REPORTS
IN
INFORMATICS

ISSN 0333-3590

CALCO Young Researchers Workshop
CALCO-jnr 2007
Selected Papers

Magne Haveraaen, John Power, Monika Seisenberger

REPORT NO 367 **February 2008**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL
http://www.ii.uib.no/publikasjoner/texrap/pdf/2008-367.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

CALCO Young Researchers Workshop
CALCO-jnr 2007
Selected Papers

Magne Haveraaen*

Department of Informatics
University of Bergen
N-5020 Bergen
Norway

John Power†

Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom

Monika Seisenberger‡

Department of Computer Science
Swansea University
Swansea, SA2 8PP
UK

February 29, 2008

Abstract

The CALCO Young Researchers Workshop, CALCO-jnr, was a satellite event for 2nd Conference on Algebra and Coalgebra in Computer Science, August 20-24, 2007, Bergen, Norway (CALCO'07). CALCO-jnr was dedicated to presentations by PhD students and by those who completed their doctoral studies within the past few years. This report contains selected, refereed papers from the workshop.

Related URLs

- The conference
  `http://www.ii.uib.no/calco07/`
- The report
  `http://www.ii.uib.no/publikasjoner/texrap/pdf/2008-367.pdf`
- Bergen Open Research Archive (BORA)
  `https://bora.uib.no/`

---

*`http://www.ii.uib.no/~magne`

†`http://www.cs.bath.ac.uk/department/contact-department/academic-staff/dr-john-power.html`

‡`http://www.cs.swan.ac.uk/~csmona/`

CALCO Young Researchers Workshop
CALCO-jnr 2007

20 August 2007

Selected Papers

edited by

Magne Haveraaen, John Power, and Monika Seisenberger

UNIVERSITY OF BERGEN

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. The study of algebra and coalgebra relates to the data, process and structural aspects of software systems.

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). The first CALCO conference was held in Swansea, Wales, in 2005; the second took place in Bergen, Norway, in 2007.

The CALCO Young Researchers Workshop, CALCO-jnr, was a CALCO 2007 satellite event dedicated to presentations by PhD students and by those who completed their doctoral studies within the past few years. Attendance at the workshop was open to all – many CALCO conference participants attended the CALCO-jnr workshop and vice versa. In total, CALCO-jnr 2007 had 12 contributions, by authors from 8 countries and 16 different institutions, – and over 40 participants.

CALCO-jnr presentations were, on the basis of submitted 2-page abstracts, selected by the CALCO-jnr PC. After the workshop, the authors 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 was organised and the final selection of papers was carried out by the CALCO-jnr PC. The volume of selected papers from the workshop is published as a Department of Informatics, University of Bergen, technical report, and it is available through the open access database http://bora.uib.no/. Authors will retain copyright, and are also encouraged to disseminate the results reported at CALCO-jnr by subsequent publication elsewhere.

The CALCO-jnr PC would like to thank the workshop participants, the reviewers, and the CALCO 2007 local organisers for their efforts to make this event a success. The support of all sponsoring institutions is gratefully acknowledged: Department of Informatics, University of Bergen, Bergen University College, The Research Council of Norway, City of Bergen, and IFIP WG1.3 on Foundations of System Specification.

February 2008

Magne Haveraaen
John Power
Monika Seisenberger

Table of Contents

The Microcosm Principle and Concurrency in Coalgebra

Ichiro Hasuo[1,3,4], Bart Jacobs[1,⋆], and Ana Sokolova[2,⋆⋆]

[1] Inst. for Computing and Information Sciences, Radboud University Nijmegen
Postbus 9010, 6500GL Nijmegen, the Netherlands
E-mail: {ichiro,bart}@cs.ru.nl
[2] Dept. of Computer Sciences, University of Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
E-mail: anas@cs.uni-salzburg.at
[3] Research Inst. for Mathematical Sciences, Kyoto University
Kitashirakawa-Oiwakecho, Kyoto 606-8502, Japan
[4] PRESTO Research Promotion Program, Japan Science and Technology Agency

Abstract. Coalgebras are categorical presentations of state-based systems. In investigating parallel composition of coalgebras (realizing *concurrency*), we observe that the same algebraic theory is interpreted in two different domains in a nested manner, namely: in the category of coalgebras, and in the final coalgebra as an object in it. This phenomenon is what Baez and Dolan have called the *microcosm principle*, a prototypical example of which is "a monoid in a monoidal category." In this paper we obtain a formalization of the microcosm principle in which such a nested model is expressed categorically as a suitable lax natural transformation. An application of this account is a general compositionality result which supports modular verification of complex systems.

1 Introduction

Design of systems with *concurrency* is nowadays one of the mainstream challenges in computer science [19]. Concurrency is everywhere: with the Internet being the biggest example and multi-core processors the smallest; also in a modular, component-based architecture of a complex system its components collaborate in a concurrent manner. However numerous difficulties have been identified in getting concurrency right. For example, a system's exponentially growing complexity is one of the main obstacles. One way to cope with it is a *modular* verification method in which correctness of the whole system $\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n$ is established using correctness of each component $\mathcal{C}_i$. *Compositionality*—meaning that behavior of $\mathcal{C} \parallel \mathcal{D}$ is determined by behavior of $\mathcal{C}$ and that of $\mathcal{D}$—is an essential property for such a modular method to work.

⋆ Also part-time at Technical University Eindhoven, the Netherlands.
⋆⋆ Supported by the Austrian Science Fund (FWF) project no. P18913-N15. During the work on this paper A.S. was employed at Radboud University Nijmegen.

Coalgebras as systems This paper is a starting point of our research program aimed at better understanding of the mathematical nature of concurrency. In its course we shall use *coalgebras* as presentations of systems to be run in parallel. The use of coalgebras as an appropriate abstract model of state-based systems is increasingly established [11,26]; the notion's mathematical simplicity and clarity provide us with a sound foundation for our exploration. The following table summarizes how ingredients of the theory of systems are presented as coalgebraic constructs.

	system		behavior-preserving map	behavior
			morphism of coalgebras	by coinduction
coalgebraically	coalgebra	$\begin{array}{c} FX \\ \uparrow \\ X \end{array}$	$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \uparrow & & \uparrow \\ X & \xrightarrow[f]{} & Y \end{array}$	$\begin{array}{ccc} FX & - - \to & FZ \\ {\scriptstyle c}\uparrow & & {\scriptstyle \text{final}}\uparrow\cong \\ X & \xrightarrow[\mathsf{beh}(c)]{} & Z \end{array}$

(1)

This view of "coalgebras as systems" has been successfully applied in the category $\mathbf{Sets}$ of sets and functions, in which case the word "behavior" in (1) refers (roughly) to bisimilarity. Our recent work [5, 6] has shown that "behavior" can also refer to trace semantics by moving from $\mathbf{Sets}$ to a suitable Kleisli category.

Compositionality in coalgebras We start with the following question: what is "compositionality" in this coalgebraic setting? Conventionally compositionality is expressed as: $\mathcal{C} \sim \mathcal{C}'$ and $\mathcal{D} \sim \mathcal{D}'$ implies $\mathcal{C} \parallel \mathcal{D} \sim \mathcal{C}' \parallel \mathcal{D}'$, where the relation $\sim$ denotes the behavioral equivalence of interest. If this is the case the relation $\sim$ is said to be a *congruence*, with its oft-heard instance being "bisimilarity is a congruence."

When we interpret "behavior" in compositionality as the coalgebraic behavior induced by coinduction (see (1)), the following equation comes natural as a coalgebraic presentation of compositionality.

$$\mathsf{beh}\left(\begin{array}{c} FX \\ c\uparrow \\ X \end{array} \;\middle\|\; \begin{array}{c} FY \\ d\uparrow \\ Y \end{array}\right) = \mathsf{beh}\left(\begin{array}{c} FX \\ c\uparrow \\ X \end{array}\right) \;\middle\|\; \mathsf{beh}\left(\begin{array}{c} FY \\ d\uparrow \\ Y \end{array}\right) \tag{2}$$

But a closer look reveals that the two "parallel composition operators" $\parallel$ in the equation have in fact different types: the first one $\mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$ combines systems (as coalgebras) and the second one $Z \times Z \to Z$ combines behavior (as states of the final coalgebra).[5] Moreover, the two domains are actually nested: the latter one $Z \xrightarrow{\cong} FZ$ is an object of the former one $\mathbf{Coalg}_F$.

The microcosm principle What we have just observed is one instance—probably the first one explicitly claimed in computer science—of the *microcosm principle* as it is called by Baez and Dolan [1]. It refers to a phenomenon that the same algebraic theory (or algebraic "specification," consisting of operations and equations) is interpreted twice in a nested manner, once in a category $\mathbb{C}$ and the other time in its object $X \in \mathbb{C}$. This

[5] At this stage the presentation remains sloppy for the sake of simplicity. Later in technical sections the first composition operator will be denoted by $\circledast$; and the second composition operator will have the type $Z \otimes Z \to Z$ instead of $Z \times Z \to Z$.

is not something very unusual, because "a monoid in a monoidal category" constitutes a prototypical example.

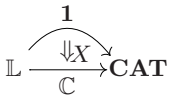
monoidal category $\mathbb{C}$		monoid $X \in \mathbb{C}$
$\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$	multiplication	$X \otimes X \xrightarrow{\mu} X$
$I \in \mathbb{C}$	unit	$I \xrightarrow{\eta} X$
$I \otimes X \cong X \cong X \otimes I$	unit law	$X \to X \otimes X \leftarrow X$, $X \to X \leftarrow X$
$X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z$	associativity law	$X \otimes X \otimes X \to X \otimes X$, $X \otimes X \to X$

(3)

Notice here that the outer operation $\otimes$ appears in the formulation of the an inner operation μ. Moreover, to be precise, in the inner "equations" the outer isomorphisms should be present in suitable places. Hence this monoid example demonstrates that, in such nested algebraic structures, the inner structure depends on the outer. What is a mathematically precise formalization of such nested models? Answering this question is a main goal of this paper.

Such a formalization has been done in [1] when algebraic structures are specified in the form of *opetopes*. Here instead we shall formalize the microcosm principle for *Lawvere theories* [18], whose role as categorical representation of algebraic theories has been recognized in theoretical computer science.

As it turns out, our formalization looks like the situation on the right. Here $\mathbb{L}$ is a category (a Lawvere theory) representing an algebraic theory; an outer model $\mathbb{C}$ is a product-preserving functor; and an inner model X is a lax natural transformation. The whole setting is 2-categorical: 2-categories (categories in categories) serve as an appropriate basis for the microcosm principle (algebras in algebras).



Applications to coalgebras: parallel composition via sync The categorical account we have sketched above shall be applied to our original question about parallel composition of coalgebras. As a main application we prove a *generic compositionality theorem*. For an arbitrary algebraic theory $\mathbb{L}$, compositionality like (2) is formulated as follows: the "behavior" functor $\mathsf{beh} : \mathbf{Coalg}_F \to \mathbb{C}/Z$ via coinduction preserves an $\mathbb{L}$-structure. This general form of compositionality holds if: $\mathbb{C}$ has an $\mathbb{L}$-structure and $F : \mathbb{C} \to \mathbb{C}$ *lax*-preserves the $\mathbb{L}$-structure.

Turning back to the original setting of (2), these general assumptions read roughly as follows: the base category $\mathbb{C}$ has a binary operation $\|$; and the endofunctor F comes with a natural transformation $\mathsf{sync} : FX \parallel FY \to F(X \parallel Y)$. Essentially, this sync is what lifts $\|$ on $\mathbb{C}$ to $\|$ on $\mathbf{Coalg}_F$, hence "parallel composition via sync." It is called a *synchronization* because it specifies the way two systems synchronize with each other. In fact, for a fixed functor F there can be different choices of sync (such as CSP-style vs. CCS-style), which in turn yield different "parallel composition" operators on the category $\mathbf{Coalg}_F$.

Related work Our interest is pretty similar to that of studies of *bialgebraic structures* in computer science (such as [3, 12, 14–16, 27]), in the sense that we are also concerned about algebraic structures on coalgebras as systems. Our current framework is distinguished in the following aspects.

First, we handle *equations* in an algebraic theory as an integral part of our approach. Equations such as associativity and commutativity appear explicitly as commutative diagrams in a Lawvere theory $\mathbb{L}$. We benefit from this explicitness in e.g. spelling out a condition for the generic associativity result (Theorem 2.4). In contrast, in the bialgebraic studies an algebraic theory is presented either by an endofunctor $X \mapsto \coprod_{\sigma\in\Sigma} X^{|\sigma|}$ or by a monad T. In the former case equations are simply not present; in the latter case equations are there but only implicitly.

Secondly and more importantly, by considering higher-dimensional, nested algebraic structures, we can now compose different coalgebras as well as different states of the same coalgebra. In this way the current work can be seen as a higher-dimensional extension of the existing bialgebraic studies (which focus on "inner" algebraic structures).

Organization of the paper We shall not dive into our 2-categorical exploration from the beginning. In Section 2, we instead focus on one specific algebraic theory, namely the one for parallel composition of systems. Our emphasis there is on the fact that the sync natural transformation essentially gives rise to parallel composition $\|$, and the fact that equational properties of $\|$ (such as associativity) can be reduced to the corresponding equational properties of sync.

These concrete observations will provide us with intuition for abstract categorical constructs in Section 3, where we formalize the microcosm principle for an arbitrary Lawvere theory $\mathbb{L}$. Results on coalgebras such as compositionality are proved here in their full generality and abstraction.

In this paper we shall focus on *strict* algebraic structures on categories in order to avoid complicated coherence issues. This means for example that we only consider *strict* monoidal categories for which the isomorphisms in (3) are in fact equalities. However, we have also obtained some preliminary observations on relaxed ("pseudo" or "strong") algebraic structures: see Section 3.3.

2 Parallel composition of coalgebras

2.1 Parallel composition via sync natural transformation

Let us start with the equation (2), a coalgebraic representation of compositionality. The operator $\|$ on the left is of type $\mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$. It is natural to require functoriality of this operation, making it a *bifunctor*. A bifunctor—especially an associative one which we investigate in Section 2.3—plays an important role in various applications of category theory. Usually such an (associative) bifunctor is called a *tensor*; we follow this convention and denote it by $\circledast$. Therefore the "compositionality"

statement now looks as follows.[6]

$$\mathsf{beh}\begin{pmatrix} FX \\ c\!\uparrow \\ X \end{pmatrix} \circledast \begin{matrix} FY \\ d\!\uparrow \\ Y \end{matrix} = \mathsf{beh}\begin{pmatrix} FX \\ c\!\uparrow \\ X \end{pmatrix} \Big\| \mathsf{beh}\begin{pmatrix} FY \\ d\!\uparrow \\ Y \end{pmatrix} \tag{4}$$

The first question is: when do we have such a tensor $\circledast$ on $\mathbf{Coalg}_F$? In many applications of coalgebras, it is obtained by lifting a tensor $\otimes$ on the base category $\mathbb{C}$ to $\mathbf{Coalg}_F$.[7] Such a lifting is possible in presence of a natural transformation

$$FX \otimes FY \overset{\mathsf{sync}_{X,Y}}{\longrightarrow} F(X \otimes Y), \quad \text{used in} \quad \begin{matrix} FX \\ c\!\uparrow \\ X \end{matrix} \circledast \begin{matrix} FY \\ d\!\uparrow \\ Y \end{matrix} := \begin{matrix} F(X \otimes Y) \\ \uparrow \mathsf{sync}_{X,Y} \\ FX \otimes FY \\ \uparrow c \otimes d \\ X \otimes Y \end{matrix} . \tag{5}$$

We shall call this sync a *synchronization* because its computational meaning is indeed a specification of the way two systems synchronize. This will be illustrated in the coming examples.

Once we have an outer parallel composition $\circledast$, an inner operator $\|$ which composes behavior (i.e. states of the final coalgebra) is also obtained immediately by coinduction as on the right. Compositionality (4) is also straightforward by finality: both sides of the equation are the unique coalgebra morphism from $c \circledast d$ to the final ζ. The following theorem summarizes the observations so far.

$$\begin{array}{ccc} F(Z \otimes Z) & \dashrightarrow & FZ \\ {\scriptstyle \zeta \circledast \zeta}\uparrow & & \uparrow{\scriptstyle \text{final}} \zeta \\ Z \otimes Z & \underset{\|}{\dashrightarrow} & Z \end{array}$$

Theorem 2.1 (Coalgebraic compositionality) *Assume that a category $\mathbb{C}$ has a tensor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ and an endofunctor $F : \mathbb{C} \to \mathbb{C}$ has a natural transformation $\mathsf{sync}_{X,Y} : FX \otimes FY \to F(X \otimes Y)$. If moreover there exists a final F-coalgebra, then:*

1. *The tensor $\otimes$ on $\mathbb{C}$ lifts to an "outer" composition operator $\circledast : \mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$.*
2. *We obtain an "inner" composition operator $\|$: $Z \otimes Z \to Z$ by coinduction.*
3. *Between the two composition operators the compositionality property (4) holds.*

□

We can put the compositionality property (4) in more abstract terms as "the functor $\mathsf{beh} : \mathbf{Coalg}_F \to \mathbb{C}/Z$ preserves a tensor," meaning that the diagram below left commutes. Here a tensor $\underline{\otimes}$ on the slice category $\mathbb{C}/Z$ is given as on the right, using the inner composition $\|$.

$$\begin{array}{ccc} \mathbf{Coalg}_F \times \mathbf{Coalg}_F & \xrightarrow{\mathsf{beh} \times \mathsf{beh}} & \mathbb{C}/Z \times \mathbb{C}/Z \\ {\scriptstyle \circledast}\downarrow & & \downarrow{\scriptstyle \underline{\otimes}} \\ \mathbf{Coalg}_F & \xrightarrow[\mathsf{beh}]{} & \mathbb{C}/Z \end{array} \qquad \begin{pmatrix} X & Y \\ \downarrow f & \downarrow g \\ Z & Z \end{pmatrix} \overset{\underline{\otimes}}{\longmapsto} \begin{matrix} X \otimes Y \\ \downarrow f \otimes g \\ Z \otimes Z \\ \downarrow \| \\ Z \end{matrix} \tag{6}$$

[6] Strictly speaking the presentation (4) is still sloppy. Since each side of the equation is of the type $X \otimes Y \to Z$, to be precise the right-hand side should be denoted by $\| \circ (\mathsf{beh}(c) \otimes \mathsf{beh}(d))$.

[7] Note that we use the symbol $\circledast$ for a tensor on $\mathbf{Coalg}_F$ while reserving a more standard tensor symbol $\otimes$ for a tensor on the base category $\mathbb{C}$.

The point of Theorem 2.1 is as follows. Those parallel composition operators which are induced by sync are well-behaved ones: good properties like compositionality come for free. We shall present some examples in Section 2.2.

Remark 2.2 The view of parallel composition of systems as a tensor structure on $\mathbf{Coalg}_F$ has been previously presented in [13]. The interest there is on categorical structures on $\mathbf{Coalg}_F$ rather than on properties of parallel composition such as compositionality. In [13] and other literature an endofunctor F with sync (equipped with some additional compatibility) is called a *monoidal endofunctor*.[8]

2.2 Examples

In $\mathbf{Sets}$: bisimilarity is a congruence We shall focus on LTSs and bisimilarity as their process semantics. For this purpose it is appropriate to take $\mathbf{Sets}$ as our base category $\mathbb{C}$ and $\mathcal{P}_\omega(\Sigma \times \_)$ as the functor F. We use Cartesian products as a tensor on $\mathbf{Sets}$. This means that a composition of two coalgebras has the product of the two state spaces as its state space, which matches our intuition. The functor $\mathcal{P}_\omega$ in F is the finite powerset functor; the finiteness assumption is needed for existence of a final F-coalgebra. It is standard (see e.g. [26]) that a final F-coalgebra captures bisimilarity via coinduction.

In considering parallel composition of LTSs, the following two examples are well-known ones.[9]

- *CSP-style* [7]: $a.P \parallel a.Q \xrightarrow{a} P \parallel Q$. For the whole system to make an a-action, each component has to make an a-action.
- *CCS-style* [21]: $a.P \parallel \overline{a}.Q \xrightarrow{\tau} P \parallel Q$, assuming $\Sigma = \{a, b, \dots\} \cup \{\overline{a}, \overline{b}, \dots\} \cup \{\tau\}$. When one component outputs on a channel a and another inputs from a, then the whole system makes an internal τ move.

In fact, each of these different ways of synchronization can be represented by a suitable sync natural transformation.

$$\begin{array}{ccc} \mathcal{P}_\omega(\Sigma \times X) \times \mathcal{P}_\omega(\Sigma \times Y) & \longrightarrow & \mathcal{P}_\omega\big(\Sigma \times (X \times Y)\big) \\ (u, v) & \overset{\mathsf{sync}^{\mathrm{CSP}}_{X,Y}}{\longmapsto} & \big\{\, (a, (x, y)) \mid (a, x) \in u \wedge (a, y) \in v \,\big\} \\ (u, v) & \overset{\mathsf{sync}^{\mathrm{CCS}}_{X,Y}}{\longmapsto} & \big\{\, (\tau, (x, y)) \mid (a, x) \in u \wedge (\overline{a}, y) \in v \,\big\} \end{array}$$

By Theorem 2.1, each of these gives (different) $\circledast$ on $\mathbf{Coalg}_F$, and $\parallel$ on Z; moreover the behavior functor beh satisfies compositionality. Since the current behavior functor beh (induced by coinduction in $\mathbf{Sets}$) gives behavior modulo bisimilarity, this specific instance of Theorem 2.1 reads: bisimilarity is a congruence with respect to both CSP-style and CCS-style parallel composition.

[8] Later in Section 3 we will observe that a functor F with sync is a special case of a *lax* $\mathbb{L}$-functor, by choosing a suitable algebraic theory $\mathbb{L}$. Such a functor F with sync is usually called a monoidal functor (as opposed to a *lax* monoidal functor), probably because it preserves (inner) monoid objects; see Proposition 3.8.1.

[9] Here we focus on synchronous interaction. Both CSP and CCS have an additional kind of interaction, namely an "interleaving" one; see Remark 2.3.

Remark 2.3 As mentioned in the introduction, in some ways this paper can be seen as an extension of the bialgebraic studies started in [27]. However there is also a drawback, namely the limited expressive power of $\mathsf{sync} : FX \otimes FY \to F(X \otimes Y)$.

Our sync specifies the way an algebraic structure interacts with a coalgebraic one. In this sense it is a counterpart of a distributive law $\Sigma F \Rightarrow F\Sigma$ in [27] representing operational rules, where Σ is a functor induced by an algebraic signature. However there are many common operational rules which do not allow representation of the form $\Sigma F \Rightarrow F\Sigma$; therefore in [27] the type of such a distributive law is eventually extended to $\Sigma(F \times \mathrm{id}) \Rightarrow F\Sigma^*$. The class of rules representable in this form coincides with the class of so-called *GSOS-rules*.

At present it is not clear how we can make a similar extension for our sync; consequently there are some operational rules which we cannot model by sync. One important example is an *interleaving* kind of interaction—such as $a.P \parallel Q \xrightarrow{a} P \parallel Q$ which leaves the second component unchanged. This is taken care of in [27] by the identity functor (id) appearing on the left-hand side of $\Sigma(F \times \mathrm{id}) \Rightarrow F\Sigma^*$. For our sync to be able to model such interleaving, we can replace F by the cofree comonad on it, as is done in [13, Example 3.11]. This extension should be straightforward but detailed treatment is left as future work.

In $\mathcal{K}\ell(T)$: trace equivalence is a congruence In our recent work [6] we extend earlier observations in [10, 25] and show that trace semantics—including trace *set* semantics for non-deterministic systems and trace *distribution* semantics for probabilistic systems—is also captured by coinduction when it is employed in a Kleisli category $\mathcal{K}\ell(T)$. Applying the present composition framework, we can conclude that trace semantics is compositional with respect to well-behaved parallel composition. The details are omitted here due to lack of space.

2.3 Equational properties of parallel composition

Now we shall investigate equational properties—associativity, commutativity, and so on—of parallel composition $\circledast$, which we have ignored deliberately for simplicity of argument. We present our result in terms of associativity; it is straightforward to transfer the result to other properties like commutativity. The main point of the following theorem is as follows: if $\otimes$ is associative and sync is "associative," then the lifting $\circledast$ is associative. The proof is straightforward.

Theorem 2.4 *Let $\mathbb{C}$ be a category with a strictly associative tensor $\otimes$,*[10] *and $F : \mathbb{C} \to \mathbb{C}$ be a functor with $\mathsf{sync} : FX \otimes FY \to F(X \otimes Y)$. If the diagram*

$$\begin{array}{ccccc} FX \otimes (FY \otimes FZ) & \xrightarrow{FX \otimes \mathsf{sync}} & FX \otimes F(Y \otimes Z) & \xrightarrow{\mathsf{sync}} & F(X \otimes (Y \otimes Z)) \\ \downarrow \mathrm{id} & & & & \downarrow \mathrm{id} \\ (FX \otimes FY) \otimes FZ & \xrightarrow[\mathsf{sync} \otimes FZ]{} & F(X \otimes Y) \otimes FZ & \xrightarrow[\mathsf{sync}]{} & F((X \otimes Y) \otimes Z) \end{array} \tag{7}$$

commutes, then the lifted tensor $\circledast$ on $\mathbf{Coalg}_F$ is strictly associative. □

[10] As mentioned already, in this paper we stick to *strict* algebraic structures.

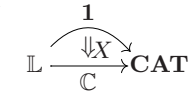
The two identity arrows in (7) are available due to strict associativity of $\otimes$. In the next section we shall reveal the generic principle behind the commutativity condition of (7), namely a coherence condition on a lax natural transformation.

As an example, $\mathsf{sync}^{\mathrm{CSP}}$ and $\mathsf{sync}^{\mathrm{CCS}}$ in Section 2.2 are easily seen to be "associative" in the sense of the diagram (7). Therefore the resulting tensors $\circledast$ are strictly associative.

3 Formalizing the microcosm principle

In this section we shall formalize the microcosm principle for an arbitrary algebraic theory presented as a Lawvere theory $\mathbb{L}$. This and the subsequent results generalize the results in the previous section. In particular, we will obtain a general compositionality result which works for an arbitrary algebraic theory.

As we sketched in the introduction, an outer model will be a product-preserving functor $\mathbb{C} : \mathbb{L} \to \mathbf{CAT}$; an inner model inside will be a lax natural transformation $X : \mathbf{1} \Rightarrow \mathbb{C}$. Here $\mathbf{1} : \mathbb{L} \to \mathbf{CAT}$ is the constant functor which maps everything to the category $\mathbf{1}$ with one object and one arrow (which is a special case of an outer model). Mediating 2-cells for the *lax* natural transformation X play a crucial role as inner interpretation of algebraic operations. In this section we heavily rely on 2-categorical notions, about which detailed accounts can be found in [4].



3.1 Lawvere theories

Lawvere theories are categorical presentations of algebraic theories. The notion is introduced in [18] (not under this name, though) aiming at a categorical formulation of "theories" and "semantics." An accessible introduction to the notion can be found in [17]. Lawvere theories are known to be equivalent to *finitary monads*. These two ways of presenting algebraic theories have been widely used in theoretical computer science, e.g. for modeling computation with effect [8, 22]. Recent developments (such as [24]) utilize the increased expressive power of *enriched* Lawvere theories.

In the sequel, by an *FP-category* we refer to a category with (a choice of) finite products. An *FP-functor* is a functor between FP-categories which preserves finite products "on-the-nose," that is, up-to-equality instead of up-to-isomorphism.

Definition 3.1 (Lawvere theory) By $\mathbf{Nat}$ we denote the category of natural numbers (as sets) and functions between them. Therefore every arrow in $\mathbf{Nat}$ is a (cotuple of) coprojection; an arrow in $\mathbf{Nat}^{\mathrm{op}}$ is a (tuple of) projection.[11]

A *Lawvere theory* is a small FP-category $\mathbb{L}$ equipped with an FP-functor $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ which is bijective on objects. We shall denote an object of $\mathbb{L}$ by a natural number k, identifying $k \in \mathbf{Nat}^{\mathrm{op}}$ and $Hk \in \mathbb{L}$.

[11] An arrow $f : n \to k$ in $\mathbf{Nat}$ can be written as a cotuple $[\kappa_{f(1)}, \ldots, \kappa_{f(n)}]$ where $\kappa_i : 1 \to k$ is the coprojection into the i-th summand of $1 + \cdots + 1$ (k times).

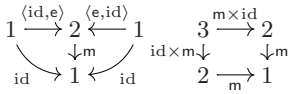
The category $\mathbf{Nat}^{\mathrm{op}}$—which is a free FP-category on the trivial category $\mathbf{1}$—is there in order to specify the choice of finite products in $\mathbb{L}$. For illustration, we make some remarks on $\mathbb{L}$'s objects and arrows.

- An object $k \in \mathbb{L}$ is a k-fold product $1 \times \cdots \times 1$ of 1.
- An arrow in $\mathbb{L}$ is intuitively understood as an algebraic operation. That is, $k \to 1$ as an k-ary operation; and $k \to n$ as an n-tuple $\langle \mathsf{f}_1, \ldots, \mathsf{f}_n \rangle$ of k-ary operations. To be precise, arrows in $\mathbb{L}$ also include projections (such as $\pi_1 : 2 \to 1$) and *terms* made up of operations and projections (such as $\mathsf{m} \circ \langle \pi_1, \pi_2 \rangle : 3 \to 1$).

Conventionally in universal algebra, an algebraic theory is presented by an *algebraic specification* (Σ, E)—a pair of a set Σ of operations and a set E of equations. A Lawvere theory $\mathbb{L}$ arises from such (Σ, E) as its so-called *classifying category* (e.g. [9,18]). An arrow $k \to n$ in the resulting Lawvere theory $\mathbb{L}$ is an n-tuple $([t_1(\overrightarrow{x})], \ldots, [t_n(\overrightarrow{x})])$ of Σ-terms with k variables $\overrightarrow{x}$, where $[\_]$ denotes taking an equivalence class modulo equations in E.

Our leading example is the Lawvere theory $\mathbf{Mon}$ for monoids.[12] It arises as a classifying category from the well-known algebraic specification of monoids. This specification has a nullary operation e and a binary one m; subject to the equations $\mathsf{m}(x, \mathsf{e}) = x$, $\mathsf{m}(\mathsf{e}, x) = x$, and $\mathsf{m}(x, \mathsf{m}(y, z)) = \mathsf{m}(\mathsf{m}(x, y), z)$.

Equivalently, $\mathbf{Mon}$ is the freely generated FP-category by arrows $0 \xrightarrow{\mathsf{e}} 1$ and $2 \xrightarrow{\mathsf{m}} 1$ subject to the commutativity on the right. These data (arrows and commutative diagrams) form an *FP-sketch* (see [2]).



3.2 Outer models: $\mathbb{L}$-categories

We start by formalizing an outer model. It is a category with an $\mathbb{L}$-structure, hence called an $\mathbb{L}$-*category*. It is standard that a (set-theoretic) model of $\mathbb{L}$—a *set* with an $\mathbb{L}$-structure—is identified with an FP-functor $\mathbb{L} \xrightarrow{X} \mathbf{Sets}$. Concretely, let $X = X1$ be the image of $1 \in \mathbb{L}$. Then $k \in \mathbb{L}$ must be sent to X^k due to preservation of finite products. Now the functor's action on arrows is what interprets $\mathbb{L}$'s operations in X, as illustrated above right. Equations (expressed as commutative diagrams in $\mathbb{L}$) are satisfied because a functor preserves commutativity.

$$\begin{array}{ccc} \mathbb{L} & \xrightarrow{X} & \mathbf{Sets} \\ 2 & \longmapsto & X^2 \\ \downarrow \mathsf{m} & & \downarrow [\![\mathsf{m}]\!] \\ 1 & & X \end{array}$$

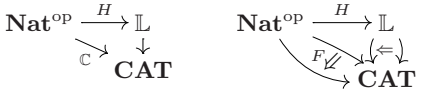
Turning back to $\mathbb{L}$-*categories*, what we have to do here is to just replace $\mathbf{Sets}$ by the category $\mathbf{CAT}$ of (possibly large and locally small) categories.

Definition 3.2 ($\mathbb{L}$-categories, $\mathbb{L}$-functors) A (strict) $\mathbb{L}$-*category* is an FP-functor $\mathbb{L} \xrightarrow{\mathbb{C}} \mathbf{CAT}$. In the sequel we denote the image $\mathbb{C}1$ of $1 \in \mathbb{L}$ by $\mathbb{C}$; and the image $\mathbb{C}(\mathsf{f})$ of an arrow f by $[\![\mathsf{f}]\!]$.

[12] The Lawvere theory $\mathbf{Mon}$ for the theory of monoids should not be confused with the category of (set-theoretic) monoids and monoid homomorphisms (which is often denoted by $\mathbf{Mon}$ as well).

An $\mathbb{L}$-*functor* $F : \mathbb{C} \to \mathbb{D}$—a functor preserving an $\mathbb{L}$-structure—is a natural transformation $\mathbb{L} \overset{\mathbb{C}}{\underset{\mathbb{D}}{\Downarrow F}} \mathbf{CAT}$.

Another way to look at the previous definition is to view an $\mathbb{L}$-structure as "factorization through $\mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$." We can identify a category $\mathbb{C} \in \mathbf{CAT}$ with a functor $\mathbf{1} \to \mathbf{CAT}$, which is in turn identified with an FP-functor $\mathbf{Nat}^{\mathrm{op}} \to \mathbf{CAT}$, because $\mathbf{Nat}^{\mathrm{op}}$ is the free FP-category on $\mathbf{1}$. We say that $\mathbb{C}$ has an $\mathbb{L}$-structure, if this FP-functor factors through $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ (as below left). Note that the factorization is not necessarily unique, because there can be different ways of interpreting the algebraic theory $\mathbb{L}$ in $\mathbb{C}$. Similarly, a functor $\mathbb{C} \overset{F}{\to} \mathbb{D}$ is identified with a natural transformation $\mathbf{1} \overset{\Downarrow F}{\Rightarrow} \mathbf{CAT}$; and then with $\mathbf{Nat}^{\mathrm{op}} \overset{\Downarrow F}{\Rightarrow} \mathbf{CAT}$ due to the 2-universality of $\mathbf{Nat}^{\mathrm{op}}$ as a free object. We say that this F preserves an $\mathbb{L}$-structure, if the last natural transformation factors through $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ (as below right).

