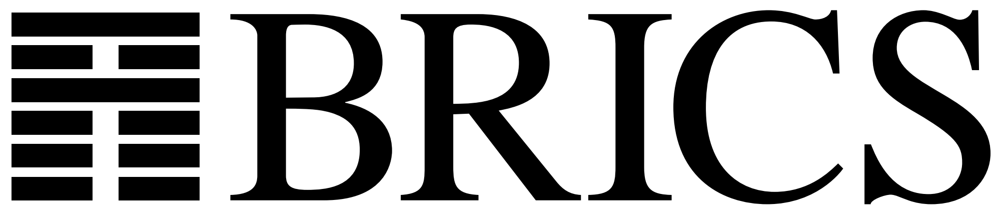


Basic Research in Computer Science

BRICS NS-02-2 Ésik & Ingólfsdóttir (eds.): FICS '02 Preliminary Proceedings

Preliminary Proceedings of the Workshop on

Fixed Points in Computer Science

Copenhagen, Denmark, July 20 and 21, 2002

Zoltán Ésik
Anna Ingólfsdóttir
(editors)

BRICS Notes Series NS-02-2

ISSN 0909-3206 June 2002

Copyright © 2002, Zoltán Ésik & Anna Ingólfsdóttir (editors).
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/02/2/`

Fixed Points in Computer Science 2002

(FICS'02)

Preliminary Proceedings

July 20 and 21, 2002

Copenhagen, Denmark

Content

1. Glynn Winskel, U Cambridge, UK (Invited talk) *Calculus for categories* 1
2. Jirí Adámek, Stefan Milius, and Jiri Velebil; Techn. U of Braunschweig, Germany *Parametric corecursion and completely iterative monads* 2
3. Neil Ghani, U Leicester, UK; Christoph Lüth, U Bremen, Germany; and Frederico De Marchi, U Leicester, UK *Coalgebraic approaches to algebraic terms* 6
4. Tarmo Uustalu, Tallinn Techn. U, Estonia *Generalizing substitution* 9
5. Nick Benton, Microsoft Research Cambridge, UK and Martin Hyland, U Cambridge, UK *Traced pre-monoidal categories* 12
6. Luca Aceto, Aalborg U, Denmark (Invited talk) *Kleene through the process algebraic glass* 20
7. Christoph Sprenger, Swedish Inst. of Computer Science, Sweden and Mads Dam, Royal Inst. of Techn., Sweden *A note on global induction in a mu-calculus with explicit approximations* 22
8. Nikolay Vyacheslavovich Shilov and Natalia Olegovna Garanina, IIS/RAS Novosibirsk, Russia *Model checking knowledge and fixpoints* 25
9. Benet Devereux, U Toronto, Canada *Strong next-time operators for multiple-valued mu-calculus* 40
10. Dexter Kozen, Cornell U, USA (Invited talk) *On two letters versus three* 44
11. Hans Leiss, U Munich, Germany *Kleenean semimodules and linear languages* 51
12. Guo-Qiang Zhang, Case Western Reserve U, USA *Decidable fragments of domain mu-calculus: an automata-theoretic perspective* 54
13. Margareta V. Korovina, U Aarhus, Denmark *Fixed points on abstract structures without the equality test* 58
14. Gerard Boudol and Pascal Zimmer, INRIA Sophia-Antipolis, France *Recursion in the call-by-value lambda;-calculus* 61
15. Anna Labella, U Rome I (La Sapienza), Italy (Invited talk) *Kleene's (unary) star in nondeterministic context* 67
16. Thomas Jensen, Florimond Ployette, and Olivier Ridoux; IRISA Rennes, France *Iteration schemes for fixed point calculation* 69
17. Luigi Santocanale, LaBRI Bordeaux, France *Congruences of modal μ-algebras* 77

Calculus for categories

Glynn Winskel
U Cambridge, UK

This talk will present and motivate a calculus for a fragment of category theory. Its judgements systematise such categorical arguments as that an expression is functorial in its free variables and that two functorial expressions are naturally isomorphic in their free variables. The calculus is particularly useful for showing the continuity (i.e., limit and colimit preserving properties) of functors and partly arose from work in a form of domain theory where domains are (special kinds of) categories. The power of the calculus derives from its liberal use of ends and coends. The talk will only assume a nodding acquaintance with category theory and, in particular, will try to give an accessible introduction to ends and coends. (The talk is based on joint work with Mario Caccamo—see BRICS report RS-01-27 and the Lecture Notes in Category Theory at www.brics.dk/ mcaccamo .)

