CALCO Young Researchers Workshop
CALCO-jnr 2007

20 August 2007

Abstracts for Presentations

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 takes place in Bergen, Norway.

The CALCO Young Researchers Workshop, CALCO-jnr, is a CALCO satellite event dedicated to presentations by PhD students and by those who completed their doctoral studies within the past few years. Attendance at the workshop is open to all - it is anticipated that many CALCO conference participants attend the CALCO-jnr workshop (and vice versa).

CALCO-jnr presentations have been selected on the basis of submitted 2-page abstracts, by the CALCO-jnr PC. This booklet contains the abstracts of the accepted contributions.

After the workshop, the author(s) of each presentation will be invited to submit a full 10-15 page paper on the same topic. They will also be asked to write (anonymous) reviews of papers submitted by other authors on related topics. Additional reviewing and the final selection of papers will be carried out by the CALCO-jnr PC. The volume of selected papers from the workshop will be published as a Department of Informatics, University of Bergen, technical report, and it will also be made 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 CALCO 2007 local organisers for their efforts to make this event possible. The support of all sponsoring institutions is gratefully acknowledged: Department of Informatics, University of Bergen, Bergen University College, Research Council of Norway, City of Bergen, and IFIP WG1.3 on Foundations of System Specification.

August 2007

Magne Haveraaen
John Power
Monika Seisenberger

Table of Contents

The Microcosm Principle and Concurrency in Coalgebras

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

[1] ICIS, Radboud University Nijmegen, the Netherlands,
[2] University of Salzburg, Austria,
[3] Technical University Eindhoven, the Netherlands
[4] RIMS, Kyoto University, Japan,
{ichiro,bart}@cs.ru.nl, anas@cs.uni-salzburg.at

Our questions. *Compositionality* is an important property in modular verification of complex component-based systems. It is usually expressed as follows: $x \sim x'$ and $y \sim y'$ implies $(x \parallel y) \sim (x' \parallel y')$.

When we take *final coalgebra semantics* as behavior of systems

$$\begin{array}{ccc} FX & \dashrightarrow & FZ \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle \text{final}} \\ X & \underset{\mathsf{beh}(c)}{\dashrightarrow} & Z \end{array}$$

the following comes natural as a "coalgebraic presentation of compositionality".

$$\mathsf{beh}\left(\begin{array}{c} FX \\ {\scriptstyle c}\uparrow \\ X \end{array} \;\middle\|\; \begin{array}{c} FY \\ {\scriptstyle d}\uparrow \\ Y \end{array}\right) \quad = \quad \mathsf{beh}\left(\begin{array}{c} FX \\ {\scriptstyle c}\uparrow \\ X \end{array}\right) \;\middle\|\; \mathsf{beh}\left(\begin{array}{c} FY \\ {\scriptstyle d}\uparrow \\ Y \end{array}\right) \tag{1}$$

Here arise some questions which we believe are important for our mathematical understanding of parallel composition, or *concurrency*, of systems.

- The operator $\parallel$ on the left gives us composition of *systems*, being an operation on the category $\mathbf{Coalg}_F$. When is it available?
- The other $\parallel$ appearing on the right has a different domain: it is an operation on the final coalgebra Z! In this way we observe
  - the same algebraic structure (or algebraic "theory") which concerns the operation $\parallel$ and possibly some axioms like associativity,
  - interpreted on two different levels—on a category $\mathbf{Coalg}_F$ and on its object $Z \in \mathbf{Coalg}_F$—in a nested manner.

  What is the mathematical principle behind this?

In the sequel we shall sketch our first answers given in our preprint [2]. Our title refers to the phenomenon of nested algebraic structures which is called the *microcosm principle* [1].

Parallel composition of coalgebras. The "outer" composition (of coalgebras) is described as a bifunctor $\mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$. Such an operation is usually denoted by $\otimes$ (rather than $\parallel$) and called a *tensor product*: we follow this tradition. Composition of systems is most of the time *associative*—$(c \otimes d) \otimes e \cong c \otimes (d \otimes e)$—making $\otimes$ an *associative tensor*.

Theorem 1 *Let $\mathbb{C}$ be a base category equipped with an associative tensor $\otimes$; and a functor $F : \mathbb{C} \to \mathbb{C}$ be equipped with the "synchronization" natural transformation $FX \otimes FY \xrightarrow{\mathsf{sync}_{X,Y}} F(X \otimes Y)$ compatible with associativity in $\mathbb{C}$. They induce a canonical associative tensor $\otimes$ on $\mathbf{Coalg}_F$.*

The final coalgebra carries an "inner associative tensor" $\|$ on Z, induced on the following left. It is associative in the sense of the diagram on the right.

$$\begin{array}{ccc} F(Z \otimes Z) & \to & FZ \\ {\scriptstyle \zeta\otimes\zeta = \mathsf{sync}_{Z,Z}\circ(\zeta\otimes_{\mathbb{C}}\zeta)}\uparrow & & \cong \uparrow{\scriptstyle \zeta} \\ Z \otimes Z & \to & Z \end{array} \qquad \begin{array}{ccccc} (Z \otimes Z) \otimes Z & \xrightarrow{\cong} & Z \otimes (Z \otimes Z) & \xrightarrow{Z\otimes\|} & Z \otimes Z \\ \|\otimes Z \downarrow & & & & \downarrow \| \\ Z \otimes Z & & \longrightarrow & & Z \end{array}$$

For such compositions we have the compositionality result (1) for free. □

When $F = \mathcal{P}_\omega(A \times \_)$ for which a coalgebra is a finitely-branching LTS, we can realize all of the ACP/CCS/CSP-style synchronizations by taking different sync.

The microcosm principle. The *microcosm principle* is exemplified by the sentence: "a monoid is defined in a monoidal category". What we saw above is one instance of such phenomena. We pursue a mathematical formulation of this principle for general algebraic theories. In its course we use 2-categorical notions since a 2-category ("categories in a category") well accommodates microcosm phenomena.

For our purpose, a *Lawvere theory* $\mathbb{L}$ is an appropriate categorical presentation of an algebraic theory. An $\mathbb{L}$-*category*—a category with the $\mathbb{L}$-structure—is a product-preserving pseudo-functor $\mathbb{L} \xrightarrow{\mathbb{C}} \mathbf{Cat}$. It is "pseudo" because equations hold only up to isomorphism. Now an object in $X \in \mathbb{C}$ which has the "inner" $\mathbb{L}$-structure is defined to be a lax natural transformation $\mathbb{L} \overset{\overset{\mathbf{1}}{\curvearrowright}}{\underset{\mathbb{C}}{\curvearrowright}} \mathbf{Cat}$. The component $\chi_1 : \mathbf{1} \to \mathbb{C}$ specifies the object X; the operations in $\mathbb{L}$ are interpreted by the mediating 2-cells of lax naturality.

In this general setting we can state the compositionality (1) as follows. It subsumes Theorem 1.

Theorem 2 *For an $\mathbb{L}$-category $\mathbb{C}$ and $F : \mathbb{C} \to \mathbb{C}$ being a lax $\mathbb{L}$-functor, the functor $\mathbf{Coalg}_F \xrightarrow{\mathsf{beh}} \mathbb{C}/Z$ as a morphism of $\mathbb{L}$-categories.* □

In [2] we also present this framework in less categorical terms, presenting an algebraic theory concretely by a pair (Σ, E) of operations and equations.

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. I. Hasuo, B. Jacobs, and A. Sokolova. The microcosm principle and concurrency in coalgebras, 2007. Preprint, available from `http://www.cs.ru.nl/~ichiro`.

A Coalgebraic View on Bi-Infinite Streams

Alexandra Silva

CWI, The Netherlands
ams@cwi.nl

Bi-infinite streams arise as a natural data structure in several contexts, such as signal processing [1], symbolic dynamics [2], (balanced) representation of real/rational numbers [3] or study of sets invariant under shift transformation [4].