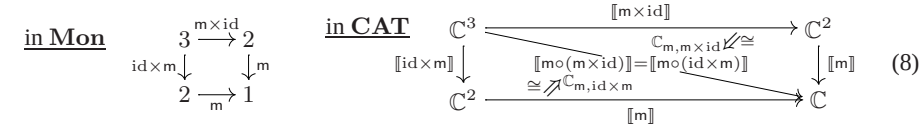


Example 3.3 The usual notion of strictly monoidal categories coincides with $\mathbb{L}$-categories for $\mathbb{L} = \mathbf{Mon}$. A tensor $\otimes$ and a unit I on a category arise as interpretation of the operations $2 \overset{\mathsf{m}}{\to} 1$ and $0 \overset{\mathsf{e}}{\to} 1$; commuting diagrams in $\mathbf{Mon}$ such as $\mathsf{m} \circ \langle \mathrm{id}, \mathsf{e} \rangle = \mathrm{id}$ yield equational properties of $\otimes$ and I.

3.3 Remarks on "pseudo" algebraic structures

As we mentioned in the introduction, in this paper we focus on *strict* algebraic structures. This means that monoidal categories (in which associativity holds only up-to-isomorphism, for example) fall out of our consideration. Extending our current framework to such "pseudo" algebraic structures is one important direction of our future work. Such an extension is not entirely obvious; we shall sketch some preliminary observations in this direction.

The starting point is to relax the definition of $\mathbb{L}$-categories from (strict) functors $\mathbb{L} \to \mathbf{CAT}$ to *pseudo* functors, meaning that composition and identities are preserved only up-to-isomorphism. Then it is not hard to see that a pseudo functor $\mathbf{Mon} \overset{\mathbb{C}}{\to} \mathbf{CAT}$ (which preserves finite products in a suitable sense) gives rise to a monoidal category. Indeed, let us denote a mediating iso-2-cell for composition by $\mathbb{C}_{\mathsf{g},\mathsf{f}} : [\![\mathsf{g}]\!] \circ [\![\mathsf{f}]\!] \overset{\cong}{\Rightarrow} [\![\mathsf{g} \circ \mathsf{f}]\!]$. The associativity diagram (below left) gives rise to the two iso-2-cells on the right.



(8)

The composition $\mathbb{C}^{-1}_{\mathsf{m},\mathrm{id}\times\mathsf{m}} \bullet \mathbb{C}_{\mathsf{m},\mathsf{m}\times\mathrm{id}}$ is what gives us a natural isomorphism $\alpha : X \otimes (Y \otimes Z) \overset{\cong}{\Rightarrow} (X \otimes Y) \otimes Z$. Moreover, the coherence condition on such isomorphisms

in a monoidal category (like the famous pentagon diagram; see [20]) follows from the coherence condition on mediating 2-cells of a pseudo functor (see [4]).

So far so good. However, at this moment it is not clear what is a canonical construction the other way round, i.e. from a monoidal category to a pseudo functor.[13] In the present paper we side-step these 2-categorical subtleties by restricting ourselves to strict, non-pseudo functors.

3.4 Inner models: $\mathbb{L}$-objects

We proceed to formalize an inner model. It is an object in an $\mathbb{L}$-category which itself carries an (inner) $\mathbb{L}$-structure, hence is called an $\mathbb{L}$-*object*. A monoid object in a monoidal category is a prototypical example. We first present an abstract definition; some illustration follows afterwards.

Definition 3.4 ($\mathbb{L}$-objects) An $\mathbb{L}$-*object* X in an $\mathbb{L}$-category $\mathbb{C}$ is a lax natural transformation $X : \mathbf{1} \Rightarrow \mathbb{C}$ (below left) which is "product-preserving": this means that the composition $X \circ H$ (below right) is strictly, non-lax natural. Here $\mathbf{1} : \mathbb{L} \to \mathbf{CAT}$ denotes the constant functor to the trivial one-object category $\mathbf{1}$.

$$\mathbb{L} \xrightarrow[\mathbb{C}]{\mathbf{1} \;\Downarrow X} \mathbf{CAT} \qquad \mathbf{Nat}^{\mathrm{op}} \xrightarrow{H} \mathbb{L} \xrightarrow[\mathbb{C}]{\mathbf{1} \;\Downarrow X} \mathbf{CAT}$$

Such a nested algebraic structure—formalized as an $\mathbb{L}$-object in an $\mathbb{L}$-category—shall be called a *microcosm model* for $\mathbb{L}$.

Let us now illustrate the definition. First, X's component at $1 \in \mathbb{L}$ is a functor $\mathbf{1} \xrightarrow{X_1} \mathbb{C}$ which is identified with an object $X \in \mathbb{C}$. This is the "carrier" object of this inner algebra. Moreover, any other component $\mathbf{1} \xrightarrow{X_k} \mathbb{C}^k$ must be the k-tuple $(X, \ldots, X) \in \mathbb{C}^k$ of X's. This is because of (strict) naturality of $X \circ H$ (see above right): for any $i \in [1, k]$ the composite $\pi_i \circ X_k$ is required to be X_1.

in $\mathbf{Nat}^{\mathrm{op}}$	in $\mathbf{CAT}$
k	$\mathbf{1} \xrightarrow{X_k=(X,\ldots,X)} \mathbb{C}^k$
$\downarrow \pi_i$	$\downarrow \llbracket H\pi_i \rrbracket = \pi_i$
1	$\mathbf{1} \xrightarrow[X_1=X]{} \mathbb{C}$

The (inner) algebraic structure on X arises in the form of mediating 2-cells of the *lax* natural transformation. For each arrow $k \xrightarrow{\mathsf{f}} n$ in $\mathbb{L}$, lax naturality of X requires existence of a mediating 2-cell $X_{\mathsf{f}} : \llbracket \mathsf{f} \rrbracket \circ X_k \Rightarrow X_n$. The diagram (above right) shows the situation when we set $\mathsf{f} = \mathsf{m}$, a binary operation. The natural transformation X_{m} can be identified with an arrow $X \otimes X \xrightarrow{\mu} X$ in $\mathbb{C}$, which gives an inner binary operation on X.

in $\mathbb{L}$	in $\mathbf{CAT}$
2	$\mathbf{1} \xrightarrow{X_2=(X,X)} \mathbb{C}^2$
$\downarrow \mathsf{m}$	$\downarrow \llbracket \mathsf{m} \rrbracket = \otimes$
1	$\mathbf{1} \xrightarrow[X]{} \mathbb{C}$

[13] For example, given a monoidal category $\mathbb{C}$, we need to define a functor $\llbracket \mathsf{m} \circ (\mathsf{m} \times \mathrm{id}) \rrbracket = \llbracket \mathsf{m} \circ (\mathrm{id} \times \mathsf{m}) \rrbracket$ in (8). It's not clear whether it should carry (X, Y, Z) to $X \otimes (Y \otimes Z)$, or to $(X \otimes Y) \otimes Z$.

How do such inner operations on X satisfy equations as specified in $\mathbb{L}$? The key is the coherence condition[14] on mediating 2-cells: it requires $X_{\mathrm{id}} = \mathrm{id}$ concerning identities; and $X_{\mathsf{g}\circ\mathsf{f}} = X_{\mathsf{g}} \bullet (\llbracket \mathsf{g} \rrbracket \circ X_{\mathsf{f}})$ concerning composition (as on the right). The following example illustrates how such coherence induces equational properties.

$$X_{\mathsf{g}\circ\mathsf{f}} = \begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathbb{C}^l \\ \| & \swarrow X_{\mathsf{f}} & \downarrow \llbracket \mathsf{f} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C}^k \\ \| & \swarrow X_{\mathsf{g}} & \downarrow \llbracket \mathsf{g} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C}^n \end{array}$$

Example 3.5 A monoid object in a strictly monoidal category is an example of an $\mathbb{L}$-object in an $\mathbb{L}$-category. Here we take $\mathbb{L} = \mathbf{Mon}$, the theory of monoids.

For illustration, let us here derive associativity of multiplication $X \otimes X \xrightarrow{\mu} X$. In the current setting the multiplication μ is identified with a mediating 2-cell X_{m} as above. The coherence condition yields the two equalities $(*)$ below.

$$\begin{array}{c} \underline{\text{in } \mathbb{L}} \\ \begin{array}{ccc} & 3 & \\ {\scriptstyle \mathrm{id}\times\mathsf{m}}\swarrow & & \searrow{\scriptstyle \mathsf{m}\times\mathrm{id}} \\ 2 & & 2 \\ {\scriptstyle \mathsf{m}}\searrow & & \swarrow{\scriptstyle \mathsf{m}} \\ & 1 & \end{array} \end{array} \qquad \begin{array}{c} \underline{\text{in } \mathbf{CAT}} \\ \begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathbb{C}^3 \\ \| & \swarrow X_{\mathrm{id}\times\mathsf{m}} & \downarrow \llbracket \mathrm{id}\times\mathsf{m} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C}^2 \\ \| & \swarrow X_{\mathsf{m}} & \downarrow \llbracket \mathsf{m} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C} \end{array} \end{array} \overset{(*)}{=} \begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathbb{C}^3 \\ \| & \swarrow X_{\mathsf{m}\circ(\mathrm{id}\times\mathsf{m})} = X_{\mathsf{m}\circ(\mathsf{m}\times\mathrm{id})} & \downarrow \\ \mathbf{1} & \longrightarrow & \mathbb{C}^1 \end{array} \overset{(*)}{=} \begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathbb{C}^3 \\ \| & \swarrow X_{\mathsf{m}\times\mathrm{id}} & \downarrow \llbracket \mathsf{m}\times\mathrm{id} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C}^2 \\ \| & \swarrow X_{\mathsf{m}} & \downarrow \llbracket \mathsf{m} \rrbracket \\ \mathbf{1} & \longrightarrow & \mathbb{C} \end{array}$$

Now it is not hard to see that: the composed 2-cell on the left corresponds to $X^3 \xrightarrow{X\times\mu} X^2 \xrightarrow{\mu} X$; and the one on the right corresponds to $X^3 \xrightarrow{\mu\times X} X^2 \xrightarrow{\mu} X$. The equalities $(*)$ above prove that these two arrows $X^3 \rightrightarrows X$ are identical.

3.5 Microcosm structures in coalgebras

In this section we return to our original question and apply the framework we just introduced to coalgebraic settings. First we present some basic results, which are used later in our main result of general compositionality. The constructs in Section 2 (such as sync) will appear again, now in their generalized form. Some details and proofs are omitted here due to lack of space. They will appear in the forthcoming extended version of this paper, although the diligent reader will readily work them out.

Let $\mathbb{C}$ be an $\mathbb{L}$-category, and $F : \mathbb{C} \to \mathbb{C}$ be a functor. We can imagine that, for the category $\mathbf{Coalg}_F$ to carry an $\mathbb{L}$-structure, F needs to be somehow compatible with $\mathbb{L}$; it turns out that the following condition is sufficient. It is weaker than F's being an $\mathbb{L}$-functor (see Definition 3.2).

Definition 3.6 (Lax $\mathbb{L}$-functor) A functor $F : \mathbb{C} \to \mathbb{D}$ between $\mathbb{L}$-categories is said to be a *lax* $\mathbb{L}$*-functor* if it is identified with[15] some lax natural transformation $\mathbb{L} \overset{\mathbb{C}}{\underset{\mathbb{D}}{\Downarrow F}} \mathbf{CAT}$ which is product-preserving (i.e. $F \circ H$ is strictly natural; see Definition 3.4).

[14] This is part of the notion of lax natural transformations; see [4].

[15] Meaning: $F : \mathbb{C} \to \mathbb{D}$ is the 1-component of such a lax natural transformation $\mathbb{C} \Rightarrow \mathbb{D}$.

Lax $\mathbb{L}$-endofunctors are natural generalization of functors with sync as in Section 2. To illustrate this, look at the lax naturality diagram on the right for a binary operation m. Here we denote the outer interpretation $[\![\mathsf{m}]\!]$ by $\otimes$. The 2-component is $F_2 = (F, F)$ because the lax natural transformation F is product-preserving. The mediating 2-cell F_{m} can be identified with a natural transformation $FX \otimes FY \to F(X \otimes Y)$; this is what we previously called sync. Moreover, F_{m} (as generalized sync) is automatically compatible with equational properties (as in Theorem 2.4); this is because of the coherence condition on mediating 2-cells like "$F_{\mathsf{g}\circ\mathsf{f}}$ is a suitable composition of F_{g} after F_{f}."

in $\mathbb{L}$	in $\mathbf{CAT}$
2	$\mathbb{C}^2 \xrightarrow{(F,F)} \mathbb{C}^2$
$\downarrow\mathsf{m}$	$\otimes\downarrow \quad \swarrow_{F_{\mathsf{m}}} \quad \downarrow\otimes$
1	$\mathbb{C} \xrightarrow[F]{} \mathbb{C}$

The following results follow from a more general result concerning the notion of *inserters*, namely: when G is an oplax $\mathbb{L}$-functor and F is a lax $\mathbb{L}$-functor, then the inserter $Ins(G, F)$ is an $\mathbb{L}$-category.

Proposition 3.7 *1. Let $\mathbb{C}$ be an $\mathbb{L}$-category and $F : \mathbb{C} \to \mathbb{C}$ be a lax $\mathbb{L}$-functor. Then $\mathbf{Coalg}_F$ is an $\mathbb{L}$-category; moreover the forgetful functor $\mathbf{Coalg}_F \xrightarrow{U} \mathbb{C}$ is a (strict, non-lax) $\mathbb{L}$-functor.*

2. Given a microcosm model $X \in \mathbb{C}$ for $\mathbb{L}$, the slice category $\mathbb{C}/X$ is an $\mathbb{L}$-category; moreover the functor $\mathbb{C}/X \xrightarrow{\mathrm{dom}} \mathbb{C}$ is an $\mathbb{L}$-functor. □

Note that $\mathbf{Coalg}_F$ being an $\mathbb{L}$-category means not only that operations are interpreted in $\mathbf{Coalg}_F$ but also that all the equational properties specified in $\mathbb{L}$ are satisfied in $\mathbf{Coalg}_F$. Therefore this result generalizes Theorem 2.4.

Concretely, an operation $\mathsf{f} : k \to 1$ in $\mathbb{L}$ is interpreted in $\mathbf{Coalg}_F$ and $\mathbb{C}/X$ as follows, respectively.

$$\begin{pmatrix} FX_1 & & FX_k \\ \uparrow c_1 & \ldots, & \uparrow c_k \\ X_1 & & X_k \end{pmatrix} \mapsto \begin{array}{c} F[\![\mathsf{f}]\!](\overrightarrow{X}) \\ \uparrow (F_{\mathsf{f}})_{\overrightarrow{X}} \\ [\![\mathsf{f}]\!](\overrightarrow{FX}) \\ \uparrow [\![\mathsf{f}]\!](\overrightarrow{c}) \\ [\![\mathsf{f}]\!](\overrightarrow{X}) \end{array} \qquad \begin{pmatrix} Y_1 & & Y_k \\ \downarrow y_1 & \ldots, & \downarrow y_k \\ X & & X \end{pmatrix} \mapsto \begin{array}{c} [\![\mathsf{f}]\!](\overrightarrow{Y}) \\ \downarrow [\![\mathsf{f}]\!](\overrightarrow{y}) \\ [\![\mathsf{f}]\!](\overrightarrow{X}) \\ \downarrow X_{\mathsf{f}} \\ X \end{array}$$

Compare these with (5) and (6); these make an essential use of F_{f} and X_{f} which generalize sync and $\|$ in Section 2, respectively.

Proposition 3.8 *1. A lax $\mathbb{L}$-functor preserves $\mathbb{L}$-objects. Hence so does an $\mathbb{L}$-functor.*

2. A final object of an $\mathbb{L}$-category $\mathbb{C}$, if it exists, is an $\mathbb{L}$-object. The inner $\mathbb{L}$-structure is induced by finality. □

We can now present our main result. It generalizes Theorem 2.1, hence is a generalized version of the "coalgebraic compositionality" equation (4).

Theorem 3.9 (General compositionality) *Let $\mathbb{C}$ be an $\mathbb{L}$-category and $F : \mathbb{C} \to \mathbb{C}$ be a lax $\mathbb{L}$-functor. Assume further that $\zeta : Z \xrightarrow{\cong} FZ$ is the final coalgebra. Then the*

functor beh : $\mathbf{Coalg}_F \to \mathbb{C}/Z$ *is a (non-lax)* $\mathbb{L}$*-functor. It makes the following diagram of* $\mathbb{L}$*-functors commute.*

$$\begin{array}{ccc} \mathbf{Coalg}_F & \xrightarrow{\mathsf{beh}} & \mathbb{C}/Z \\ & {\scriptstyle U}\searrow \quad \swarrow{\scriptstyle \mathrm{dom}} & \\ & \mathbb{C} & \end{array}$$

□

The proof is straightforward by finality. Here $\mathbf{Coalg}_F$ is an $\mathbb{L}$-category (Proposition 3.7.1). So is $\mathbb{C}/Z$ because: $\zeta \in \mathbf{Coalg}_F$ is an $\mathbb{L}$-object (Proposition 3.8.2); $Z = U\zeta$ is an $\mathbb{L}$-object (Propositions 3.8.1 and 3.7.1); hence $\mathbb{C}/Z$ is an $\mathbb{L}$-category (Proposition 3.7.2).

We have also observed some facts which look interesting but are not directly needed for our main result (Theorem 3.9). They include: the category $\mathbb{L}\text{-}\mathbf{obj}_{\mathbb{C}}$ of $\mathbb{L}$-objects in $\mathbb{C}$ and morphisms between them forms the lax limit of a diagram $\mathbb{C} : \mathbb{L} \to \mathbf{CAT}$; the simplicial category Δ is the "universal" microcosm model for $\mathbf{Mon}$ (cf. [20, Proposition VII.5.1]). The details will appear in the forthcoming extended version.

4 Conclusions and future work

In this paper we have observed that the microcosm principle (as called by Baez and Dolan) brings new mathematical insights into computer science. Specifically, we have looked into parallel composition of coalgebras, which would serve as a mathematical basis for the study of concurrency. As a purely mathematical expedition, we have presented a 2-categorical formalization of the microcosm principle, where an algebraic theory is presented by a Lawvere theory. Turning back to our original motivation, the formalization was applied to coalgebras and yielded some general results which ensure compositionality and equational properties such as associativity.

There are many questions yet to be answered. Some of them have been already mentioned, namely: extending the expressive power of sync (Remark 2.3), and a proper treatment of "pseudo" algebraic structures (Section 3.3).

On the application side, one direction of future work is to establish a relationship between sync and *(syntactic) formats* for process algebras. Our sync represents a certain class of operational rules; formats are a more syntactic way to do the same. Formats which guarantee certain good properties (such as commutativity, see [23]) have been actively studied. Such a format should be obtained by translating e.g. a "commutative" sync into a format.

On the mathematical side, one direction is to identify more instances of the microcosm principle. Mathematics abounds with the (often implicit) idea of nested algebraic structures. To name a few: a topological space in a topos which is itself a "generalized topological space"; a category of domains which itself carries a "structure as a domain." We wish to turn such an informal statement into a mathematically rigorous one, by generalizing the current formalization of the microcosm principle. As a possible first step towards this direction, we are working on formalizing the microcosm principle for finitary monads which are known to be roughly the same thing as Lawvere theories.

Another direction is a search for n-folded nested algebraic structures. In the current paper we have concentrated on two levels of interpretation; an example with more levels might be found e.g. in an internal category in an internal category.

Acknowledgments Thanks are due to Kazuyuki Asada, John Baez, Masahito Hasegawa, Bill Lawvere, Duško Pavlović, John Power and the participants of CALCO-jnr workshop 2007 including Alexander Kurz for helpful discussions and comments.

References

1. J.C. Baez and J. Dolan. Higher dimensional algebra III: n-categories and the algebra of opetopes. *Adv. Math*, 135:145–206, 1998.
2. M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer, Berlin, 1985. Available online.
3. F. Bartels. *On generalised coinduction and probabilistic specification formats. Distributive laws in coalgebraic modelling*. PhD thesis, Free Univ. Amsterdam, 2004.
4. F. Borceux. *Handbook of Categorical Algebra*, vol. 50, 51 and 52 of *Encyclopedia of Mathematics*. Cambridge Univ. Press, 1994.
5. I. Hasuo. Generic forward and backward simulations. In C. Baier and H. Hermanns, editors, *International Conference on Concurrency Theory (CONCUR 2006)*, vol. 4137 of *Lect. Notes Comp. Sci.*, pp. 406–420. Springer, Berlin, 2006.
6. I. Hasuo, B. Jacobs and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Comp. Sci.*, 3(4:11), 2007.
7. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
8. M. Hyland and A.J. Power. Discrete Lawvere theories and computational effects. *Theor. Comp. Sci.*, 366(1–2):144–162, 2006.
9. B. Jacobs. *Categorical Logic and Type Theory*. North Holland, Amsterdam, 1999.
10. B. Jacobs. Trace semantics for coalgebras. In J. Adámek and S. Milius, editors, *Coalgebraic Methods in Computer Science*, vol. 106 of *Elect. Notes in Theor. Comp. Sci.* Elsevier, Amsterdam, 2004.
11. B. Jacobs. Introduction to coalgebra. Towards mathematics of states and observations, 2005. Draft of a book, `www.cs.ru.nl/B.Jacobs/PAPERS/index.html`.
12. B. Jacobs. A bialgebraic review of deterministic automata, regular expressions and languages. In K. Futatsugi, J.P. Jouannaud and J. Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, vol. 4060 of *Lect. Notes Comp. Sci.*, pp. 375–404. Springer, 2006.
13. P.T. Johnstone, A.J. Power, T. Tsujishita, H. Watanabe and J. Worrell. An axiomatics for categories of transition systems as coalgebras. In *Logic in Computer Science*. IEEE, Computer Science Press, 1998.
14. M. Kick, A.J. Power and A. Simpson. Coalgebraic semantics for timed processes. *Inf. & Comp.*, 204(4):588–609, 2006.
15. B. Klin. From bialgebraic semantics to congruence formats. In *Workshop on Structural Operational Semantics (SOS 2004)*, vol. 128 of *Elect. Notes in Theor. Comp. Sci.*, pp. 3–37. 2005.
16. B. Klin. Bialgebraic operational semantics and modal logic. In *Logic in Computer Science*, pp. 336–345. IEEE Computer Society, 2007.
17. A. Kock and G.E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pp. 283–313. North-Holland, Amsterdam, 1977.
18. F.W. Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963. Reprints in Theory and Applications of Categories, 5 (2004) 1–121.
19. E.A. Lee. Making concurrency mainstream. Invited talk at CONCUR 2006, 2006.
20. S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 2nd edn., 1998.
21. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

22. E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
23. M.R. Mousavi, M.A. Reniers and J.F. Groote. A syntactic commutativity format for SOS. *Inform. Process. Lett.*, 93(5):217–223, 2005.
24. K. Nishizawa and A.J. Power. Lawvere theories enriched over a general base. *Journ. of Pure & Appl. Algebra*, 2006. To appear.
25. J. Power and D. Turi. A coalgebraic foundation for linear time semantics. In *Category Theory and Computer Science*, vol. 29 of *Elect. Notes in Theor. Comp. Sci.* Elsevier, 1999.
26. J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 249:3–80, 2000.
27. D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Logic in Computer Science*, pp. 280–291. IEEE, Computer Science Press, 1997.

CSP-CASL-Prover — Tool integration and algorithms for automated proof generation

Liam O'Reilly[1], Yoshinao Isobe[2], Markus Roggenbach[1⋆]

[1] Swansea University, United Kingdom
[2] AIST, Tsukuba, Japan

Abstract. The specification language CSP-CASL allows one to model data as well as processes of distributed systems within one framework. In our paper, we describe how a combination of the existing tools HETS and CSP-Prover can solve the challenges that CSP-CASL raises on integrated theorem proving for processes and data. For building this new tool, the automated generation of theorems and their proofs in Isabelle/HOL plays a fundamental role. A case study of industrial strength demonstrates that our approach scales up to complex problems.

1 Introduction

Distributed computer applications like flight booking systems, web services, and electronic payment systems such as the EP2 standard [ep202], require parallel processing of data. Consequently, these systems have concurrent aspects (e.g. deadlock-freedom) as well as data aspects (e.g. functional correctness). Often, these aspects depend on each other.

In [Rog06], we present the language CSP-CASL, which is tailored to the specification of distributed systems. CSP-CASL integrates the process algebra CSP [Hoa85,Ros98] with the algebraic specification language CASL [Mos04]. Its novel aspects include the combination of denotational semantics in the process part and, in particular, loose semantics for the data types covering both concepts of partiality and sub-sorting. In [GRS05] we apply CSP-CASL to the EP2 standard and demonstrate that CSP-CASL can deal with problems of industrial strength.

Here, we develop theorem proving support for CSP-CASL and show that our approach scales up to practically relevant systems such as the EP2 standard. CSP-CASL comes with a simple, but powerful notion of refinement. CSP-CASL refinement can be decomposed into first a refinement step on data only and then a refinement step on processes. Data refinement is well understood in the CASL context and has good tool support already. Thus, we focus here on process refinement. The basic idea is to re-use existing tools for the languages CASL and CSP, namely for CASL the tool HETS [MML07] and for CSP the tool CSP-Prover [IR05,IR06], both of which are based on the theorem prover Isabelle/HOL [NPW02]. This re-use is possible thanks to the definition of the CSP-CASL semantics in a two step approach: First, the data specified in CASL is translated

⋆ This cooperation was supported by the EPSRC Project EP/D037212/1.

into an alphabet of communications, which, in the second step, is used within the processes, where the standard CSP semantics are applied.

The main issue in integrating the tools HETS and CSP-Prover into a CSP-CASL-Prover is to implement – in Isabelle/HOL – CSP-CASL's construction of an alphabet of communications out of an algebraic specification of data written in CASL. The correctness of this construction relies on the fact that a certain relation turns out to be an equivalence relation. Although this has been proven to hold under certain conditions, we chose to prove this fact for each CSP-CASL specification individually. This adds an additional layer of trust. It turns out that the alphabet construction, the formulation of the justification theorems (establishing the equivalence relation), and also the proofs of these theorems can be automatically generated.

Closely related to CSP-CASL is the specification language μCRL [GP95]. Here, data types have loose semantics and are specified in equational logic with total functions. The underlying semantics of the process algebraic part is operational. [BFG$^+$05] presents - on the fly, as the focus of the paper is on protocol verification – the prototype of a μCRL-Prover based on the interactive theorem prover PVS [ORS92]. The chosen approach is to represent the abstract μCRL data types directly by PVS types, and to give a subset of μCRL processes, namely the linear process equations, an operational semantics in terms of labelled transition systems. Thanks to μCRL's simple approach to data – neither sub-sorting nor partiality are available – there is no need for an alphabet construction – as it is also the case in CSP-CASL in the absence of sub-sorting and partiality. Concerning processes, CSP-CASL provides semantics to full CSP by reusing the implementation of various denotational CSP semantics in CSP-Prover.

Our paper is organized as follows: Section 2 introduces the CSP-CASL semantics along with a case study from the EP2 system. Section 3 describes the existing tools which we make use of. The overall architecture of CSP-CASL-Prover is presented in Section 4. Section 5 discusses in detail how we build a single alphabet which can be used as a parameter for the process type in CSP-Prover. Then we consider how integration theorems can lift proof obligations on the alphabet back onto proof obligations over the data from a CSP-CASL specification. In Section 7 we analyze which parts of our Isabelle code are specification dependent. Section 8 finishes our paper with a case study on how to prove deadlock freedom of a dialog within the EP2 system.

2 CSP-CASL

CSP-CASL [Rog06] is a comprehensive language which combines *processes* written in CSP [Hoa85,Ros98] with the specification of *data types* in CASL [Mos04]. The general idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types, which are loosely specified in CASL. All standard CSP operators are included, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, communication over channels. Concerning CASL fea-

tures, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting.

Syntactically, a CSP-CASL specification with name N consists of a data part Sp, which is a structured CASL specification, an (optional) channel part Ch to declare channels, which are typed according to the data part, and a process part P written in CSP, within which CASL terms are used as communications, CASL sorts denote sets of communications, relational renaming is described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions – see Figure 1 for an instance of this scheme:

$$\textbf{ccspec } N = \textbf{data } Sp \textbf{ channel } Ch \textbf{ process } P \textbf{ end}$$

2.1 EP2 in CSP-CASL

As a running example, we choose a dialog nucleus of the EP2 system [ep202], see [GRS05] for further details of the modelling approach. In this dialog, the credit card terminal and another component, the so-called acquirer, are supposed to exchange initialization information over the channel `C_SI_Init`. The messages on this channel can be classified into `SessionStart`, `SessionEnd`, `ConfigDataRequest` and `ConfigDataResponse`. In order to prove that the dialog is deadlock-free, we need to ensure that messages of type `SessionEnd` are different from messages of type `ConfigDataResponse`. The terminal initiates the dialog by sending a message of type `SessionStart`, see the process `Ter_Init`. The acquirer receives this message, see the process `Acq_Init`. In `Acq_ConfigManagement`, the acquirer then takes the internal decision either to end the dialog by sending the message `e` of type `SessionEnd` or to start a data exchange with the terminal. The terminal, on the other side, waits in the process `Ter_ConfigManagement` for a message from the acquirer. Depending on the type of this message, the terminal ends the dialog with `SKIP`, engages in a data exchange, or executes the deadlock process `STOP`. The system consists of the parallel composition of terminal and acquirer. Should one of these two components be in a deadlock, the whole system will be in deadlock.

The original dialog in EP2 has many more possibilities for the data exchange. To this end, it involves 11 sorts, but it exhibits the same structure. For simplicity, we present here only the above nucleus. However, we successfully applied our approach to the full dialog. Our proof on deadlock freedom (see Section 8) scales up from the nucleus to the real version.

2.2 CSP-CASL semantics

Semantically, a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part Sp gives rise to one process denotation. The definition of the language CSP-CASL is generic in the choice of a specific CSP semantics. For example, all denotational CSP models mentioned in [Ros98] are possible parameters.

```
ccspec GetInitialisationData =
  data  sorts SessionStart, SessionEnd,
              ConfigDataRequest, ConfigDataResponse < D_SI_Init
        forall x:ConfigDataRequest; y:SessionEnd . not (x=y)
        ops r: ConfigDataRequest; e: SessionEnd
  channel C_SI_Init: D_SI_Init
  process
  let  Ter_Init = C_SI_Init ! sessionStart: SessionStart
                  -> Ter_ConfigManagement
       Ter_ConfigManagement = C_SI_Init ? configMess
         -> IF (configMess: SessionEnd) THEN SKIP ELSE
            (IF (configMess: ConfigDataRequest) THEN
              C_SI_Init ! response: ConfigDataResponse
              -> Ter_ConfigManagement ELSE STOP)
       Acq_Init = C_SI_Init ? sessionStart: SessionStart
                  -> Acq_ConfigManagement
       Acq_ConfigManagement =
             C_SI_Init ! e -> SKIP
         |~| C_SI_Init ! r -> C_SI-Init ? response: ConfigDataResponse
             -> Acq_ConfigManagement
  in Ter_Init |[ C_SI_Init ]| Acq_Init
```

Fig. 1. Nucleus of an EP2 dialog.

The semantics of CSP-CASL is defined in a two-step approach [3], see Figure 2. Given a CSP-CASL specification (Sp, P), in the first step we construct for each model M of Sp a CSP process $P'(A(\beta(M)))$. To this end, we define for each model M, which might include partial functions, an equivalent model $\beta(M)$ in which partial functions are totalized. $\beta(M)$ gives rise to an alphabet of communications $A(\beta(M))$. In the second step we point-wise apply a denotational CSP semantics. This translates a process $P'(A(\beta(M)))$ into its denotation d_M in the semantic domain of the chosen CSP model.

In the following we sketch the alphabet construction – see [Rog06] for the full details. The purpose of the alphabet construction is to turn a CASL model into an alphabet of communications. CASL models are defined in two steps: First, we define what a model over a many-sorted signature is. Using this concept we then define what a model over a sub-sorted signature is.

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of

- a set S of sorts,
- two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of *total function symbols* and *partial function symbols*, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$ for each $(w, s) \in S^* \times S$, and
- a family $P = (P_w)_{w \in S^*}$ of *predicate symbols*.

[3] We omit the syntactic encoding of channels into the data part.

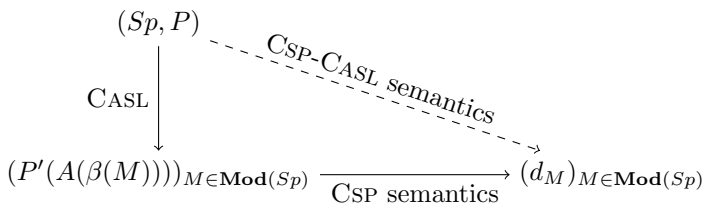


Fig. 2. CSP-CASL semantics.

Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *many-sorted Σ-model* M consists of

- a non-empty carrier set M_s for each $s \in S$,
- a partial function $(f_{w,s})_M : M_w \to M_s$ for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, the function being total for $f \in TF_{w,s}$, and
- a relation $(p_w)_M \subseteq M_w$ for each predicate symbol $p \in P_w$.

Together with the standard definition of first order logic formulae and their satisfaction, this definition yields the institution $PFOL^=$, see [Mos02] for the details.

A *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leq)$ consists of a many-sorted signature (S, TF, PF, P) together with a reflexive and transitive *sub-sort relation* $\leq_S \subseteq S \times S$. With each sub-sorted signature $\Sigma = (S, TF, PF, P, \leq)$ we associate a many-sorted signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$, which extends the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\mathtt{inj} : s \to s'$ for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\mathtt{pr} : s' \to ?s$ for each pair of sorts $s \leq_S s'$, and
- an unary *membership* predicate symbol $\epsilon^s_{s'} : s'$ for each pair of sorts $s \leq_S s'$.

