two models X_1, X_2 in $\mathbf{Mod}(A, \alpha)$, and states $x_1 \in X_1$, $x_2 \in X_2$, we say x_1 and x_2 are **behaviourally R-related** if there exists in $\mathbf{Mod}(A, \alpha)$ a cospan*

$$\begin{tikzcd} X_1 \arrow[r, "f_1"] & X_3 & X_2 \arrow[l, "f_2"'] \end{tikzcd}$$

such that $f_1(x_1)\, R_{X_3}\, f_2(x_2)$.

To see that these are the correct definitions we first consider the case where the type R is equality. In this case logically R-related simply becomes equality of theories, as expected. For the definition of behaviourally R-related we see that the forgetful functor from $\mathbf{Mod}(A, \alpha)$ to $\mathbf{CoAlg}(T)$ yields the usual definition of behavioural equivalence as a cospan in $\mathbf{CoAlg}(T)$ [8], but in addition, the forgetful functor to $\mathbb{X}$ yields a condition that the theory maps are compatible.

This is because we are working with arbitrary L-algebras, and not just the initial L-algebra, and is similar to the definition of bisimulation in [2].

The other cases of relation type R are best justified by an example. Consider the logical connection of Example 24 (2). In this case, if the type R represents preorders or partial orders, then logically R-related corresponds to inclusion of theories, and if x is behaviourally R-related to y, then x is simulated by y. The explanation for this was first noted in [6] for the case of enrichment over **Pos**, but we now extend this to $\mathbf{Set}_R$ as follows.

In [12] it is shown that for a functor F on **Set**, and an F-relator Γ, a general notion of Γ-simulation for F-coalgebras can be defined. Further, associated with Γ is a functor T on **Preord**, and the final T-coalgebra characterises Γ-similarity of F-coalgebras. Moreover, this characterisation of similarity arises from the preorder on the carrier of the final T-coalgebra, and the observation that every set carries a discrete preorder (equality). Thus for every F-coalgebra there is a corresponding T-coalgebra, and given two F-coalgebras, the Γ-similarity relation is given by the preorder on the images of states under the corresponding unique cospan of morphisms to the final T-coalgebra [12, Theorem 2].

Our notion of behaviourally R-related can thus be seen as taking this as a definition, rather than a consequence, and extending it to arbitrary cospans in $\mathbf{Mod}(A, \alpha)$, not just those with the final T-coalgebra as the target, and also to arbitrary functors T, rather than that arising from a functor F on **Set** and an F-relator Γ.

The following result is a simple consequence of the fact that morphisms in $\mathbb{X}$ are R-preserving.

Proposition 27. *Given two models X_1, X_2 in $\mathbf{Mod}(A, \alpha)$, and states $x_1 \in X_1$, $x_2 \in X_2$, if x_1 and x_2 are behaviourally R-related then x_1 and x_2 are logically R-related.*

Definition 28. *An L-algebra (A, α) is **R-expressive** for $\mathbf{Mod}(A, \alpha)$ if for all models in $\mathbf{Mod}(A, \alpha)$, states are logically R-related if and only if they are behaviourally R-related.*

To use internal models to investigate the phenomena of R-expressivity we must choose the class M to be a subclass of the class of monomorphisms that have underlying functions that are injective and R-reflecting. A function $f\colon X \to Y$ is R-reflecting if for all $x, y \in X$ if $f(x) R_Y f(y)$ then $x R_X y$.

Theorem 29. *Given an L-algebra (A, α), and if M is a subclass of the class of monomorphisms in $\mathbb{X}$ that have underlying functions that are injective and R-reflecting, then if the following hold:*

1. *every model in $\mathbf{Mod}(A, \alpha)$ factors via some model in $\mathbf{IntMod}_M(A, \alpha)$,*
2. *for every pair I_1, I_2 in $\mathbf{IntMod}_M(A, \alpha)$ there is a cospan $I_1 \to I_3 \leftarrow I_2$ in $\mathbf{IntMod}_M(A, \alpha)$,*

then (A, α) is R-expressive for $\mathbf{Mod}(A, \alpha)$.

Proof. Take any pair of models X_1 and X_2 in $\mathbf{Mod}(A,\alpha)$. Then these factor via the internal models I_1 and I_2 respectively, and by assumption there exists an internal model I_3 such that there exists a cospan $I_1 \to I_3 \leftarrow I_2$ in $\mathbf{IntMod}_M(A,\alpha)$. Thus both X_1 and X_2 factor via I_3.