In this paper, we will present a coalgebraic view of the set of bi-infinite streams which we shall denote by $A^{\mathbb{Z}}$ and is formally defined as

$$A^{\mathbb{Z}} = \{\sigma \mid \sigma : \mathbb{Z} \to A\}$$

We can easily prove that $A^{\mathbb{Z}} \cong (A \times A)^{\omega}$ and therefore the set $A^{\mathbb{Z}}$ is the final coalgebra for the functor $FX = (A \times A) \times X$.

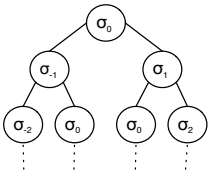
This reflects the fact that one can think about a bi-infinite stream denoted by $(\ldots, \sigma_{-2}, \sigma_{-1}, \underline{\sigma_0}, \sigma_1, \sigma_2, \ldots)$, as two infinite streams growing in parallel.

σ_0	σ_1	σ_2	$\ldots$
σ_{-1}	σ_{-2}	σ_{-3}	$\ldots$

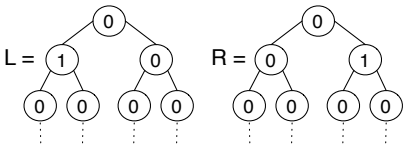
Using this observation, and defining a semiring structure on $A \times A$, we could reuse the calculus developed for streams [5] to deal with the bi-infinite case.

However, is this the only/best way to view bi-infinite streams coalgebraically? Will this reduction to the infinite case be too restrictive and not allow us to fully benefit from the structure of bi-infinite streams?

We shall now present another possible representation for bi-infinite streams. We can see $(\ldots, \sigma_{-2}, \sigma_{-1}, \underline{\sigma_0}, \sigma_1, \sigma_2, \ldots)$ as an infinite binary tree as follows:



The set T_A of infinite binary trees is the final coalgebra for the functor $GX = X \times A \times X$ and as showed in [6], by viewing trees as formal power series a very simple but surprisingly powerful coinductive calculus can be developed. In this framework, definitions are presented as behavioural differential equations and very compact closed formulae can be deduced for (rational) trees. For instance, in this framework the bi-infinite stream $(\ldots, 0, 1, 0, 1, \underline{0}, 1, 0, 1, 0, \ldots)$ would be represented by the formula $(L + R)(1 + (L - R)^2)^{-1}$, where L and R represent the following constant trees.



Note that not all $t \in T_A$ are representations of bi-infinite streams. However, we can prove that the subset of T_A containing valid representations of bi-infinite streams is a subcoalgebra of T_A and therefore, the existing calculus can be used to reason about bi-infinite streams.

Further questions still remain to be answered. We would like to classify the closed formulae that we have for trees in such a way that from its syntax could immediately be deduced if it is a valid representation of a bi-infinite stream.

We would also like to further exploit a specific class of bi-infinite streams, the ones which correspond to finite-tailed Laurent series and see if they give rise to a different type of coalgebra/calculus.

References

1. M. Barnabei, C. Guerrini, and L. B. Montefusco. Some algebraic aspects of signal processing. *Linear Algebra and Its Applications*, 284(1-3):3–17, 1998.
2. P. Collins. Dynamics of surface diffeomorphisms relative to homoclinic and heteroclinic orbits. *Dynamical Systems*, 19(1):1–39, 2004.
3. K. Culik and J. Kari. On aperiodic sets of wang tiles. In *Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday*, pages 153–162, London, UK, 1997. Springer-Verlag.
4. D. Perrin and J. Éric Pin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004. ISBN 0-12-532111-2.
5. J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.
6. A. Silva and J. Rutten. Behavioural differential equations and coinduction for binary trees. In *Proceedings of the 14th Workshop on Logic, Language, Information and Computation (WoLLIC'2007)*. Springer-Verlag, 2007. To appear.

Categorical Design Patterns

Ondrej Rypacek

School of CS&IT, University of Nottingham, Nottingham, NG8 1BB, UK
oxr@cs.nott.ac.uk

1 Introduction

Design patterns [1] in object-oriented programming (OO programming) describe good software design practise. A design pattern is a systematic yet informal description of a recurring design problem with its solution. Unfortunately, most design patterns are not formalisable in OO programming languages and therefore have to be implemented over and over again. As a result, their abstract structure is partially lost. In the following text we outline an approach to semantics of design patterns in the coalgebraic setting of Category Theory. We observe that many of the fundamental and most common patterns are encodings of primitive mathematical notions, which can be directly formalised as language features. Others are instances and combinations of the fundamental ones. This should ultimately give rise to an algebra of OO design with well founded mathematical semantics and provable properties of the programs constructed.

2 Products, Sums and Exponentials

We consider functional objects with local state in a suitable category with enough structure. Here, object implementations correspond to coalgebras [2, 3] and carriers of final coalgebras correspond to abstract object types.

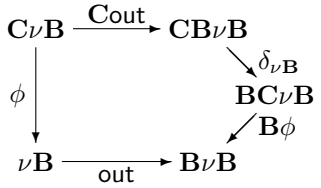
Object-oriented programming is based on the notion of *object* and mainstream OO languages lack separate notions of *products*, *exponentials* and *sums*. These are instead inherent in the complex notion of *object* and the purpose of some design patterns is to recover them via encodings. For instance, sums are defined by the *Visitor* pattern, which essentially describes the impredicative encoding of sums as found in System-F [4]. Likewise for products and exponentials (*Command* pattern). We argue that formal products, sums and exponentials are good formalisations of the essence of the above-mentioned patterns and make good candidates for becoming primitive in an OO calculus.

The idea of impredicative encoding can be generalised to polynomial functors to define arbitrary initial algebras conservatively within the OO setting as limits of large functors [5]. This defines algebraic datatypes in object oriented programming and sets the scene for comparison of functional datatypes and induction (algebras) on the one hand and objects (coalgebras) on the other.

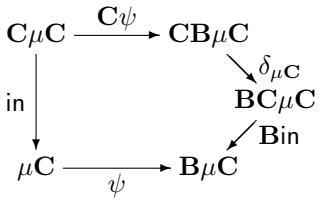
3 Composites

Algebraic datatypes and catamorphisms (generic *folds*) [6] play a central role in functional programming as they allow one to define recursive tree-like data structures with functions defined uniformly on them. Likewise, instances of the *Composite* pattern, roughly speaking, recursive hierarchies of objects with the same interface but different implementations, play a central role in OO programming as they model essentially the same.

Formally, the composite pattern can be defined coalgebraically as follows. We formalise the shape of an OO composite by a polynomial functor $\mathbf{C}$. Given a behaviour functor $\mathbf{B}$, which captures the interface of each object in the composite, we observe that a composite is defined by a natural transformation $\delta : \mathbf{CB} \to \mathbf{BC}$ in the following situation.



The universal arrow ϕ into the terminal coalgebra defines the joined constructor of the various kinds of objects in the composite structure. Alternatively, we may consider putting the behaviour $\mathbf{B}$ directly on the algebraic datatype, $\mu\mathbf{C}$. This gives rise to the following situation.



Here, the universal arrow ψ defines an implementation of behaviour $\mathbf{B}$ on structures with shape $\mathbf{C}$. We observe the exact situation occurs in categorical semantics as the *adequacy of operational and denotational semantics* [7]. We conclude that these two approaches to defining functions on recursive structures are essentially the same.

4 Conclusion and Future Work

Our work makes a contribution to formalisation of OO programming and its relation to functional programming. We formalise the informal *Composite* pattern. We define the notion of *object structure* with a common interface and formalise the relation of OO *composites* to functional traversals of datatypes. We observe that the relation of functional and OO programming is similar to the relation of denotational and operational semantics and adopt the notion of adequacy to

formalise this observation. This allows us to view OO composites as catamorphisms and vice versa. We believe this also opens new possibilities in reasoning about the so-called expression problem [8].