Sub-sorted Σ-models are many-sorted $\hat{\Sigma}$-models satisfying in $PFOL^=$ a set of axioms $\hat{J}(\Sigma)$, which prescribe how the injection, projection, and membership behave[4]. A typical axiom in $\hat{J}(\Sigma)$ is $\mathtt{inj}_{s,s}(x) \stackrel{e}{=} x$ for $s \in S$. Together with the definition of sub-sorted first order logic formulae and their satisfaction, this definition yields the institution $SubPFOL^=$, see [Mos02] for the details.

Given a sub-sorted model M on carrier sets, its strict extension $\beta(M)$ is defined as: $\beta(M)_s = M_s \cup \{\bot\}$ for all $s \in \hat{S}$, where $\bot \notin M_s$ for all $s \in \hat{S}$. The strict extension is uniquely determined. We say that a signature $\Sigma = (S, TF, PF, P, \leq)$ has local top elements, if for all $u, u', s \in S$ the following holds: if $u, u' \geq s$ then there exists $t \in S$ with $t \geq u, u'$. Relatively to the extension $\beta(M)$ of a model M for a sub-sorted signature with local top elements, we define an alphabet of communications

$$A(\beta(M)) := (\biguplus_{s \in S} \beta(M)_s)_{/\sim}$$

[4] and also define how overloading works.

where $(s, x) \sim (s', x')$ iff either

- $x = x' = \bot$ and there exists $u \in S$ such that $s \leq u$ and $s' \leq u$,

or

- $x \neq \bot$, $x' \neq \bot$, there exists $u \in S$ such that $s \leq u$ and $s' \leq u$, and
- for all $u \in S$ with $s \leq u$ and $s' \leq u$ the following holds:

$$(\mathtt{inj}_{(s,u)})_M(x) = \mathtt{inj}_{(s',u)\,M}(x')$$

for $s, s' \in S$, $x \in M_s, x' \in M_{s'}$. For signatures with local top elements the relation $\sim$ turns out to be an equivalence relation [Rog06].

2.3 CSP-CASL refinement

Given a denotational CSP model with domain $\mathcal{D}$, the semantic domain of CSP-CASL consists of families of process denotations $d_M \in \mathcal{D}$. Its elements are of the form $(d_M)_{M \in I}$ where I is a class of algebras. As refinement $\rightsquigarrow_{\mathcal{D}}$ we define on these elements

$$\begin{gathered}(d_M)_{M \in I} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in I'} \\ \text{iff} \\ I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'},\end{gathered}$$

where $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen CSP model $\mathcal{D}$. Concerning *data refinement*, we directly obtain the following characterisation:

$$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P) \quad \text{if} \quad \begin{cases} 1.\ \Sigma(Sp) = \Sigma(Sp'), \\ 2.\ \mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp) \end{cases}$$

The crucial point is that we fix both the signature of the data part and the process P. For *process refinement*, a similar characterisation is obvious:

$$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp, Q) \quad \text{if} \quad \begin{cases} \text{for all } M \in \mathbf{Mod}(Sp) : \\ P'(A(\beta(M))) \sqsubseteq_{\mathcal{D}} Q'(A(\beta(M))) \end{cases}$$

CSP-CASL refinement can be decomposed into first a data refinement and then a process refinement[5]:

$$\begin{gathered}(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P') \\ \Longleftrightarrow \\ (Sp, P) \overset{\mathtt{data}}{\rightsquigarrow} (Sp', P) \ \wedge \ (Sp', P) \overset{\mathtt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp', P')\end{gathered}$$

Here, $\overset{\mathtt{data}}{\rightsquigarrow}$ and $\overset{\mathtt{proc}}{\rightsquigarrow_{\mathcal{D}}}$ represent data refinement and process refinement, respectively. Data refinement does not deal with the process part at all. Thus, to prove data refinement we can re-use the existing tool support for CASL. Consequently, we focus in this paper on tool support for process refinement.

[5] Note that the order of the decomposition is essential: $(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P') \not\Rightarrow (Sp, P) \overset{\mathtt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp, P') \ \wedge \ (Sp, P') \overset{\mathtt{data}}{\rightsquigarrow} (Sp', P')$.

3 Tools involved

Csp-Casl-Prover makes appropriate re-use of existing technology and tools. In this section we will explain what these tools are and what they do.

3.1 Isabelle/HOL

Isabelle/HOL [NPW02] is a widely used, generic interactive theorem prover for Higher Order Logic. Theorems are entered into Isabelle/HOL via *commands*. Isabelle/HOL then displays proof goals which need to be discharged. For example the command `theorem T1: "a+b = b+a"` creates a theorem with the name `T1` and a goal of `a+b = b+a`.

To prove such a theorem, *proof commands* are issued which transform goals into other goals (or possibly many sub-goals). A goal is discharged if it is transformed into the truth value `True`. A theorem is proven when all of its proof obligations have been discharged. Previously established theorems can be used within further proofs as new proof commands. Proof commands can be combined in various ways to form tactics, which can ease the burden of discharging proof goals.

Theory files consist of scripts of Isabelle commands and proof commands. Such theory files can use theorems, proofs, data structures and functions written in other theory files. This brings in a concept of modularity to Isabelle/HOL.

Commands allow the user to extend the logic, for example, by adding new data structures, types, and function definitions to Isabelle/HOL. This allows the user to accommodate for the particular area of interest.

For example, the command `datatype Num = N nat | I int` adds a new data type with the name `Num`. This creates a new type which is the sum of natural numbers and integers. Here, `N` and `I` are user chosen type constructors while `nat` and `int` are the built in types of natural numbers and integers, respectively. Such a datatype declaration comes with a built in induction tactic within Isabelle/HOL. In our example of `Num`, this induction tactic simplifies to a complete case distinction.

The commands that Isabelle/HOL offers are able to create new data structures, functions, relations, etc. Theorems can then be used to prove properties of data structures, functions and relations, while proof commands can use the definition of such structures, functions and relations. For example, we define a new function `plus:: Num => Num => Num` such that a natural number will be returned only if both arguments are natural numbers, else an integer will be returned. The definition of `plus` is as expected. The following Isabelle/HOL code proves our new function `plus` to be commutative:

```
theorem comm: "plus a b = plus b a"
apply(induct_tac a)
```

The first line sets up the theorem with the name `comm` and states that `plus` is commutative. The second line applies the induction tactic on the variable `a`. This simplifies to a finite case distinction over our sum type, i.e. `a` can have the form

`N nat` or `I int`. Hence, after application of the induction tactic `induct_tac`, the following two sub-goals are shown:

```
goal (theorem (comm), 2 subgoals):
 1. !!nat. plus (N nat) b = plus b (N nat)
 2. !!int. plus (I int) b = plus b (I int)
```

Here `b`, `nat`, and `int` are variables where `nat` and `int` are locally bound in each subgoal (indicated by the `!!` symbol). Further induction on the variable `b` along with simplification proves our theorem.

3.2 HETS

HETS (the Heterogeneous Tool Set) [MML07] is a parsing, static analysis and proof management tool for various specification languages centred around CASL [Mos04].

One of the features of HETS is the ability of translating a specification from one specification language into a specification from another language, while preserving its semantics. An important instance of this is the translation of CASL specifications into suitable code for use in the theorem prover Isabelle/HOL. In our setting we use HETS as an input/output tool, loading specifications written in CASL and encoding them into Isabelle/HOL code. This translation process is non-trivial and CSP-CASL-Prover exploits this functionality heavily.

3.3 CSP-Prover

CSP-Prover [IR05,IR06] is a theorem prover built upon Isabelle/HOL. CSP-Prover is dedicated to refinement proofs over CSP processes. It is generic in the models of CSP that can be used. It can be instantiated with all main CSP models. The trace model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$ are available, while implementations of the stable-revivals model $\mathcal{R}$ and failure-divergences model $\mathcal{N}$ are underway. CSP-Prover provides a deep-encoding of CSP within Isabelle/HOL. Consequently, it offers a type `'a proc` (which is used within Section 5.2), the type of CSP processes that are built over the alphabet `'a`, where `'a` is an Isabelle/HOL type variable.

CSP-Prover supports two proof methods, namely syntactical and semantical proofs. Syntactical proofs transform the syntax of CSP processes into equivalent CSP processes until syntactical identity is reached. Semantical proofs evaluate the denotational semantics of CSP processes and compare the denotations.

CSP-Prover comes with a large collection of CSP laws and tactics. CSP-Prover tactics combine these laws to powerful proof principles. One typical example is the tactic `cspF_hsf_tac`, which transforms CSP processes to a 'head normal form' over the model $\mathcal{F}$.

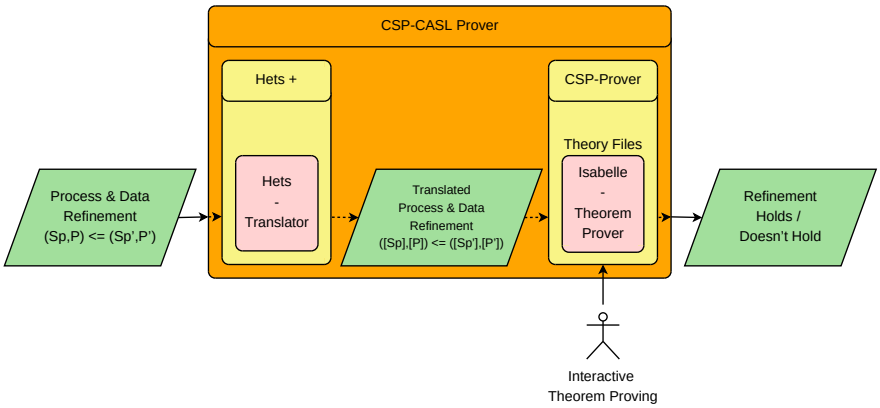


Fig. 3. Diagram of the basic architecture of Csp-Casl-Prover.

4 Basic architecture of CSP-CASL-Prover

Csp-Casl-Prover uses the existing tools HETS and Csp-Prover discussed in Sections 3.2 and 3.3. Its proposed architecture is shown in Figure 3. The overall idea is that Csp-Casl-Prover takes a Csp-Casl process refinement statement as its input. The Csp-Casl specifications involved are parsed and transformed by Csp-Casl-Prover into a new file suitable for use in Csp-Prover. This file can then be directly used within Csp-Prover to interactively prove if the Csp-Casl process refinement holds. For example, dead lock freedom of a system of processes can be proven using such a refinement statement, see Section 8 for details.

Csp-Casl-Prover re-uses the existing functionality of HETS in order to produce part of the file that will be used as input to Csp-Prover. We take the data part of a Csp-Casl specification and translate this into Isabelle/HOL via HETS. This generates (in general) several types in Isabelle/HOL, which need to be transformed into one alphabet to become the parameter of the Csp-Prover process type `'a proc`. This is expressed in Figure 3 by HETS being labelled as "Hets +", which represents the extra encoding that needs to be done. This is discussed in more detail Section 5.2.

The final form of the file which is produced by Csp-Casl-Prover (i.e. HETS and the extra encoding) is labelled as "Translated Processes and Data Refinement" in Figure 3. Figure 4 shows how this file is split up into five distinct parts. The first three parts can all be automatically generated from the original Csp-Casl specification. The final two parts are dependent on the application. Csp-Casl-Prover provides place holder code that the user can fill in and expand.

The first part of the file shown in Figure 4 "Hets Translation of CASL" is the direct encoding of the data part of the Csp-Casl specification which is produced by HETS – see Section 5.1. The second part "Alphabet Construction & Justi-

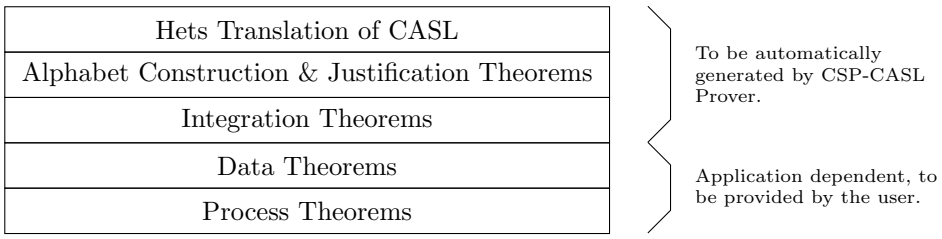


Fig. 4. Structure of a translated Csp-Casl specification using Csp-Casl-Prover.

fication Theorems" provides the Csp-Casl semantics, namely the alphabet of communications, over which Csp processes can be constructed – see Section 5.2. The third part "Integration Theorems" provides the user with a mechanism to lift proof obligations on processes to proof obligations on data in the HETS encoding only – see Section 6. These Integration Theorems are crucial in keeping the final proof of the process refinement small, readable and manageable – see Section 8 for an example. The forth part is where the user shall write auxiliary theorems and proofs which are helpful for the specific refinement to be proven. The final part is where the user shall provide the proof of the refinement between the processes.

5 Alphabet construction

In this section we first discus the encoding that is produced by HETS. Then we describe how to encode the alphabet construction in Isabelle/HOL.

5.1 HETS encoding

We use HETS in order to encode the data part of a Csp-Casl specification[6]. Essentially, HETS produces Isabelle/HOL commands such as `typedecl` and function declarations, followed by axioms which define the properties of such declared types and functions. Casl sub-sorting and partiality are encoded within Isabelle/HOL by adding undefined elements to each sort and by providing injection and projection functions between sorts in the sub-sort relation. Full details of this encoding can be found in Chapter 4 of [Mos02].

Figure 5 shows part of the encoding[7] that HETS produces for the Csp-Casl specification for the nucleus of the EP2 dialog in Figure 1.

HETS produces one `typedecl` for each sort declared in the Csp-Casl specification. A `typedecl` command extends the signature by a new type which is

[6] Currently, the chosen encoding of HETS does not allow for the use of free and generated types, however, this difficulty will be over come in future versions of HETS.

[7] For the purposes of a clear presentation in the paper, we have slightly adapted the naming scheme of HETS.

```
typedecl D_SI_Init
typedecl D_SI_Init_ConfigDataRequest ...
consts
e :: "D_SI_Init_SessionEnd"   r :: "D_SI_Init_ConfigDataRequest"
g__bottom_1 :: "D_SI_Init" ...
g__defined_1 :: "D_SI_Init => bool"
g__defined_2 :: "D_SI_Init_ConfigDataRequest => bool"
g__defined_4 :: "D_SI_Init_SessionEnd => bool" ...
g__inj_1 :: "D_SI_Init_ConfigDataRequest => D_SI_Init"
g__inj_3 :: "D_SI_Init_SessionEnd => D_SI_Init" ...
g__proj_1 :: "D_SI_Init => D_SI_Init_ConfigDataRequest" ...
ga_nonEmpty : "EX x. g__defined_1(x)" ...
ga_notDefBottom : "ALL x. (~ g__defined_1(x)) = (x = g__bottom_1)" ...
Ax1 : "ALL x. ALL y. g__defined_2(x) & g__defined_4(y) -->
      ~ g__inj_1(x) = g__inj_3(y)"
```

Fig. 5. HETS Encoding for the nucleus of the EP2 specification (Figure 1).

assumed to be non-empty. After introducing all types for messages, the constants `e` and `r` are declared. Their type is the translated version of the sort from the specification, i.e. `D_SI_Init_SessionEnd` and `D_SI_Init_ConfigDataRequest`, respectively. Then constants representing an undefined element of each sort are declared. The constant `g__bottom_1` represents the undefined element of type `D_SI_Init`. This is where the strict encoding of the models is produced - see Section 2.2. Next, functions are declared to capture definedness, injection and projection functions. This is followed by axioms that control how these function behave, including the axioms of $\hat{J}(\Sigma)$ from Section 2.2 which describe the encoding of the sub-sorted signature into a many-sorted signature. Besides this, the axioms state that there is a single unique undefined element in each sort - see axiom `ga_notDefBottom` – and that each sort has at least one defined element – see axiom `ga_nonEmpty`. Full details of these axioms can be found in [Mos02]. Finally, the original axiom from the CSP-CASL specification has been added with the name of `Ax1`. This axiom changes slightly from the specification because of the encoding of undefined elements. Now the axiom states that two messages of types `D_SI_Init_ConfigDataRequest` and `D_SI_Init_SessionEnd` are never equal if they are both defined.

5.2 Alphabet construction within Isabelle/HOL

The goal of the alphabet construction is to create an alphabet of communications (the new type `Alphabet`) in Isabelle/HOL as set out in Section 2.2. We then use this type with CSP-Prover to form the type `Alphabet proc` of CSP processes over this alphabet of communications.

As HETS produces a shallow encoding of CASL, it is impossible to give a single alphabet definition within Isabelle/HOL. To overcome this obstacle, we

```
consts
  compare_with_A :: "D_SI_Init => PreAlphabet => bool"
primrec
  compare_with_A_A:  "compare_with_A ax (C_A ay) = (ax = ay)"
  compare_with_A_B:  "compare_with_A ax (C_B by) = (ax = g__inj(by))"
...
consts
  eq :: "PreAlphabet => PreAlphabet => bool"
primrec
  eq_A:  "eq(C_A ax) = compare_with_A ax"
  eq_B:  "eq(C_B bx) = compare_with_B bx"
...
```

Fig. 6. Alphabet construction for the nucleus of the EP2 dialog (Figure 1).

produce an encoding which is specifically crafted, however in a systematic way for each data part of a CSP-CASL specification. This encoding can automatically be produced by an algorithm that we report on in this paper.

The algorithm generates the code in multiple stages. First the algorithm produces the *construction section* followed by the *justification section*. The *construction section* creates a new type what we call `PreAlphabet` and defines a relation over this type. The *justification section* is a collection of theorems and proofs which make sure that we are allowed to use the code from the *construction section* in the way we want. Then the alphabet of communications is produced using both the type `PreAlphabet` and the relation.

The *construction section* consists of a new data-type called the `PreAlphabet` which is the disjoint union of all the sorts that HETS produces. The particular code for the creation of the `PreAlphabet` for the nucleus is:

```
datatype  PreAlphabet = C_A D_SI_Init
                      | C_B D_SI_Init_ConfigDataRequest
                      | C_C D_SI_Init_ConfigDataResponse
                      | C_D D_SI_Init_SessionEnd
                      | C_E D_SI_Init_SessionStart
```

Next a relation called `eq` is defined. This relation takes as parameters two elements of the `PreAlphabet` and checks weather they are equal with respect to the CSP-CASL semantics (this is the relation $\sim$ from Section 2.2).

Figure 6 shows part of the code that is produced for the `eq` relation of the nucleus. Here, auxiliary functions are used to compare each constructor of the data-type `PreAlphabet` with every other constructor. These auxiliary functions are used in order to make use of primitive recursion in Isabelle/HOL. Finally the `eq` relation is defined. Basically, two elements of the `PreAlphabet` are equal if they are equal in all super-sorts. This is accomplished using the injection functions to test the elements of the `PreAlphabet` at the correct sorts.

The CSP-CASL-semantics requires the relation `eq` to be an equivalence relation. The *justification section* checks that this property holds. The code for checking reflexivity and symmetry is simple. Thus, we focus on the proof of transitivity. The main idea behind this proof is to induct all the variables until only finitely many case distinctions remain. Isabelle/HOL can then automatically solve all of the cases by using some previously proven lemmas. Figure 7 shows part of the code that is produced to check that the `eq` relation is transitive. We carefully apply induction to the variables `x` and `y` in specific sub-goals by first pulling the sub-goal to the top of the list (using the `prefer` command) and then applying induction to the variable in the first sub-goal. The numbers associated with each `prefer` command are systematically generated by our algorithm.

```
lemma  eq_trans: "[| eq x y ; eq y z |] ==> eq x z"
apply(induct x)
prefer 1 apply(induct y)
prefer 6 apply(induct y)
...
prefer 16 apply(induct y)
prefer 21 apply(induct y)
prefer 1 apply(induct z)
prefer 6 apply(induct z)
...
prefer 116 apply(induct z)
prefer 121 apply(induct z)
apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)
done
```

Fig. 7. Proof of transitivity of the `eq` relation.

We illustrate this proof idea by a concrete example. Consider the sub-sort structure shown in Figure 8 where the functions shown are the injections functions which HETS provides[8] . After applying induction one of the resulting proof obligations is $x \sim y \wedge y \sim z \Rightarrow x \sim z$, where x, y and z are variables of the types S, T and U, respectively. Expanding the definition of $x \sim z$ yields two new sub-goals: $inj\_S\_U(x) = z$ and $inj\_S\_V(x) = inj\_U\_V(z)$. We focus here on proving $inj\_S\_V(x) = inj\_U\_V(z)$. This equation means that x is equal to z in the sort V. Expanding the definition of $x \sim y$ we obtain the equation $inj\_S\_V(x) = inj\_T\_V(y)$. From $y \sim z$ we obtain $inj\_T\_V(y) = inj\_U\_V(z)$. These two facts together yield $inj\_S\_V(x) = inj\_U\_V(z)$. This proves one part of the goal, the other can be proven in a similar way using the fact that the functions we use are injections (these axioms are provided by HETS). Isabelle/HOL can carry out all these proofs fully automatically, provided the simplifier is en-

[8] We use the notation of $\sim$ inplace of the Isabelle function `eq`.

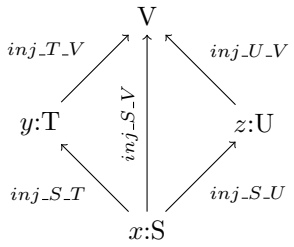


Fig. 8. Example of a possible sub-sort structure with injection functions.

riched with the right injection axioms, see the last but one line `apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)` of Figure 7.

Figure 9 shows part of the algorithm which produces the theorem and proof of transitivity of the `eq` relation.

```
Let n = Number of Sorts in the Specification.
output lemma eq_trans: "[| eq x y; eq y z |] ==> eq x z"
output apply(induct x)
for i = 1 to n   {output prefer (i*n)+1 apply(induct y)}
for i = 1 to n^2 {output prefer (i*n)+1 apply(induct z)}
output apply(auto simp add:‘{all Inject, all Decomp}’)
output done
```

Fig. 9. Algorithm for producing the theorem and proof of transitivity of the `eq` relation.

Finally, after the *justification section*, the alphabet of communications is constructed:

```
instance PreAlphabet::eqv
by intro_classes

defs (overloaded) preAlphabet_sim_def : "x ~ y == eq x y"

instance PreAlphabet::equiv
apply(intro_classes)
apply(unfold preAlphabet_sim_def)
apply(rule eq_refl)
apply(rule eq_trans, auto)
apply(rule eq_symm, simp)
done

types Alphabet = "PreAlphabet quot"
```

First we instantiate `PreAlphabet` as the class `eqv` which allows us to define a relation $\sim$. Then we define this relation in terms on the `eq` function. In the next step we instantiate `PreAlphabet` as the class `equiv`, which comes with proof obligations that the $\sim$ relation is indeed an equivalence relation. Finally we create a type synonym called `Alphabet` as the quotient of `PreAlphabet`

6 Integration theorems

CSP processes communicate within the alphabet of communications. As the alphabet of communications is a quotient, CSP processes actually communicate equivalence classes. Arguing about the elements of the communications alphabet can therefore be difficult. However, CSP-CASL-semantics asks only three different questions on the alphabet of communications, see [Rog06]. The most prominent is the test whether two elements of the alphabet of communications are equal or not. This test, for example, is used when two processes synchronise.

In order for the end-user to be able to easily argue on the CSP-CASL process part they need to be able to easily test whether two equivalence classes are equal or not. CSP-CASL-Prover provides integration theorems which allow tests on the alphabet of communications to be lifted back to tests on the data from the HETS encoding. Figure 10 shows an example of one such integration theorem from the nucleus of the EP2 dialog.

```
lemma integration_theorem: "(class(C_B t1) = class(C_B t2)) =
                            (g__inj(t1) = g__inj(t2))"
apply(simp add: quot_equality)
apply(unfold preAlphabet_sim_def)
apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)
done
```

Fig. 10. Example of an integration theorem and it's proof.

The integration theorem of Figure 10 states that two equivalence classes, which are based on the type "data request" (as they have the form `C_B x`), are equal if and only if their underlying elements of the pre-alphabet are equal in their top most sort (i.e. `D_SI_Init`). Such data theorems and their proofs can be automatically generated by algorithms.

Proof practice shows that with these integration theorems available, reasoning about the behavioural aspects of a CSP-CASL specification becomes as easy (or challenging) as reasoning on data and processes separately, where reasoning on processes usually depends on theorems concerning data.

7 Dependencies

The following table shows the dependencies of the pre-alphabet construction and the integration theorems. $T(D)$ denotes that the theorem is dependent on the parameter in the column heading, while $T(I)$ expresses that the theorem is independent of the parameter in the column heading, and similar $P(\_)$ expresses the dependencies of the proofs on the parameter in the column heading.

	Specification	# of Sorts	Sub-sort Structure
Pre-Alphabet Construction	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$
eq_Reflexivity	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$
eq_Symmetry	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(I)$
eq_Transitivity	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$
Integration Theorems	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$

The reflexivity property of the `eq` relation is completely independent of the specification whereas the proof of symmetry relies only on the number of sorts and the proof of transitivity relies on the number of sorts and the sub-sort structure(indirectly). The integration theorems are the most dependent on the specification. All these proofs can be automatically generated by algorithms.

8 Proof of deadlock freedom of EP2

```
spec D_ACL_GetInitialisation =
    sorts SessionStart, SessionEnd,
          ConfigDataRequest, ConfigDataResponse < D_SI_Init
    forall x:ConfigDataRequest; y:SessionEnd . not (x=y)
    ops r: ConfigDataRequest; e: SessionEnd
end
ccspec sequential_system =
  data D_ACL_GetInitialisation
  channels C_SI_Init: D_SI_Init
  process
 let
    Abstract =
       C_SI_Init ! sessionStart: SessionStart -> Loop
    Loop =   C_SI_Init ! e -> SKIP
         |~| C_SI_Init ! r -> C_SI_Init ! response: ConfigDataResponse
                           -> Loop
  in Abstract
end
```

Fig. 11. CSP-CASL specification of a sequential system.

As an application of Csp-Casl-Prover we show how to prove deadlock freedom in an industrial setting. Here we prove deadlock freedom of the nucleus as shown in Figure 1. We have also proven deadlock freedom of the full EP2 dialog: the proof script scales up.

Our approach is to prove that, in the stable failures model $\mathcal{F}$, the nucleus is a refinement of the sequential system shown in Figure 11. Here, we have an `Abstract` process that sends a `SessionStart` value and then enters a loop. The `Loop` process either sends a `SessionEnd` message and terminates, or it sends a `ConfigDataRequest` message followed by a `ConfigDataResponse` message and then repeats the loop. `Loop` chooses internally, which of these two branches is taken. As this system has no parallelism it is impossible for it to deadlock. Process refinement within stable failures model preserves deadlock freedom. Hence if we can show that the EP2 nucleus is indeed a refinement of the sequential system, the EP2 nucleus is guaranteed to be deadlock free.

For our refinement proof we apply the algorithms discussed in this paper on both the EP2 nucleus as well as on the sequential system specification. Adding the integration theorems to Isabelle/HOL's simplifier set then allows us to prove deadlock freedom as shown in Figure 12 (we actually show more, namely that both systems are equivalent). This refinement proof involves recursive process definitions. These are first unfolded, then (metric) fixed point induction is applied. A powerful tactic from Csp-Prover finally discharges the proof obligation. The whole proof script involves syntactic proof techniques only.

```
theorem ep2: "Abs_System =F System"
apply (unfold System_def Abs_System_def)
apply (rule cspF_fp_induct_left[of _ "Abs_System_to_System"])
apply (simp_all)
apply (induct_tac p)
apply (tactic {* cspF_hsf_tac 1 *} | rule cspF_decompo |
       auto simp add: csp_prefix_ss_def image_iff inj_on_def)+
done
```

Fig. 12. Proof of deadlock freedom of the nucleus (see Figure 1).

9 Summary and future work

We have shown how to combine the tools HETS and Csp-Prover into a proof tool for Csp-Casl. The main challenges turned out to be the encoding of Csp-Casl's alphabet construction in Isabelle/HOL as well as the automated generation of integration theorems. The alphabet construction turns a many-sorted algebra into a flat set of communications. The integration theorems translate questions

on the alphabet of communications back into the language of many-sorted algebra. In both cases, we managed to come up with an algorithm that – take a Csp-Casl specification as their input – produce the required types, functions, theorems, and proofs in Isabelle/HOL. A case study on the EP2 system, for the moment carried out manually, demonstrates that our approach scales up on problems of industrial strength.

Future work will include the implementation of the algorithms described as well as further case studies on distributed computer applications. Another direction of work is to consider a semi-deep encoding of Casl. In such a setting the justification theorem which states that the relation `eq` is an equivalence relation becomes specification independent.

Acknowledgement Thanks to Temesghen Kahsai for his work on decomposition theorems for Csp-Casl refinement and also to Erwin R. Catesbeiana (jr) for his valuable insights into the very nature of electronic payment systems.

References

[BFG+05] B. Badban, W. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[ep202] *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[GRS05] A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in Csp-Casl. In *WADT 2004*, LNCS 3423, pages 61–78. Springer, 2005.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[IR05] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.

[IR06] Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *CONCUR'06*, LNCS 4137, pages 158–172. Springer, 2006.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, Hets. In *TACAS 2007*, LNCS 4424, pages 519–522. Springer, 2007.

[Mos02] T. Mossakowski. Relating Casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[Mos04] P. Mosses, editor. Casl *Reference Manual.* LNCS 2960 (IFIP Series). Springer, 2004.

[NPW02] T. Nipkow, L.C. Paulon, and M. Wenzel. *Isabelle/HOL.* LNCS 2283. Springer, 2002.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE-11*, LNAI 607, pages 748–752. Springer, 1992.

[Rog06] M. Roggenbach. CSP-CASL - A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[Ros98] A.W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.

Generalized Sketches and Model Driven Architecture

Adrian Rutle[1], Uwe Wolter[2], and Yngve Lamo[1]

[1] Bergen University College, p.b. 7030, 5020 Bergen, Norway aru@hib.no yla@hib.no
[2] University of Bergen, p.b. 7803, 5020 Bergen, Norway Uwe.Wolter@ii.uib.no

Abstract. The diversity and heterogeneity of modeling languages make the need for formal model definitions and mechanisms for automatic model integration and transformation more pressing than ever. These mechanisms are the cornerstones in Model Driven Architecture, which is a natural evolutionary step in raising the abstraction level of programming languages. This paper provides an introduction to Model Driven Architecture and to the various approaches that are central in the definition of models and model transformations. Then, a generic formalism, Generalized Sketches, will be presented for use in specifying modeling languages and their transformations. In addition, how concepts like instances, models and metamodels correspond to the Generalized Sketches formalism will be discussed.

1 Introduction

The rise of the abstraction level of programming languages was one of the most important trends in software development in the second half of last century. It started in the 50s with the replacement of raw machine code by assembly languages, which in their turn were replaced in the 60s by procedural programming languages. In the 80s object-oriented programming languages dominated the field. These changes in programming paradigms were viewed with sceptism by the programming communities. This sceptism was not without reason since the compilers of the time were not very good. Thus, many programmers doubted that the generated assembler code could be as efficient as handwritten code.

These changes in the programming paradigms affected the entire software engineering discipline; for example, flowcharts came with assemblers and diagrammatic modeling followed the object-oriented languages. Currently, building applications by first modeling them is considered one of the key evolutionary steps in raising the level of abstraction of the software development process to a higher level. Thus, models, model transformations, as well as automatization of model transformations are key issues in the current approach of software development process, which is standardized by the Object Management Group (OMG) as Model Driven Architecture (MDA) [6]. Various approaches for formalization of models and transformation definitions are proposed. Some of these approaches with features and shortcomings will be outlined in this paper. Then

the potentials of the formalism, Generalized Sketches (GS) [8], in MDA will be discussed.

The outline of the paper is as follows. Section 2 provides an introduction to MDA with a short explanation of models, model transformations as well as transformation definition languages. In Section 3, some OMG standards directly related to MDA are explained. Finally, Section 4 provides a gentle introduction to the theory of GS, a presentation of some generic approaches for model specification and transformation based on GS, as well as a comparison of the GS methodology to some OMG standards. The last section concludes the paper by summarizing the presented ideas, presenting the state-of-the-art of our project and our future work.

2 Model Driven Architecture (MDA)

MDA is a software development methodology in which modeling, model transformations and automatization of model transformations are important issues. In MDA, the first step in building applications is to construct abstract, platform independent models (PIM) of system properties and behavior. PIMs are transformed into one or more platform specific models (PSM) which are used by code generators to generate application code. In addition, PIMs can be used as a common communication basis among software designers, programmers and domain experts.

The advantages of MDA are many, including the fact that it enhances for domain specificity and platform independence of models which in turn make possible the separation of business logic from the application technologies. Another advantage is portability and cross-platform interoperability which facilitate the integration of different systems. Moreover, MDA increases productivity by letting the developers shift focus from code to models in which no platform-specific details need to be designed and written down. Computing infrastructures are expanding continuously and in every dimension, in response to business needs and developments in implementation technologies. Using MDA-based standards, business logic and application technologies can evolve independently of each other and organizations are able to integrate present systems with systems that will be built in the future [6].

2.1 The Development Process in MDA

The activities in the MDA development process (requirements specification, analysis, system design, implementation and coding, testing and deployment) are the same as those in the traditional development processes (Figure 1). The main difference is in the artifacts that are produced during and after each activity. In MDA, these artifacts are (formal) models – PIM, PSM and Code – while in the traditional development processes most of these artifacts before the implementation and coding phases are informal diagrammatic and textual documents.

These documents often lose their value quickly and are outdated soon after coding has started because of changes in the requirements, bug-fixing, evolution of the application and programmers' short cuts (the blue, dashed line in Figure 1).

Another difference is in the automatization of the steps in software development processes, from PIM to PSM (right column in Figure 1.) Since this is not always a feasible task, some human activity and heuristics is often needed in the first transformation processes.