PARAMETRIC CORECURSION AND COMPLETELY ITERATIVE MONADS

JIŘÍ ADÁMEK, STEFAN MILIUS, AND JIŘÍ VELEBIL

One algebraic approach to infinite computations abstracting from the nature of external memory are iterative algebraic theories, introduced by C. Elgot in [E]. This notion has been extended to the notion of completely iterative theories by Elgot, Bloom and Tindell, see [EBT]. The latter are infinitary algebraic theories (or equivalently, monads on the category of sets) that allow for unique solutions of systems of recursive equations of a certain type.

It has recently been discovered independently by L. Moss [Mo] and P. Aczel and the present authors [AAV], [AAMV] that a coalgebraic approach to infinite computations makes it possible to obtain more general categorical results, where, moreover, the proofs are conceptually clearer than in the setting of algebraic theories. However, unique solutions of recursive equations can only be obtained for a certain restricted class of functors. That excludes such important examples as the power-set functor with applications in process algebra. In recent work [AMV] we have shown how to uniquely solve recursive equations for arbitrary endofunctors in the category of sets. Here we shall extend those results to all locally presentable categories. But first we recall the basic results from [AAMV].

In the coalgebraic approach one considers a category $\mathcal{A}$ with binary coproducts such that coproduct injections are monomorphic, and an *iteratable* endofunctor H on $\mathcal{A}$, i. e., such that for every object X a final coalgebra

$$TX$$

of $H(_) + X$ exists.

Basic example: Given a signature Σ, consider for a given set X of variables the Σ-algebra $T_\Sigma X$ consisting of all finite and infinite Σ-labelled trees over X (i. e., nodes with n children are labelled by n-ary operation symbols and leaves by constant symbols or variables in X). It is well-known that $T_\Sigma X$ is the final $H_\Sigma(_) + X$ coalgebra, where $H_\Sigma : \mathsf{Set} \longrightarrow \mathsf{Set}$ is the polynomial endofunctor associated with the signature Σ, see [AK]. Trees admit unique solutions of recursive systems

$$\begin{array}{rcl} x_1 & \approx & t_1(x_1, x_2, \ldots, y_1, y_2, \ldots) \\ x_2 & \approx & t_2(x_1, x_2, \ldots, y_1, y_2, \ldots) \\ & \vdots & \end{array}$$

where $X = \{x_1, x_2, \ldots\}$ is a set of variables, $Y = \{y_1, y_2, \ldots\}$ is a set of parameters, and for each variable x_i, t_i is a *guarded* tree (i. e., a tree in $T_\Sigma(X + Y)$ which is not just a single variable). A solution of such a system is a sequence $x_1{}^\dagger, x_2{}^\dagger, \ldots$ of trees in $T_\Sigma Y$ such that the trees $x_i{}^\dagger$ and $t_i(x_1{}^\dagger, x_2{}^\dagger, \ldots, y_1, y_2, \ldots)$ are identical, for all i. The existence of unique solutions for trees is an instance of a very general categorical fact.

More precisely, if we start with an iteratable endofunctor H as above, notice first that by the Lambek Lemma [L] the structure map $TX \longrightarrow HTX + X$ is an isomorphism, i. e., TX is a coproduct of HTX and X. Denote the injections as follows:

$$\begin{array}{rrcl} \tau_X : & HTX \longrightarrow TX & & (\text{“}TX \text{ becomes an } H\text{-algebra”}) \\ \eta_X : & X \longrightarrow TX & & (\text{“embedding of variables”}) \end{array}$$

The final coalgebras TX have rich structure. Firstly, the way how substitution works on trees generalizes smoothly.

Theorem 1. *For any arrow $s : X \longrightarrow TY$ there exists a unique homomorphism of $\widehat{s} : TX \longrightarrow TY$ of H-algebras extending s, i. e., such that $\widehat{s} \cdot \eta_X = s$.*

It follows that T carries the structure of a monad (T, η, μ). Moreover, recursive equations have unique solutions. To make this more precise, let us call an arrow $e : X \longrightarrow T(X + Y)$ in $\mathcal{A}$ an *equation morphism.*

Date: June 13, 2002.
Key words and phrases. completely iterative, monad, solution theorem, parametric corecursion.