In the future, we want to generalise the approach beyond polynomial functors to cover structures corresponding to nested datatypes and possibly indexed families of datatypes. This would allow us to capture formally a much wider range of OO programs defined as hierarchies of nonuniform objects.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Reichel, H.: An approach to object semantics based on terminal co-algebras. Mathematical Structures in Computer Science **5**(2) (1995) 129–152
3. Jacobs, B.P.F.: Objects and classes, coalgebraically. In Freitag, B., Jones, C.B., Lengauer, C., Schek, H.J., eds.: Object-Orientation with Parallelism and Persistence. Kluwer Academic Publishers, Boston (1996) 83–103
4. Buchlovsky, P., Thielecke, H.: A type-theoretic reconstruction of the visitor pattern. In: 21st Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). Electronic Notes in Theoretical Computer Science (ENTCS) (2005)
5. Neil Ghani, T.U., Vene, V.: Build, augment, destroy. universally. In: Proceedings of Programming Languages and Systems: Second Asian Symposium, APLAS, 2004. Volume 3302 of Lecture Notes in Computer Science., Springer Verlag (2004) 327–341
6. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Proceedings of the 5th ACM conference on Functional programming languages and computer architecture, New York, NY, USA, Springer-Verlag New York, Inc. (1991) 124–144
7. Turi, D., Plotkin, G.D.: 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. IEEE Computer Society Press, Los Alamitos, CA (1997) 280–291
8. Cook, W.R.: Object-oriented programming versus abstract data types. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, London, UK, Springer-Verlag (1991) 151–178

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

Tomoyuki Suzuki

Japan Advanced Institute of Science and Technology,
1-1 Asahidai Nomi Ishikawa, Japan
tsuzuki@jaist.ac.jp

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 [4]).

In modal logic, however, relational semantics introduced by Kripke are also attractive with their intuitive character and connection with applicative 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's duality.

In author's Master's Thesis [7], a relational semantics for a large class of substructural logics: distributive substructural logics (DFL logics) was introduced via Stone's duality. These logics are including well studied logics like 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 obtained can be summed up as follows:

- For all basic extensions of DFL, we identified corresponding frame conditions and proved completeness results.
- We extended Stone's duality to duality between DFL-algebras and DFL-frames.
- We have obtained general completeness result: every DFL logic is complete with respect to a class of descriptive frames.
- We have studied the categorical duality between DFL-algebras and descriptive frames.
- Finally, we have found the topological characterization of descriptive frames. This is a natural generalization of similar characterization for intuitionistic [2] or relevance [6] frames: differentiation, tightness and compactness.

Distinct points of our approach from other authors, like [3] or [5], are the following:

- Well-researched relational semantics, for example, for relevance logics or superintuitionistic logics, can be thought of as specializations of our semantics.
- Our relational semantics consist of just one underlying set and just one ternary relation.
- Moreover, the single ternary relation provides an interpretation for almost all connectives, that is, $\vee$, $\wedge$, $\circ$, $\backslash$ and $/$.

Through our relational semantics, we have obtained new approaches for research of substructural logics. For example, many modal techniques (eg. [1] or [2]) will be applicable in substructural logics.

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. A. Chagrov and M. Zakharyaschev. *Modal Logic*, volume 35 of *Oxford Logic Guides*. Oxford Science Publications, 1997.
3. N. Galatos. *Varieties of residuated lattices*. PhD thesis, Graduate School of Vanderbilt University, May 2003.
4. 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.
5. M. Gehrke, H. Nagahashi, and Y. Venema. A Sahlqvist theorem for distributive modal logic. *Annals of Pure and Applied Logic*, 131:65-102, 2005.
6. T. Seki. General frames for relevant modal logics. *Notre Dame Journal of Formal Logic*, 44:93-109, 2003.
7. T. Suzuki. Kripke completeness of some distributive substructural logics. Master's thesis, Japan Advanced Institute of Science and Technology, March 2007.

Modal Logic on Topological Coalgebras

Levan Uridia

Institute for Logic, Language and Computation, University of Amsterdam,
Plantage Muidergracht 24, NL-1018 TV Amsterdam, The Netherlands
luridia@science.uva.nl

In this paper we consider topological spaces and open-continuous maps from a coalgebraic perspective. Our main result is that the coalgebraic modal language, defined in the style of Moss [2] has the same expressive power as the basic modal language with it's standard topological interpretation.

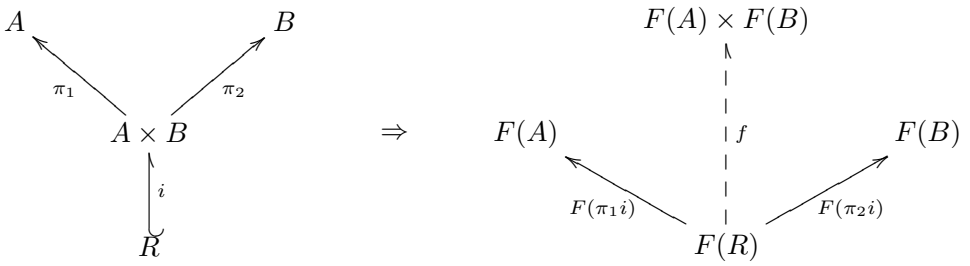
We will consider topological spaces for the following so-called filter functor defined in [4].

Definition 1 *Let $f : A \to B$ be a map and $\mathcal{F}$ be a filter over A, then by $f(\mathcal{F})$ we denote the following family of subsets of B, $f(\mathcal{F}) = \{f(W) | W \in \mathcal{F}\}$.*

Let $\uparrow f(\mathcal{F})$ denote the filter generated by $f(\mathcal{F})$, so $\uparrow f(\mathcal{F}) = \{V |$ there is $U \in f(\mathcal{F})$ such that $U \subseteq V\}$.

Definition 2 *Define the filter functor $\Phi : Set \longrightarrow Set$, by putting $\Phi(A) = \{\mathcal{F} | \mathcal{F}$ is a filter over $A\}$ and for a morphism $f : A \to B$ let $\Phi(f) : \Phi(A) \to \Phi(B)$ be the function associating with every filter $\mathcal{F} \in \Phi(A)$, the filter $\uparrow f(\mathcal{F})$.*

Definition 3 *Given a set functor F, the relation lifting $\overline{F}(R)$ of R is defined as the image of $F(R)$ under the unique map from $F(R)$ to $F(A) \times F(B)$ given as the pair $(F(\pi_1 i), F(\pi_2 i))$, see the diagrams below:*



Proposition 4 *For a given relation $R \subseteq A \times B$, a pair of filters $(\mathcal{F}_1, \mathcal{F}_2)$ (over A and B respectively) belongs to the relation lifting $\overline{\Phi}(R)$ of R iff*

$$(\forall U \in \mathcal{F}_1)(\exists U_0 \in \mathcal{F}_1)(\exists V \in \mathcal{F}_2)(U_0 \subseteq U \wedge (U_0, V) \in \overline{P}(R)) \tag{1}$$

and

$$(\forall V \in \mathcal{F}_2)(\exists V_0 \in \mathcal{F}_2)(\exists U \in \mathcal{F}_1)(V_0 \subseteq V \wedge (U, V_0) \in \overline{P}(R)) \tag{2}$$

where $\overline{P}(R)$ denotes the relation lifting for the power set functor ($(U, V) \in \overline{P}(R)$ iff $(\forall u \in U, \exists v \in V)(uRv) \& (\forall v \in V, \exists u \in U)(uRv)$).

Definition 5 (Gumm [4]) *A Φ-coalgebra (X, τ) is topological iff for every element $x \in X$ and every $U \subseteq X$ we have: $U \in \tau(x)$ implies, that there exists a subcoalgebra $(X_0, \tau_0) \leq (X, \tau)$ such that $x \in X_0 \subseteq U$.*

Fact 6 *There is a one-to-one correspondence between topological Φ-coalgebras and topological spaces.*

We give the general definition of finitary coalgebraic modal language and describe how coalgebras serve as semantical structures for this language.

Definition 7 (Venema[3]) *For a given set functor F, finitary F-coalgebraic modal language Λ_F is defined inductively as follows: $\phi ::= p|\neg\phi|\phi \vee \phi|\nabla_F\Sigma$, where $\Sigma \in F(S)$ for some finite set S of formulas.*

Definition 8 *For a given functor $F : Set \to Set$ and a F-coalgebra (X, τ), a coalgebraic F-model based on (X, τ) is a pair (X, υ), where $\upsilon : X \to F(X) \times P(Prop)$ such that $\pi_1\upsilon = \tau$ for all $x \in X$.*

The semantics for Λ_F is given as follows.