Spelling this out, the models $((X_1,\gamma_1),f_1)$ and $((X_2,\gamma_2),f_2)$ factor via the internal model $((I_3,\zeta_3),m_3)$ via T-coalgebra morphisms $g_1\colon (X_1,\gamma_1)\to(I_3,\zeta_3)$ and $g_2\colon (X_2,\gamma_2)\to(I_3,\zeta_3)$ such that $f_1 = m_3\circ g_1$ and $f_2 = m_3 \circ g_2$.

Now suppose two states $x_1 \in X_1$ and $x_2 \in X_2$ are logically R-related for (A,α). Then $f_1(x_1)\, R_{S(A)}\, f_2(x_2)$, which means $m_3\circ g_1(x_1)\, R_{S(A)}\, m_3\circ g_2(x_2)$, and since m_3 is R-reflecting, $g_1(x_1)\, R_{I_3}\, g_2(x_2)$, and x_1 and x_2 are behaviourally R-related.

The converse direction is given by Proposition 27. □

Making some mild assumptions about the category $\mathbb{X}$, and using our results on colimits we have the following corollary.

Corollary 30. *If given an L-algebra (A,α) the following hold:*

1. *$\mathbb{X}$ has a factorisation system (E,M),*
2. *$\mathbb{X}$ has binary coproducts,*
3. *M is a subclass of the class of monomorphisms in $\mathbb{X}$ that have underlying functions that are injective and R-reflecting,*
4. *$m\in M \Rightarrow \delta_A^*\circ T(m)\in M$,*

then (A,α) is R-expressive for $\mathbf{Mod}(A,\alpha)$.

Proof. By Proposition 15 every model in $\mathbf{Mod}(A,\alpha)$ factors via an internal model in $\mathbf{IntMod}_M(A,\alpha)$. Also since $\mathbb{X}$ has binary coproducts, by Theorem 18 the coproduct of every pair of objects in $\mathbf{IntMod}_M(A,\alpha)$ exists. So by Theorem 29 we have the result. □

Example 31.

1. In the case where R is the type equality, then enrichment over $\mathbf{Set}_R$ is just ordinary category theory, and Corollary 30 becomes a straightforward generalisation of [7, Theorem 4.2] and [4, Theorem 4] to arbitrary L-algebras. Many examples abound, for example the obvious extension of those of [4] to logics with propositional variables and further axioms.
2. For the logical connection of Example 24 (2) we can consider a generalisation of the finite powerset functor on $\mathbf{Set}_R$ for the two cases where the type R represents preorders and equality. We define $\mathcal{P}_{\mathrm{fin}}(X,R_X) = (\mathcal{P}_{\mathrm{fin}}(X), R_{\mathcal{P}_{\mathrm{fin}}(X)})$, where if the type R represents preorders

$$UR_{\mathcal{P}_{\mathrm{fin}}(X)}V \Leftrightarrow \forall x\in U\ \exists y \in V.xR_Xy$$

Using this we define $T(X,R_X) = \mathcal{P}_{\mathrm{fin}}((\Sigma,R_\Sigma)\times(X,R_X))$, making the T-coalgebras finite branching labelled transition systems with labels from Σ. Note we assume that R_Σ is the equality relation. If the type R represents preorders, for the T-coalgebra $\gamma\colon (X,R_X)\to T(X,R_X)$, we have that xR_Xy implies for all $(l,z)\in\gamma(x)$ there exists $(l,z')\in\gamma(y)$ such that zR_Xz'. This is the usual notion that x is simulated by y. To define the modalities we make use of the forgetful functor $U\colon \mathbf{MSL}\to\mathbf{Set}_R$ and its left adjoint $F\colon \mathbf{Set}_R\to\mathbf{MSL}$ that creates free meet semilattices with a top element. Specifically, $F(X,R_X)$ is the usual free meet semilattice $F(X)$, over the set of variables X, with the relation $R_{F(X)}$ given by the equality relation extended by $[x]R_{F(X)}[y]\Leftrightarrow xR_xy$ for all $x,y\in X$. The functor L is then defined by $L(A)=F\coprod_{l\in\Sigma}U(A)$, and we choose δ as follows