The following three steps in MDA raise the abstraction level of the activities in the development process. The steps outlined below and the output models are defined by OMG [6]:

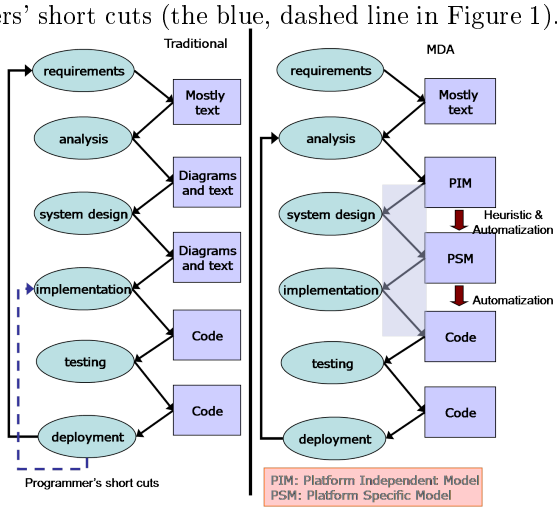


Fig. 1: MDA vs Traditional development process. Adopted from [5]

Platform Independent Model (PIM) The development process in MDA starts with the specification of an abstract, diagrammatic model which is independent of the application's platform. This kind of model is referred to as a Platform Independent Model (PIM) [5,6]. In PIM, one can specify the business logic of an application without restriction to a specific system design or implementation technology. Hence, PIM is a domain-specific model which is crucial for developing software systems where application logic and domain logic are separated. In addition, PIMs can be reused easily in other applications since they are not polluted with constructs from a specific implementation technology.

A PIM must be formal, unambiguous and consistent since it is used as input to transformation tools. Writing PIMs is a human activity which cannot be automated.

Platform Specific Model (PSM) The next step in the development process in MDA consists of specifying the transformations for transforming the PIM into a (set of) Platform Specific Model(s) (PSM). PSMs are also formal models, but they are restricted to a specific application platform; for example, Enterprise Java Beans (EJB,) .NET, Web Services or Relational Schemes. Since this transformation process builds a more concrete model from the abstract PIM, it can be considered as a refinement process.

Obviously, specifying transformations between PIM and PSM is not a minor task. However, these transformations must be defined only once for the modeling languages in which the PIMs and PSMs are written, then it can be used over and over again (possibly with minor changes.)

Code Generation The last step consists of the automatic transformation of the PSMs to application code. Considering the static aspects of software systems, this step is usually straightforward because of the similarity between PSMs and Code. In order to transform the dynamic aspects of a system automatically, both the behavioral and structural aspects of the system must be modeled in PIMs and PSMs. This task requires modeling languages (or specification formalisms) which have formal definitions for constructs that can be used to model the dynamics of software systems. At the time of writing, it's possible to model only a fraction of the dynamics of software systems formally using the existing modeling languages.

MDA includes the two development steps; PIM-to-PSM and PSM-to-Code, rather than generating code directly from PIM in order to bridge the large abstraction gap between PIM and Code, to allow for the modularization of transformations, as well as for debugging purposes [2].

2.2 Model Transformations

As mentioned above, the steps in MDA involve transformations between models, i.e. the generation of a target model from a source model. These transformations are carried out automatically by tools in transformation processes. Each transformation process is described by a transformation definition, which in turn consists of a set of transformation rules. The transformation definition is written in a transformation definition language. The transformation rules define how (and which) constructs from a source model are transformed to (which) constructs in a target model.

Research in this field has established a set of features which the languages used for writing transformation definitions should provide [5,6]. Some of these features are the following:

- Tunability: applications of the transformation rules must be adjustable to provide flexibility and more user control.
- Traceability: it should be possible to trace target model constructs back to their counterpart construct(s) in the source model. During bug-fixing, traceability will help the developers to find which part of the PIM is the source of errors in the generated code.
- Incremental consistency: changes in the target model, for example hand-written code, must persist in spite of re-transformation.
- Bidirectionality: the source model can be generated from the target model by application of the inverse of the transformation. This feature is useful in the case of reverse engineering, but it is difficult to achieve.
- Rule scheduling: transformation rules can be applied in a user-defined sequence. This feature is important for optimization purposes since it provides facilities for adding a hierarchy or structure to the rules and the sequence in which they are applied [2].

Design and specification of transformation definition languages is a relatively new field in software engineering. OMG's initial request for proposals in 2002 on Query, Views and Transformations was the first call for a standardization of transformation definition languages. A large number of approaches (for example; GreAT, UMLX, VIATRA, ATL, QVTP etc) have been proposed in reply to the OMG's request, however, many of those proposals were already forced by practical needs independent of the OMG's proposal.

These approaches are categorized into two major categories: Model-To-Model and Model-To-Code [2]. The latter can be considered a special case of the former where the target metamodel is the metamodel of a programming language. Different approaches such as relational, graph-transformation-based and hybrid are used in the Model-To-Model category.

The relational approach is a declarative approach in which the main concepts are mathematical relations and mapping rules based on set-theory. Relations between element types from the source and target models are stated in mathematical relations which are specified by constraints. This approach has the advantage of a good balance between declarative expressiveness, flexibility (rule scheduling) and simplicity [2].

The graph-transformation-based approach is also a declarative approach which is inspired by theoretical work on graph transformations between typed, directed graphes [2]. In this approach, the models to be transformed are graphs. Graph transformation rules define patterns in the source graph that will be transformed to patterns in the target graph. This approach is extremely powerful and declarative, but is also very complex [2].

Hybrid approaches where different concepts and paradigms are applied depending on the application domain seem to be more useful. In the hybrid approach, users can combine the expressive power of graph-based transformations with the flexibility of the relational approach to design their transformation definitions.

Some examples of transformation definition languages – Query, Views and Transformations (QVT) and ATLAS Transformation Language (ATL) – will be outlined in the next sections.

3 MDA and OMG Standards

The challenges in MDA are to obtain a formalism for specifying the models and choosing mechanisms for definition of (and automatically execution of) transformations between these models. OMG has some standards that are directly related to MDA and these challenges.

3.1 The Meta Object Facility

The OMG has defined a special language, the Meta Object Facility (MOF) to describe all other languages specified by OMG. MOF is also reflective, which means it is capable of describing itself. According to OMG, MDA compliant

languages must be MOF-based, i.e. instances of the MOF model in order to enable automatic model transformations [6].

There are four layers of metamodeling defined by the OMG architecture (Table 1). The M_0-level, an instance where one defines the running system, must conform to a model. This model is at the M_1-level; it is a structure specifying what instances should look like. This model, in its turn, must conform to a metamodel, which is a model at the M_2-level, against which models can be checked for validity, and which can also be used to specify models. Metamodels correspond to modeling languages, for example UML and Common Warehouse Model (CWM). The highest level of metamodeling which is defined by OMG is the M_3-level, where MOF is located.

OMG levels	OMG Standards/examples
M_3	MOF
M_2	UML language
M_1	A UML model: Class "Person" with attributes "name" and "address"
M_0	An instance of "Person": "Ola Nordmann" living in "Sotraveien 1, Bergen"

Table 1: OMG metamodeling levels

3.2 UML and the Object Constraint Language

OMG suggests UML, in combination with the Object Constraint Language (OCL), as a language for writing PIMs. This approach involves combining the text-based language OCL with the graphical language UML to formally describe static aspects of software systems. In some cases, the dynamics of software systems can also be expressed using the UML-OCL combination – by pre- and post-conditions on operations. In most cases, however, the body of the operations must be written manually in the PSM.

3.3 Query, Views and Transformations (QVT)

QVT is an OMG standard proposed for describing transformation definitions. QVT is currently in the finalization phase. MOF 2.0 is used to define the abstract syntax of QVT, and OCL is used for querying the models and implementing the transformations. QVT is composed of three languages: Relations, Core and Operational Mappings. The first two are declarative languages and the third is imperative. Relations language is at a higher level of abstraction than the Core language. The semantics of Relations language is described as a transformation into the Core language, a transformation that may be defined in the Relations language itself [4]. The semantics of the Core language is given in a semi-formal set-theoretical notation [7]. The Relations language defines transformations as a set of relations (each containing a set of patterns) among models.

The Operational Mappings language and the Black Box implementation extend the Relations and Core languages, and are mechanisms intended to define

transformations that are difficult to express in the Relations language. Traceability links are handled automatically by the Relations and Operational Mappings languages, while these links must be handled manually in the Core language [7]. Rules in the Relations and Core languages are multidirectional, while they are unidirectional in the Operational Mappings.

4 Generalized Sketches (GS)

Since models in MDA are not only for documentation, but are also used as input to transformation tools, there is a tremendous need for a formal, generic specification formalism which also supports the definition of transformations. This section provides a gentle introduction to GS. Then the potentials of GS as a generic approach for the specification of models and metamodels of modeling languages as well as their transformations will be discussed.

GS is a graph-based specification format that borrows its main ideas from both categorical and first-order logic, and adapts them to software engineering needs [8]. The claim behind GS is that any diagrammatic specification technique in software engineering can be viewed as a specific instance of the GS specification pattern. GS is a pattern, i.e. generic, in the sense that we can instantiate this pattern by a signature that corresponds to a specific specification technique, like UML class diagrams, ER diagrams or XML. The technique in which a modeling language is represented by a signature in GS is called "sketching" the modeling language [3]. Signatures and sketches in the GS framework are defined as follows:

Definition 1. *A signature $\Sigma := (\Pi, ar)$ is an abstract structure consisting of a collection of predicate symbols Π with a mapping that assigns an arity (graph) $ar(p)$ to each predicate symbol $p \in \Pi$.*

Definition 2. *A diagram (p, δ) labeled with the predicate p in a graph $G(S)$ is a graph homomorphism $\delta : ar(p) \rightarrow G(S)$ where $ar(p)$ is the arity of p.*

Definition 3. *A Σ-sketch $S := (G(S), S(\Pi))$, is a graph $G(S)$ with a set $S(\Pi)$ of diagrams in G(S) labeled with predicates from the signature Σ.*

Definition 4. *A $\Sigma-$sketch morphism $\phi : S_1 \rightarrow S_2$ between two $\Sigma-$sketches $S_1 = (G(S_1), S_1(\Pi))$ and $S_2 = (G(S_2), S_2(\Pi))$ is a graph homomorphism $\phi : G(S_1) \rightarrow G(S_2)$ compatible with marked diagrams, i.e., $(p, \delta : ar(p) \rightarrow G(S_1)) \in S_1(\Pi)$ implies $(p, \delta; \phi : ar(p) \rightarrow G(S_2)) \in S_2(\Pi)$ for all diagrams $(p, \delta) \in S_1(\Pi)$.*

***Ske*(Σ)** is used to denote the category of all $\Sigma-$sketches, where objects are Σ-sketches and morphisms are $\Sigma-$sketch morphisms.

Table 2 shows signature Σ_{UML} which consists of the predicates [objectNode], [valueNode], [cover] etc. These predicates allow to specify most of UML class diagrams. The arities of the predicates are shown in the second column and the third column shows a possible visualization of these predicates. In the fourth column, the semantics of each predicate is specified.

Sematically, arrows are interpreted as multivalued functions $f : A \rightarrow \wp(B)$, i.e. a constraintless arrow stands for an arbitrary multivalued function.

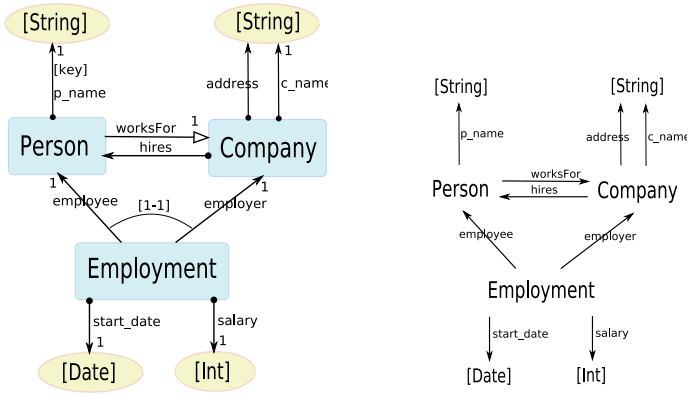
name	**arity**	**visualization**	**semantic**
[objectNode]	1	$\boxed{A}$	set of objects
[valueNode]	1	(A)	set of values
[total]	$1 \xrightarrow{x} 2$	$\boxed{A} \bullet\!\!\xrightarrow{f} \boxed{B}$	$\forall a \in A : \exists b \in B \mid b \in f(a)$
[key]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f\ [key]} \boxed{B}$	$\forall a, a' \in A : a \neq a'$ implies $f(a) \neq f(a')$
[singlevalued]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f\ \ 1} \boxed{B}$	$\forall a \in A : \lvert f(a)\rvert \leq 1$
[cover]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f} \triangleright \boxed{B}$	$\forall b \in \wp(B) : \exists a \in A \mid b \in f(a)$
[inverse]	$1 \underset{y}{\overset{x}{\rightleftarrows}} 2$	$\boxed{A} \underset{g}{\overset{f}{\underset{[inv]}{\rightleftarrows}}} \boxed{B}$	$\forall a \in A$, $\forall b \in B : b \in f(a)$ iff $a \in g(b)$
[disjoint-cover]	$1 \xrightarrow{x} 2$, $2 \xrightarrow{y} 3$	$\boxed{A} \xrightarrow{f} \boxed{B} \xleftarrow{g} \boxed{C}$	$\bigcup\{f(a) \mid a \in A\} \cap \bigcup\{g(c) \mid c \in C\} = \emptyset$
[jointly-mono] or [1-1]	$1 \xrightarrow{x} 2$, $1 \xrightarrow{y} 3$	$\boxed{A} \xrightarrow{f} \boxed{B}$, $\boxed{A} \xrightarrow{g} \boxed{C}$	$\forall a, a' \in A : a \neq a'$ *implies* $f(a) \neq f(a')$ or $g(a) \neq g(a')$

Table 2: Signature Σ_{UML}.

Example 5. The Σ_{UML}-sketch $S = (G(S), S(\Pi))$ in Figure 2a specifies the class diagram of a simplified software system of persons, companies and employments. The carrier graph $G(S)$ is shown in 2b. Some of the diagrams in the set $S(\Pi)$ are:

- $([objectNode], \delta_1 : (1) \mapsto (Person))$
- $([objectNode], \delta_2 : (1) \mapsto (Company))$
- $([valueNode], \delta_3 : (1) \mapsto ([String]))$
- $([total], \delta_4 : \ (1 \xrightarrow{x} 2) \ \mapsto \ (Person \xrightarrow{p\_name} [String]) \ )$
- $([total], \delta_5 : \ (1 \xrightarrow{x} 2) \ \mapsto \ (Company \xrightarrow{hires} Person) \ )$
- $([singlevalued], \delta_6 : \ (1 \xrightarrow{x} 2) \ \mapsto \ (Employment \xrightarrow{employee} Person) \ )$
- $([singlevalued], \delta_7 : \ (1 \xrightarrow{x} 2) \ \mapsto \ (Employment \xrightarrow{employer} Company) \ )$
- $([1-1], \delta_8 : \begin{pmatrix} 1 \xrightarrow{x} 2 \\ \downarrow y \\ 3 \end{pmatrix} \mapsto \begin{pmatrix} Employment \xrightarrow{employer} Company \\ \downarrow employee \\ Person \end{pmatrix})$

In the class diagram in Figure 2a every person may work for *zero or one* company, but every company must hire *one or more* persons. The first constraint



(a) A Σ-sketch S (b) The carrier graph $G(S)$

Fig. 2: A sketch and its carrier graph

is set by the predicate [singlevalued] on the arrow *worksFor*. While the second constraint is set by the predicate [total] on the arrow *hires*.

4.1 Models and Metamodels in GS

In GS, software models are represented by $\Sigma-$sketches where Σ is the signature which represents the modeling language used to specify the model. Instances of models and the semantic interpretation of their signatures are explained/formalized in the next definitions.

Definition 6. *The semantic interpretation of a signature $\Sigma := (\Pi, ar)$ is given by a mapping that assigns to each $p \in \Pi$ a set $[\![p]\!]$ of graph homomorphisms $\tau : O \to ar(p)$ which are called valid instances of p, where O may vary over all graphs.*

Example 7. Consider the predicates [total] and [cover], both with arity $1 \xrightarrow{x} 2$. A function $f : \{a_1, a_2\} \to \{b_1, b_2\}$ with $f(a_1) = f(a_2) = b_1$ is represented by a bipartite graph, i.e. as a graph homomorphism from the "graph of f" into $1 \xrightarrow{x} 2$. For example,

$$\tau_1 : \begin{pmatrix} a_1 & \xrightarrow{e_1} & b_1 \\ & \overset{e_2}{\nearrow} & \\ a_2 & & b_2 \end{pmatrix} \to (1 \xrightarrow{f} 2)$$

with $\tau_1(a_1) = \tau_1(a_2) = 1$ and $\tau_1(b_1) = \tau_1(b_2) = 2$. $\tau_1 \in [\![[total]]\!]$ since both a_1 and a_2 are mapped to some elements of the set $\{b_1, b_2\}$, while $\tau_1 \notin [\![[cover]]\!]$ since b_2 is not in the image of any element from the set $\{a_1, a_2\}$

Definition 8. *An instance of a sketch S is a graph I together with a graph morphism $\iota : I \to G(S)$, where $G(S)$ is the carrier graph of S, such that $\iota^* \in [\![p]\!]$, i.e. ι^* is a valid instance of p, for each diagram $\delta : ar(p) \to G(S)$ where ι^* is given by the pullback diagram in Figure 3.*

Thus, for I to be a valid instance of S, for each diagram δ in $S(\Pi)$, the part of I related to the diagram δ, must be a valid instance of p. However, the meaning of being a valid instance of a predicate remains to be specified by the designer of the signature as explained in Definition 6 and Example 7.

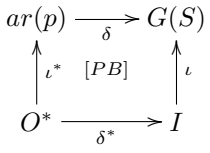


Fig. 3: Instance of Sketch S

Example 9. Figure 4 shows an example of an instance of the sketch S (from Figure 2a)[3]. To verify this, construct the pullback for each diagram (p, δ) in $S(\Pi)$. E.g. the pullback of $ar([1-1]) \to G(S) \leftarrow I$ will consist of $ar([1-1]) \leftarrow O^* \to I$ (see Figure 3) where $\iota^* : O^* \to ar([1-1])$ is as in Figure 5. The set of the spans represents a valid instance of the predicate [1-1] since the semantic of the constraint which is set by the predicate is not violated. The constraint here is that every Employment element is uniquely identified by a pair of elements from the sets Person and Company.

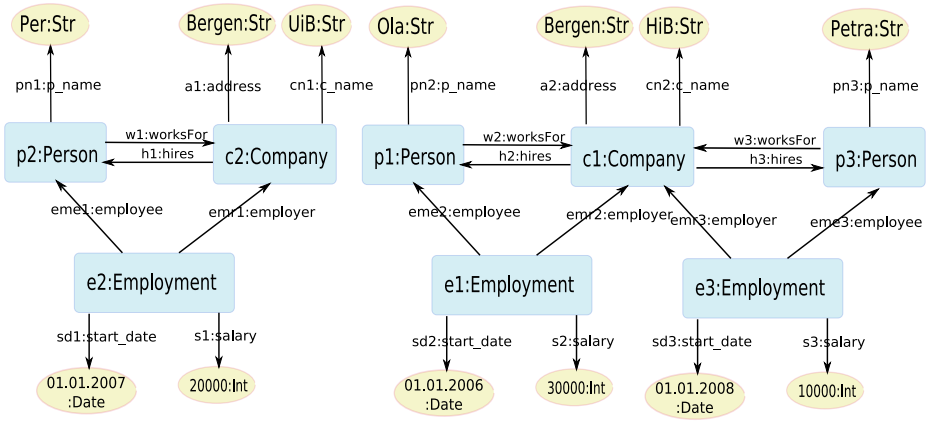


Fig. 4: An instance, $\iota : I \to G(S)$, of the Σ-sketch S

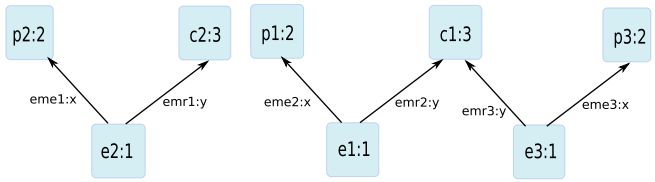


Fig. 5: $\iota^* : O^* \to ar([1-1])$

[3] Notice that $(p1 : Person)$ is a "user-friendly" notation for the assignment $(\iota : p1 \mapsto Person)$

Metamodeling is a mechanism for defining graphical modeling languages which is used in the way grammars in Bakus Naur Form (BNF) are used to define text-based languages such as programming languages [5]. A BNF grammar describes which series of tokens are valid expressions in a language. In the same way, a metamodel describes which graphs are valid models in a given modeling language. A fundamental difference here is that the representation of the structure of text-based languages is based on terms (abstract syntax trees,) while graphical languages have a graph-like structure which makes it impossible to apply BNF for their representation [1]. The metamodels of the graphical languages are usually represented by typed graphs.

The concepts of models and instances of models are present in the GS formalism. However, to complete the task of the formalization of modeling languages it is also necessary to introduce the concept of metamodeling. This is because signatures alone are not enough to set all restrictions on sketches. In other words, if a modeling language L is represented by a signature Σ_L, then $\boldsymbol{Ske(\Sigma_L)}$ may contain more Σ-sketches than intended to be modeled by L. In the proposed formalization of modeling via GS, this means that for any signature Σ a meta-signature Γ_Σ and a Γ_Σ-sketch M_Σ must be found such that the intended subset of $\boldsymbol{Ske(\Sigma)}$ is described by the instances of M_Σ.

Definition 10. *A signature $\Gamma_\Sigma = (\Pi_\Sigma, ar)$ together with a $\Gamma_\Sigma-$sketch $M_\Sigma = (G(M_\Sigma), M_\Sigma(\Pi_\Sigma))$ is the metamodel of $\boldsymbol{Ske(\Sigma)}$ iff for any instance $\iota : I \to G(M_\Sigma)$ of M_Σ there exists a Σ-sketch $S^\iota = (I, S^\iota(\Pi_\Sigma))$.*

The MDA vision of OMG suggests MOF as the metamodel for modeling languages like UML and CWM (Section 3). This means to define a signature Γ_{MOF} such that $\Gamma_{\Sigma_{UML}} \subseteq \Gamma_{MOF}$ and a $\Gamma_{\Sigma_{UML}}$-sketch $M_{\Sigma_{UML}}$ (or $M_{\Gamma_{MOF}}$) such that all Σ_{UML}-sketches are instances of $M_{\Sigma_{UML}}$. Table 3 shows a comparison of the four layers of metamodeling from the OMG architecture with the GS methodology.

Instances of metamodels are defined in the same way as instances of models as in definitions 6 and 8.

OMG levels	**OMG standards**	**GS methodology**
M_3	MOF	Γ_{MOF} where $\Gamma_{\Sigma_{UML}} \subseteq \Gamma_{MOF}$
M_2	UML	A $\Gamma_{\Sigma_{UML}}$-Sketch $M_{\Sigma_{UML}}$ representing the metamodel of UML
M_1	A UML model	An instance $\iota : G(S) \to G(M_{\Sigma_{UML}})$ of $M_{\Sigma_{UML}}$
M_0	An instance	An instance $\iota' : I \to G(S)$ of the Σ_{UML}-sketch $S^\iota = (G(S), S^\iota(\Pi))$

Table 3: OMG metamodeling levels vs GS

It should be noted that in the OMG architecture, each metamodeling layer is defined as an instance of the layer above itself. Thus, an instance of a UML class model for example, is defined as $\iota : I \to G(S)$, without taking $\delta(\Pi)$ and the pullback diagram in Figure 3 into consideration, which means that a valid instance only needs to be valid syntactically.

4.2 Model and Metamodel Transformations in GS

Since models and metamodels are represented by sketches, both model and metamodel transformations correspond to sketch morphisms (Definition 4).

Proposition 11. *(Figure 6)If $\phi : S_1 \to S_2$ is a sketch morphism, and $\iota_2 : I_2 \to G(S_2)$ is an instance of S_2, then $\phi^\bullet(\iota_2 : I_2 \to G(S_2)) = \iota_1 : I_1 \to G(S_1)$ is an instance of S_1, where $\phi^\bullet$ is the reduct transformation of ϕ [8].*

Proof. Having an instance I_2, a graph morphism $\iota_1 : I_1 \to G(S_1)$ can be constructed by the pullback PB_1, as shown in the right side of Figure 6. In addition, since I_2 is a valid instance of $G(S_2)$, we have $\iota \in [\![P]\!]$ for all $\delta;\phi$ where the span $ar_1(p_1) \leftarrow O \to I_2$ is a pullback of the sink $ar_1(p_1) \to G(S_2) \leftarrow I_2$. This ensures the validity of I_1 as an instance of $G(S_1)$ since we have $\iota \in [\![P]\!]$ and the pullback PB_2. We have the pullback PB_2 by a well-known result from CT; if the external rectangle and the right square are pullbacks, then the left square is also a pullback.

Definition 12. ***Inst*(S)** *is the category of all instances $\iota : I \to G(S)$ of a sketch S in which objects are instances ι and morphisms are commutative diagrams.*

Theorem 13. *Any sketch morphism $\phi : S_1 \to S_2$ defines a functor $\phi^\bullet :$ **Inst**(S_2) $\to$ **Inst**(S_1) which is called reduct transformation.*

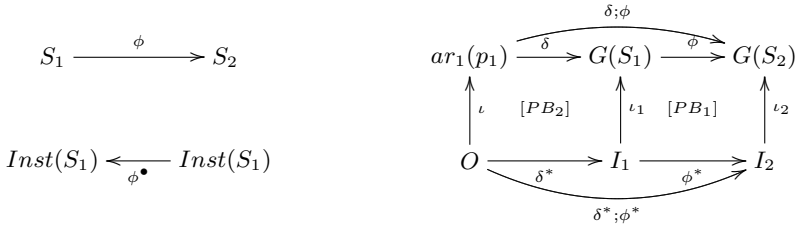


Fig. 6: Generic Model Transformation

Some model transformations can be achieved by the reduct transformation. However, most model transformations in software engineering are used for model extension and thus require a morphism in the other direction, that is, a morphism $\phi^\circ :$ **Inst**$(S_1) \to$ **Inst**(S_2) . In the case of conservative extension, this arrow is persistent, i.e. $\phi^\circ;\phi^\bullet = id_{Inst(S_1)}$. This constraint is too restrictive for practical purposes, i.e. it does not cover all interesting model transformations. The semantical conditions for model extensions will be a topic of a future work.

4.3 Generalized Sketches and MDA

Since GS can be used to specify modeling languages and transformations between them, and since it is a generic specification format, the focus of this section will

be on using GS to specify PIMs, PSMs and Code[4], and the transformations between them.

The metamodels of the languages of PIMs, PSMs and Code can be represented by sketches S_{PIM}, S_{PSM} and S_{Code} respectively. As mentioned in section 4.2 the transformations between PIMs, PSMs and Code are represented by sketch morphisms. As evident from Figure 6, model transformations based on their metamodel transformations can be achieved by defining sketch morphisms between sketches, $\phi_1 : S_{PIM} \rightarrow S_{PSM}$ and $\phi_2 : S_{PSM} \rightarrow S_{Code}$. Then a sketch morphism $\phi^*_{pim-psm} : PIM \rightarrow PSM$ will represent the transformation between PIM and PSM, given that $PIM \in Inst(S_{PIM})$ and $PSM \in \boldsymbol{Inst(S_{PSM})}$. The same procedure can be applied to transform PSM to Code. The automatization of the transformation is given by the automatic construction of the reduct transformations of ϕ_1 and ϕ_2.

In addition to the features mentioned in Section 2, transformation definitions which are specified in GS will ensure compositionality of rules. Compositionality is defined as follows. If the transformation t_1 transforms a model m_1 to m_2 and t_2 transforms m_2 to m_3, then the composition of the transformations, written $t_1; t_2$, must transform m_1 to m_3. This property facilitates stepwise refinement, simplifying the transformation by applying smaller, simpler transformations in multiple steps. This feature also simplifies the verification of transformations since each step will preserve correctness independent of its neighboring steps. For example, for $T = t_1; t_2; t_3$, if t_1, t_2 and t_3 are correct, then T is also correct. On the other hand, features that are providing mechanisms for rule scheduling, in which transformation rules can be applied in a user-defined sequence, are not needed since transformation rules correspond to mappings between diagrams and the sequence of their execution is handled automatically.

5 Conclusion

GS may be considered a suitable specification formalism to define all other diagrammatic modeling languages with a strong mathematical foundation, first, because models and metamodels in software engineering are graph-based, and also because GS is based on Category Theory (CT), which is the mathematics of diagrammatic notations.

Models which are specified using a specific specification technique will appear as a (possibly ambiguous) visualization of a sketch which is parameterized by the corresponding signature. Thus, the claim is that GS can be used as a standard notation for representing both the syntax and the semantics of diagrammatic specification languages, since the syntax and the semantics of GS are well-defined and unambiguous.

The (meta)model transformation approach in GS is expressed by sketch morphisms. The power of GS lies in its genericness, which makes it applicable to all modeling languages and their transformations. This is because the relations

[4] By regarding programming languages as modeling languages.

between constructs from the source and target (meta)models can be expressed as functors (morphisms between categories). Thus, unlike the OMG standards which require MOF-compliance of modeling languages to enable transformation between models, using GS makes possible to relate models written in any modeling language.

At the time of writing, there was no implementation of QVT. The QVT specification is so huge and complicated that it may not be possible for a single tool to meet all the requirements in a reasonable time. In addition, as experience with programming languages has shown, no single language can fit all application domains. Modeling languages and transformation definition languages are not exceptions of this rule. Rather than writing a standard specification for a language that all tool vendors must implement in order to obtain interoperability, therefore, we would suggest that a generic formalism for the specification of transformation definitions based on CT and mappings between sets is a better approach for standardization.

By developing tools that support GS as a generic pattern for specifying and developing diagrammatic specification techniques, we can prove and exploit the practical value of GS in all aspects of (meta)modeling and MDA, such as model transformation and integration, as well as model decomposition and modularization.

One major focus of our project is on developing tools which can be used to design signatures corresponding to existing specification techniques, like UML class diagrams, SQL Schemas and ER diagrams. Designing signatures for existing modeling languages (the so-called "sketching") involves exhaustive exploration of the syntax and semantics of those languages to identify a set of predicates – for example, total, partial, jointly mono, disjoint-cover, etc – which are needed to express all properties that can be expressed by them. Preferred graphical notations (or visualizations) for the predicates can then be chosen. Diagrammatic models can then be specified using the signatures/specification techniques. The tool will also support the definition of relations between metamodels, which correspond to transformation definition; as well as automatic construction of pullback, which may be used to both construct and identify valid instances of sketches.

In a future work, existing transformation languages like QVT and ATL as well as concrete transformation rules specified by these languages will be studied. In addition, the capabilities of GS to analyze constraints on rules, correctness of transformations, as well as checking for ambiguities and contradictions in transformation rules will be investigated.

References

1. Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT*, pages 431–433, 2004.
2. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In OOPSLA 2003, editor, *Generative Techniques in the context of MDA*, 2003.

3. Zinovy Diskin. Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Research Report M-97, Faculty of Physics and Mathematics, University of Latvia, August 1997.
4. Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, volume Model transformation (MT 2006), pages 1188 – 1195. ATLAS Group, INRIA and LINA, University of Nantes, ACM Press, 2006.
5. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: practice and promise*. Addison-Wesley, 1 edition, April 2003.
6. Object Management Group (OMG). Model Driven Architecture Guide. (1.0.1), june 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.
7. Object Management Group (OMG). MOF Query, Views and Transformations. Number 2.0, November 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.
8. Uwe Wolter and Zinovy Diskin. Generalized sketches: Towards a universal logic for diagrammatic modeling in software engineering. 2007. Proceedings, ACCAT 2007, ENTCS, Submitted.

Objects Versus Abstract Data Types: Bialgebraically

Ondrej Rypacek

School of Computer Science, University of Nottingham, NG8 1BB, UK
oxr@cs.nott.ac.uk

Abstract. Algebraic data types and catamorphisms (generic folds) play a central role in functional programming as they allow programmers to define recursive tree-like data structures and operations on them uniformly by structural recursion. Likewise, in object-oriented programming recursive hierarchies of objects play a central role for the same reason, although the execution is quite different. There is a duality between these two approaches which we formalise using a distributive law and define a notion of behavioural equivalence of the dual functional and object-oriented programs. In passing we also show how to put an algebraic structure on an object hierarchy.

1 Introduction

Algebraic data types and catamorphisms (generic folds) [9] play a central role in functional programming (FP) as they allow programmers to define recursive tree-like data structures and operations on them uniformly by structural recursion. Program development is often centered around data types, which are fixed upfront while operations are defined later. Likewise in object-oriented programming (OOP), recursive hierarchies of objects play a central role for the same reason, although the realisation is quite different. Here, the operations are fixed upfront and define a single common interface while the different kinds of data that can appear in the data structure are defined as implementations of the interface. Objects (instances of these implementations) are then linked in a tree-shaped pattern where method calls are recursively propagated down the tree. This is called the *Composite pattern* in object-oriented software engineering and described informally in [5].

In this paper, we present a formalisation of this situation using the category-theoretical notion of a bialgebra. We illustrate the idea informally in Section 2, where we go through a simple example of a recursive data structure. We observe, and in Section 3 formalise, that the behaviour of (operations on) such a data structure is often defined by a single natural transformation: a distributive law of the functor representing the signature of the data-structure over the behaviour functor. This distributive law gives rise to two arrows: one corresponds to the functional program – an abstract datatype – while its dual corresponds to the object-oriented program. We give a direct computational proof of their behavioural equivalence.

The example we start with in Section 2 is an instance of a strictly more general situation, previously described in the context of bialgebraic semantics [12, 13], where one considers distributive laws between a monad freely generated by the signature and a comonad cofreely generated by the behaviour. We introduce this generalisation in Section 4. Moreover, we show how our proof of the equivalence from the simplified setting generalises straightforwardly to the more general monadic case. In Section 6 we conclude and sketch directions for further work.