It is called *guarded* if it factorizes as follows:

$$\begin{array}{ccc} X & \xrightarrow{e} & T(X+Y) \\ & \searrow & \uparrow{\scriptstyle [\tau_{X+Y},\, \eta_{X+Y}\mathsf{inr}]} \\ & & HT(X+Y)+Y \end{array} \tag{1}$$

A *solution* of an equation morphism e is a morphism $e^\dagger : X \longrightarrow TY$ such that the following triangle

$$\begin{array}{ccc} X & \xrightarrow{e^\dagger} & TY \\ {\scriptstyle e}\downarrow & \nearrow{\scriptstyle \widehat{[e^\dagger, \eta_Y]}} & \\ T(X+Y) & & \end{array}$$

commutes.

The following result is called *Parametric Corecursion* in [Mo] and Solution Theorem in [AAMV]:

Theorem 2. *Every guarded equation morphism has a unique solution.*

The first topic in the present paper are free completely iterative monads. It has been proved in [EBT] that the algebraic theory of finite and infinite trees is a free completely iterative theory over the given signature. This result has been extended in [AAMV] to the setting of iteratable endofunctors.

Informally, monads that admit unique solutions of recursive equations are called *completely iterative*. It turns out that for any iteratable endofunctor H the above monad T is a free completely iterative monad on H, see [AAMV]. More precisely, notice first that the above notions of equation morphism and solution make sense for any monad on $\mathcal{A}$. To be able to express guardedness of equation arrows the notion of *idealized monad* is introduced:

Definition 3. Let (S, η, μ) be a monad. A subfunctor $\sigma : S' \rightarrowtail S$ is called a (*right*) *ideal* provided that the multiplication μ of the given monad restricts to S', i. e., there exists a (necessarily unique) $\mu' : S'S \longrightarrow S$ such that the following square

$$\begin{array}{ccc} S'S & \xrightarrow{\mu'} & S' \\ {\scriptstyle \sigma S}\downarrow & & \downarrow{\scriptstyle \sigma} \\ SS & \xrightarrow[\mu]{} & S \end{array}$$

commutes.

A monad together with an ideal of it is called an *idealized monad.* If furthermore we have $S = S' + Id$, with injections σ and η, then S is called an *ideal monad.*

An *idealized-monad morphism* between idealized monads S and R with ideals $\sigma : S' \rightarrowtail S$ and $\varrho : R' \rightarrowtail R$, respectively, is a monad morphism $\alpha : S \longrightarrow R$ that restricts to the ideals of the monads, i. e., there exists a (necessarily unique) $\alpha' : S' \longrightarrow R'$ such that the following square

$$\begin{array}{ccc} S' & \xrightarrow{\alpha'} & R' \\ {\scriptstyle \sigma}\downarrow & & \downarrow{\scriptstyle \varrho} \\ S & \xrightarrow[\alpha]{} & R \end{array}$$

commutes.

Remark 4. Notice that the notion of an ideal of a monad corresponds precisely to the notion of a right ideal for a monoid. Indeed, a right ideal of a monoid M is a subset I of M such that $I \cdot M \subseteq I$. Now a monad S is just a monoid in the monoidal category $[\mathcal{A}, \mathcal{A}]$ of endofunctors on $\mathcal{A}$ with tensor product given by composition of functors.

Recall that the above monad T is a coproduct of HT and Id. And the ideal $\tau : HT \rightarrowtail T$, where μ' is given by $H\mu$, makes T an ideal monad. Moreover, T comes with a natural "embedding of H":

$$\tau^* \equiv H \xrightarrow{H\eta} HT \xrightarrow{\tau} T$$

For idealized monads one easily defines the notion of a *guarded* equation morphism as an arrow e that factorizes as follows:

$$\begin{array}{ccc} X & \xrightarrow{e} & S(X+Y) \\ & \searrow & \uparrow [\sigma_{X+Y}, \eta_{X+Y} \cdot \mathsf{inr}] \\ & & S'(X+Y)+Y\,. \end{array}$$

For the ideal monad T this is precisely the same notion as the one defined above (see (1)).

An idealized monad S is called *completely iterative* if every guarded equation arrow has a unique solution. Thus, Theorem 2 says that T is a completely iterative monad.