Definition 9 *We define the truth of formula at a point $x \in X$ in the F-coalgebraic model (X, υ) inductively as follows:*
$x \Vdash p$ *iff* $p \in \pi_2\upsilon(x)$,
$x \Vdash \neg\alpha$ *iff* $x \nVdash \alpha$,
$x \Vdash \alpha \vee \beta$ *iff* $x \Vdash \alpha$ *or* $x \Vdash \beta$,
$x \Vdash \nabla_F(\Sigma)$ *iff* $(\pi_1\upsilon_\tau(x), \Sigma) \in \overline{F}(\Vdash)$.

The semantics of the standard modal language on the topological spaces is defined as follows.

Definition 10 *The truth definition of the modal formula at a point $x \in X$ in the topological model (X, Ω, V) is given in the following way:*
$x \Vdash p$ *iff* $p \in V(x)$,
$x \Vdash \neg\alpha$ *iff* $x \nVdash \alpha$,
$x \Vdash \alpha \vee \beta$ *iff* $x \Vdash \alpha$ *or* $x \Vdash \beta$,
$x \Vdash \Diamond\alpha$ *iff* $x \in Cl(\{y | y \Vdash \alpha\})$.

Theorem 11 *Over the class of all topological spaces, the language Λ_Φ is equivalent to standard modal language.*

References

1. J. McKinsey, A. Tarski, The algebra of topology, Annals of Mathematics 45 (1944) 141-191.
2. L. Moss, Coalgebraic Logic, Annals of Pure and Applied Logic 96 (1999) 277-317 (Erratum published APAL 99:241-259, 1999).
3. Y. Venema, Automata and fixed point logic: A coalgebraic perspective, Information and Computation 204 (2006) 637-678.
4. H. P. Gumm, Functors for Coalgebras, Algebra Universalis 45 (2001) 135-147.

Limits and Colimits in Categories of Institutions

Adam Warski

Institute of Informatics, University of Warsaw, Poland
adam.warski@students.mimuw.edu.pl

The theory of institutions, first introduced by Goguen and Burstall in 1984 ([4, 3]), 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. Examples can be found in papers by Sannella and Tarlecki ([13], [12, chapters 4 and above]), Diaconescu ([1]), Mossakowski ([7, 8]), and many others.

There are two main ways of moving between institutions, using either institution morphisms or comorphisms (which were first introduced under the name "simple maps of institutions" by Meseguer in [6], and then renamed to representations by Tarlecki in [15]). 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 show that there is some duality between the two concepts. A very thorough and systematic paper dealing with various properties of (co)morphisms is [5].

In my work, I am going to analyse the relationships between limits and colimits of diagrams built from institutions linked by morphisms and comorphisms. As mentioned above, morphisms and comorphisms may seem as dual concepts at first. However, intuitively similar universal constructions associated with morphisms and comorphisms, turn out to be rather different.

The main motivation behind this work takes source in heterogeneous specifications [9, 16], 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; thus the need to translate one into another. Also, given such a diagram, it may be useful to represent a family of models of a heterogeneous distributed specification, or specifications themselves in a (co)limiting institution. Limits/colimits of institutions haven't proved to be the best tool for "putting institutions together" (see for example [2, 11]), however it may be suitable to use them as concise representations of whole diagrams of institutions. This approach is different from the one taken by, for example, Mossakowski ([8, 10]) and Diaconescu ([1]), where, over a diagram, a corresponding Grothendieck institution is built. Using this technique, institutions are put into one, essentially "side by side", without much interaction. Considering (co)limits, we are after a more compact representation, where some combination of signatures, models and sentences of the institutions involved takes place.

Taking morphisms or comorphism, we can build two categories: **INS** and **coINS**, with institutions as objects. It has been proved long ago ([14]) that

the category **INS** is complete. It can be quite easily seen and proved, that the construction of products and equalizers (and hence limits of arbitrary diagrams) works in a "component-by-component" manner—that is, independently on each of the 3 parts of a morphism (namely, a functor between signature categories and natural transformations between model and sentence functors). The construction of limits in **coINS** is almost the same as above, and this category is also complete.

The case of colimits is not as straightforward. First of all, it turns out that both categories (**INS** and **coINS**) are not cocomplete. This, however, is due to purely set-theoretical problems with the size of components of the institutions that would be the colimit. Restricting attention to institutions, whose signatures form a small category, solves the problem. However, the construction here cannot be done for each component separately as before. The problem lies in defining the model and sentence functors on morphisms of signatures. To do this properly, when constructing a model category (or sentence set) for a given signature, we have to take into account all signatures, from which morphisms exists to the considered one. The definition of the satisfaction relation is also far from trivial. Firstly, I will explain in more detail the problem with constructing the colimits, and then show how the construction of arbitrary coproducts and coequalizers looks.

As the concepts of an institution morphism and comorphism are largely dual, a question arises if they can be represented by one another. One way to do that is by spans of (co)morphisms, as introduced for example in [9]. This enables us to represent a morphism by two comorphisms, each leading from an "intermediary" institution, which has signatures taken from the domain of the morphism, and model and sentence functors taken from the second one. This construction can be also done the other way round, to replace a comorphism with a span of morphisms.

Now comes the question: how do (co)limits of diagrams relate to (co)limits of diagrams built by replacing each (co)morphism by a span of (co)morphisms? Intuitions behind, for example, a limit in **INS** and a colimit in **coINS** may seem to some extent similar. I will try to answer the above question, show what difficulties arise when trying to construct a (co)morphism between (co)limiting institutions, when this is possible and when not. When changing morphisms into comorphisms, the shape of the diagram changes, hence the "procedure" for constructing (co)limits changes also; but because the new morphisms are of a special form, it can be sometimes simplified—I will show how. In general, there is, however, no simple and straightforward way to translate between limits/colimits of the two diagrams, which again shows that morphisms and comorphisms are not really dual. For example, considering a product of two institutions. As there are no morphisms, there is nothing to change. Now, if we look at the categories of signatures of the limiting and colimiting institutions, one is a product of categories, the other one a coproduct. The only reasonable functors that can be defined are projections from the product to the coproduct. However, this does not "capture" the whole of the limiting institution.

References

1. Razvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402, 2002.
2. Joseph Goguen and Rod Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*, pages 313–333, New York, NY, USA, 1986. Springer-Verlag New York, Inc.
3. Joseph Goguen and Rod Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, January 1992.
4. Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In Edmund M. Clarke and Dexter Kozen, editors, *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer, 1983.
5. Joseph A. Goguen and Grigore Rosu. Institution morphisms. *Formal Asp. Comput.*, 13(3-5):274–307, 2002.
6. Jose Meseguer. General logics. In *H.-D. Ebbinghaus et al., editors, Proceedings, Logic Colloquium 1987*, pages 275–329. North-Holland, 1989.
7. Till Mossakowski. *Representations, Hierarchies and Graphs of Institutions*. PhD thesis, Bremen University, 1996.
8. Till Mossakowski. Comorphism-based grothendieck logics. In *K. Diks and W. Rytter, editors, Mathematical Foundations of Computer Science, LNCS 2420*, pages 593–604. Springer, 2002.
9. Till Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, Lecture Notes in Computer Science, pages 359–375. Springer Verlag, London, 2003.
10. Till Mossakowski. Institutional 2-cells and grothendieck institutions. In *Essays Dedicated to Joseph A. Goguen*, pages 124–149, 2006.
11. Wieslaw Pawlowski. Context institutions. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *COMPASS/ADT*, volume 1130 of *Lecture Notes in Computer Science*, pages 436–457. Springer, 1995.
12. Donald Sannella and Andrzej Talecki. *Foundations of Algebraic Specifications and Formal Program Development*. To appear. 2007.
13. Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Inf. Comput.*, 76(2-3):165–210, 1988.
14. Andrzej Tarlecki. Bits and pieces of the theory of institutions. In David H. Pitt, Samson Abramsky, Axel Poigné, and David E. Rydeheard, editors, *CTCS*, volume 240 of *Lecture Notes in Computer Science*, pages 334–365. Springer, 1985.
15. Andrzej Tarlecki. Moving between logical systems. In M. Haveraaen, O.-J. Dahl, and O. Owe, editors, *Recent Trends in Data Type Specifications. Selected Papers, 11th Workshop on Specification of Abstract Data Types ADT'95*, LNCS 1130, pages 478–502. Springer, 1996.
16. 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.