Our contribution is two-fold:

1. We contribute to understanding of the relation between functional and object-oriented programming and formally connect two important phenomena in both worlds: namely a large class of generalised folds and a large class of instances of the Composite pattern. In passing we also show how to put an algebraic structure on an object hierarchy.
2. We give an intuitive presentation of bialgebras from the programmers' point of view. We provide a very concise calculational proof of *adequacy* in the sense of Turi and Plotkin [13]. Although this is not a new result, we believe our presentation will appeal to many readers because of its straightforward computational nature.

Throughout the text, we assume the reader to be conversant with basic categorical notions such as functors, natural transformations, adjunctions, monads and algebras of a functor. Steve Awodey's Category Theory [1] is an excellent reference for the purpose.

2 Motivating Example

Consider the following object interface of a memory cell.

```
interface MemCell {
  attribute get : () -> nat
  method    set : nat -> ()
}
```

Here, in a hypothetical object-based language, `MemCell` is an object interface and also a name for the type of objects with the given interface. Attribute `get` is an observation of type `nat` on `MemCell`s. As an attribute, it can't change the local state of the object it's being invoked on. Method `set` is an operation with a possible side-effect in the local state of the object. It is parameterised by a natural number. Note that the interface doesn't specify the behaviour of objects carrying it, apart from the suggestive names of the operations. All implementations[1] with the given signature are admissible. Following are two examples.

[1] We avoid the term *class* here, because a *class* in class-based languages defines both a type and its implementation. We keep implementations and types separate.

```
implementation SimpleMC {
  var n : nat
  get    = n
  set n' = do { n := n'; return ();}
} : MemCell

implementation CompositeMC {
  var l : MemCell
  var r : MemCell
  get    = (l!get + r!get)
  set n' = do { l!set n'; r!set n'; return (); }
} : MemCell
```

`SimpleMC` is just a simple memory cell whose `get` attribute returns the current value and its `set` method resets the cell to the number given. `CompositeMC` combines two other (arbitrary) MemCells. Its `get` value is the sum of its components' values, its `set` method sets both components.

In the above definitions, we use a variation of the Haskell do-notation for monads in definitions of methods with local side-effects. Informally speaking, it is assumed that the monad in question is a *state monad* where the state is defined by `var` declarations in the definitions of the implementations: a single `nat` and a pair of `MemCell`s respectively. Also method calls, as in `l!set` are assumed to be monadic in that they update the local state of the calling object with the new state of the called object.

The above example illustrates a standard approach to recursive data structures in OOP (c.f. *Composite pattern* in [5]). In functional programming, one would typically model the same data and operations using an algebraic data type and a pair of functions defined by induction on the data. The following is a definition in Haskell.

$$\texttt{data BTree} \quad = \quad \texttt{Val nat} \mid \texttt{Node BTree BTree} \tag{1}$$

$$\begin{aligned} &\texttt{getf} && : \texttt{BTree} \to \texttt{nat} \\ &\texttt{getf (Node l r)} && = (\texttt{getf l}) + (\texttt{getf r}) \\ &\texttt{getf (Val n)} && = \texttt{n} \end{aligned} \tag{2}$$

$$\begin{aligned} &\texttt{setf} && : \texttt{nat} \to \texttt{BTree} \to \texttt{BTree} \\ &\texttt{setf n (Node l r)} && = \texttt{Node (setf n l) (setf n r)} \\ &\texttt{setf n (Val m)} && = \texttt{Val n} \end{aligned}$$

Function `getf` adds all numbers in a `BTree` and `setf` replaces all leaf values in the tree with the given value.

Intuitively, we can "see" that the object-based and functional views on trees of numbers with two operations *get* and *set* are just different models of the same data structure. In the rest of this section we formalise this data structure as two functors and a distributive law between them from which the two models can be canonically derived. In the following sections, we generalise the construction.

But first we must introduce a suitable formalism for objects. We adopt the coalgebraic view of functional objects, where object interfaces are modeled as *interface endo-functors*, and implementations are modeled as *coalgebras* of these functors. See e.g. [11, 7] for a gentle introduction, or [10] for a type-theoretical view. In the following text, we introduce the required notions as we go along. For simplicity, we work in the category **SET** of sets and total functions.

Interfaces of functional objects with local state induce so-called *interface functors*, which are products of function types, one for each attribute or method in the signature. For instance, the interface functor corresponding to `MemCell` is the following.

$$\mathsf{MemCellF}.X \quad \triangleq \quad \mathsf{nat}^{1} \times (1 \times X)^{\mathsf{nat}} \quad \cong \quad \mathsf{nat} \times X^{\mathsf{nat}} \tag{3}$$

Here X is a placeholder for the type of the local state of the object, nat is a constant. We use dot "." for functor and function application whenever we feel it improves readability.

In the above functor, the first component of the product corresponds to the observation `get`; the second component, X^{nat}, corresponds to the method `set`, which produces a new value of the local state given a parameter of type nat. An implementation of `MemCell` with local state of type A defines a MemCellF-coalgebra [2]: an arrow

$$\varphi : A \longrightarrow \mathsf{nat} \times A^{\mathsf{nat}}$$

The terminal MemCellF-coalgebra:

$$\mathsf{out}_{\mathsf{MemCellF}} : \mathsf{MemCell} \longrightarrow \mathsf{nat} \times \mathsf{MemCell}^{\mathsf{nat}}$$

corresponds to the abstract object type, MemCell, and a pair of message sending functions:

$$\begin{array}{lcl} \mathsf{send\_get} & \triangleq & \pi_1 \circ \mathsf{out}_{\mathsf{MemCellF}} \\ \mathsf{send\_set} & \triangleq & \pi_2 \circ \mathsf{out}_{\mathsf{MemCellF}} \\ \mathsf{out}_{\mathsf{MemCellF}} & = & \langle\, \mathsf{send\_get}, \mathsf{send\_set} \,\rangle \end{array} \tag{4}$$

Here, $\langle\, f, g \,\rangle$ denotes the *tuple* of f and g. As MemCellF is a polynomial functor, its terminal coalgebra in **SET** exists and $\mathsf{out}_{\mathsf{MemCellF}}$ is an isomorphism.

The implementations `SimpleMC` and `CompositeMC` can now be rewritten as coalgebras $\mathsf{s} : \mathsf{nat} \longrightarrow \mathsf{nat} \times \mathsf{nat}^{\mathsf{nat}}$ and $\mathsf{c} : \mathsf{MemCell} \times \mathsf{MemCell} \longrightarrow \mathsf{nat} \times (\mathsf{MemCell} \times \mathsf{MemCell})^{\mathsf{nat}}$

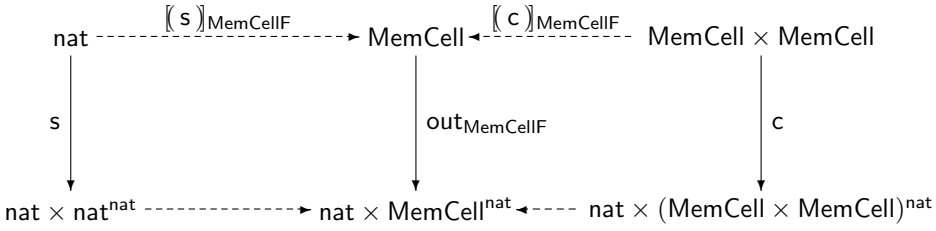
$$\begin{array}{lcl} \mathsf{s} & \triangleq & \langle\, \mathsf{id}_{\mathsf{nat}}, (\pi_2)^* \,\rangle \\ \mathsf{c} & \triangleq & \langle\, \mathsf{plus} \circ (\mathsf{send\_get} \times \mathsf{send\_get}), \Phi \circ (\mathsf{send\_set} \times \mathsf{send\_set}) \,\rangle \end{array} \tag{5}$$

Here, $^* : \mathrm{Hom}(X \times Y, Z) \Rightarrow \mathrm{Hom}(X, Z^Y)$ is the currying natural isomorphism so that $(\pi_2)^*$ has type $\mathsf{nat} \longrightarrow \mathsf{nat}^{\mathsf{nat}}$ and Φ is the obvious natural transformation

[2] Throughout the text, we use the `typewriter` font for references to the example source code; the sans-serf font is reserved for their formal categorical counterparts.

of type $(X^{\mathsf{nat}}) \times (X^{\mathsf{nat}}) \Rightarrow (X \times X)^{\mathsf{nat}}$. The function $\mathsf{plus} : \mathsf{nat} \times \mathsf{nat} \longrightarrow \mathsf{nat}$ is addition of natural numbers.

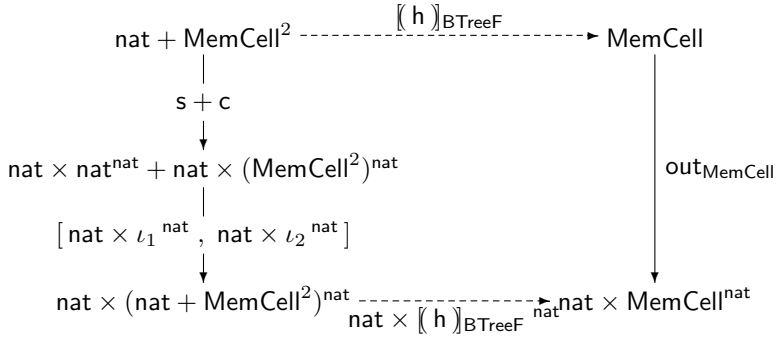
The following diagram illustrates the current situation in **SET**. We are using the anamorphism brackets, as in $(\!|\, \mathsf{s} \,|\!)_{\mathsf{MemCellF}}$, to denote the unique arrow into a final coalgebra from a coalgebra.



Alternatively, when we introduce a functor BTreeF as:

$$\mathsf{BTreeF}.X \quad = \quad \mathsf{nat} + X^2 \tag{6}$$

we can rewrite the two coalgebras s and c as one, with carrier of type $\mathsf{BTreeF}.\mathsf{MemCell} \equiv \mathsf{nat} + \mathsf{MemCell}^2$:



where $\mathsf{h} : \mathsf{nat} + \mathsf{MemCell}^2 \longrightarrow \mathsf{nat} \times (\mathsf{nat} + \mathsf{MemCell}^2)^{\mathsf{nat}}$ is the arrow:

$$\mathsf{h} \quad \triangleq \quad [\, \mathsf{nat} \times \iota_1^{\mathsf{nat}} \,,\, \mathsf{nat} \times \iota_2^{\mathsf{nat}} \,] \circ (\mathsf{s} + \mathsf{c}) \tag{7}$$

Here $[\, f \,,\, g \,]$ denotes the *co-tuple* of f and g defined by the universal arrow from a binary sum; ι_1 and ι_2 are the left and right injections. Now, when we define $\lambda_{\mathsf{MemCell}}$ as follows:

$$\lambda_{\mathsf{MemCell}} \quad \triangleq \quad [\, \mathsf{nat} \times \iota_1^{\mathsf{nat}} \circ \langle\, \mathsf{id}_{\mathsf{nat}}, (\pi_2)^* \,\rangle \,,\, \mathsf{nat} \times \iota_2^{\mathsf{nat}} \circ \langle\, \mathsf{plus} \circ (\pi_1)^2, \Phi \circ (\pi_2)^2 \,\rangle \,]$$

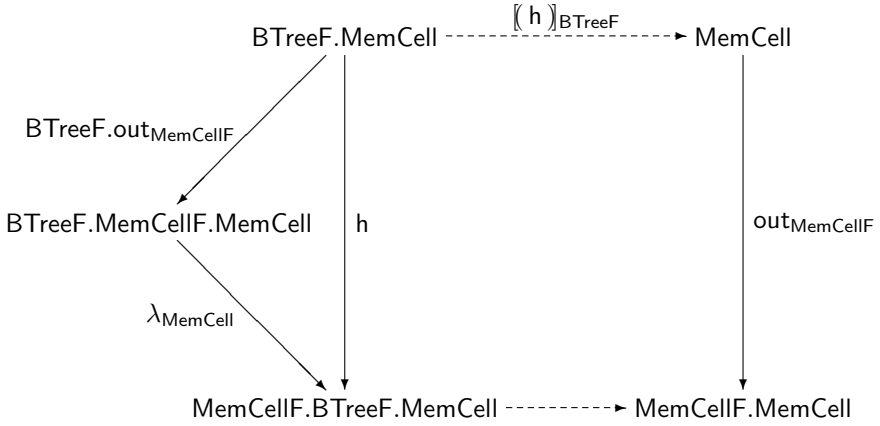
we get, by unfolding the definitions of s and c and factorising $\mathsf{out}_{\mathsf{BTreeF}}$:

$$\mathsf{h} \quad = \quad \lambda_{\mathsf{MemCell}} \circ (\mathsf{BTreeF}.\mathsf{out}_{\mathsf{MemCellF}})$$

The following diagram illustrates the types in the definition.

$$
\begin{array}{c}
\mathsf{nat} + \mathsf{MemCell}^2 \\
\big| \\
\mathsf{id} + \mathsf{out}_{\mathsf{MemCellF}}{}^2 \\
\downarrow \\
\mathsf{nat} + (\mathsf{nat} \times \mathsf{MemCell}^{\mathsf{nat}})^2 \\
\big| \\
\langle\, \mathsf{id}_{\mathsf{nat}}, (\pi_2)^* \,\rangle + \langle\, \mathsf{plus} \circ (\pi_1)^2, \Phi \circ\ (\pi_2)^2 \,\rangle \\
\downarrow \\
\mathsf{nat} \times \mathsf{nat}^{\mathsf{nat}} + \mathsf{nat} \times (\mathsf{MemCell}^2)^{\mathsf{nat}} \\
\big| \\
[\, \mathsf{nat} \times \iota_1{}^{\mathsf{nat}}\,,\ \mathsf{nat} \times \iota_2{}^{\mathsf{nat}} \,] \\
\downarrow \\
\mathsf{nat} \times (\mathsf{nat} + \mathsf{MemCell}^2)^{\mathsf{nat}}
\end{array}
$$

Note that everything in $\lambda_{\mathsf{MemCell}}$ is natural, thus λ is a natural transformation $\lambda : \mathsf{BTreeF} \circ \mathsf{MemCellF} \Rightarrow \mathsf{MemCellF} \circ \mathsf{BTreeF}$. The following diagram summarises the current situation.



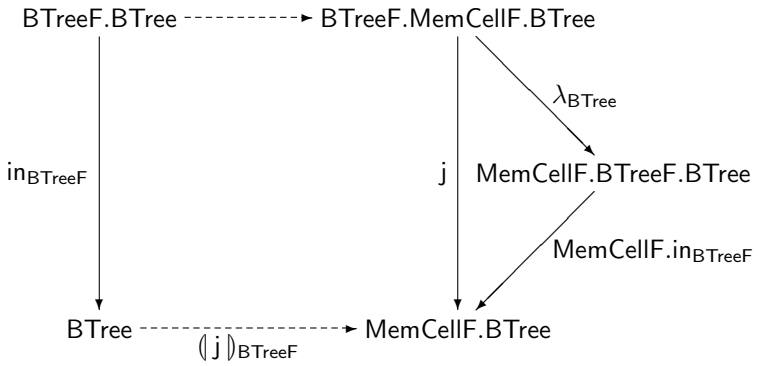
The careful reader will notice that BTreeF, which arose from the disjoint union of states of implementations involved in our little system of memory cells, corresponds to our intuition about its shape: it is the shape functor of data type BTree.

$$\mathsf{BTree} \quad = \quad \mu\, \mathsf{BTreeF}$$

where μ denotes the least fixed point. Thus we have, by careful analysis of the object-oriented code, recovered the structure of the object system as a functional data type together with behaviour of all its components. This behaviour is defined by a $\lambda : \mathsf{BTreeF} \circ \mathsf{MemCellF} \Rightarrow \mathsf{MemCellF} \circ \mathsf{BTreeF}$, which arose systematically from s and c. This construction also defines a simple operator on such distributive laws. It will be defined formally in the next section.

Alternatively, one could have started from BTree in the first place, and directly defined two functions as in (2). These are both catamorphisms and thus

correspond categorically to universal arrows out of the carrier of the initial BTreeF-algebra, BTree. Remarkably, when we use the same λ in their definition, everything else is forced and it is easy to verify that the result is equal the original definition (2):



In the diagram $(\!|\, \mathsf{j} \,|\!)_{\mathsf{BTreeF}}$ denotes the "catamorphism" of j, i.e. the universal arrow out of the initial BTreeF-algebra: $\mathsf{in}_{\mathsf{BTreeF}}$.

Now, $[\!(\, \mathsf{h} \,)\!]_{\mathsf{MemCellF}}$ is a BTreeF-algebra and thus implies a universal arrow

$$(\!|\, [\!(\, \mathsf{h} \,)\!]_{\mathsf{MemCellF}} \,|\!)_{\mathsf{BTreeF}} : \mathsf{BTree} \longrightarrow \mathsf{MemCell}$$

Intuitively, this is a fold over a BTree of states of MemCells, turning inductively, by the action of $[\!(\, \mathsf{h} \,)\!]_{\mathsf{MemCellF}}$, every leaf into a `SimpleMC` and every node with two subordinate MemCells into a `CompositeMC`.

Dually, $(\!|\, \mathsf{j} \,|\!)_{\mathsf{BTreeF}}$ is a MemCellF-coalgebra and also implies a universal arrow

$$[\!(\, (\!|\, \mathsf{j} \,|\!)_{\mathsf{BTreeF}} \,)\!]_{\mathsf{MemCellF}} : \mathsf{BTree} \longrightarrow \mathsf{MemCell}$$

This is a constructor of a single object – an abstract data-type – with realisation of type BTree whose two operations `getf` and `setf` are implemented via a fold over the tree.

In the following text, we show that these two arrows are equal and therefore the two MemCells, constructed in either way, are observationally equivalent. That is, either we are composing small objects into a large object where the structure of this composition is given by induction on the intended datatype, or we are programming functions directly on an algebraic datatype, as in functional programming. This is formalised in Theorem 2 in the next section.

3 Object Systems Formally

In this section, we abstract from the motivating example and formalise object systems of the kind introduced in Section 2. We start with a quick summary of our coalgebraic understanding of objects.

Definition 1. *An* interface functor *(or simply* interface*) is a polynomial endo-functor on* **SET** *of the form*

$$\prod_{i\in I} A_i^{B_i} \times \prod_{j\in J} (C_j \times -)^{D_j}$$

where all As , Bs , Cs and Ds are constant functors, − is the identity functor and all products and exponents are lifted to functors. I and J are finite sets.

For an example of an interface functor, see (3). Given an interface functor B, we can see coalgebras $\varphi : X \longrightarrow BX$ as *implementations* of objects with interface B. Examples of implementations are given in (5).

Carriers of the *terminal B-coalgebra*, νB, correspond to object-types, i.e. types (sets) of objects with interface B.

Definition 2. *An* object system *is a triple $\langle B, \{F_i\}_{i\in I}, \{\lambda_i\}_{i\in I}\rangle$ where B is an interface functor, $\{F_i\}$ is a finite collection of endo-functors and each λ_i is a natural transformation of type $F_iB \Rightarrow BF_i$, both indexed by a finite set I*

Example 1. The correspondence to the motivating example in Section 2 is the following: $B = \mathsf{MemCellF}$, $F_1 = \mathsf{nat}$, $F_2 = (-)^2$, $\lambda_1 = \langle\, \mathsf{id}_{\mathsf{nat}}, (\pi_2)^* \,\rangle$, $\lambda_2 = \langle\, \mathsf{plus} \circ (\pi_1)^2, \Phi \circ (\pi_2)^2 \,\rangle$.

For an object system $\langle B, \{F_i\}_{i\in I}, \{\lambda_i\}_{i\in I}\rangle$, the following is a collection of B-coalgebras:

$$\lambda_i \circ F_i\mathsf{out}_B : F_i\nu B \longrightarrow BF_i\nu B \quad , i \in I$$

Moreover, for

$$F \quad \triangleq \quad \sum_{i\in I} F_i$$

and

$$\lambda \quad \triangleq \quad [\, B\iota_i \circ \lambda_i \,]_{i\in I} \;:\; FB \Rightarrow BF \tag{8}$$

these can be combined into a single B-coalgebra as follows:

$$\lambda_{\nu B} \circ F\mathsf{out}_B \;:\; F\nu B \longrightarrow BF\nu B \tag{9}$$

Now (9) is a B-coalgebra and implies a unique arrow

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \;:\; F\nu B \longrightarrow \nu B$$

which in turn is an F-algebra and thus implies a unique arrow from the initial F-algebra:

$$(\!|\, [\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \,|\!)_F : \mu F \longrightarrow \nu B$$

Dually,

$$B\mathsf{in}_F \circ \lambda_{\mu F} \;:\; FB\mu F \longrightarrow B\mu F$$

is an F-algebra,

$$(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!) \;:\; \mu F \longrightarrow B\mu F \tag{10}$$

is a B-coalgebra and implies an arrow:

$$[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B : \mu F \longrightarrow \nu B$$

Theorem 1. *Let* out_B *be the terminal* B*-coalgebra and* μ_F *be the initial* F*-algebra. Let* $\lambda : FB \Rightarrow BF$*. Then*

$$(\!|\, [\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B \,|\!)_F \quad = \quad [\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B$$

Proof. We show that the right-hand side, $[\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B$, satisfies the universal property of the left-hand side:

$$[\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B \circ F[\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B \quad = \quad [\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B \circ \mathsf{in}_F \quad (11)$$

The calculation is straightforward by a two-fold application of the following rule, called "AnaFusion" in [9]:

$$[\![\, \varphi \,]\!]_B \circ f \;=\; [\![\, \psi \,]\!]_B \quad \Leftarrow \quad \varphi \circ f \;=\; Bf \circ \psi \qquad (12)$$

The premise of the rule is precisely the statement that f is a B-coalgebra morphism to φ from ψ. The proof is immediate by compositionality of coalgebra morphisms and uniqueness of the universal arrow. Using this rule we proceed as follows:

$$\begin{array}{ll}
& [\![\, \lambda \circ F\mathsf{out}_B \,]\!]_B \circ F[\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B \\
= & \quad \{ \text{ By (12) and the following:} \\
& \quad \lambda \circ F\mathsf{out}_B \circ F[\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B \\
& = \quad \{ \text{ functor composition } \} \\
& \quad \lambda \circ F(\mathsf{out}_B \circ [\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B) \\
& = \quad \{ [\![\, \ldots \,]\!]_B \text{ is a coalgebra morphism } \} \\
& \quad \lambda \circ F(B[\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B \circ (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F) \\
& = \quad \{ \lambda \text{ is natural } \} \\
& \quad (BF[\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B) \circ \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \quad \} \\
& [\![\, \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B \\
= & \quad \{ \text{ By (12) and the following fact:} \\
& \quad (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \circ \mathsf{in}_F \\
& = \quad \{ (\!|\, \ldots \,|\!)_F \text{ is a } F\text{-algebra morphism } \} \\
& \quad B\mathsf{in}_F \circ \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \quad \} \\
& [\![\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,]\!]_B \circ \mathsf{in}_F
\end{array}$$

□

The following is standard, here taken from [7].

Lemma 1. *Let* $\phi : A \to GA$ *and* $\psi : B \to GB$ *be two* G*-coalgebras. Then two elements* $a : 1 \longrightarrow A$ *and* $b : 1 \longrightarrow B$ *of their carriers are bisimilar if and only if*

$$[\![\, \phi \,]\!]_G \circ a = [\![\, \psi \,]\!]_G \circ b \quad : \quad 1 \longrightarrow \nu G$$

□

The following fact will also be useful.

$$[\![\, \mathsf{out}_B \,]\!]_B = \mathsf{id} \tag{13}$$

Theorem 2. *Let* out_B, μ_F *and* λ *be as in Theorem 1. Then for all* $t : 1 \longrightarrow \mu F$, t *is* B*-bisimilar to* $(\!|\, [\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B \,|\!)_F \circ t$.

Proof. Both $(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!) : \mu F \longrightarrow B\mu F$, first mentioned in (10), and $\mathsf{out}_B : \nu B \longrightarrow B\nu B$ are B-coalgebras. By Theorem 1 and fact (13):

$$[\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B \circ t = [\![\, \mathsf{out}_B \,]\!]_B \circ (\!|\, [\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B \,|\!)_F \circ t$$

The conclusion follows by Lemma 1. □

4 Lifting to Monads and Comonads

In the previous section, we considered distributive laws $\lambda : FB \Rightarrow BF$ for functors F and B. We showed a simple proof that the two arrows from the initial F-algebra to the final B-coalgebra, defined by induction and coinduction, are equal. In this section, we generalise the proof to distributive laws $\mathbf{\Lambda}$ between a monad T and the comonad D. Often, T is free over a signature F and D is cofree over a behaviour functor B, but this does not play any role in the proof. As distributive laws give definitions of dualisable data structures, a stronger distributive law will allow us to express more, while still preserving the same notion of duality.

We proceed by straightforward generalisation of the simple case. Simply put, if we just write T instead of F and D instead of B everywhere in the proof of Theorem 1, everything works just fine. In the remainder of this section, we carry out this lifting, which essentially consists of a verification that everything makes sense in the monadic setting. We begin with reminding the reader of the following standard definitions.

Definition 3. *Let* $\mathbf{C}$ *be a category, let* $\langle T, \eta, \mu \rangle$ *be a monad on* $\mathbf{C}$. *The* category of T-algebras, *denoted* $\mathbf{C}^T$, *has as objects arrows* $\alpha : TX \longrightarrow X$ *in* $\mathbf{C}$ *such that the following equations hold:*

$$1_X \;=\; \alpha \circ \eta_X \tag{14}$$

$$\alpha \circ \mu_X \;=\; \alpha \circ\; T\alpha \tag{15}$$

Arrows $f : \langle X, \alpha \rangle \longrightarrow \langle Y, \beta \rangle$ *in* $\mathbf{C}^T$ *are arrows* $f : X \longrightarrow Y$ *in* $\mathbf{C}$ *such that*

$$f \circ \alpha \;=\; \beta \circ Tf$$

The notion of the *category of* D*-coalgebras*, $\mathbf{C}_D$, for a *comonad* D is exactly dual. This cuts all work down to a half.

Lemma 2. *Let* 0 *be the initial object in* $\mathbf{C}$*. Then* $\langle T0, \mu_0 \rangle$ *is the initial* T*-algebra in* $\mathbf{C}^T$*. Dually,* $\langle D1, \delta_1 \rangle$ *is the final* D*-coalgebra.*

Proof. The monad $T : \mathbf{C} \longrightarrow \mathbf{C}$ splits into the adjunction $F \dashv U$, where $U : \mathbf{C}^T \longrightarrow \mathbf{C}$ is the underlying object functor and $FX = \langle X, \mu_X \rangle$ (see [1] for details). The conclusion follows from the fact that left adjoints preserve colimits and the initial object is the colimit of the empty diagram. □

We have thus lifted induction and coinduction to monads and comonads, respectively, and can give the following definition.

Definition 4. *Let* ϕ *be a* T*-algebra. Then* $(\!|\, \phi \,|\!)_T$ *denotes the underlying arrow in* $\mathbf{C}$ *of the unique morphism from the initial* T*-algebra. Dually, for a* D*-coalgebra* ψ *and* $[\!(\, \psi \,)\!]_D$*.*

The fusion rule we used in Theorem 1 can be lifted as follows.

Lemma 3. *For* D*-coalgebras* α *and* β*:*

$$[\!(\, \alpha \,)\!]_D \circ f \;=\; [\!(\, \beta \,)\!]_D \quad \Leftarrow \quad \alpha \circ f = Df \circ \beta \tag{16}$$

Proof. Immediate by uniqueness of the terminal morphism, as before. □

The following is standard, here taken from [13].

Definition 5. *Let* $\langle T, \eta, \mu \rangle$ *be a monad and* $\langle D, \varepsilon, \delta \rangle$ *be a comonad in a category* $\mathcal{C}$*. A* distributive law *of* T *over* D *is a natural transformation*

$$\Lambda \;:\; TD \Rightarrow DT$$

satisfying the following:

$$\Lambda \circ \eta_D \;=\; D\eta \tag{17}$$

$$\Lambda \circ \mu_D \;=\; D\mu \circ \Lambda_T \circ T\Lambda \tag{18}$$

$$\varepsilon_T \circ \Lambda \;=\; T\varepsilon \tag{19}$$

$$\delta_T \circ \Lambda \;=\; D\Lambda \circ \Lambda_D \circ T\delta \tag{20}$$

Lemma 4. *For all* X *in* $\mathbf{C}$*, the arrow*

$$D\mu_X \circ \Lambda_{TX} \;:\; TDTX \longrightarrow DTX$$

is a T*-algebra. Dually, the arrow*

$$\Lambda_{DX} \circ T\delta_X \;:\; TDX \longrightarrow DTDX$$

is a D*-coalgebra.*

Proof. We must verify that the two T-algebra laws (14) and (15) hold. This is done by a simple calculation. We give here just the proof of (15), (14) is even simpler. The dual part of the lemma follows by duality.

$$
\begin{aligned}
& D\mu \circ \Lambda_T \circ \mu_{DT} \\
= \quad & \{ \text{ by (18) } \} \\
& D\mu \circ D\mu_T \circ \Lambda_{T^2} \circ T\Lambda_T \\
= \quad & \{ \text{ monad laws } \} \\
& D\mu \circ DT\mu \circ \Lambda_{T^2} \circ T\Lambda_T \\
= \quad & \{ \Lambda \text{ is a natural transformation } \} \\
& D\mu \circ \Lambda_T \circ TD\mu \circ T\Lambda_T
\end{aligned}
$$

□

At this point, we can rephrase Theorem 1 for the generalised monadic setting.

Theorem 3. *Let $\langle T, \eta, \mu \rangle$ be a monad and $\langle D, \eta, \delta \rangle$ be a comonad. Let $\Lambda : TD \Rightarrow DT$ be a distributive law of the monad T over D. Then the following holds:*

$$(\!| [\![\Lambda_{D1} \circ T\delta_1]\!]_D |\!)_T \quad = \quad [\![(\!| D\mu_0 \circ \Lambda_{T0} |\!)_T]\!]_D$$

Proof. The proof has exactly the same structure as that of Theorem 1 except that we have to check at all relevant places that the algebras and coalgebras in question satisfy the additional properties (14) and (15) or their duals. All these proofs are by straightforward application of the monad laws, properties (17) - (20) of distributive laws and by naturality. We give the outline here but omit the routine checks.

$$
\begin{aligned}
& [\![\Lambda_D \circ T\delta]\!]_D \circ T[\![(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \\
= \quad & \{ \text{ By (16) } \} \\
& [\![\Lambda_D \circ T(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \\
= \quad & \{ \text{ By (16) } \} \\
& [\![(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \circ \mu
\end{aligned}
$$

□

Theorem 2 now generalises in the obvious way.

5 Related Work

We are not aware of any similar treatment of the relation of FP and OOP we describe. As for bialgebras per se, the key point of reference is Turi's thesis [12] and Turi and Plotkin's joint paper [13] on the same subject. There, our FP side of

the picture corresponds to *denotational semantics* and the OOP side corresponds to *operational semantics*. Our behavioural equivalence of the dual functional and object-oriented programs is *adequacy of denotational and operational semantics*. However, any further correspondence, for instance of their *operational rules* (c.f. [13],Section 3), is not quite clear and it is likely that in the different setting different rule formats or additional structure will be useful.

Distributive laws of a functor over a functor (both possibly with additional structure) [2] have recently enjoyed renewed interest in the research community. Here we mention just a few most relevant contributions.

In the context of bialgebraic semantics, Fiore, Plotking and Turi have worked on semantics of languages with binders in a presheaf category [4]. On the other hand, Bartek Klin has worked on adding recursive constructs to bialgebraic semantics. Both theoretical contributions could lead to a model of object structures with cycles and sharing in our interpretation.

Bart Jacobs in [6] gives several simple examples of modular constructions on distributive laws. This naturally links to modularity of programs as illustrated in Section 3.

Tarmo Uustalu with various colleagues has been using distributivity laws for recursion and corecursion schemes. Together with Varmo Vene and Alberto Pardo in [14] they specify a generalised coinduction scheme where a distributivity law specifies the pattern of mutual recursion between several functions defined by coinduction. This seems to be related to coalgebraic OOP where all methods in an interface are (possibly) mutually recursive.

6 Conclusions and Further Work

We demonstrated how distributive laws between a monad and a comonad arise naturally from everyday programming practice. By abstraction from a simple programming example we arrived at the same notion of adequacy Plotkin and Turi originally coined for denotational and operational semantics. We gave an alternative computational proof of equality of the two dual operational models defined by induction and coinduction, respectively (c.f. Corollary 7.3 in [13]).

In our future work we want to tackle data structures with cycles and sharing (pointers). This seems to be possible by introducing recursion and or signatures with binders [8, 4]. Another plausible direction is exploration of modularity of distributivity laws so as to tackle modularity of programs. We have seen an example of this in Section 3. This and a few other simple examples can be found in [6], but other more intricate ways of combining distributive laws would allow us to express more intricate patterns of mutual recursion between components. This strand of research is also closely related to modular operational semantics.

7 Acknowledgements

The author would like to thank the referees for their constructive feedback. This research has been funded by EPSRC grant EP/D502632/1.

References

1. Steve Awodey. *Category Theory*. Clarendon Press, 2006.
2. Jon Beck. Distributive laws. *Lecture Notes in Mathematics*, 80:119–140, 1969.
3. William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag.
4. Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 193, Washington, DC, USA, 1999. IEEE Computer Society.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. Bart Jacobs. Distributive laws for the coinductive solution of recursive equations. *Inf. Comput.*, 204(4):561–587, 2006.
7. Bart P. F. Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H. J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, Boston, 1996.
8. Bartek Klin. Adding recursive constructs to bialgebraic semantics. *J. Log. Algebr. Program.*, 60-61:259–286, 2004.
9. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
10. Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.
11. Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.
12. Daniele Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, June 1996.
13. Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings 12th Ann. IEEE Symp. on Logic in Computer Science, LICS'97, Warsaw, Poland, 29 June – 2 July 1997*, pages 280–291. IEEE Computer Society Press, Los Alamitos, CA, 1997.
14. Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–??, Fall 2001.