$$\begin{aligned}\delta_{(X,R_X)}\colon LP(X,R_X)&\to PT(X,R_X)\\ \top_{LP(X,R_X)}&\mapsto\mathcal{P}_{\text{fin}}((\Sigma,R_\Sigma)\times(X,R_X))\\ [V_l]_{LP(X,R_X)}&\mapsto\{W\in\mathcal{P}_{\text{fin}}((\Sigma,R_\Sigma)\times(X,R_X))\mid\exists(l,x)\in W.x\in V_l\}\\ [V_{l_1}\wedge V_{l_2}]_{LP(X,R_X)}&\mapsto\delta_{(X,R_X)}(V_{l_1})\cap\delta_{(X,R_X)}(V_{l_2})\end{aligned}$$

where the notation V_l indicates that V is from the copy of $UP(X,R_X)$ indexed by l. This corresponds to a modal operator $\langle l\rangle$ for each $l\in\Sigma$, where $\langle l\rangle a$ is satisfied at a state if there is an l transition from that state to one where a is satisfied. From this we get

$$\begin{aligned}\delta^*_A\colon TS(A)&\to SL(A)\\ V&\mapsto\{[W]_{L(A)}\in L(A)\mid V\in\delta_{S(A)}\circ L(\rho_A)(W)\}\end{aligned}$$

where ρ_A, the unit of the logical connection, is given by $\rho_A(a)=\{s\in S(A)\mid a\in s\}$, and so

$$\begin{aligned}\delta_{S(A)}\circ L(\rho_A)\colon L(A)&\to PTS(A)\\ \top_{L(A)}&\mapsto\mathcal{P}_{\text{fin}}((\Sigma,R_\Sigma)\times(S(A),R_{S(A)}))\\ [a_l]_{L(A)}&\mapsto\{V\in\mathcal{P}_{\text{fin}}((\Sigma,R_\Sigma)\times(S(A),R_{S(A)}))\mid\exists(l,s)\in V.a_l\in s\}\\ [a_{l_1}\wedge a_{l_2}]_{L(A)}&\mapsto\delta_{S(A)}\circ L(\rho_A)(a_{l_1})\cap\delta_{S(A)}\circ L(\rho_A)(a_{l_2})\end{aligned}$$

where again the notation a_l indicates that a is from the copy of $U(A)$ indexed by l. Now it is quite easy to see that the category $\mathbf{Set}_R$ has a factorisation system (E,M), where E is the class of morphisms with surjective underlying functions, and M is the class of morphisms with underlying functions that are injective and R-reflecting. Further, it is also easy to show that T preserves the morphisms of M, and so by Proposition 4 what is left to show is that δ^*_A is injective and R-reflecting. To show that δ^*_A is R-reflecting suppose $V\not R_{TS(A)}V'$, then

$$V\not R_{TS(A)}V'\Leftrightarrow\exists(l,s)\in V.\,\forall(l,s')\in V'\text{ either }l\neq l'\text{ or }s\not R_{S(A)}s'$$

In the case of R representing equality we can always swap the labels V and V' so this holds. Now, our plan is to find $[a_l]_{L(A)}\in L(A)$ such that $a_l\in s$, but for all $(l',s')\in V'$ either $l\neq l'$ or $a_l\notin s'$. If there is no $(l',s')\in V'$ such that $l=l'$ then we can take $a_l=(\top_A)_l$. If that is not the case, then there is a finite set of pairs $(l, s') \in V'$ such that $s \not R_{S(A)} s'$. Now if the type R represents preorders $s \not R_{S(A)} s'$ means $s \not\subseteq s'$, whereas if R represents equality, then $s \not R_{S(A)} s'$ means $s \neq s'$. In the former case it is possible to find an element of s that is not in any of the s' (do it pairwise and then take the meet - can do this as V' is finite), but in the latter case in general it is not. Therefore if the type R represents preorders $\delta^*_A(V) \not\subseteq \delta^*_A(V')$, which means $\delta^*_A(V) \not R_{SL(A)} \delta^*_A(V')$, and thus δ^*_A is R-reflecting, but if the type R represents equality δ^*_A need not be. To show that δ^*_A is injective, suppose $V \neq V'$, and assume that there exists $(l, s) \in V$ such that $(l, s) \notin V'$. Once again we aim to find $[a_l]_{L(A)} \in L(A)$ such that $a_l \in s$, but for all $(l', s') \in V'$ either $l \neq l'$ or $a_l \notin s'$. A similar argument to that above shows that if the type R represents preorders then such an $[a_l]_{L(A)}$ can be found, and thus $\delta^*_A(V) \neq \delta^*_A(V')$, but if the type R represents equality it cannot.