Layered Completeness

Daniel Găina

Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan,
daniel@jaist.ac.jp

We present a three layered completeness result, for the logics/institutions having the sentences of the form $(\forall X)e$, where e is a sentence constructed from the atoms of the given logic by means of Boolean connectives. Our study isolates the particular aspects of the logics from the general ones in order to obtain an abstract completeness result in three steps.

Firstly, we identify a system of proof rules for the atomic sentences which constitute the bricks for building sentences in concrete institutions. The results obtained are institution-dependent, and can not be formulated at the abstract level.

Secondly, we give generic rules that deal with Boolean connectives and prove, institution-independent, the soundness and completeness of the system of proof rules obtained by adding the specific rules to the general ones. For this we use the forcing techniques, a method of construction of models satisfying some properties by means of consistency results.

One important contribution is the introduction of the notion of forcing property in the institution-independent model theory. The forcing construction in set theory was introduced by Cohen [3], has led to the solution of many classical problems by means of consistency results. A. Robinson [5] developed an analogous theory of forcing in model theory, and Barwise extended Robinson's theory to infinitary logic and used it to give a new proof of Omitting Types Theorem. We show that this technique is also suitable for this framework. In the case of institutions admitting also quantifications the definition of forcing property is the same and the definition of forcing relation and all results regarding generic models are naturally extended.

Thirdly, we give proof rules that deal with universal quantifications and prove a completeness result for the institutions having the sentences constructed from a class of quantifier-free sentences by means of universal quantification over a class of signature morphisms. This class of sentences may be either all the sentences constructed from the atoms of the institution by means of Boolean connectives, or all the sentences of the form $H \Rightarrow C$, where H is a finite conjunction of atoms and C is an atom of the given logical system. In [2] is considered the second case.

There are several aspects that motivate and justify our study. One of them is the importance for model theory. The completeness results may be obtained in the same way as sentences are constructed. In our case, starting from the atoms of the institution under investigation, the results are institution-dependent. At the opposite side, for the sentences constructed from the atoms by means of

Boolean connectives, the proof rules and the results are institution-independent. The proof rules that deal with universal quantified sentences, namely the Substitutivity rules, are given at the abstract level and the completeness for the system of proof rules obtained by adding the Substitutivity rules to the proof rules given for the second layer is institution-independent.

This result has also great significance for computer science. Modern specification languages (such as CafeOBJ [4], CASL [1], Maude) are rigorously based on logic, in the sense that each feature and construct in a language can be expressed within a certain logic underlying it. Completeness results are essential for an operational semantics of executable specification languages. In the context of proliferation of a multitude of specification languages, these abstract results provide complete systems of proof rules for the logical systems underlying the languages.

The present contribution sets a general framework and incorporates many examples. It introduces new techniques, such as forcing technique, in institutional model theory and exploits many notions related to this field, yielding to a very general completeness result which shows the connection between the structure of the sentences and the structure of the proof of completeness.

References

1. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theor. Comp. Sci.*, 286(2):153–196, 2002.
2. M. Codescu and D. Găina. Birkhoff completeness in institutions. Submitted.
3. P. J. Cohen. The independence of the continuum hypothesis. In *Proc. of the National Academy of Sciences of the U.S.A.*, volume 50, pages 1143–1148, 1963.
4. R. Diaconescu and K. Futatsugi. Logical Foundations of CafeOBJ. *Theor. Comp. Sci.*, 285(2):289–318, 2002.
5. A. Robinson. Forcing in model theory. *Symposia Mathematica*, 5:69–82, 1971.

Software Development and Categories

Magne Haveraaen[1] and Adis Hodzic[2]

[1] Department of Informatics, University of Bergen, PB.7800, N-5020 Bergen, Norway
[2] Faculty of Engineering, Bergen University College, PB.7030, N-5020 Bergen, Norway
Magne.Haveraaen@ii.uib.no, Adis.Hodzic@hib.no

One of the uses of algebraic specifications is to specify, verify and understand programs. This contribution presents a general framework enabling the use of programs as model categories in institutions. It shows how, given any programming language (over an arbitrary signature) and any compiler (model algebra), we have categorical constructions appropriate for representing and reasoning about programs written in the language. The constructions are quite general and are not restricted to a specific programming paradigm.

We show how we can represent concepts such as data structures, data invariants and equivalence relations (equality). Together with the notion of encapsulation, we get full data abstraction, taking us from a program category with only products and a (non-trivial) sum object, to a complete and cocomplete category for the same programming language constructs and base signature.

The concepts can be considered at the purely syntactic level (generated from the programming language), or at the semantical level (as interpreted by the compiler), or at any intermediate level, as given by algebraic specifications of the base signature. This allows us to control the legal interpretations (compiler correctness) of the signature using standard algebraic techniques.

In addition, we will show how quantitative properties of programs, such as runtime and memory consumption, are captured in this setting.

Seen together, this gives us a basis for reasoning about meaning (semantics) and properties (complexity) of programs at a mathematically sound and highly abstract level.

The work presented here is illustrated through two simple programming languages, showing how functional and imperative programs can be treated within the framework.

Integrating Theorem Proving for Processes and Data

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

[1] University of Wales Swansea, United Kingdom
[2] AIST, Tsukuba, Japan
{csliam,m.roggenbach}@swansea.ac.uk, y-isobe@aist.go.jp

Distributed applications such as flight booking systems, web services, electronic payment systems such as the EP2 standard [1], require parallel processing of data. Consequently, such 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 [12], we designed the language Csp-Casl, which is tailored to the specification of distributed systems. Csp-Casl integrates the process algebra Csp [4, 13] with the algebraic specification language Casl [10, 2]. 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 [3] we applied Csp-Casl to the EP2 standard and demonstrated that Csp-Casl can deal with problems of industrial strength.

The combination of process algebra and algebraic specification raises various integration issues. In [12] we identified four basic integration problems, illustrated them by prototypical examples and showed how Csp-Casl copes with them. Here, we develop theorem proving support for Csp-Casl by translating Csp-Casl specifications into the input language of the already established tool Csp-Prover [5, 6, 8, 7]. Csp-Prover is based on the interactive theorem prover Isabelle [11]. Part of this translation is carried out by the tool HETS [9]. Fig. 1 shows the overall architectural concept for Csp-Casl-Prover.

Concerning tool support, the above introduced basic integration problems, which lead to challenges for integrated theorem proving. At the current state of our project, we solved these challenges for the prototypical examples stated in [12]. As simple as they are, they capture the very nature of the integration problem between processes and data. It turns out that a systematic analysis of these specifications leads to a set of automatically provable theorems. With these 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 rea-

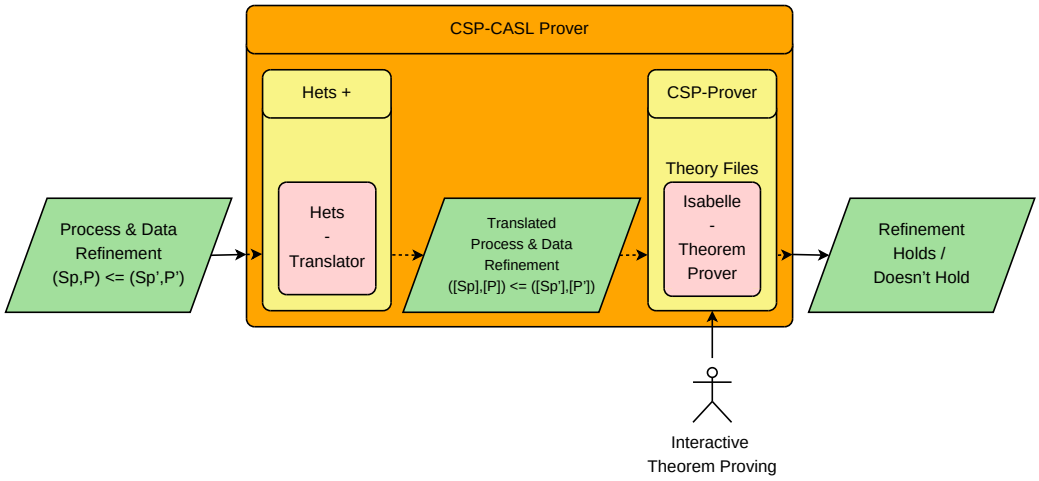


Fig. 1. Architecture of CSP-CASL-Prover.

soning on processes usually depends on theorems concerning data. This view is justified by theoretical results on the CSP-CASL semantics stated in [12]. Fig. 2 shows the prototypical structure of an Isabelle theory file.