A Relational Semantics for Distributive Substructural Logics and the Topological Characterization of the Descriptive Frames

Tomoyuki Suzuki

Dept.Computer Science, University of Leicester,**
Leicester, LE1 7RH, UK
E-mail: ts119@mcs.le.ac.uk

Abstract. In this paper, we will show the *topological characterization* of descriptive DFL-frames and discuss the *categorical duality* between the classes of DFL-algebras and descriptive DFL-frames, as in the case of modal logic (eg. [1], [3] or [10]). Through these arguments, we can consider DFL-frames as a natural extension of Kripke frames for intuitionistic logic. Plus, we will also introduce the *quasi-description* and show the categorical duality. Then, we will obtain a deeper understanding of these topological conditions.

1 Introduction

By a substructural logic, we understand an extension of *the basic sequent calculus FL* - a sequent system obtained by deleting contraction, exchange and weakening rules from Gentzen's sequent calculus LJ. Substructural logics include well-researched logics, such as many-valued logics, fuzzy logics, relevance logics, superintuitionistic logics, etc. By the Lindenbaum-Tarski method, we usually define classes of *residuated lattices* having a constant 0, as algebraic counterparts of substructural logics. Therefore, algebraic techniques are often used and have generated several results (see [8]).

In modal logic, on the other hand, relational semantics introduced by Kripke is also attractive with its intuitive character and connection with applied structures like automata or transition systems in computer science, although algebraic counterparts, like classes of BAOs [1], also exist. Stone's representation theorem provides a bridge between algebraic semantics and relational semantics. For example, it is known that relational completeness results for *canonical* modal logics can be immediately proved using Stone duality (eg. [11]).

In author's Master's Thesis [15], a relational semantics for a large class of substructural logics, namely distributive substructural logics (DFL logics), was introduced via Stone duality. These logics are including well studied logics like

** Author has moved from Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa, 923-1292 Japan

relevance logics or superintuitionistic logics which have their own relational semantics, Routley-Meyer semantics or Kripke frames for intuitionistic logic, respectively. They can be naturally seen as special cases of our relational semantics.

The main results we have obtained (including the results in author's Master's Thesis) can be summed up as follows:

- For all basic extensions of **DFL**, we identified corresponding frame conditions and proved completeness results.
- We have introduced p-morphisms and some specific conditions; embeddings, surjection and isomorphisms. While some of them are standard in modal logic, we also defined another type of these conditions to study the categorical duality over distributive lattices from logical points of view.
- We introduced two types of descriptive frames via the difference of p-morphism conditions. Besides, we have studied the categorical duality between DFL-algebras and these two types of descriptive frames.
- Finally, we have found the topological characterization of descriptive frames. This is a natural generalization of similar characterization for intuitionistic [3] or relevance frames [14]: differentiation, tightness and compactness. Moreover, differentiation can be derived from tightness condition as well as in the case of intuitionistic logic. Besides, we have also obtained that differentiation generates the difference among descriptive frames.

Distinct points of our approach from other authors;

- as opposed to [14] or [16], we focus on DFL logics. Besides, we research p-morphism lemmas and duality between injections (surjection) and surjection (embedding),
- as opposed to [13], we introduce a more Kripke-like frame based on one underlying set and one relation,
- as opposed to [6] or [7], we focus on relational semantics and modal approach like p-morphism or descriptive frames. Although the author's Kripke completeness results [15] are almost same with their results, the author's results are now extending to Sahlqvist-type theorem and under preparation for submission,
- we compare our topological characterization with that of intuitionistic logic. This gives us that our relational semantics is a natural extension of Kripke semantics for intuitionistic logic.
- in this paper, we introduce two types of descriptive frames, and show the categorical duality and topological conditions. Through this study, we see that differentiation restricts descriptive frames to anti-symmetric frames.

Hereafter, we will define distributive substructural logics (DFL logics) in Chapter 2, algebraic semantics in Chapter 3, relational semantics (DFL-frames), general frames (general DFL-frames) and frame morphisms in Chapter 4, descriptive frames and topological characterization in Chapter 5, and finally, categorical duality in Chapter 6.

2 Distributive Substructural Logics

Formulas are built up from propositional variables, four constants $\mathbf{t}, \mathbf{f}, \mathbf{T}$, $\mathbf{F}$, and logical connectives. In this paper, small Roman letters $p, q, r, \ldots$ are used for propositional variables (Φ: the set of all propositional variables), small Greek letters $\phi, \psi, \chi, \ldots$ are used for formulas, capital Greek letters $\Gamma, \Sigma, \Delta, \ldots$ are used for lists of formulas (*Frm*: the set of all formulas), and φ is used for a list including at most one formula. As logical connectives, $\vee, \wedge, \circ, \rightarrow, \leftarrow$ are used. Informally, we might use parentheses, but we save them as the priority of logical connectives is given as usual; $\vee = \wedge > \circ > \rightarrow = \leftarrow$. Then, we define *formulas* as follows:

$$\phi ::= p \mid \mathbf{t} \mid \mathbf{f} \mid \mathbf{T} \mid \mathbf{F} \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \circ \phi \mid \phi \rightarrow \phi \mid \phi \leftarrow \phi.$$

To define distributive substructural logics, we firstly introduce *the basic sequent calculus DFL*. In this system, a *sequent* is defined as $\Gamma \Rightarrow \varphi$.

Definition 1 (Basic sequent calculus DFL)
Initial sequents:

$$\phi \Rightarrow \phi \qquad\qquad \phi \wedge (\psi \vee \chi) \Rightarrow (\phi \wedge \psi) \vee (\phi \wedge \chi)$$

$$\Gamma \Rightarrow \mathbf{T} \qquad\qquad \Gamma, \mathbf{F}, \Sigma \Rightarrow \varphi \qquad\qquad \Rightarrow \mathbf{t} \qquad\qquad \mathbf{f} \Rightarrow$$

Cut rule:

$$\frac{\Gamma \Rightarrow \phi \qquad \Sigma, \phi, \Xi \Rightarrow \varphi}{\Sigma, \Gamma, \Xi \Rightarrow \varphi}\ (\text{cut})$$

Rules for logical connectives:

$$\frac{\Gamma, \Delta \Rightarrow \varphi}{\Gamma, \mathbf{t}, \Delta \Rightarrow \varphi}\ (\mathbf{t}\ \text{w}) \qquad\qquad \frac{\Gamma \Rightarrow}{\Gamma \Rightarrow \mathbf{f}}\ (\mathbf{f}\ \text{w})$$

$$\frac{\Gamma, \phi, \Delta \Rightarrow \varphi \qquad \Gamma, \psi, \Delta \Rightarrow \varphi}{\Gamma, \phi \vee \psi, \Delta \Rightarrow \varphi}\ (\vee \Rightarrow)$$

$$\frac{\Gamma \Rightarrow \phi}{\Gamma \Rightarrow \phi \vee \psi}\ (\Rightarrow \vee 1) \qquad\qquad \frac{\Gamma \Rightarrow \psi}{\Gamma \Rightarrow \phi \vee \psi}\ (\Rightarrow \vee 2)$$

$$\frac{\Gamma, \phi, \Delta \Rightarrow \varphi}{\Gamma, \phi \wedge \psi, \Delta \Rightarrow \varphi}\ (\wedge 1 \Rightarrow) \qquad\qquad \frac{\Gamma, \psi, \Delta \Rightarrow \varphi}{\Gamma, \phi \wedge \psi, \Delta \Rightarrow \varphi}\ (\wedge 2 \Rightarrow)$$

$$\frac{\Gamma \Rightarrow \phi \qquad \Gamma \Rightarrow \psi}{\Gamma \Rightarrow \phi \wedge \psi}\ (\Rightarrow \wedge)$$

$$\frac{\Gamma, \phi, \psi, \Delta \Rightarrow \varphi}{\Gamma, \phi \circ \psi, \Delta \Rightarrow \varphi}\ (\circ \Rightarrow) \qquad\qquad \frac{\Gamma \Rightarrow \phi \qquad \Delta \Rightarrow \psi}{\Gamma, \Delta \Rightarrow \phi \circ \psi}\ (\Rightarrow \circ)$$

$$\frac{\Gamma \Rightarrow \phi \qquad \Xi, \psi, \Delta \Rightarrow \varphi}{\Xi, \Gamma, \phi \to \psi, \Delta \Rightarrow \varphi}\ (\to\Rightarrow) \qquad \frac{\phi, \Gamma \Rightarrow \psi}{\Gamma \Rightarrow \phi \to \psi}\ (\Rightarrow\to)$$

$$\frac{\Gamma \Rightarrow \phi \qquad \Xi, \psi, \Delta \Rightarrow \varphi}{\Xi, \psi \leftarrow \phi, \Gamma, \Delta \Rightarrow \varphi}\ (\leftarrow\Rightarrow) \qquad \frac{\Gamma, \phi \Rightarrow \psi}{\Gamma \Rightarrow \psi \leftarrow \phi}\ (\Rightarrow\leftarrow)$$

We say that a formula ϕ is provable, if the sequent $\Rightarrow \phi$ is provable. Before introducing logics, we define structural rules and the corresponding sequents.

Definition 2 (Structural rules)
Contraction :

$$\phi \Rightarrow \phi \circ \phi \qquad \frac{\Gamma, \phi, \phi, \Delta \Rightarrow \varphi}{\Gamma, \phi, \Delta \Rightarrow \varphi}\ (\mathrm{c}\Rightarrow)$$

Exchange :

$$\phi \circ \psi \Rightarrow \psi \circ \phi \qquad \frac{\Gamma, \phi, \psi, \Delta \Rightarrow \varphi}{\Gamma, \psi, \phi, \Delta \Rightarrow \varphi}\ (\mathrm{e}\Rightarrow)$$

Left-weakening :

$$\phi \Rightarrow \mathbf{t} \qquad \frac{\Gamma, \Delta \Rightarrow \varphi}{\Gamma, \phi, \Delta \Rightarrow \varphi}\ (\mathrm{w}\Rightarrow)$$

Right-weakening :

$$\mathbf{f} \Rightarrow \phi \qquad \frac{\Gamma \Rightarrow}{\Gamma \Rightarrow \phi}\ (\Rightarrow \mathrm{w})$$

Now, we define logics as sets of formulas (see [8]).

Definition 3 (Logic)
A set $\mathbf{S}$ of formulas is a *logic*, if $\mathbf{S}$ satisfies the following conditions;

1. $\mathbf{S}$ includes all provable formulas in DFL,
2. if ϕ and $\phi \to \psi$ in $\mathbf{S}$, then $\psi \in \mathbf{S}$,
3. if $\phi \in \mathbf{S}$, then $\phi \wedge \mathbf{t} \in \mathbf{S}$,
4. if $\phi \in \mathbf{S}$ and ψ is arbitrary formula, then $\psi \to \phi \circ \psi$ and $\psi \circ \phi \leftarrow \psi$ in $\mathbf{S}$,
5. $\mathbf{S}$ is closed under substitution.

The set of formulas which are provable in DFL is a logic, and we call it *the distributive substructural logic* **DFL**. Moreover, we call the extensions of **DFL** *distributive substructural logics (DFL logics)*. Well-researched logics like relevance logics, fuzzy logics or superintuitionistic logics are DFL logics (see [12]). We use bold face letters for logics.

Definition 4 (Basic DFL logics)
$\mathbf{DFL}_X$ is called *basic*, where X is a subset of $\{c(\text{Contraction}), e(\text{Exchange}), w(\text{Left- and Right-weakening})\}$. $\mathbf{DFL}_X$ is the set of provable formulas in DFL plus X as initial sequents or equivalently inference rules.

3 Algebraic Semantics for DFL logics

Algebraic counterparts of substructural logics are usually defined by classes of residuated lattices with a constant 0, called FL-algebras (see [7], [8] or [12]). Here, we will introduce algebraic semantics for DFL logics on the same line.

Definition 5 (DFL-algebra)
A tuple $\mathfrak{A} = \langle A, \vee, \wedge, \cdot, \backslash, /, 1, 0, \top, \bot \rangle$ is a *DFL-algebra*, if $\langle A, \vee, \wedge, \top, \bot \rangle$ is a bounded distributive lattice, $\langle A, \cdot, 1 \rangle$ a monoid, 0 arbitrary element in A, and $\mathfrak{A}$ satisfies the residuation law below.

$$a \cdot b \leq c \iff b \leq a \backslash c \iff a \leq c / b^{1}$$

Given a DFL-algebra $\mathfrak{A} = \langle A, \vee, \wedge, \cdot, \backslash, /, 1, 0, \top, \bot \rangle$, an *assignment* f is a function from Φ to A. Then, we inductively extend f to $\bar{f} : Frm \to A$ as follows.

- $\bar{f}(p) := f(p)$.
- $\bar{f}(\mathbf{t}) := 1$.
- $\bar{f}(\mathbf{f}) := 0$.
- $\bar{f}(\mathbf{T}) := \top$.
- $\bar{f}(\mathbf{F}) := \bot$.
- $\bar{f}(\phi \vee \psi) := \bar{f}(\phi) \vee \bar{f}(\psi)$.
- $\bar{f}(\phi \wedge \psi) := \bar{f}(\phi) \wedge \bar{f}(\psi)$.
- $\bar{f}(\phi \circ \psi) := \bar{f}(\phi) \cdot \bar{f}(\psi)$.
- $\bar{f}(\phi \rightarrow \psi) := \bar{f}(\phi) \backslash \bar{f}(\psi)$.
- $\bar{f}(\psi \leftarrow \phi) := \bar{f}(\psi) / \bar{f}(\phi)$.

Based on this, we define the truth relation $\models$;

- $\mathfrak{A}, f \models \phi \iff 1 \leq \bar{f}(\phi)$,
- $\mathfrak{A} \models \phi \iff \mathfrak{A}, f \models \phi$, for any assignment f.

If $\mathfrak{A}, f \models \phi$, we say that ϕ is *true* on $\mathfrak{A}$ through an assignment f, and if $\mathfrak{A} \models \phi$, we say that ϕ is *valid* on $\mathfrak{A}$. Moreover, given a class $\mathfrak{C}$ of DFL-algebras and a set S of formulas, $\mathfrak{C} \models S$, if $\mathfrak{A} \models \phi$ for each DFL-algebra $\mathfrak{A} \in \mathfrak{C}$ and for all formulas $\phi \in S$. Among DFL-algebras, we define *homomorphism*, *injection*, *surjection* and *isomorphism* in the standard way (eg. [2]).

4 Relational Semantics for DFL logics

Relational semantics of some specific substructural logics has already been researched by several authors (relevance logics [5], [14] or [16], superintuitionistic logics [3], BCK logics [13]). Here, we introduce a relational semantics for DFL logics which is a natural extension of the above relational semantics.

[1] Although we use a binary relation $\leq$ without the definition, this is the usual order relation in Lattice theory [2]. i.e. $a \leq b \iff a \wedge b = a \iff a \vee b = b$. We use this relation without saying anything, when we consider algebraic contexts.

Definition 6 (DFL-frame)
A tuple $\mathfrak{F} = \langle W, W_t, W_f, R_\circ \rangle$ is a *DFL-frame*, if W is a non-empty set, W_t a non-empty subset of W, W_f a subset of W, $R_\circ$ a ternary relation on W, and $\mathfrak{F}$ satisfies the following conditions;

1. there exist t_1, t_2 in W_t such that $R_\circ(w, t_1, w)$ and $R_\circ(w, w, t_2)$,
2. if $R_\circ(w, v, u)$, $w \preceq w'$, $v' \preceq v$ and $u' \preceq u$, then $R_\circ(w', v', u')$,
3. there exists $x \in W$ such that $R_\circ(w, x, s)$ and $R_\circ(x, v, u)$, if and only if, there exists $y \in W$ such that $R_\circ(w, v, y)$ and $R_\circ(y, u, s)$,
4. if $w \in W_t$ and $w \preceq w'$, then $w' \in W_t$,
5. if $w \in W_f$ and $w \preceq w'$, then $w' \in W_f$,

where we define a (order-like) binary relation $\preceq$ on W as an abbreviation as below.

$$w \preceq w' \iff \text{there exists } t \in W_t \text{ such that } R_\circ(w', t, w) \text{ or } R_\circ(w', w, t).$$

Given a DFL-frame $\mathfrak{F} = \langle W, W_t, W_f, R_\circ \rangle$, a *valuation* V is a function from Φ to $Up(W)$, where $Up(W)$ is the set of all $\preceq$-upward closed subsets of W. Then, given an element $w \in W$ and a formula ϕ, we inductively define the truth relation $\Vdash$ as follows;

- $\mathfrak{F}, V, w \Vdash p \iff w \in V(p)$,
- $\mathfrak{F}, V, w \Vdash \mathbf{t} \iff w \in W_t$,
- $\mathfrak{F}, V, w \Vdash \mathbf{f} \iff w \in W_f$,
- $\mathfrak{F}, V, w \Vdash \mathbf{T}$ always holds,
- $\mathfrak{F}, V, w \Vdash \mathbf{F}$ never hold,
- $\mathfrak{F}, V, w \Vdash \phi \vee \psi \iff \mathfrak{F}, V, w \Vdash \phi$ or $\mathfrak{F}, V, w \Vdash \psi$,
- $\mathfrak{F}, V, w \Vdash \phi \wedge \psi \iff \mathfrak{F}, V, w \Vdash \phi$ and $\mathfrak{F}, V, w \Vdash \psi$,
- $\mathfrak{F}, V, w \Vdash \phi \circ \psi \iff$ there exist $v, u \in W$ such that $R_\circ(w, v, u)$, $\mathfrak{F}, V, v \Vdash \phi$ and $\mathfrak{F}, V, u \Vdash \psi$,
- $\mathfrak{F}, V, w \Vdash \phi \rightarrow \psi \iff$, for each $v, u \in W$, if $R_\circ(u, v, w)$ and $\mathfrak{F}, V, v \Vdash \phi$, then $\mathfrak{F}, V, u \Vdash \psi$,
- $\mathfrak{F}, V, w \Vdash \psi \leftarrow \phi \iff$, for each $v, u \in W$, if $R_\circ(u, w, v)$ and $\mathfrak{F}, V, v \Vdash \phi$, then $\mathfrak{F}, V, u \Vdash \psi$.

We can check that any valuation is extended to a function from *Frm* to $Up(W)$, straightforwardly (see [15]). Besides, we define $\mathfrak{F}, V, w \Vdash_t \phi$ as $\mathfrak{F}, V, w \Vdash \phi$ and $w \in W_t$. Then, we call $\mathfrak{F}, V, w \Vdash_t \phi$ that a formula ϕ is *true* at w on a DFL-model $\mathfrak{F}, V$. Moreover, we define *globally truth* ($\mathfrak{F}, V \Vdash_t \phi$) as $\mathfrak{F}, V, w \Vdash_t \phi$ for any $w \in W_t$ and *validity* ($\mathfrak{F} \Vdash_t \phi$) as $\mathfrak{F}, V \Vdash_t \phi$ for each valuation V.

Based on this relational semantics, we introduce general frames as in the case of modal logic (eg. [1]). Then, our DFL-frames are also considered as a special case of the general frames.

Definition 7 (General DFL-frame)
A tuple $\mathfrak{G} = \langle \mathfrak{F}, A \rangle$ is a *general DFL-frame*, if $\mathfrak{F}$ is a DFL-frame, and A a subset of $Up(W)$ satisfying the following conditions;

1. W_t, W_f, W and $\emptyset$ in A,
2. A is closed under $\cup, \cap, *, \setminus$ and $\swarrow$,

where $*$, $\setminus$ and $\swarrow$ are defined below.

$$X * Y := \{w \in W \mid \exists v \in X, \exists u \in Y.[R_\circ(w, v, u)]\}$$

$$X \setminus Y := \{w \in W \mid \forall u \in W, \forall v \in X.[R_\circ(u, v, w) \Rightarrow u \in Y]\}$$

$$Y \swarrow X := \{w \in W \mid \forall u \in W, \forall u \in X.[R_\circ(u, w, v) \Rightarrow u \in Y]\}$$

We can prove that $Up(W)$ is closed under these operators.

Given a general DFL-frame $\mathfrak{G} = \langle \mathfrak{F}, A \rangle$, an *admissible valuation* V is a function from Φ to A. Moreover, we can extend it to the function from *Frm* to A as in the case of DFL-frames. The truth relation $\Vdash$ and the other terms are also defined in the same way.

Next, we define p-morphisms between (general) DFL-frames. This definition is different from modal logic's one. That is, the conditions 5, 6 and 7 below are defined by the inequality $\preceq$, not the equality $=$. This is because we can only prove the inequality, when we consider the categorical duality, see Lemma 30 in Section 6.

Definition 8 (P-morphism)
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f_1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f_2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. A morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ is a *p-morphism*, if ρ satisfies the following conditions;

1. ρ is a function from W_1 to W_2,
2. $w \in W_{t1} \iff \rho(w) \in W_{t2}$,
3. $w \in W_{f_1} \iff \rho(w) \in W_{f_2}$,
4. if $R_{\circ 1}(w, v, u)$, then $R_{\circ 2}(\rho(w), \rho(v), \rho(u))$,
5. if $R_{\circ 2}(\rho(w), v', u')$, there exist $v, u \in W_1$ such that $R_{\circ 1}(w, v, u)$, $v' \preceq \rho(v)$ and $u' \preceq \rho(u)$,
6. if $R_{\circ 2}(u', v', \rho(w))$, there exist $v, u \in W_1$ such that $R_{\circ 1}(u, v, w)$, $v' \preceq \rho(v)$ and $\rho(u) \preceq u'$,
7. if $R_{\circ 2}(u', \rho(w), v')$, there exist $v, u \in W_1$ such that $R_{\circ 1}(u, w, v)$, $v' \preceq \rho(v)$ and $\rho(u) \preceq u'$
8. for each $X' \in A_2$, $\rho^{-1}(X') \in A_1$.

Here, we define two types of embeddings; a *quasi-embedding* and an *embedding*. Finally, this distinction will lead to a deeper understanding of descriptive frames and their duality, see Table 1 in Section 5 and Theorem 34 in Section 6.

Definition 9 (Quasi-embedding)
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f_1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f_2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. A morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ is a *quasi-embedding*, if ρ is a p-morphism from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ and satisfies 9 and 10.

9. For any $X \in A_1$, $\uparrow(\rho(X)) := \{w' \in W_2 \mid \exists w \in X.[\rho(w) \preceq w']\}$ is in A_2.

10. If $\rho(w) \preceq \rho(v)$, then $w \preceq v$.

Definition 10 (Embedding)
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. A morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ is an *embedding*, if ρ is a injective p-morphism from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ and satisfies 9'.

9'. For any $X \in A_1$, there exists $Y \in A_2$, such that $\rho(X) = \rho(W_1) \cap Y$.

The reason we call quasi-embeddings is that the conditions 9 and 10 generate the condition 9'. Moreover, if $\mathfrak{G}_1$ and $\mathfrak{G}_2$ are anti-symmetric, the injectivity is also derived from the condition 10. Hence, for anti-symmetric frames, quasi-embeddings and embeddings coincide.

Definition 11 (Surjection)
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. A morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ is *surjective*, if ρ is a p-morphism from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ and satisfies 11.

11. For any $w' \in W_2$, there exists $w \in W_1$ such that $\rho(w) = w'$.

We define an *quasi-isomorphism* as a surjective quasi-embedding.

Definition 12 (Isomorphism)
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. A morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ is an *isomorphism*, if ρ is a bijection from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ and satisfies,

1. $w \in W_{t1} \iff \rho(w) \in W_{t2}$ for any $w \in W_1$.
2. $w \in W_{f1} \iff \rho(w) \in W_{t2}$ for any $w \in W_2$.
3. $R_{\circ 1}(w, v, u) \iff R_{\circ 2}(\rho(w), \rho(v), \rho(u))$ for any $w, v, u \in W_1$.
4. $X \in A_1 \iff \rho(X) \in A_2$.

We can say that every isomorphism is a bijective p-morphism.

The reason we introduce both quasi-isomorphism and isomorphism is to discuss both a usual descriptive frame like modal logic and our categorical duality or the following p-morphism lemmas.

Then, we can show the following p-morphism lemmas. These are analogously proved in the case of modal logics.

Lemma 13
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. If there exists a p-morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$ satisfying 9 above, for any valuation V_1 on $\mathfrak{G}_1$, there exists a valuation V_2 on $\mathfrak{G}_2$ such that $\mathfrak{G}_1, V_1, w \Vdash \phi \iff \mathfrak{G}_2, V_2, \rho(w) \Vdash \phi$.

Lemma 14
Let $\mathfrak{G}_1 = \langle W_1, W_{t1}, W_{f1}, R_{\circ 1}, A_1 \rangle$ and $\mathfrak{G}_2 = \langle W_2, W_{t2}, W_{f2}, R_{\circ 2}, A_2 \rangle$ be general DFL-frames. Suppose that there exists a p-morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$. The following holds:

1. if ρ is a quasi-embedding, $\mathfrak{G}_2 \Vdash \phi \Rightarrow \mathfrak{G}_1 \Vdash \phi$,
2. if ρ is surjective, $\mathfrak{G}_1 \Vdash \phi \Rightarrow \mathfrak{G}_2 \Vdash \phi$,
3. if ρ is quasi-isomorphic, $\mathfrak{G}_1 \Vdash \phi \iff \mathfrak{G}_2 \Vdash \phi$.

5 Topological characterization of descriptive frames

Hereafter, we consider a logic **L** as a DFL logic and L-algebras, L-frames or general L-frames as DFL-algebras, DFL-frames or general DFL-frames validating each formula in **L**, respectively. Here, we introduce descriptive frames for DFL logics via Stone duality.

Stone duality for DFL logics are the following.

Definition 15 (Dual general frame)
Given a L-algebra $\mathfrak{A} = \langle A, \vee, \wedge, \cdot, \backslash, /, 1, 0, \top, \bot \rangle$, the tuple $\mathfrak{A}_* = \langle Pf(A), Pf_1(A), Pf_0(A), R., \widehat{A} \rangle$ is the *dual general frame*, where $Pf(A)$ is the set of all prime filters over A, $Pf_1(A)$ the set of all prime filters containing 1, $Pf_0(A)$ the set of all prime filters containing 0,

$$R.(F_1, F_2, F_3) \iff \text{if } a \in F_2, b \in F_3, \text{ then } a \cdot b \in F_1, \text{ for any } a, b \in A,$$

$$\widehat{A} := \{\hat{a} \mid a \in A\}, \text{ where } \hat{a} := \{F \in Pf(A) \mid a \in F\}.$$

If we introduce a binary operation on prime filters $\times$ defined as follows;

$$F_1 \times F_2 := \{a \in A \mid \exists b \in F_1, \exists c \in F_2 [b \cdot c \leq a]\},$$

then $R.(F_1, F_2, F_3)$ can be seen as $F_2 \times F_3 \subseteq F_1$, and the abbreviation $\preceq$ can be thought of the set inclusion $\subseteq$. Hereafter, we sometimes use this notation.

Definition 16 (Dual L-algebra)
Given a general L-frame $\mathfrak{G} = \langle W, W_t, W_f, R_\circ, A \rangle$, the tuple $\mathfrak{G}^* = \langle A, \cup, \cap, *, \backslash, \swarrow, W_t, W_f, W, \emptyset \rangle$ is the *dual algebra*, where $\cup$ and $\cap$ are the set union and intersection, $*$, $\backslash$, $\swarrow$ defined in Definition 7.

The following two proposition 17 and 18 are proved by the author in his Master's Thesis [15].

Proposition 17
For any L-algebra, the dual general frame is a general L-frame.

Proposition 18
For any general L-frame, the dual algebra is a L-algebra.

Next, we define the two types of descriptive frames; a *quasi-descriptive frame* and a *descriptive frame*.[2]

[2] This difference comes only from the definition of embeddings; a quasi-embedding or an embedding.

Definition 19 (Quasi-descriptive frame)
A general L-frame $\mathfrak{G}$ is *quasi-descriptive*, if $\mathfrak{G}$ is quasi-isomorphic to the bidual general frame $(\mathfrak{G}^*)_*$.

Definition 20 (Descriptive frame)
A general L-frame $\mathfrak{G}$ is *descriptive*, if $\mathfrak{G}$ is isomorphic to the bidual general frame $(\mathfrak{G}^*)_*$.

We introduce some conditions about general DFL-frames as in the case of modal logic (eg. [1] or [3]). Given a general DFL-frame $\mathfrak{G} = \langle W, W_t, W_f, R_\circ, A\rangle$, we call it;

Differentiated for each $w, v \in W$,
$w = v \iff \forall X \in A.[w \in X \iff v \in X]$,
Totally order disconnected for each $w, v \in W$,
$w \preceq v \iff \forall X \in A.[w \in X \Rightarrow v \in X]$,
Tight for each $w, v, u \in W$,
$R_\circ(w, v, u) \iff \forall X, Y \in A.[v \in X \text{ and } u \in Y \text{ imply } w \in X * Y]$,
Compact for each family $\mathcal{X} \subseteq A$ and $\mathcal{Y} \subseteq \bar{A}(:= \{W - X \mid X \in A\})$,
$\bigcap(\mathcal{X} \cup \mathcal{Y}) \neq \emptyset$, whenever $\mathcal{X} \cup \mathcal{Y}$ has the finite intersection property (see [1]).

In our settings, since $\preceq$ is defined by $R_\circ$, totally order disconnectedness follows from tightness. If we consider general DFL-frames as Priestley spaces with the topology given by the base $\mathfrak{B} = A \cup \{W - X \mid X \in A\}$, the above differentiation and compactness correspond to the standard Hausdorffness and compactness in the topological space. Moreover, $\preceq$ coincides with the order of the Priestley spaces by totally order disconnectedness.

Before we prove the topological characterization theorem, some lemmas and propositions are introduced.

The Squeeze lemma in [5] provides the following lemma.

Lemma 21
$\hat{a} * \hat{b} = \widehat{a \cdot b}$.

The following proposition is standard (see eg. [3]).

Proposition 22
For any general DFL-frame $\mathfrak{G} = \langle W, W_t, W_f, R_\circ, A\rangle$, $\mathfrak{G}$ is compact if and only if every prime filter over A can be expressed by $\hat{w}$ for some $w \in W$, where $\hat{w} := \{F \in A \mid w \in F\}$.

Firstly, we show the standard topological characterization theorem for descriptive L-frames.

Theorem 23 (Topological characterization)
A general L-frame $\mathfrak{G} = \langle W, W_t, W_f, R_\circ, A\rangle$ is descriptive if and only if it is differentiated, tight and compact.

Proof
($\Rightarrow$). We will prove that $(\mathfrak{G}^*)_*$ is differentiated, tight and compact. Suppose $\mathfrak{G} \cong (\mathfrak{G}^*)_*$.

Differentiation The 'only if' part is obvious. Let F_1 and F_2 be arbitrary elements of $Pf(A)$. If $F_1 \neq F_2$, then there exists $a \in A$ such that either $a \in F_1$ and $a \notin F_2$ or $a \notin F_1$ and $a \in F_2$. So, either $F_1 \in \hat{a}$ and $F_2 \notin \hat{a}$ or $F_1 \notin \hat{a}$ and $F_2 \in \hat{a}$.

Tightness The 'only if' part is very straightforward. Let F_1, F_2 and F_3 be arbitrary elements of $Pf(A)$. For arbitrary $a, b \in A$, if $a \in F_2$ and $b \in F_3$, then $F_2 \in \hat{a}$ and $F_3 \in \hat{b}$. By the assumption, $F_1 \in \hat{a} * \hat{b}$. By Lemma 21, $F_1 \in \widehat{a \cdot b}$.

Compactness Since $\mathfrak{G}$ is isomorphic to $(\mathfrak{G}^*)_*$ and $\hat{w} := \{X \in A \mid w \in X\}$ is a prime filter over $Pf(A)$ for any $w \in W$, every prime filter can be represented by some element of W as $\hat{w}(:= \{X \in A \mid w \in X\})$. By Proposition 22, $(\mathfrak{G}^*)_*$ is compact.

($\Leftarrow$). We define a function $f_{\mathfrak{G}}$ from $\mathfrak{G}$ to $(\mathfrak{G}^*)_*$ as $f_{\mathfrak{G}}(w) := \hat{w}$ for any $w \in W$. Then, we will show that $f_{\mathfrak{G}}$ is a isomorphism (Definition 12). Firstly, it is obvious that $f_{\mathfrak{G}}$ is well-defined. For any $w \in W$, $w \in W_t \iff \hat{w} \in Pf_{W_t}(A)$ and $w \in W_f \iff \hat{w} \in Pf_{W_f}(A)$ are obvious. By compactness and tightness, $R_\circ(w, v, u) \iff R_*(\hat{w}, \hat{v}, \hat{u})$ is trivial. By definition, $X \in A \iff \widehat{X} \in \widehat{A}$. If $w \neq v$, then, by differentiation, there exists $X \in A$ such that either $w \in X$ and $v \notin X$ or $w \notin X$ and $v \in X$. So, $\hat{w} \neq \hat{v}$. Therefore, if $\hat{w} = \hat{v}$, $w = v$ for each $w, v \in W$. By Proposition 22, $f_{\mathfrak{G}}$ is surjective, since $\mathfrak{G}$ is compact.(Q.E.D)

Theorem 23 gives us that descriptive frames are topologically characterized by three conditions: differentiation, tightness and compactness. This result is precisely the same with modal logic's one. However, in our settings, we can obtain the following results;

Here, we introduce *the anti-symmetry*. For each (general) DFL-frame $\mathfrak{G} = \langle W, W_t, W_f, R_\circ, A\rangle$ and $w, v \in W$, if $w \preceq v$ and $v \preceq w$, $w = v$.

Proposition 24
For each DFL-algebra $\mathfrak{A}$, the dual (general) DFL-frame $\mathfrak{A}_*$ is anti-symmetric.