Putting this together, we see that Corollary 30 gives the following results. If we take the type R to represent preorders, and if we take (A, α) to be the initial L-algebra, then the logic given by the syntax

$$\mathcal{L} \ni \phi ::= \mathrm{tt} \mid \phi \wedge \phi \mid \langle l \rangle \phi \quad \text{where } l \in \Sigma$$

is expressive for simulation of image finite labelled transition systems. However, if we take the type R to represent equality, δ^*_A need not be injective or R-reflective (which in this case is the same thing), and thus we cannot deduce expressivity. This is consistent with the famous result that Hennessy-Milner logic ($\mathcal{L}$ with negation) is expressive for bisimulation of image finite labelled transition systems. To fix the above proof to get expressivity for bisimulation would require use of the category **BA** instead of **MSL**, and ultrafilters instead of filters [4].

8 Conclusion

We have introduced internal models for a modal logic, and shown their utility for exploring properties of a logic. Indeed, it should be noted that with the exception of the proofs of the existence of colimits in $\mathbf{Mod}(A, \alpha)$ and $\mathbf{IntMod}_M(A, \alpha)$, the proofs using internal models are relatively short. Most of the structure of $\mathbf{Mod}(A, \alpha)$ is related to whether models always factor via internal models.

An examination of corollaries 20 and 30 reveals that with the exception of the exact choice of the class M, the primary difference is between requiring the base category $\mathbb{X}$ to have small coproducts or just binary coproducts. In most such categories of interest finite cocompleteness usually also means cocompleteness, and so the existence of a dual adjunction between $\mathbf{Alg}(L)$ and $\mathbf{CoAlg}(T)$ essentially amounts to every L-algebra being expressive (for bisimulation) for its class of models.

The category $\mathbf{IntMod}_M(A, \alpha)$ is not yet fully understood, and indeed, an obvious question is that, given that internal models can be thought of as generalisations of the canonical models of Kripke semantics, do internal models have anything to say about completeness? To answer this will require a systematic treatment of the different possible notions of semantic consequence that can be defined for the coalgebraic semantics of modal logics - local/global, and frame/model.

Acknowledgements

The author would like to thank Corina Cîrstea for the many interesting discussions and her invaluable guidance. Special thanks also go to Alexander Kurz for sharing an unpublished version of [6].

References

1. Adámek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories. Wiley New York, 1990.
2. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, 2001.
3. Doberkat, E.-E.: Stochastic Coalgebraic Logic. Springer Berlin Heidelberg, 2009.
4. Jacobs, B., Sokolova, A.: Exemplaric Expressivity of Modal Logics. Journal of Logic and Computation 20(5):1041–1068, 2010.
5. Kapulkin, K., Kurz, A., Velebil, J.: Expressivity of Coalgebraic Logic over Posets. CMCS 2010 Short contributions CWI Technical report SEN-1004:16–17, 2010.
6. Kapulkin, K., Kurz, A., Velebil, J.: Expressivity of Coalgebraic Logic over Posets. Unpublished extended version (2011)
7. Klin, B.: Coalgebraic modal logic beyond sets. Electronic Notes in Theoretical Computer Science 173:177–201, 2007.
8. Kupke, C., Kurz, A., Pattinson, D.: Algebraic semantics for coalgebraic logics. Electronic Notes in Theoretical Computer Science 106:219–241, 2004.
9. Kupke, C., Kurz, A., Venema, Y.: Stone coalgebras. Theoretical Computer Science 327(1–2):109–134, 2004.
10. Kupke, C., Kurz, A., Pattinson, D.: Ultrafilter extensions for coalgebras. Lecture Notes in Computer Science 3629:263–277, 2005.
11. Kurz, A., Velebil, J.: Enriched Logical Connections. Applied Categorical Structures (to appear), 2011.
12. Levy, P.: Similarity Quotients as Final Coalgebras. Lecture Notes in Computer Science 6604:27–41, 2011.
13. Pavlovic, D., Mislove, M., Worrell J.: Testing Semantics: Connecting Processes and Process Logic. Lecture Notes in Computer Science 4019:308–322, 2006.
14. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theoretical Computer Science 249(1):3–80, 2000.
15. Schröder, L., Pattinson, D.: Strong completeness of coalgebraic modal logics. Leibniz International Proceedings in Informatics 3:673–684, 2009.

Author Index