The next steps in our project will be to gain more experience with our concept of integrated theorem proving, e.g. by analysing the CSP-CASL specifications of the EP2 system [3], and to implement CSP-CASL-Prover.

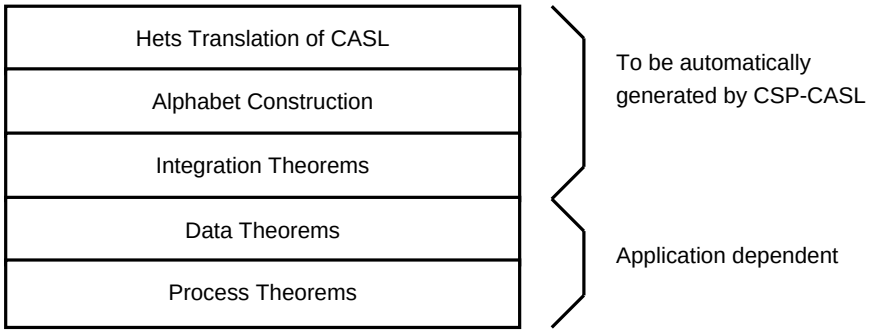


Fig. 2. Prototypical Structure of a Theory File.

References

1. *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

2. M. Bidoit and P. D. Mosses. CASL *User Manual*. LNCS 2900. Springer, 2004.
3. 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.
4. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
5. Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
6. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.
7. 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.
8. Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.
9. T. Mossakowski, C. Maeder, and K. Luettich. The heterogeneous tool set. In *TACAS 2007*, LNCS 4424. Springer, 2007.
10. P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960. Springer, 2004.
11. T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.
12. M. Roggenbach. CSP-CASL - a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.
13. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

Generalized Sketches for Model-driven Development

Adrian Rutle

Bergen University College, p.b. 7030, 5020 Bergen, Norway,
aru@hib.no

Abstract. The diversity and heterogeneity of modeling languages make the needs for formal model specifications and automatic model integration and transformation mechanisms more relevant than ever. These mechanisms are the corner stones in Model-driven Development, which is a natural evolutionary step in raising the abstraction level of programming languages. In this talk, we propose a generic formalism, Generalized Sketches, for specifying modeling languages and their transformations.

1 Model-driven Development (MDD)

MDD is a software development process in which modeling, transformations and automatization of model transformations are important issues. In MDD, an application is built by working at the model level. The process starts by specifying an abstract and formal (diagrammatic) model which is independent of the application's platform, i.e. the implementation technology, design, programming language ... etc. This kind of model is referred to as Platform Independent Model (PIM) [1]. In PIM, one can specify the business logic of the application without restriction to a special system design.

The next step in MDD consists of specifying a transformation 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 implementation technology and programming language; like OO-design or relational schemes.

The last step considers transforming the PSMs to application code. There are many tools that support this step, but existing tools only allow developers to choose among a predefined set of transformation definitions, for example, transforming UML class diagrams to Java, C++, SQL code ...etc.

The challenge in MDD is in finding a formalism for specifying the models and choosing mechanisms for definition of (and automatically execution of) transformations between those models.

2 Generalized Sketches (GS)

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 [2]. The claim behind GS is that any diagrammatic specification technique in

software engineering can be seen 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. A signature is an abstract structure consisting of a collection (or a graph) of predicate symbols with a mapping that assigns a shape (or an arity) to each predicate symbol. A $\Sigma - sketch$ is a graph with a set of diagrams labeled with predicates from the signature Σ [3]. Diagrams drawn using a specific specification technique, will appear as a (possibly ambiguous) visualization of a sketch which is parameterized by the corresponding signature Σ.

Thus we claim that GS can be used as a standard notation for representing both the syntax and the semantics of diagrammatic specification languages, as the syntax and in most cases also the semantics of GS is mathematically well-defined and unambiguous.

3 Generalized Sketches and MDD

As mentioned above, GS can be used to specify modeling languages and transformations between them. Since GS is a generic specification format, it can be used to specify PIMs, PSMs and the transformations between them. Also by regarding programming languages as modeling languages, one can use the following generic mechanism for transformation between PIMs, PSMs and code.

Models M that are specified by a given modeling language ML must conform to the metamodel MM of ML. MM is considered as a specification technique which corresponds to a (graphical) signature Σ_{ML} in GS, i.e. $MM \cong \Sigma_{ML}$ [4]. Having abstract definitions of signatures (or metamodels,) say MM_1 and MM_2, a relationship or transformation between them can be defined as a morphism $rel : MM_2 \rightarrow MM_1$ from the target metamodel MM_2 to the source metamodel MM_1 (see the figure.) Then any model M_1 conforming to (i.e. which is an instance of) MM_1 can be transformed automatically to a model M_2 conforming to MM_2 by computing the pullback (M_2, i_2, rel^*) of the sink (MM_1, i_1, rel) [5]. The underlying category of the models and metamodels is GRAPH and thus the pullback exists. The application of the pullback construction opens for automatization of the transformation as it's needed by MDD [6].

$$\begin{array}{ccc} M_1 & \xleftarrow{rel^*} & M_2 \\ {\scriptstyle i_1}\downarrow & & \downarrow{\scriptstyle i_2} \\ MM_1 & \xleftarrow{rel} & MM_2 \end{array}$$

4 Tools

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 MDD from transformation and integration to decomposition and code-generation.

The transformation definition in most existing transformation tools is composed of a set of transformation rules that define how elements or constructs from a source model can be transformed to a target model [1]. These rules are restricted to element-wise transformations and in the best case to binary relations between elements. While in the GS methodology, a transformation is a morphism capable of transforming structures and relationships spanning over many (meta)model elements. Other drawbacks of the methodologies used today are that transformation rules are only based on heuristics –they must be defined and hard-coded for each two metamodels, and you are not guarantied the existence of compositionality and associativity between rules.

Our tool will be used to design signatures corresponding to existing specification techniques, like UML class diagrams and ER diagrams. Designing signatures for existing modeling languages (the so-called "sketching" or "formalizing" in [3]) involves exhausting exploration of the syntax and semantics of those languages to find the adequate set of predicates; like total, partial, jointly mono, disjoint-cover ... etc needed to express all properties that can be expressed by them. Then, preferred graphical notations for the predicates can be chosen. (This step corresponds to the specification of MM1 and MM2 in the figure.) Diagrams, i.e. visualizations of sketches, can be drawn using the signatures/specification techniques (this step corresponds to the specification of M1 and M2 in the figure.) The tool will also support definition of transformation between (meta)models and automatic construction of pullback.

References

1. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: practice and promise.* Addison-Wesley, 1 edition, April 2003.
2. Uwe Wolter and Zinovy Diskin. Generalized sketches: A universal logic for diagrammatic modeling in software engineering. Technical report, University og Bergen, Norway and School of Computing, Kingston, Canada, 2007. To appear in Proceedings ACCAT.
3. Zinovy Diskin and Boris Kadish. 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. Uwe Wolter and Zinovy Diskin. The next hundred diagrammatic specification techniques a gentle introduction to generalized sketches. Technical report, University og Bergen, Norway and School of Computing, Kingston, Canada, 2006.
5. Zinovy Diskin. Model transformation via pull-backs: algebra vs. heuristics. Technical Report 521, School of Computing, Kingston, Canada, september 2006.
6. Zinovy Diskin and Juergen Dingel. A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. Technical report, 2006. 3rd Int.Workshop on Metamodels, Schemas, Grammas and Ontologies for reverse engineering, ATEM.