Proof
This is trivial, because, for each $F, G \in Pf(A)$, $F \preceq G \iff F \subseteq G$.(Q.E.D)

Proposition 25
Every descriptive L-frame $\mathfrak{G}$ satisfies the anti-symmetry.

Proof
Let $f : \mathfrak{G} \to (\mathfrak{G}^*)_*$ be an isomorphism. For arbitrary $w, v \in W$, if $w \preceq v$ and $v \preceq w$, $f(w) \preceq f(v)$ and $f(v) \preceq f(w)$. By Proposition 24, $f(w) = f(v)$. Since f is injective, $w = v$.(Q.E.D)

Lemma 26
Given a descriptive L-frame $\mathfrak{G}$, differentiation is derived from tightness.

Proof
By the anti-symmetry, it is straightforwardly obtained that any general DFL-frame is differentiated, whenever it is totally order disconnected. Now, we derive differentiation from tightness. The 'only if' part is trivial. Conversely, assume $w \not\preceq v$. By the condition 1 of Definition 6, there exists $t_w \in W_t$ such that $R_\circ(w, t_w, w)$. By the assumption, neither $R_\circ(v, t, w)$ nor $R_\circ(v, w, t)$ hold for each $t \in W_t$. Surely, $R_\circ(v, t_w, w)$ does not hold. By tightness, there exist $X, Y \in A$ such that $t_w \in X$, $w \in Y$ and $v \notin X * Y$, while $w \in X * Y$ because of $R_\circ(w, t_w, w)$ and tightness. (Q.E.D)

Corollary 27
Descriptive L-frames are characterized only by the tightness and compactness.

This result perfectly matches the topological characterization of descriptive frames for intuitionistic logic (eg. [3]).

Next, we consider the topological characterization of quasi-descriptive L-frames.

Theorem 28
For any general L-frame $\mathfrak{G}$, $\mathfrak{G}$ is quasi-descriptive if and only if $\mathfrak{G}$ is tight and compact.

Proof
Analogous to the proof of Theorem 23.(Q.E.D)

As we see before, we can say that both quasi-descriptive L-frames and descriptive L-frames are characterized by tightness and compactness. However, the following proposition gives us the witness of the difference.

Proposition 29
If general L-frame is differentiated, it is anti-symmetric.

We can sum up these results as follows;

Table 1. Topological conditions

	Quasi-descriptive	Descriptive
Differentiation	May not	Must
Totally order disconnected	Must	Must
Tightness	Must	Must
Compactness	Must	Must

Remark: This result says that if we topologically define descriptive frames with differentiation, it sees every descriptive frame as anti-symmetric frames.

6 Categorical duality for DFL logics

In this chapter, we will discuss the categorical duality between L-algebras and descriptive L-frames like [1], [3], [10] or [14] and quasi-descriptive L-frames.

Let $\mathcal{A}_L$ be the category of L-algebras defined as follows:

1. objects are all L-algebras,
2. morphisms are all homomorphisms over L-algebras.

Let $\mathcal{D}_L$ be the category of descriptive L-frames defined as follows:

1. objects are all descriptive L-frames,
2. morphisms are all p-morphisms over descriptive L-frames.

Let $\mathcal{D}_L^w$ be the category of quasi-descriptive L-frames defined as follows;

1. objects are all quasi-descriptive L-frames,
2. morphisms are all p-morphisms over descriptive L-frames.

By definition, $\mathcal{D}_L$ is a subcategory of $\mathcal{D}_L^w$.

Next, we define $(\cdot)^*$ and $(\cdot)_*$ as functors between $\mathcal{A}_L$ and $\mathcal{D}_L$ or $\mathcal{D}_L^w$. For objects, these values are the dual objects defined in Definition 15 and 16. For morphisms, we define h_* and ρ^* as the inverse images. In other words,

$$h_*(F) := h^{-1}(F)$$

$$\rho^*(X) := \rho^{-1}(X)$$

Then, we check the following lemmas.

Lemma 30
Let $\mathfrak{A}$ and $\mathfrak{B}$ be DFL-algebras. If h is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, then h_* is a p-morphism from $\mathfrak{B}_*$ to $\mathfrak{A}_*$.

Proof
Here, we check only the conditions 2 and 6 in Definition 8.

2. If $F \in Pf_{1_B}(B)$, then $1_B \in F$. Since h is a homomorphism, $1_A \in h^{-1}(1_B)$ i.e. $1_A \in h_*(F)$. Therefore, $h_*(F) \in Pf_{1_A}(A)$. Conversely, if $h_*(F) \in Pf_{1_A}(A)$, then $1_A \in h_*(F)$. Since $h(1_A) = 1_B$, $F \in Pf_{1_B}(B)$.
6. Assume $R_{\cdot_A}(G_1, G_2, h_*(F_3))$. Let $\uparrow (h(G_2))$ be $\{b \in B \mid \exists a \in G_2[h(a) \leq b]\}$ and $\downarrow (h(A - G_1))$ $\{b \in B \mid \exists a \in A - G_1[b \leq h(a)]\}$. We claim here that $((\uparrow (h(G_2))) \times F_3) \cap \downarrow (h(A - G_1)) = \emptyset$. If not, there exist $a \in G_2, b \in F_3, c \notin G_1$ such that $h(a) \cdot_B b \leq h(c)$. By the residuation law, $b \leq h(a \backslash c) \in F_3$. Therefore, $c \in G_1$, which a contradiction. So, by the prime filter theorem, there exists a prime filter F_1 such that $(\uparrow (h(G_2)) \times F_3 \subseteq F_1$ and $F_1 \cap \downarrow (h(A - G_1)) = \emptyset$ i.e $h_*(F_1) \subseteq G_1$. Moreover, by the squeeze lemma, there exists a prime filter F_2 such that $\uparrow (h(G_2)) \subseteq F_2$ i.e. $G_2 \subseteq h_*(F_2)$ and $F_2 \times F_3 \subseteq F_1$ i.e $R_{\cdot_B}(F_1, F_2, F_3)$. (Q.E.D)

Lemma 31
Let $\mathfrak{G}_1$ and $\mathfrak{G}_2$ be general DFL-frames. If ρ is a p-morphism from $\mathfrak{G}_1$ to $\mathfrak{G}_2$, then ρ^* is a homomorphism from ${\mathfrak{G}_2}^*$ to ${\mathfrak{G}_1}^*$.

Proof
We here check only $\rho^*(X *_2 Y) = \rho^*(X) *_1 \rho^*(Y)$. $w \in \rho^*(X *_2 Y)$ if and only if $\rho(w) \in X *_2 Y$. By the definition, there exist $v', u' \in W_2$ such that $R_{\circ 2}(\rho(w), v', u'), v' \in X$ and $u' \in Y$. By Definition 8, there exist $v, u \in W_1$ such that $R_{\circ 1}(w, v, u)$, $v' \preceq \rho(v)$ and $u' \preceq \rho(u)$. Here, X and Y are $\preceq$-upward closed set. So, $\rho(v) \in X$ and $\rho(u) \in Y$. Therefore, $w \in \rho^*(X) *_1 \rho^*(Y)$. Conversely, assume $w \in \rho^*(X) *_1 \rho^*(Y)$. By the definition, there exist $v, u \in W_1$ such that $R_{\circ 1}(w, v, u)$, $v \in \rho^*(X)$ i.e. $\rho(v) \in X$ and $u \in \rho^*(Y)$ i.e. $\rho(u) \in Y$. By Definition 8, $R_{\circ 2}(\rho(w), \rho(v), \rho(u))$. Therefore, $w \in \rho^*(X *_2 Y)$. (Q.E.D)

Proposition 32
For each L-algebras $\mathfrak{A}, \mathfrak{B}$ and each homomorphism $h : \mathfrak{A} \to \mathfrak{B}$, $(h_*)^* \circ f_{\mathfrak{A}} = f_{\mathfrak{B}} \circ h$, where $f_{\mathfrak{A}}$ is defined as $f_{\mathfrak{A}}(a) := \hat{a}$ and $f_{\mathfrak{B}}(b) := \hat{b}$.

Proof
For each $a \in A$, we need to prove $(h_*)^*(\hat{a}) = \widehat{h(a)}$. $G \in (h_*)^*(\hat{a}) \iff h_*(G) \in \hat{a} \iff a \in h_*(G) \iff h(a) \in G \iff G \in \widehat{h(a)}$. (Q.E.D)

Proposition 33
For each descriptive L-frames $\mathfrak{G}_1, \mathfrak{G}_2$ and each p-morphism ρ from $\mathfrak{G}_1$ to $\mathfrak{G}_2$, $(\rho^*)_* \circ f_{\mathfrak{G}_1} = f_{\mathfrak{G}_2} \circ \rho$, where $f_{\mathfrak{G}}$ is defined as $f_{\mathfrak{G}_1}(w) := \hat{w}$ and $f_{\mathfrak{G}_2}(w) := \hat{w}$.

Proof
For each $w \in W_1$, we need to prove $(\rho^*)_*(\hat{w}) = \widehat{\rho(w)}$. $X \in (\rho^*)_*(\hat{w}) \iff \rho^*(X) \in \hat{w} \iff w \in \rho^*(X) \iff \rho(w) \in X \iff X \in \widehat{\rho(w)}$. (Q.E.D)

From the above lemmas and propositions, we can prove the following theorem.

Theorem 34
1. The category $\mathcal{A}_L$ is dually equivalent to the category $\mathcal{D}_L$ by the contravariant functors $(\cdot)^*$ and $(\cdot)_*$.
2. The category $\mathcal{D}_L$ is a full reflexive subcategory of the category $\mathcal{D}_L^w$ through the reflector (left adjoint) $R : \mathcal{D}_L^w \to \mathcal{D}_L$ and the inclusion right adjoint $I : \mathcal{D}_L \to \mathcal{D}_L^w$. Moreover, the natural transformation $1_{\mathcal{D}_L^w} \to IR$ is the collection of quasi-isomorphisms.

Finally, we show the following lemma.

Lemma 35
1. if a homomorphism h is injective, then h_* is surjective,
2. if a homomorphism h is surjective, then h_* is both a quasi-embedding and an embedding,

3. if a p-morphism ρ is a quasi-embedding or an embedding, then ρ^* is surjective,
4. if a p-morphism ρ is surjective, then ρ^* is injective.

Proof

1. For each prime filter $G \in Pf(A)$, we define $\uparrow (h(G))$ and $\downarrow (h(A-G))$. Since h is injective, $\bot_B \notin \uparrow (h(G))$. Now, since $\uparrow (h(G)) \cap \downarrow (h(A-G)) = \emptyset$, by the prime filter theorem, there exists a prime filter F such that $\uparrow (h(G)) \subseteq F$ i.e. $G \subseteq h_*(F)$ and $F \cap \downarrow (h(A-G)) = \emptyset$. For each $a \in A$, if $a \in G$, $h(a) \in F \iff a \in h_*(F)$. Therefore, $G \subseteq h_*(F)$. Conversely, suppose that there exists $a \in h_*(F)$ but $a \notin G$. Then, $a \in A - G$. Hence, $h(a) \in P$ and $h(a) \in \downarrow (h(A-G))$, which a contradiction.
2. By Proposition 24, all we need to show is that h_* is a quasi-embedding. For each $b \in B$, since h is surjective, there exists $a \in A$ such that $h(a) = b$. If $G \in \uparrow (h_*(\hat{b}))$, there exists F such that $h_*(F) \subseteq G$ and $h_*(F) \in h_*(\hat{b})$. So, $F \in \hat{b} \iff h(a) = b \in F \iff a \in h_*(F) \Rightarrow G \in \hat{a}$. Conversely, if $G \in \hat{a}$, $G \in h_*(\hat{b})$. Suppose $h_*(F_1) \subseteq h_*(F_2)$. $b \in F_1 \iff h(a) \in F_1 \iff a \in h_*(F_1) \subseteq h_*(F_2) \iff h(a) = b \in F_2$.
3. The condition 9' in Definition 10 directly derive surjectivity of ρ^*. Moreover, quasi-embeddings also satisfy the condition 9'.
4. For each $w' \in W_2$, there exists $w \in W_1$ such that $\rho(w) = w'$. Assume $\rho^*(X) = \rho^*(Y)$, $w' \in X \iff \rho(w) \in X \iff w \in \rho^*(X) = \rho^*(Y) \iff \rho(w) \in Y \iff w' \in Y$. (Q.E.D)

Acknowledgements: The author would like to thank firstly Hiroakira Ono, Tadeusz Litak and Alexander Kurz; they always encourage him well. He also thanks the referees giving him a lot of fruitful comments. Finally, he thanks all of CALCO, CALCO-jnr participants and organizers, and the beautiful nature of Bergen.

References

1. P. Blackburn, M. D. Rijike, and Y. Venema. *Modal logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2002.
2. S. Burris and H.P. Sankappanavar. *A course in universal algebra*. Springer-Verlag, 1981.
3. A. Chagrov and M. Zakharyaschev. *Modal logic*, volume 35 of *Oxford Logic Guides*. Oxford Science Publications, 1997.
4. B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.
5. M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume III, chapter 3, pages 117-224. Kluwer Academic Publishers, 1986.
6. M. Dunn, M. Gehrke and A. Palmigiano. Canonical extensions and relational completeness of some substructural logics. The Journal of Symbolic Logic, 70:713-740, 2005.

7. N. Galatos. *Varieties of residuated lattices*. PhD thesis, Graduate School of Vanderbilt University, May 2003.
8. N. Galatos, P. Jipsen, T. Kowalski, and H. Ono. *Residuated lattices: an algebraic glimpse at substructural logics*, volume 151 of *Studies in Logics and the Foundation of Mathematics*. Elsevier, 2007.
9. M. Gehrke, H. Nagahashi, and Y. Venema. A Sahlqvist theorem for distributive modal logic. *Annals of Pure and Applied Logic*, 131:65-102, 2005.
10. R. Goldblatt. *Mathematics of modality*, volume 43 of *CSLI Lecture notes*. CSLI Publications, 1993.
11. P.T. Johnstone. *Stone spaces*, volume 3 of *Cambridge studies in advanced mathematics*, Cambridge University Press, 1982.
12. H. Ono. Substructural logics and residuated lattices - an introduction. *Trends in Logic: 50 Years of Studia Logica*, 21:193-228, 2003.
13. H. Ono and Y. Komori. Logics without the contraction rule. *The Journal of Symbolic Logic*, 50:169-201, 1985.
14. T. Seki. General frames for relevant modal logics. *Notre Dame Journal of Formal Logic*, 44:93-109, 2003.
15. T. Suzuki. Kripke completeness of some distributive substructural logics. Master's thesis, Japan Advanced Institute of Science and Technology, March 2007.
16. A. Urquhart. Duality for algebras of relevant logics, *Studia Logica*, 56:263-276, 1996.

Limits and colimits in categories of institutions

Adam Warski

University of Warsaw, Poland
adam@warski.org

1 Introduction

The theory of institutions, first introduced by Goguen and Burstall in 1984 ([GB83,GB92]), quickly gained ground and proved to be a very useful tool to construct and reason about logics in a uniform way. Since then, it has found many applications and has been widely developed. An institution is a formalization of a logical system—for example, we can build an institution of equational logic or first order logic.

There are two main ways of moving between institutions, using either institution morphisms or comorphisms. Informally, morphisms express how a "richer" institution is built over a "simpler" one; comorphisms express a relation going the other way round: how a "simpler" institution can be encoded in a "richer" one. These intuitions hint at some duality between the two concepts. Various properties of (co)morphisms are presented very thoroughly and systematically in [GR02]. Taking morphisms or comorphism, we can build two categories: **INS** and **coINS**, with institutions as objects.

In this article, I am going to analyse some of the relationships between limits and colimits of diagrams built from institutions linked by morphisms and comorphisms, as well as show the constructions of those limits and colimits. Even though morphisms and comorphisms may seem to be dual concepts at first, universal constructions associated with morphisms and comorphisms turn out to be rather different.

The main motivation behind this work takes source in heterogeneous specifications [Mos02b,Tar00], which are built over a number of institutions linked with morphisms or comorphisms. It is sometimes important to have the underlying diagram of institutions represented in a uniform way, using only morphisms or only comorphisms; hence the need to translate one into another. One way to do that is by transforming a morphism into a span of (co)morphisms (or vice versa), as introduced for example in [Mos02b]. Also, given such a diagram, it may be useful to represent a family of models of a heterogeneous distributed specification, or specifications themselves in an institution, which combines the institutions involved. Limits/colimits of institutions haven't proved to be the best tool for "putting institutions together" (see for example [GB85,Paw95]), however it may be suitable to use them as concise representations of institutions diagrams. This approach is different from the one taken by, for example, Mossakowski ([Mos02a,Mos06]) and Diaconescu ([Dia02]), where, given a diagram, the corresponding Grothendieck institution is built. Using this technique,

institutions are put into one, essentially "side by side", without much interaction. We are after a more compact representation, where some combination of signatures, models and sentences of the institutions involved takes place. One of the constructions to consider, when pursuing such a goal, is the construction of limits and colimits of diagrams of institutions.

Now comes the question: how do (co)limits of diagrams relate to (co)limits of diagrams built by replacing each morphism by a span of comorphisms? (Or each comorphism by a span of morphisms.) Here, we will show what the answer is for the case of limits. When morphisms are replaced by spans of comorphisms, the shape of the diagram changes. Hence the "procedure" for constructing limits changes as well; even though the new morphisms are of a special form. In general it seems there is no simple and straightforward way to translate between limits/colimits of the two diagrams, which shows that morphisms and comorphisms are not entirely dual.

All proofs of correctness of constructions and of theorems are left out. They are available in [War07].

2 Definitions

This section presents definitions used later in the article: of institutions, institution morphisms and comorphisms and various institution categories. Examples of these concepts can be found in [GB83], [GR02], [ST88] and many other papers dealing with institutions.

Definition 1. *An* institution $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$ *consists of:*

- *a category* $\mathbf{Sign}$ *of signatures,*
- *a functor* $\mathbf{Mod} \colon \mathbf{Sign}^{op} \to \mathbf{Cat}$*, which assigns to each signature a category of models.* $\mathbf{Cat}$ *is a category of "all" categories and functors between them,*
- *a functor* $\mathbf{Sen} \colon \mathbf{Sign} \to \mathbf{Set}$*, which assign to each signature a set of sentences,*
- *for each signature* $\Sigma \in |\mathbf{Sign}|$*, a satisfaction relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$*,*

Such that for each signature morphism $\sigma \colon \Sigma \to \Sigma'$*, sentence* $\varphi \in \mathbf{Sen}(\Sigma)$ *and model* $m' \in |\mathbf{Mod}(\Sigma')|$*, the satisfaction condition holds (***SC***):*

$$m' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(m') \models_\Sigma \varphi.$$

The following notations are used: $\sigma(\varphi)$ stands for $\mathbf{Sen}(\sigma)(\varphi)$ and $m'|_\sigma$ stands for $\mathbf{Mod}(\sigma)(m')$.

The satisfaction condition takes then the form:

$$m' \models_{\Sigma'} \sigma(\varphi) \iff m'|_\sigma \models_\Sigma \varphi.$$

Definition 2. *An* institution morphism $\mu : \mathbf{I} \to \mathbf{I}'$*,* $\mu = \langle \Phi, \alpha, \beta \rangle$*, where* $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$ *and* $\mathbf{I}' = \langle \mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models' \rangle$ *consists of:*

- *a functor between signature categories* $\Phi\colon \mathbf{Sign} \to \mathbf{Sign}'$
- *a natural transformation between model functors* $\alpha\colon \mathbf{Mod} \to (\Phi^{op});\mathbf{Mod}'$
- *a natural transformation between sentence functors* $\beta\colon \Phi;\mathbf{Sen}' \to \mathbf{Sen}$.

Also here, the satisfaction condition must hold, for each signature $\Sigma \in |\mathbf{Sign}|$, *sentence* $\varphi' \in \mathbf{Sen}'(\Phi(\Sigma))$, *and model* $m \in |\mathbf{Mod}(\Sigma)|$:

$$m \models_\Sigma \beta_\Sigma(\varphi') \iff \alpha_\Sigma(m) \models'_{\Phi(\Sigma)} \varphi'.$$

Note that the domain of the sentence functor is a "re-indexed" sentence functor of the institution $\mathbf{I}'$, and the codomain is the sentence functor of $\mathbf{I}$.

Intuitively, the institution $\mathbf{I}$ is more complicated than the institution $\mathbf{I}'$. A morphism between them shows how $\mathbf{I}$ is built upon $\mathbf{I}'$.

Definition 3. *An* institution comorphism $\rho\colon \mathbf{I} \to_{co} \mathbf{I}'$, $\rho = \langle \Phi, \alpha, \beta \rangle$, *where we have* $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$ *and* $\mathbf{I}' = \langle \mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models' \rangle$ *consists of:*

- *a functor* $\Phi\colon \mathbf{Sign} \to \mathbf{Sign}'$
- *a natural transformation* $\alpha\colon (\Phi^{op});\mathbf{Mod}' \to \mathbf{Mod}$
- *a natural transformation* $\beta\colon \mathbf{Sen} \to \Phi;\mathbf{Sen}'$

such that for each signature $\Sigma \in |\mathbf{Sign}|$, *sentence* $\varphi \in \mathbf{Sen}(\Sigma)$ *and model* $m' \in |\mathbf{Mod}'(\Phi(\Sigma))|$ *the satisfaction condition holds:*

$$m' \models'_{\Phi(\Sigma)} \beta_\Sigma(\varphi) \iff \alpha_\Sigma(m') \models_\Sigma \varphi.$$

Intuitively, ρ is a representation of institutions—it shows, how a simpler institution can be embedded into a richer one. Institution comorphisms were first introduced under the name "simple maps of institutions" by Meseguer, and as "representations" by Tarlecki in [Tar95].

Definition 4. *Having institutions and morphisms between them, we can build a* category of institutions **INS**.

- ***Objects:*** *institutions*
- ***Morphisms:*** *institution morphisms as defined in Def. 2.*
- ***Identities:*** *morphisms* $id = \langle id_{\mathbf{Sign}}, id_{\mathbf{Mod}}, id_{\mathbf{Sen}} \rangle$.
- ***Composition:*** *a composition of a morphism* $\mu_1 : \mathbf{I} \to \mathbf{I}'$ *with a morphism* $\mu_2 : \mathbf{I}' \to \mathbf{I}''$ *is a morphism* $\mu = \mu_1;\mu_2 : \mathbf{I} \to \mathbf{I}''$, *where for* $\mu_1 = \langle \Phi_1, \alpha_1, \beta_1 \rangle$, $\mu_2 = \langle \Phi_2, \alpha_2, \beta_2 \rangle$ *we define* $\mu = \langle \Phi, \alpha, \beta \rangle$:
  $\Phi = \Phi_1;\Phi_2 \quad : \mathbf{Sign} \to \mathbf{Sign}''$
  $\alpha = \alpha_1;((\Phi_1^{op}) \cdot \alpha_2) \; : \mathbf{Mod} \to (\Phi_1;\Phi_2)^{op};\mathbf{Mod}''$
  $\beta = (\Phi_1 \cdot \beta_2);\beta_1 \quad : \Phi_1;\Phi_2;\mathbf{Sen}'' \to \mathbf{Sen}$

Here $\cdot$ is the horizontal composition of natural transformations, and ; is the composition of functors or the vertical composition of natural transformations (depending on context). It is easy to check that the definition of identities is correct, that composition is associative, and that μ is indeed an institution morphism.

Definition 5. *Using comorphisms instead of morphisms we can also build another* category of institutions, **coINS**.

- ***Objects:*** *institutions*
- ***Morphisms:*** *comorphisms of institutions, as defined in Def. 3.*
- ***Identities:*** *comorphisms* $id = \langle id_{\mathbf{Sign}}, id_{\mathbf{Mod}}, id_{\mathbf{Sen}} \rangle$.
- ***Composition:*** *composition of a comorphism* $\rho_1 : \mathbf{I} \to \mathbf{I}'$ *with a comorphism* $\rho_2 : \mathbf{I}' \to \mathbf{I}''$ *is a comorphism* $\rho = \rho_1;\rho_2 : \mathbf{I} \to \mathbf{I}''$, *where for* $\rho_1 = \langle \Phi_1, \alpha_1, \beta_1 \rangle$, $\rho_2 = \langle \Phi_2, \alpha_2, \beta_2 \rangle$ *we define* $\rho = \langle \Phi_1;\Phi_2, \beta_1;(\Phi_1 \cdot \beta_2), ((\Phi_1^{op}) \cdot \alpha_2);\alpha_1 \rangle$.

Again it is easy to check that **coINS** is a category.

Definition 6. Categories of institutions **sINS** and **scoINS** *are full subcategories of, respectively,* **INS** *and* **coINS**, *where objects are only those institutions, in which signature categories are small (objects and morphisms of the signature category form a proper set).*

Definition 7. Categories of institutions $\mathbf{INS_{Sign}}$ and $\mathbf{coINS_{Sign}}$ (with a fixed signature category), where $\mathbf{Sign} \in |\mathbf{Cat}|$ is an arbitrary category *are subcategories of, respectively,* **INS** *and* **coINS**, *where objects are all institutions with a fixed signature category* **Sign**, *and morphisms are all institution morphisms/comorphisms, in which the functor between signature categories is an identity.*

Definition 8. *The* signature-projecting functor $\mathbf{C} : \mathbf{INS} \to \mathbf{Cat}$ *is defined as follows*

- $\mathbf{C}(\mathbf{I}) = \mathbf{Sign}$, *for each institution* $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$
- $\mathbf{C}(\mu) = \Phi$, *for each institution morphism* $\mu : \mathbf{I} \to \mathbf{I}' \in \mathbf{INS}$, *where* $\mu = \langle \Phi, \alpha, \beta \rangle$.

This is a functor which projects an institution on its signature category. We can also define an analogous functor with domain **coINS**.

3 Limits in INS

As mentioned in the introduction, diagrams of institutions often appear in heterogeneous specifications [Mos02b,Tar00]. One way of compactly representing such diagrams is by considering their limits.

Theorem 9. *The category* **INS** *is complete.*

This result is well-known, and the proof can be found for example in [Tar85]. However, the construction given there proceeds rather indirectly in several quite involved steps. Instead, here we give an explicit construction directly in terms of institutions and their morphisms in the diagram, thus offering a better "feel" and direct handle on the result. Here we will describe the construction of arbitrary limits; to do that it is enough to construct products of an arbitrary family of categories and equalizers of any two morphisms ([Mac71, Ch. V]). The constructions are easy and are done in a component-wise manner; the construction of model categories and sentence sets on each signature doesn't depend on the overall structure of the signature category.

3.1 Products in INS

For a given family of institutions, $\mathbf{I}_j \in |\mathbf{INS}|$, $j \in J$, where J is a set of indices and $\mathbf{I}_j = \langle \mathbf{Sign}_j, \mathbf{Mod}_j, \mathbf{Sen}_j, \models_j \rangle$, we define a product of this family, an institution $\mathbf{I} = \Pi_{j \in J} \mathbf{I}_j$.

- $\mathbf{Sign} = \Pi_{j \in J} \mathbf{Sign}_j$ is a product of categories:
  - objects are functions $\xi \colon J \to \biguplus_{j \in J} |\mathbf{Sign}_j|$, such that $\xi(j) \in |\mathbf{Sign}_j|$ for $j \in J$.
  - morphisms between ξ and ξ' are functions $\chi : J \to \biguplus_{j \in J} \mathbf{Sign}_j$, such that $\chi(j) \colon \xi(j) \to \xi'(j)$ in $\mathbf{Sign}_j$.
- $\mathbf{Mod}(\xi) = \Pi_{j \in J} \mathbf{Mod}_j(\xi(j))$ for $\xi \in |\mathbf{Sign}|$ (product of categories)
- $\mathbf{Mod}(\chi) = \chi^{mod}$, where the functor $\chi^{mod} \colon \mathbf{Mod}(\xi') \to \mathbf{Mod}(\xi)$ is defined as follows: $\chi^{mod}(m')(j) = \mathbf{Mod}_j(\chi(j))(m'(j))$, for $\chi \colon \xi \to \xi'$ in $\mathbf{Sign}$, $j \in J$ and $m' \in |\mathbf{Mod}(\xi')|$ (analogically for model morphisms).
- $\mathbf{Sen}(\xi) = \biguplus_{j \in J} \mathbf{Sen}_j(\xi(j))$ (coproduct of sets, its elements are pairs $\langle \varphi, j \rangle$, where $j \in J$ and $\varphi \in \mathbf{Sen}_j(\xi(j))$).
- $\mathbf{Sen}(\chi) = \chi^{sen}$, where $\chi^{sen} : \mathbf{Sen}(\xi) \to \mathbf{Sen}(\xi')$ is defined as follows: $\chi^{sen}(\langle \varphi, j \rangle) = \langle \mathbf{Sen}_j(\chi(j))(\varphi), j \rangle$, for $\chi : \xi \to \xi'$ in $\mathbf{Sign}$, $j \in J$ and $\varphi \in \mathbf{Sen}_j(\xi(j))$.
- satisfaction relation $\models_\xi$ for $\xi \in |\mathbf{Sign}|$, $m \in |\mathbf{Mod}(\xi)|$, $j \in J$ and $\varphi \in \mathbf{Sen}_j(\xi(j))$: $m \models_\xi \langle \varphi, j \rangle \iff m(j) \models^j_{\xi(j)} \varphi$

The projections $\pi_j : \mathbf{I} \to \mathbf{I}_j$ for $j \in J$ are defined in a straightforward way, $\pi_j = \langle \Phi_j, \alpha_j, \beta_j \rangle$:

- $\Phi_j(\xi) = \xi(j)$
- $\alpha_j(\xi)(m) = m(j)$, for $m \in |\mathbf{Mod}(\xi)|$ and similarly for model morphisms
- $\beta_j(\xi)(\varphi) = \langle \varphi, j \rangle$, for $\varphi \in \mathbf{Sen}_j(\xi(j))$

Lemma 10. *α and β are natural transformations and π_j for $j \in J$ are institution morphisms.*

Lemma 11. *$\Pi_{j \in J} \mathbf{I}_j$ with projections π_j for $j \in J$ is a product of institutions $\mathbf{I}_j$ for $j \in J$.*

3.2 Equalizers in INS

Given two "parallel" institution morphisms μ_1, $\mu_2 : \mathbf{I}_1 \to \mathbf{I}_2$, we define their equalizer $\mu \colon \mathbf{I} \to \mathbf{I}_1$, with domain $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$.

Sign is the subcategory of $\mathbf{Sign}_1$ such that $\Sigma \in |\mathbf{Sign}| \iff \Phi_1(\Sigma) = \Phi_2(\Sigma)$ and $\sigma \in \mathbf{Sign}(\Sigma, \Sigma') \iff \Phi_1(\sigma) = \Phi_2(\sigma)$. The functor $\Phi \colon \mathbf{Sign} \to \mathbf{Sign}_1$ is the inclusion. Hence, Φ is an equalizer of Φ_1, $\Phi_2 \colon \mathbf{Sign}_1 \to \mathbf{Sign}_2$ in **Cat**.

For $\Sigma \in |\mathbf{Sign}|$, $\mathbf{Mod}(\Sigma)$ is the subcategory of $\mathbf{Mod}_1(\Phi(\Sigma)) = \mathbf{Mod}_1(\Sigma)$, such that

$$m \in |\mathbf{Mod}(\Sigma)| \iff \alpha_1(\Sigma)(m) = \alpha_2(\Sigma)(m)$$
$$h \in \mathbf{Mod}(\Sigma)(m, m') \iff \alpha_1(\Sigma)(h) = \alpha_2(\Sigma)(h).$$

For $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Mod}(\sigma) = \mathbf{Mod}_1(\sigma)|_{\mathbf{Mod}(\Sigma)}$ (functor domain restriction).

For $\Sigma \in |\mathbf{Sign}|$, let $\mathbf{Sen}(\Sigma) = \mathbf{Sen}_1(\Phi(\Sigma))/\equiv_\Sigma = \mathbf{Sen}_1(\Sigma)/\equiv_\Sigma$, where $\equiv_\Sigma$ is the smallest equivalence relation such that $\beta_1(\Sigma)(\varphi) \equiv_\Sigma \beta_2(\Sigma)(\varphi)$ for all $\varphi \in \mathbf{Sen}_2(\Phi_1(\Sigma))$. The full relation satisfies this condition, and an intersection of two relations satisfying this condition also satisfies it, hence a smallest relation exists.

For $\sigma\colon \Sigma \to \Sigma'$ and $[\psi]_{\equiv_\Sigma} \in \mathbf{Sen}(\Sigma)$, we define:

$$\mathbf{Sen}(\sigma)([\psi]_{\equiv_\Sigma}) = [\mathbf{Sen}_1(\Phi(\sigma))(\psi)]_{\equiv_{\Sigma'}} = [\mathbf{Sen}_1(\sigma)(\psi)]_{\equiv_{\Sigma'}}.$$

Remark 12. The above definition is correct, that is, it does not depend on the choice of ψ.

Hence $\mathbf{Mod}(\Sigma)$ with inclusion $\alpha(\Sigma)\colon \mathbf{Mod}(\Sigma) \to \mathbf{Mod}_1(\Sigma)$ is an equalizer of functors $\alpha_1(\Sigma)$ and $\alpha_2(\Sigma)$ between categories $\mathbf{Mod}_1(\Sigma)$ and $\mathbf{Mod}_2(\Phi_1(\Sigma))$ (the choice of Φ_1 or Φ_2 is not important, as $\Phi_1(\Sigma) = \Phi_2(\Sigma)$ from the construction of the signature category), and $\beta(\Sigma)$ is a coequalizer of $\beta_1(\Sigma), \beta_2(\Sigma)\colon \mathbf{Sen}_2(\Phi_1(\Sigma)) \to \mathbf{Sen}_1(\Sigma)$.