An Algebraic Structure for Concurrent Actions*

Cristian Prisacariu

Department of Informatics – University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
cristi@ifi.uio.no

Abstract

In [2] we have provided a formal language for specifying contracts, which allows to write (conditional) obligations, permissions and prohibitions of the different contract signatories, based on the so-called *ought-to-do* approach. In such an approach the above normative notions are specified over (names of human) *actions*, as for example "The client is obliged to pay after each delivery". There, we have given a formal semantics of the contract language in a variant of μ-calculus, but we have left the formalization of the underlying action algebra underspecified.

In this paper we introduce a new algebraic structure to provide a well-founded formal basis for the action-based contract language presented in [2]. Though the algebraic structure we define is somehow similar to Kleene algebra with tests [1], there are substantial differences due mainly to our application domain. A first difference is that we do not include the Kleene star as it is not needed in our context. A second difference is that we introduce an operator in the algebra to model true concurrency. The main contributions of the paper are: (1) A formalization of concurrent actions; (2) The introduction of a different kind of action negation; (3) A restricted notion of resource-awareness; and (4) A standard interpretation of the algebra over specially defined rooted trees.

The *algebra of concurrent actions and tests* ($\mathcal{CAT}$) that we present in this abstract is formed of an algebraic structure $\mathcal{CA} = (\mathcal{A}, +, \cdot, \&, \mathbf{0}, \mathbf{1})$ which defines the concurrent actions, and a Boolean algebra which defines the tests. Special care is taken when combining actions and tests under the different operators.

The algebraic structure $\mathcal{CA}$ is defined by a carrier set of elements (which we call *compound actions*, or just actions) denoted $\mathcal{A}$ and by the signature $\Sigma = \{\&, \cdot, +, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$ which gives the action operators and the *basic actions*. The non-constant functions of Σ are: $+$ for *choice* of two actions, $\cdot$ for *sequence* of actions (or concatenation), and $\&$ for *concurrent* composition of two actions. The constant function symbols of the finite set $\mathcal{A}_B \subseteq \mathcal{A}$ are called basic (atomic) actions. The special elements $\mathbf{1}$ and $\mathbf{0}$ are also constant function symbols. The set of basic actions is called the *generator set* of the algebra. In Table 1 we collect the axioms that define the structure $\mathcal{CA}$.

We want to have a resource-aware algebra similarly to what has been done for linear logic. Therefore we do not allow the idempotence property for the $\&$ operator ($a \& a \neq a$). As an example, if α represents the action of paying 100$

* Partially supported by the Nordunet3 project "Contract-Oriented Software Development for Internet Services".

(1)	$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$	(10)	$\alpha\&(\beta\&\gamma) = (\alpha\&\beta)\&\gamma$
(2)	$\alpha + \beta = \beta + \alpha$	(11)	$\alpha\&\beta = \beta\&\alpha$
(3)	$\alpha + \mathbf{0} = \mathbf{0} + \alpha = \alpha$	(12)	$\alpha\&\mathbf{1} = \mathbf{1}\&\alpha = \alpha$
(4)	$\alpha + \alpha = \alpha$	(13)	$\alpha\&\mathbf{0} = \mathbf{0}\&\alpha = \mathbf{0}$
(5)	$\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$	(14)	$\alpha\&(\beta + \gamma) = \alpha\&\beta + \alpha\&\gamma$
(6)	$\alpha \cdot \mathbf{1} = \mathbf{1} \cdot \alpha = \alpha$	(15)	$(\alpha + \beta)\&\gamma = \alpha\&\gamma + \beta\&\gamma$
(7)	$\alpha \cdot \mathbf{0} = \mathbf{0} \cdot \alpha = \mathbf{0}$	(16)	$\alpha\&(\alpha' \cdot \beta) = \alpha(1)\&\alpha'(1) \cdot \ldots \cdot \alpha(n)\&\alpha'(n) \cdot \beta$
(8)	$\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$		where $length(\alpha) = length(\alpha') = n$
(9)	$(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$		

Table 1. Axioms of $\mathcal{CA}$

then paying 200$ would be represented as $\alpha\&\alpha$. Note that we can represent only discrete quantities with this approach. We consider a *conflict relation* over the set of basic actions $\mathcal{A}_B$ (denote by $\#_{\mathcal{C}}$) defined as: $a \#_{\mathcal{C}} b \stackrel{def}{\iff} a\&b = \mathbf{0}$. The intuition of the conflict relation is that if two actions are in conflict then the actions cannot be executed concurrently.

The structure $\mathcal{CAT} = (\mathcal{CA}, \mathcal{B})$ combines the previous defined algebraic structure $\mathcal{CA}$ with a Boolean algebra $\mathcal{B}$ in a special way. A Boolean algebra is a structure $\mathcal{B} = (\mathcal{A}_\mathbf{1}, \vee, \wedge, \neg, \bot, \top)$ where the function symbols ($\vee$, $\wedge$, and $\neg$) and the constants ($\bot$ and $\top$) have the usual meaning. Moreover, the elements of set $\mathcal{A}_\mathbf{1}$ are called *tests* and are included in the set of actions of the $\mathcal{CA}$ algebra (i.e. tests are special actions; $\mathcal{A}_\mathbf{1} \subseteq \mathcal{A}$). We denote tests by letters from the end of the Greek alphabet $\phi, \varphi, \ldots$ followed by ?.

We give the standard interpretation of the actions of $\mathcal{A}$ by defining a homomorphism $I_{\mathcal{CAT}}$ which takes an action of the $\mathcal{CAT}$ algebra and returns a special guarded rooted tree preserving the structure of the action given by the constructors. A guarded rooted tree has labels (representing basic actions) on edges and tests as the types of the nodes. We define special operators on these trees: $\cup$ join, $\hat{}$ concatenation, and $\|$ concurrent join. In order to have the same behavior of $\mathbf{1}$ and $\mathbf{0}$ from $\mathcal{CAT}$ under the interpretation as trees we give a special procedure for *pruning* the trees.

For actions α defined with the operators $+$, $\cdot$, $\&$, and tests we have a canonical form denoted $\alpha!$ and defined as: $\alpha! = +_{\rho \in R}\ \rho \cdot \alpha'!$, where R contains either basic actions, concurrent actions, or tests, and α' is a compound action in canonical form. The action negation is denoted by $\overline{\alpha}$ and is defined as: $\overline{\alpha} = \overline{+_{\rho \in R} \rho \cdot \alpha'} = +_{b \in \overline{R}} b\ +\ +_{\rho \in R} \rho \cdot \overline{\alpha'}$, where ρ and α' are as before. The set $\overline{R}$ is defined to contain: $\{(\neg\phi)? \mid \phi \in R\} \cup \{\alpha \mid \alpha \in \mathcal{A}_\&$, and $\forall \beta \in R, \beta \not<_\& \alpha\}$ where $\mathcal{A}_\&$ contains concurrent actions generated only by means of $\&$, and $<_\&$ is a strict partial order which basically compares two actions to see which contains the other with respect to the $\&$ operator. Note that because $\&$ is not idempotent the set $\overline{R}$ becomes infinite (thus having infinite branching in the associated tree). We overcome this problem by defining *action schemas* and *tree schemas*.

In conclusion we mention some works which are close related to our work. J.J.Meyer '88 investigates algebraic properies of the actions he has in its Dynamic Deontic Logic. D.Kozen's extensive work on Kleene algebras forms a basis for our algebra. Our work goes well with Pratt's work on pomsets for true concurrency.

A nice introduction to rooted trees can be found in the work of M.Hennessy on algebraic theory of processes.

Acknowledgements

All the results of the paper have been obtained jointly with Gerardo Schneider.

References

1. D. Kozen. Kleene algebra with tests. *TOPLAS'97*, 19(3):427–443, 1997.
2. C. Prisacariu and G. Schneider. A formal language for electronic contracts. In M. Bonsangue and E. B. Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.