The satisfaction relation for $\Sigma \in |\mathbf{Sign}|$ is defined as follows:

$$m \models_\Sigma [\psi]_{\equiv_\Sigma} \iff \text{for each } \psi' \in [\psi]_{\equiv_\Sigma}, m \models^1_\Sigma \psi'.$$

Lemma 13. *A morphism $\mu\colon \mathbf{I} \to \mathbf{I}_1$ defined as: $\mu = \langle \Phi, \alpha, \beta \rangle$, where $\beta(\Sigma) = [-]_{\equiv_\Sigma}$ for $\Sigma \in |\mathbf{Sign}|$ is an institution morphism.*

Lemma 14. *μ is an equalizer of μ_1 and μ_2.*

4 Colimits in INS

Another way of combining institutions in a diagram is taking its colimit. Dually to limits, to construct a colimit it suffices to show the construction of coproducts and coequalizers (see [Mac71, Chap. V]). However, colimits of arbitrary diagrams of institutions connected by morphisms do not always exist, because it is not always possible to construct a coequalizer of two morphisms. A counter example can be found in [GR02, Ex. 4.10].

However, the problems are purely set-theoretical. If we restrict our attention only to institutions, in which signature categories are small (in typical examples it is enough to restrict the alphabet of symbols used to build operation names), we will get the following result.

Theorem 15. *The category* **sINS** *is cocomplete.*

This result is also not new, and is mentioned for example in [GR02] and proved in [Ros99]. However again, no direct and explicit constructions are given there.

Below the constructions of coproducts and coequalizers are briefly described. It is relatively easy to construct coproducts (the construction is dual to the construction of products in **INS**) but the construction of coequalizers is much harder. Here, as opposed to limits, the constructions of model and sentence functors heavily depend on the overall structure of the signature category.

4.1 Coproducts in INS

For a given family of institutions $\mathbf{I}_j$, $j \in J$, where J is a set of indices, we define its coproduct, an institution $\mathbf{I} = \uplus_{j \in J} \mathbf{I}_j$.

- $\mathbf{Sign} = \biguplus_{j \in J} \mathbf{Sign}_j$ is a coproduct of categories:
  - objects are pairs $\langle \Sigma, j \rangle$, where $j \in J$ and $\Sigma \in |\mathbf{Sign}_j|$.
  - morphisms are pairs $\langle \sigma, j \rangle : \langle \Sigma, j \rangle \to \langle \Sigma', j \rangle$, where $j \in J$ and $\sigma : \Sigma \to \Sigma'$ is a morphism in $\mathbf{Sign}_j$; for $j \neq j'$, there are no morphisms between $\langle \Sigma, j \rangle$ and $\langle \Sigma', j' \rangle$.
- for $\langle \Sigma, j \rangle \in |\mathbf{Sign}|$, $\mathbf{Mod}(\langle \Sigma, j \rangle) = \mathbf{Mod}_j(\Sigma)$, $\mathbf{Sen}(\langle \Sigma, j \rangle) = \mathbf{Sen}_j(\Sigma)$
- for $\langle \sigma, j \rangle \in \mathbf{Sign}$, $\mathbf{Mod}(\langle \sigma, j \rangle) = \mathbf{Mod}_j(\sigma)$, $\mathbf{Sen}(\langle \sigma, j \rangle) = \mathbf{Sen}_j(\sigma)$
- satisfaction relation: for a signature $\langle \Sigma, j \rangle$, model $m \in |\mathbf{Mod}(\langle \Sigma, j \rangle)| = |\mathbf{Mod}_j(\Sigma)|$ and sentence $\varphi \in \mathbf{Sen}(\langle \Sigma, j \rangle) = \mathbf{Sen}_j(\Sigma)$, $m \models_{\langle \Sigma, j \rangle} \varphi \iff m \models^j_\Sigma \varphi$

The inclusions $\iota_j : \mathbf{I}_j \to \mathbf{I}$, $\iota_j = \langle \Phi_j, \alpha_j, \beta_j \rangle$ for $j \in J$, are defined as follows: $\Phi_j(\Sigma) = \langle \Sigma, j \rangle$, $\alpha_j(\Sigma) = id_{\mathbf{Mod}_j(\Sigma)}$ and $\beta_j(\Sigma) = id_{\mathbf{Sen}_j(\Sigma)}$.

Lemma 16. *α and β are natural transformations and ι_j for $j \in J$ are institution morphisms.*

Lemma 17. **I** *with inclusions ι_j for $j \in J$ is a coproduct of institutions $\mathbf{I}_j$, for $j \in J$.*

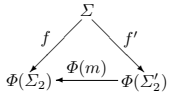
4.2 Coequalizers in sINS

Given two "parallel" morphisms $\mu_1, \mu_2 : \mathbf{I}_1 \to \mathbf{I}_2$ ($\mathbf{I}_1, \mathbf{I}_2 \in |\mathbf{sINS}|$, we will define their coequalizer $\mu : \mathbf{I}_2 \to \mathbf{I}$. The following construction is inspired by [TBG91, Ch. 3, Ex. 4], and coincides with the construction of a left Kan extension in a category of functors with a fixed codomain ([Mac71, Ch. X], [Ros99]).

Sign is the domain of a coequalizer of functors $\Phi_1, \Phi_2 : \mathbf{Sign}_1 \to \mathbf{Sign}_2$. The construction of coequalizers in **Cat** can be found in [MB99]. It is a bit more complicated then in **Set**, but they are roughly analogous. Objects in **Sign** are equivalence classes of the smallest equivalence relation $\equiv \subseteq |\mathbf{Sign}_2| \times |\mathbf{Sign}_2|$ such that for all $\Sigma \in |\mathbf{Sign}_1|$, $\Phi_1(\Sigma) \equiv \Phi_2(\Sigma)$. Morphisms can be defined in a similar way.

Let $\Sigma \in |\mathbf{Sign}|$ be an arbitrary signature. We define a graph $\mathbf{G}_\Sigma$ as follows:

- **nodes**:
  - $\langle \Sigma_1, f, 1 \rangle$, where $\Sigma_1 \in |\mathbf{Sign}_1|$, $f : \Sigma \to \Phi(\Phi_1(\Sigma_1))$ in **Sign** (choosing Φ_1 or Φ_2 does not matter, because from the construction of Φ we have $\Phi_1;\Phi = \Phi_2;\Phi$).
  - $\langle \Sigma_2, f, 2 \rangle$, where $\Sigma_2 \in |\mathbf{Sign}_2|$, $f : \Sigma \to \Phi(\Sigma_2)$ in **Sign**
- **edges**:



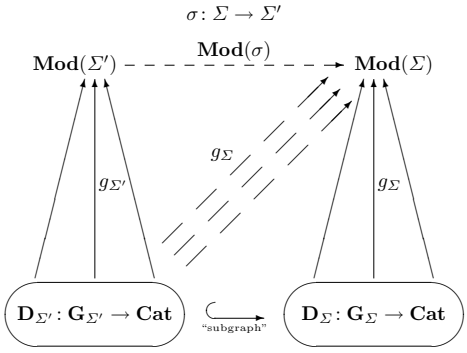
- $m : \langle \Sigma_2, f, 2\rangle \to \langle \Sigma_2', f', 2\rangle$, where $m : \Sigma_2' \to \Sigma_2$ in $\mathbf{Sign}_2$, is such that $f';\Phi(m) = f$.
- $\langle n_i, m\rangle : \langle \Sigma_1, f, 1\rangle \to \langle \Sigma_2', f', 2\rangle$, $i = 1, 2$, $m : \Sigma_2' \to \Phi_i(\Sigma_1)$ in $\mathbf{Sign}_2$, is such that $f';\Phi(m) = f$. □

Informally, all of the above nodes are needed so that we can define the model functor on signature morphisms. The first type of edges ("m") is needed to ensure that the resulting construction will be universal; and finally the "$\langle n_i, m\rangle$" edges are there so that the construction will have the coequalizer property.

Remark 18. *Note that only when the category* **Sign** *is small, we can be sure that we will be able to define the graph* $\mathbf{G}_\Sigma$ *(with a* set *of nodes and edges). This is provided by the fact that when both* $\mathbf{Sign}_1$ *and* $\mathbf{Sign}_2$ *are small categories,* **Sign** *is also a small category.*

Next, we define a diagram $\mathbf{D}_\Sigma : \mathbf{G}_\Sigma \to \mathbf{Cat}$ as follows:

- $\mathbf{D}_\Sigma(\langle \Sigma_2, f, 2\rangle) = \mathbf{Mod}_2(\Sigma_2)$
- $\mathbf{D}_\Sigma(\langle \Sigma_1, f, 1\rangle) = \mathbf{Mod}_1(\Sigma_1)$
- $\mathbf{D}_\Sigma(m) = \mathbf{Mod}_2(m)$
- $\mathbf{D}_\Sigma(\langle n_i, m\rangle) = \alpha_i(\Sigma_1);\mathbf{Mod}_2(m)$, where $\langle n_i, m\rangle : \langle \Sigma_1, f, 1\rangle \to \langle \Sigma_2', f', 2\rangle$. □



Let $\mathbf{Mod}(\Sigma)$ be a colimit of the diagram $\mathbf{D}_\Sigma$ in the category **Cat**. The injection (functor) of $\mathbf{D}_\Sigma(\langle \Sigma_i, f, i\rangle)$ into the colimit $\mathbf{Mod}(\Sigma)$ we will denote by $g_\Sigma^{\langle \Sigma_i, f, i\rangle} : \mathbf{Mod}_i(\Sigma_i) \to \mathbf{Mod}(\Sigma)$, $i = 1, 2$.

Let $\sigma : \Sigma \to \Sigma'$ be an arbitrary morphism in **Sign**. We define **Mod** on this morphism. Firstly, we build a cocone for the diagram $\mathbf{D}_{\Sigma'}$, with a vertex $\mathbf{Mod}(\Sigma)$. Injections into this cocone will be denoted by $k_{\Sigma'}^{\langle \Sigma_i', f, i\rangle} : \mathbf{Mod}_i(\Sigma_i') \to \mathbf{Mod}(\Sigma)$. The graph $\mathbf{G}_{\Sigma'}$ is a "subgraph" of $\mathbf{G}_\Sigma$: each node of the form $\langle \Sigma_i', f, i\rangle$ in $\mathbf{G}_{\Sigma'}$ has a corresponding node $\langle \Sigma_i', \sigma; f, i\rangle$ in $\mathbf{G}_\Sigma$; moreover, the values of the two nodes and of any edges between corresponding nodes (in $\mathbf{G}_{\Sigma'}$) are identical in diagrams $\mathbf{D}_\Sigma$ and $\mathbf{D}_{\Sigma'}$. Hence, if for an injection into the cocone's vertex from the value of a node $\langle \Sigma_i', f, i\rangle$ we take $k_{\Sigma'}^{\langle \Sigma_i', f, i\rangle} = g_\Sigma^{\langle \Sigma_i', \sigma; f, i\rangle}$, we will get a cocone over $\mathbf{D}_{\Sigma'}$ with a vertex in $\mathbf{Mod}(\Sigma)$. Let $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$

be the unique morphism (which exists, as $\mathbf{Mod}(\Sigma')$ is a colimit of the diagram $\mathbf{D}_{\Sigma'}$) such that for all nodes $\langle \Sigma'_i, f, i\rangle$ in $\mathbf{G}_{\Sigma'}$ (and corresponding nodes in $\mathbf{G}_\Sigma$): $g_{\Sigma'}^{\langle \Sigma'_i,f,i\rangle};\mathbf{Mod}(\sigma) = k_{\Sigma'}^{\langle \Sigma'_i,f,i\rangle} = g_{\Sigma}^{\langle \Sigma'_i,\sigma;f,i\rangle}$.

Lemma 19. $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$ *is a functor.*

We then define the transformation $\alpha\colon \mathbf{Mod}_2 \to \Phi;\mathbf{Mod}$, let $\Sigma_2 \in |\mathbf{Sign}_2|$:

$$\alpha(\Sigma_2) = g_{\Phi(\Sigma_2)}^{\langle \Sigma_2, id, 2\rangle} \colon \mathbf{Mod}_2(\Sigma_2) \to \mathbf{Mod}(\Phi(\Sigma_2)).$$

The sentence functor **Sen** is defined in similar way; for $\Sigma \in |\mathbf{Sign}|$, $\mathbf{Sen}(\Sigma)$ is a limit of a diagram $\mathbf{E}_\Sigma : \mathbf{G}_\Sigma^{op} \to \mathbf{Set}$, which is defined similarly as above. Also, **Sen** is extended to a functor analogously. The projections on the value of a node $\langle \Sigma_i, f, i\rangle$ in $\mathbf{G}_\Sigma^{op}$ will be denoted by $h_\Sigma^{\langle \Sigma_i,f,i\rangle} : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}_i(\Sigma_i)$. The transformation $\beta\colon \Phi;\mathbf{Sen} \to \mathbf{Sen}_2$ is defined on $\Sigma_2 \in |\mathbf{Sign}_2|$ as: $\beta(\Sigma_2) = h_{\Phi(\Sigma_2)}^{\langle \Sigma_2, id_{\Phi(\Sigma_2)}, 2\rangle}$.

Lemma 20. α *and* β *are natural transformations.*

The satisfaction relation in $\mathbf{I}$ is defined, for $\Sigma \in |\mathbf{Sign}|$, $m \in |\mathbf{Mod}(\Sigma)|$ and $\varphi \in \mathbf{Sen}(\Sigma)$, as follows:

$$m \models_\Sigma \varphi \iff m_2 \models^2_{\Sigma_2} h_\Sigma^{\langle \Sigma_2,f,2\rangle}(\varphi),$$

where $m_2 \in |\mathbf{Mod}_2(\Sigma_2)|$ is such, that $g_\Sigma^{\langle \Sigma_2,f,2\rangle}(m_2) = m$.

Lemma 21. *The required* m_2 *always exists, and the definition of the satisfaction relation is independent of the choice of* m_2*.*

5 Limits and colimits in coINS

Similar results hold for the category **coINS**. The constructions are much like the ones presented above. Like the results on completeness and cocompleteness of **INS** and **coINS**, these theorems have also been known to be true before ([GR02,Ros99]), but I have not found explicit constructions. Again, the category **coINS** is not cocomplete, for a reason analogous to **INS** not being cocomplete.

Theorem 22. *The category* **coINS** *is complete.*

Theorem 23. *The category* **scoINS** *is cocomplete.*

6 The categories $\mathbf{INS_{Sign}}$ and $\mathbf{coINS_{Sign}}$

Categories of institutions with a fixed signature category exhibit some interesting properties. In particular, as the category of signatures does not change, there is no significant difference between a morphism and a comorphism, and, moreover, the construction of a colimit of a diagram is as easy as the construction of a limit. Also, the constructions of limits and colimits in $\mathbf{INS_{Sign}}$ and $\mathbf{coINS_{Sign}}$ can be used to construct limits of diagrams in **INS** and **coINS**.

6.1 Limits and colimits

The construction of an equalizer of two morphisms in $\mathbf{INS_{Sign}}$ is exactly the same as in Sect. 3.2, as it easily follows from that construction that the signature category of the domain of an equalizer will be equal to **Sign**.

To construct products in $\mathbf{INS_{Sign}}$, we need to make a slight change to the construction presented in Sect. 3.1, by making a requirement that the signature category of the product must be **Sign**, and not $\mathbf{Sign} \times \mathbf{Sign}$. However it's the only change, and the rest of the construction remains the same.

Analogously, we can define products and equalizers in $\mathbf{coINS_{Sign}}$. Thus, we get:

Theorem 24. *The categories* $\mathbf{INS_{Sign}}$ *and* $\mathbf{coINS_{Sign}}$ *are complete.*

Moreover, let's consider an arbitrary morphism $\mu\colon \mathbf{I} \to \mathbf{I}'$ in $\mathbf{INS_{Sign}}$, where $\mu = \langle id_{\mathbf{Sign}}, \alpha, \beta \rangle$. It is easy to check, that $\rho\colon \mathbf{I}' \to \mathbf{I}$, $\rho = \langle id_{\mathbf{Sign}}, \alpha, \beta \rangle$, is an comorphism in $\mathbf{coINS_{Sign}}$ (in fact, we can change a morphism into a comorphism using such a technique whenever the functor between signature categories has a left adjoint, see [AF95]). More formally:

Fact 25. $\mu\colon \mathbf{I} \to \mathbf{I}'$*, where* $\mu = \langle id_{\mathbf{Sign}}, \alpha, \beta \rangle$ *is an institution morphism if and only if* $\rho\colon \mathbf{I}' \to \mathbf{I}$*,* $\rho = \langle id_{\mathbf{Sign}}, \alpha, \beta \rangle$*, is an institution comorphism.*

Corollary 26. $\mathbf{INS_{Sign}} \cong (\mathbf{coINS_{Sign}})^{op}$.

It easily follows from Thm. 24 and Cor. 26 that:

Theorem 27. *The categories* $\mathbf{INS_{Sign}}$ *and* $\mathbf{coINS_{Sign}}$ *are cocomplete.*

6.2 "Flattening" a diagram in INS to a diagram in coINS

Suppose we have a diagram $\mathbf{D}\colon \mathbf{G} \to \mathbf{INS}$, which has nodes $\mathbf{I}_i$, $\mathbf{I}_j$ for $i, j \in |\mathbf{G}|$, and morphisms $\mu_{k,i,j}\colon \mathbf{I}_i \to \mathbf{I}_j$, for $k \in K_{i,j}$, where $K_{i,j}$ is a set of indices. For notational convenience, the coordinate k will be omitted.

Let **Sign** and morphisms $\Phi_i : \mathbf{Sign} \to \mathbf{Sign}_i$ be a limit of the diagram $\mathbf{D};\mathbf{C}\colon \mathbf{G} \to \mathbf{Cat}$ (see Def. 8).

Given $\mathbf{D}$, we build another diagram $\mathbf{D}'\colon \mathbf{G} \to \mathbf{INS_{Sign}}$, with nodes $\mathbf{I}'_i$ and morphisms between them $\mu'_{i,j}\colon \mathbf{I}'_i \to \mathbf{I}'_j$.

Each node $\mathbf{I}_i = \langle \mathbf{Sign}_i, \mathbf{Mod}_i, \mathbf{Sen}_i, \models_i \rangle$ in diagram $\mathbf{D}$ we change to a node $\mathbf{I}'_i$ in $\mathbf{D}'$ with the signature category **Sign** in the following way:

$$\mathbf{I}'_i = \langle \mathbf{Sign}, \Phi_i^{op}; \mathbf{Mod}_i, \Phi_i; \mathbf{Sen}_i, \Phi_i; \models^i \rangle,$$

where $\Phi_i; \models^i$ is a relation that for $\Sigma \in |\mathbf{Sign}|$ is equal to $\models^i_{\Phi_i(\Sigma)}$.

It is easy to check that the satisfaction condition in $\mathbf{I}'_i$ holds.

A morphism $\mu_{i,j} = \langle \Phi_{i,j}, \alpha_{i,j}, \beta_{i,j} \rangle$ in $\mathbf{D}$ is changed to a morphism in $\mathbf{D}'$:

$$\mu'_{i,j} = \langle id, \Phi_i^{op} \cdot \alpha_{i,j}, \Phi_i \cdot \beta_{i,j} \rangle.$$

This definition is correct, as from the construction of **Sign** we have $\Phi_i;\Phi_{i,j} = \Phi_j$, hence:

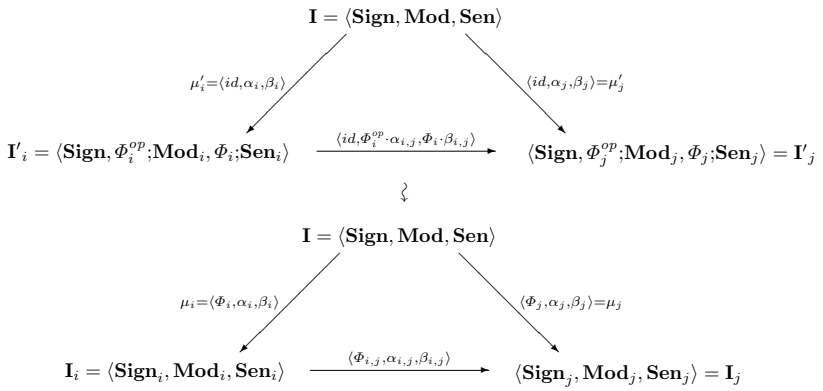
$$\Phi_i^{op} \cdot \alpha_{i,j} \colon \Phi_i^{op};\mathbf{Mod}_i \to \Phi_i^{op};\Phi_{i,j}^{op};\mathbf{Mod}_j = \Phi_j^{op};\mathbf{Mod}_j,$$

and similarly for β. The satisfaction condition for that morphism holds, which follows immediately from the satisfaction condition for $\mu_{i,j}$.

For the diagram $\mathbf{D}' \colon \mathbf{G} \to \mathbf{INS}_{\mathbf{Sign}}$ we can construct a limit, as it is described in Sect. 6.1, which will be denoted as $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$, where $\mathbf{Mod} \colon \mathbf{Sign}^{op} \to \mathbf{Cat}$, $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$. We also get projections $\mu'_i = \langle id, \alpha_i, \beta_i \rangle : \mathbf{I} \to \mathbf{I}'_i$, with natural transformations $\alpha_i \colon \mathbf{Mod} \to \mathbf{Mod}'_i$ and $\beta_i \colon \mathbf{Sen}'_i \to \mathbf{Sen}$.

6.3 Translating a limit of the "flattened" diagram to a limit of the original diagram

Having a limit $\mathbf{I}$ of $\mathbf{D}'$, it is easy to construct a limit of $\mathbf{D}$:



For each node, we get a morphism $\mu_i : \mathbf{I} \to \mathbf{I}_i$ by taking a functor projecting **Sign** on $\mathbf{Sign}_i$ and natural transformations from the projections in $\mathbf{D}'$: $\mu_i = \langle \Phi_i, \alpha_i, \beta_i \rangle$. From the definitions of "flattening" a node the natural transformations α_i and β_i are such, that: $\alpha_i \colon \mathbf{Mod} \to \Phi_i^{op};\mathbf{Mod}_i$ and $\beta_i \colon \Phi_i;\mathbf{Sen}_i \to \mathbf{Sen}$. It is easy to check, that μ_i is an institution morphism. It is not quite trivial to verify that $\mathbf{I}$ with projections $\mu_i \colon \mathbf{I} \to \mathbf{I}_i$ is in fact a limit of $\mathbf{D}$.

Theorem 28. *The institution* $\mathbf{I}$ *with projections* $\mu_i \colon \mathbf{I} \to \mathbf{I}_i$ *is a limit of diagram* $\mathbf{D}$.

A construction similar to the one presented above can be found for example in [TBG91, Ch. 4, Lem. 2].

7 Changing morphisms into comorphisms

When examining the definitions of morphisms and comorphisms (2, 3), one can see some duality between the two concepts. It would also be useful to have a way

of representing morphisms as comorphisms and vice versa. Also, in specific diagrams morphisms and comorphisms may coexists, and it is easier to reason about a diagram if it has only one type of morphisms. One way, described for example in [Mos02b,Mos06] is to replace a morphism with a span of comorphisms. It is also possible to represent a comorphism as a span of two morphisms. However below we will concentrate on the former, as the category **coINS** appears to be the most suitable for investigating the properties of heterogeneous specifications ([Mos02b]).

7.1 Spans of comorphisms

Suppose we have an institution morphism: $\mu\colon \mathbf{I}_1 \to \mathbf{I}_2 = \langle \Phi, \alpha, \beta \rangle$. We define an "intermediary" institution $\mathbf{I}' = \langle \mathbf{Sign}_1, \Phi^{op}; \mathbf{Mod}_2, \Phi; \mathbf{Sen}_2, \Phi; \models^2 \rangle$, which consists of a category of signatures from the first institution, and sentence and model functors from the second institution (here, $\Phi;\models^2$ is a relation, which for $\Sigma \in |\mathbf{Sign}_1|$ is equal to $\models^2_{\Phi(\Sigma)}$).

It is easy to check that this definition is correct. We can also define two morphisms, $\mu_1 : \mathbf{I}_1 \to \mathbf{I}'$ and $\mu_2 : \mathbf{I}' \to \mathbf{I}_2$, where $\mu_1 = \langle id, \alpha, \beta \rangle$ and $\mu_2 = \langle \Phi, id, id \rangle$, which are such that $\mu_1;\mu_2 = \mu$.

Morphism, in which the functor between signature categories is an identity, can be easily changed to comorphism (Fact 25). Moreover, starting with a morphism, in which the natural transformations between model and sentence functors are identities, we can easily build a comorphism: it will consist of exactly the same parts (but with the identity natural transformations considered in the "opposite" direction). The domain or codomain of the morphism doesn't change either. Hence, having a morphism, we can build a span of two comorphisms.

Thus, if we have a morphism: $\mathbf{I}_1 \overset{\langle \Phi, \alpha, \beta \rangle}{\longrightarrow} \mathbf{I}_2$ we can change it to a pair of morphisms: $\mathbf{I}_1 \overset{\langle id, \alpha, \beta \rangle}{\longrightarrow} \mathbf{I}' \overset{\langle \Phi, id, id \rangle}{\longrightarrow} \mathbf{I}_2$ and next to a pair of comorphisms, "reversing" the first, and leaving the second without any changes: $\mathbf{I}_1 \overset{\langle id, \alpha, \beta \rangle}{\underset{co}{\longleftarrow}} \mathbf{I}' \overset{\langle \Phi, id, id \rangle}{\underset{co}{\longrightarrow}} \mathbf{I}_2$. Informally, a span of comorphisms expresses "the same" relation between institutions, as the original morphism.

In a very similar way we can change a comorphism into a span of morphisms.

8 Constructing limits of diagrams with each morphism replaced by a span

Having a way of representing morphisms as spans of comorphisms, it is natural to ask, how do (co)limits of diagrams of institutions correspond to (co)limits of diagrams, in which each morphism has been changed into a span of comorphisms. As the comorphisms used in the spans that replace institution morphisms are quite specific (contain many identities), in the case of limits there exists an easy way to construct them for diagrams obtained in such a way.

Consider a diagram $\mathbf{D}\colon \mathbf{G} \to \mathbf{INS}$ of institutions and their morphisms./

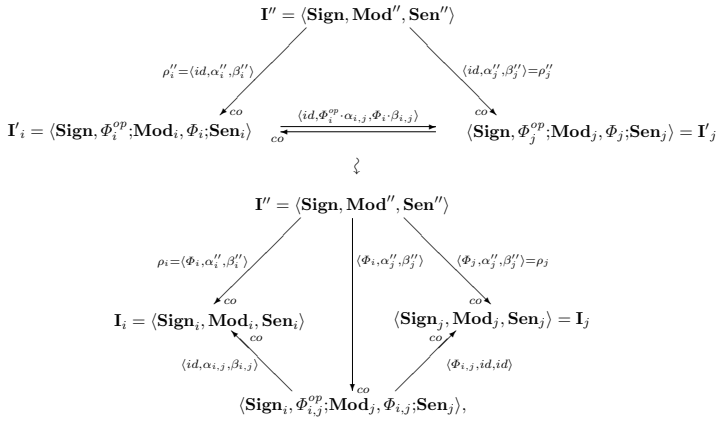
8.1 "Flattened" diagrams and spans

Suppose we "flatten" $\mathbf{D}$ to a diagram $\mathbf{D}'$ as in Sect. 6.2, where also the category **Sign** is defined. We construct new diagrams:

- diagram $\mathbf{coD}\colon \mathbf{coG} \to \mathbf{coINS}$, is a diagram $\mathbf{D}$, in which each morphism has been changed to a span of comorphisms
- diagram $\mathbf{D}''\colon \mathbf{G}^{op} \to \mathbf{coINS}_{\mathbf{Sign}}$ is a diagram $\mathbf{D}'$, in which each morphism has been changed to a comorphism (as in fact 25); its vertices are institutions $\mathbf{I}'_i$ ("flattened" institutions $\mathbf{I}_i$).

The institution that is the vertex of the limit of the diagram $\mathbf{D}''$ will be denoted by $\mathbf{I}'' = \langle \mathbf{Sign}, \mathbf{Mod}'', \mathbf{Sen}'', \models'' \rangle$, with projections $\rho''_i : \mathbf{I}'' \to \mathbf{I}'_i = \langle id, \alpha''_i, \beta''_i \rangle$ for $i \in |\mathbf{G}|$.

From a limit of the diagram $\mathbf{D}''$ we can easily get a limit of the diagram **coD**:



where we put $\rho_i\colon \mathbf{I}'' \to \mathbf{I}_i = \langle \Phi_i, \alpha''_i, \beta''_i \rangle$, for $i \in |\mathbf{G}|$.

Theorem 29. *The institution and comorphisms constructed above are a limit of the diagram* **coD**.

8.2 Relations between limits of diagrams and limits of diagrams of spans

So, we can construct a limit of a diagram of institutions and institution morphisms, and a limit of a diagram, in which each morphism has been replaced by a span of comorphisms. The natural question is how much the two limits are related. Informally, the limit of the original diagram in **INS** is an institution that is "richer" than all the institutions in the diagram, while the limit of this diagram in **coINS** is "poorer" then all institutions in the diagram. Moreover, an institution morphism "represents" a richer institution in a simpler one, so, if a relation exists, it can be in the form of a morphism from $\mathbf{I}$ to $\mathbf{I}''$ (or a comorphism from $\mathbf{I}''$ to $\mathbf{I}$).

It is the case that such a morphism always exist, and in some diagrams there can be many of them. We can build one morphism from $\mathbf{I}$ to $\mathbf{I}''$ for each node of the diagram, but morphisms built for vertices connected by any path in the graph turn out to be the same. Hence, we can build one morphism for each connected component of the graph. Of course, some of them may turn out to coincide—but only in specific cases.

Fact 30. *For each node* $i \in |\mathbf{G}|$, $\langle id, \alpha_i;\alpha_i'', \beta_i'';\beta_i\rangle : \mathbf{I} \to \mathbf{I}''$ *is an institution morphism. Moreover, from fact 25, for each node* $i \in |\mathbf{G}|$, $\langle id, \alpha_i;\alpha_i'', \beta_i'';\beta_i\rangle : \mathbf{I}'' \to \mathbf{I}$ *is an institution comorphism.*

Fact 31. *For vertices* $i, j \in |\mathbf{G}|$ *connected by any path in the graph we have:* $\langle id, \alpha_i;\alpha_i'', \beta_i'';\beta_i\rangle = \langle id, \alpha_j;\alpha_j'', \beta_j'';\beta_j\rangle$.

9 Conclusions and further work

In this paper, the constructions of limits and colimits in categories of institutions have been presented; the results on completeness and cocompleteness of these categories have been known before, however the proofs did not show direct constructions. Explicit constructions are needed when these theorems are to be apllied to a specific diagram of institutions.

Moreover, as the constructions of limits and colimits turn out to be rather different, it seems that institution morphisms and comorphisms are not dual concepts, as a first intuition may suggest.

The properties of diagrams of institutions with a fixed signature category are also presented, as well as means of translating an arbitrary diagram to a diagram with a fixed signature category, and the relations between the two diagrams. The final part of the article describes some connections between limits of diagrams with morphisms, and limits of diagrams in which each morphism has been changed into a span of comorphisms.

What remains to be investigated, is the possible relation between (co)limits of diagrams of institutions and corresponding Grothendieck institutions, as well as how these results apply to specification theory.

References

[AF95] M. Arrais and José Luiz Fiadeiro. Unifying theories in different institutions. In Haveraaen et al. [HOD96], pages 81–101.

[Dia02] Razvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402, 2002.

[GB83] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In Edmund M. Clarke and Dexter Kozen, editors, *Logic of Programs*, volume 164 of *LNCS*, pages 221–256. Springer, 1983.

[GB85] Joseph A. Goguen and Rod M. Burstall. A study in the functions of programming methodology: Specifications, institutions, charters and parchments. In Pitt et al. [PAPR86], pages 313–333.

[GB92] Joseph Goguen and Rod Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, January 1992.

[GR02] Joseph A. Goguen and Grigore Rosu. Institution morphisms. *Formal Asp. Comput.*, 13(3-5):274–307, 2002.

[HOD96] Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors. *Recent Trends in Data Type Specification, 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop, Oslo, Norway, September 19-23, 1995, Selected Papers*, volume 1130 of *LNCS*. Springer, 1996.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[MB99] Wiesaw Pawlowski Marek Bednarczyk, Andrzej Borzyszkowski. Generalized congruences. *Theory and Applications of Categories*, 5(11):266–280, 1999.

[Mos02a] Till Mossakowski. Comorphism-based Grothendieck logics. In Krzysztof Diks and Wojciech Rytter, editors, *MFCS*, volume 2420 of *LNCS*, pages 593–604. Springer, 2002.

[Mos02b] Till Mossakowski. Foundations of heterogeneous specification. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *WADT*, volume 2755 of *LNCS*, pages 359–375. Springer, 2002.

[Mos06] Till Mossakowski. Institutional 2-cells and Grothendieck Institutions. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *LNCS*, pages 124–149. Springer, 2006.

[PAPR86] David H. Pitt, Samson Abramsky, Axel Poigné, and David E. Rydeheard, editors. *Category Theory and Computer Programming, Tutorial and Workshop, Guildford, UK, September 16-20, 1985 Proceedings*, volume 240 of *LNCS*. Springer, 1986.

[Paw95] Wiesaw Pawlowski. Context institutions. In Haveraaen et al. [HOD96], pages 436–457.

[Ros99] Grigore Rosu. Kan extensions of institutions. *J. UCS*, 5(8):482–493, 1999.

[ST88] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Inf. Comput.*, 76(2-3):165–210, 1988.

[Tar85] Andrzej Tarlecki. Bits and pieces of the theory of institutions. In Pitt et al. [PAPR86], pages 334–365.

[Tar95] Andrzej Tarlecki. Moving between logical systems. In Haveraaen et al. [HOD96], pages 478–502.

[Tar00] Andrzej Tarlecki. Towards heterogeneous specifications. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Studies in Logic and Computation, pages 337–360. Research Studies Press, 2000.

[TBG91] Andrzej Tarlecki, Rod M. Burstall, and Joseph A. Goguen. Some fundamental algebraic tools for the semantics of computation: Part 3: Indexed categories. *Theor. Comput. Sci.*, 91(2):239–264, 1991.

[War07] Adam Warski. Granice i kogranice w roznych kategoriach instytucji (Limits and colimits in various categories of institutions). Master's thesis, University of Warsaw, 2007. in Polish.

Author Index