Church-Rosser for Borrowed Context Rewriting

Filippo Bonchi[1] and Tobias Heindel[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Institut für Informatik und interaktive Systeme, Universität Duisburg-Essen, Germany fibonchi@di.unipi.it,tobias.heindel@uni-due.de

Modelling distributed and mobile systems at a suitable level of abstraction maybe considered the main application area of process calculi and graph transformation systems. Analysis and verification methods for the resulting models abound. Here we focus on two lines of research: on the one hand the work following the influential theory of *Reactive Systems* (RS) [6] (originally developed for process calculi), and on the other hand the classical concurrency theory of the *double pushout approach* (DPO) to graph transformation[4, 7].

Recall the idea of the theory of RS: one derives from a given set of reaction rules a labelled transition system (LTS) such that the induced bisimulation relation is a congruence. This powerful technique has been adapted to DPO transformation over graphs [3] and even to rewriting in any adhesive category [8]. This generalization is known as DPO with *borrowed contexts* (DPOBC) and it is the main object of study in this paper.

The question is whether the natural notion of true concurrency of DPO rewriting, which is in contrast to the "interleaving only" semantics of process calculi, carries over to DPOBC. In other words, we set out to develop a DPO-style parallelism theory for DPOBC. Below we illustrate how borrowed context rewriting faithfully models the concurrency aspects of distributed and mobile systems. As a proof of concept we present the local Church-Rosser theorem for DPOBC.

A reader which is not familiar with DPOBC might skim the main ideas from the following model of an interactive system. We have only one reaction rule $(\circlearrowright\circ\to\circ) \leftarrowtail (\circ \quad \circ) \rightarrowtail (\circ\to\circ\circlearrowright)$, which models the dispatching of the message $\circlearrowright$ from one network node to the other using a channel of unit capacity between them. Now suppose we have the network $\oplus\leftrightarrows\odot$, consisting of two nodes $\oplus$ and $\odot$ which are connected by two complementary channels of unit capacity. However we do not want the channels themselves to be visible, but only the "access points" $\oplus$ and $\odot$. This system (state) is succinctly modelled by the inclusion $(\oplus \quad \odot) \rightarrowtail (\oplus\leftrightarrows\odot)$, which we also write as $\restriction^{\oplus\ \odot}_{\oplus\leftrightarrows\odot}$.

Now the LTS automatically derived using the borrowed context technique contains for example the following two transitions

$$\restriction^{\oplus\ \odot}_{\circlearrowright\oplus\leftrightarrows\odot} \xleftarrow{(\oplus\ \odot)\rightarrowtail(\oplus\ \odot\circlearrowright)\leftarrowtail(\oplus\ \odot)} \restriction^{\oplus\ \odot}_{\oplus\leftrightarrows\odot} \xrightarrow{(\oplus\ \odot)\rightarrowtail(\circlearrowright\oplus\ \odot)\leftarrowtail(\oplus\ \odot)} \restriction^{\oplus\ \odot}_{\oplus\leftrightarrows\odot\circlearrowright}$$

which correspond to the fact that the system can make transitions if the *environment* supplies messages, namely $\odot\circlearrowright$ or $\circlearrowright\oplus$. Further these two transitions are *independent* of each other and actually they form the first two sides of a local Church-Rosser square which is closed as follows.

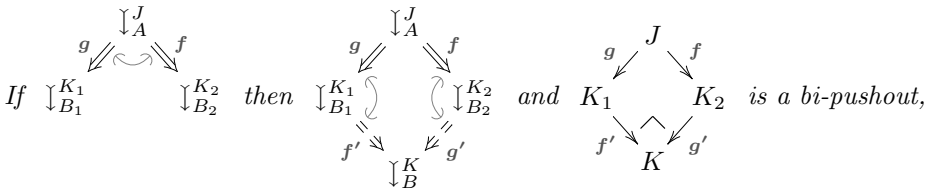
$$\restriction^{\oplus\ \odot}_{\circlearrowright\oplus\leftrightarrows\odot} \xrightarrow{(\oplus\ \odot)\rightarrowtail(\circlearrowright\oplus\ \odot)\leftarrowtail(\oplus\ \odot)} \restriction^{\oplus\ \odot}_{\circlearrowright\oplus\leftrightarrows\odot\circlearrowright} \xleftarrow{(\oplus\ \odot)\rightarrowtail(\oplus\ \odot\circlearrowright)\leftarrowtail(\oplus\ \odot)} \restriction^{\oplus\ \odot}_{\oplus\leftrightarrows\odot\circlearrowright}$$

Finally note that the two messages also could be sent concurrently, as the two transmissions use two different channels.

In general, given a DPO grammar in an adhesive category $\mathbf{C}$, a state in the derived LTS is just a mono $m\colon J \rightarrowtail A$, where J is the *interface* of the system A. The labels of the LTS describe the minimal contexts that a system needs to interact with the environment via its interface. Formally they are arrows in the bi-category of co-spans over $\mathbf{C}$, which has the same objects as $\mathbf{C}$ and the morphisms between objects J and K are monic *co-spans* $J \overset{j}{\rightarrowtail} F \overset{k}{\leftarrowtail} K$. Summarizing, states in the LTS are monos $\downarrow_A^J$ and transition have the form $\downarrow_A^J \overset{f}{\Rightarrow} \downarrow_B^K$ where the label $f = J \overset{j}{\rightarrowtail} F \overset{k}{\leftarrowtail} K$ is an arrow from J to K in the co-span category over $\mathbf{C}$.

So far we have sketched just enough about the categorical background to be able to present (the crucial point of) the local Church-Rosser theorem.

Theorem 1 (Local Church-Rosser for DPOBC)



where $\leftrightarrows$ denotes the natural generalization of parallel and sequential independence known from DPO *rewriting; further the rightmost figure is a bi-pushout in the bi-category of co-spans over* $\mathbf{C}$.

On top of the fact that the labels of the Church-Rosser square actually describe a bi-pushout in the co-span bi-category, we further have a *parallel* step $\downarrow_A^J \overset{f' \circ g}{\Longrightarrow} \downarrow_B^K$ along the diagonal of the Church-Rosser square (see the authors' [2] for the details of parallel DPO rules).

Future work As an application one might want to develop unfolding based verification techniques for DPOBC rewriting generalizing or using the methods implemented in tools like [5]. Especially secrecy properties could be handled naturally since DPOBC systems come equipped with a "built in" notion of visibility. Moreover the additional information of the interfaces might prove useful for abstraction refinement techniques.

On the theoretical side, we would like to justify our claim that the bisimulation presented in [2] is "aware" or "respects" concurrency. We plan to do so by comparing it with history preserving bisimulation in the style of [1]. However the latter work depends on the process semantics of DPO grammars, whence the need for a theory of DPOBC processes arises.

Conclusion We have presented the local Church-Rosser theorem for DPOBC. It exemplifies how the classical parallelism theory known from graph transformation systems naturally carries over to borrowed context rewriting. Moreover this

generalized result is in harmony with the rich bi-categorical structure of the labels of the automatically derived LTSs. In any case it can serve as a starting point for future studies of true concurrency in DPOBC models of interactive systems.

References

1. Paolo Baldan, Andrea Corradini, and Ugo Montanari. Bisimulation equivalences for graph grammars. In Wilfried Brauer, Hartmut Ehrig, Juhani Karhumäki, and Arto Salomaa, editors, *Formal and Natural Computing*, volume 2300 of *Lecture Notes in Computer Science*, pages 158–190. Springer, 2002.
2. Filippo Bonchi and Tobias Heindel. Adhesive DPO parallelism for monic matches. In *Graph Transformation for Verification and Concurrency, GT-VC2006*.
3. Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2004.
4. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
5. Barbara König and Vitali Kozioura. AUGUR – a tool for the analysis of graph transformation systems. *Bulletin of the EATCS*, 87:126–137, 2005.
6. James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2000.
7. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
8. Vladimiro Sassone and Pawel Sobocinski. Reactive systems over cospans. In *LICS*, pages 311–320. IEEE Computer Society, 2005.

Author Index