The following is the main result of [AAMV]:

Theorem 5. *For every iteratable endofunctor H, the monad T is a free completely iterative monad on H. More precisely, given a completely iterative monad S and an idealized transformation $\lambda : H \longrightarrow S$ (i.e., such that λ factorizes through the ideal $S' \rightarrowtail S$) there exists a unique idealized-monad morphism $\overline{\lambda} : T \longrightarrow S$ such that $\overline{\lambda} \cdot \tau^* = \lambda$:*

$$\begin{array}{ccc} H & \xrightarrow{\tau^*} & T \\ & {\scriptstyle\lambda}\searrow & \downarrow \overline{\lambda} \\ & & S\,. \end{array}$$

It has been shown in [M] that, conversely, iteratable endofunctors are precisely those that admit free completely iterative monads.

Theorem 6. *If S is a free completely iterative monad on H, then H is iteratable, and SX is a final coalgebra of $H(_)+X$.*

Thus, for non-iteratable functors there is no free completely iterative monad. However, this does not imply that recursive equations are not uniquely solvable. It is possible to extend Theorem 2 to arbitrary endofunctors. For the category of sets this has been shown in [AMV]. We generalize this result to locally presentable categories. In order to do this, we first need to obtain parametrized final coalgebras for arbitrary endofunctors. We work in the following setting proposed by M. Barr, see [B]. Fix an inaccessible cardinal, $\aleph_\infty$, say. A *small set* is a set X of cardinality less than $\aleph_\infty$. These sets form the category Set. A *class* is any set with cardinality not greater than $\aleph_\infty$. The category of classes is denoted by Set^∞. This generalizes as follows: let $\mathcal{K}$ be a category that is

(i) cocomplete, i.e., small diagrams have a colimit in $\mathcal{K}$,
(ii) cowellpowered, i.e., each object of $\mathcal{K}$ has only a small set of quotients,
(iii) locally small, i.e., $\mathcal{K}$ has only a class of objects and each hom-set is small.

We denote by $\mathcal{K}^\infty$ the free cocompletion of $\mathcal{K}$ under *small-filtered colimits*, i.e., colimits of class-indexed diagrams that are λ-filtered for each infinite cardinal $\lambda < \aleph_\infty$. Notice that any endofunctor H on $\mathcal{K}$ extends essentially uniquely to an endofunctor H^∞ on $\mathcal{K}^\infty$ preserving small-filtered colimits.

Theorem 7. *For any endofunctor $H : \mathcal{K} \longrightarrow \mathcal{K}$ there exists an initial algebra and a final coalgebra of H^∞.*

Now suppose we have an endofunctor $H : \mathcal{K} \longrightarrow \mathcal{K}$, where $\mathcal{K}$ is a locally presentable category satisfying conditions (i)–(iii) above. The extension $H^\infty : \mathcal{K}^\infty \longrightarrow \mathcal{K}^\infty$ is iteratable. Denote by $T^\sharp$ the free completely iterative monad on H^∞. A *small equation morphism* is a morphism $X \longrightarrow T^\sharp(X+Y)$ in $\mathcal{K}^\infty$, where X and Y are objects of $\mathcal{K}$. The notions of guardedness and solution are the same as above. Consider the λ-accessible coreflections $H_\lambda : \mathcal{K} \longrightarrow \mathcal{K}$ of H given by left Kan extension of HJ_λ along J_λ, where $J_\lambda : \mathcal{K}_\lambda \longrightarrow \mathcal{K}$ denotes the inclusion of the small category $\mathcal{K}_\lambda$ of (representatives of all) λ-presentable objects of $\mathcal{K}$ into $\mathcal{K}$. Recall that accessible endofunctors are iteratable (see [AAMV], Example 2.9). Hence, all the H_λ are iteratable. Denote by T_λ the completely iterative monad generated by H_λ.

Lemma 8. $T^\sharp = \operatorname*{colim}_{i<\aleph_\infty} T_\lambda^\infty$ *is a small-filtered colimit.*

This allows us to prove the following result:

Theorem 9. (General Solution Theorem) *Every small guarded equation morphism $e : X \longrightarrow T^\sharp(X+Y)$ has a unique solution, which can be found in $\mathcal{K}$.*

More precisely, since X is small-presentable in $\mathcal{K}^\infty$, e factorizes through an arrow in $\mathcal{K}$

$$\begin{array}{ccc} X & \xrightarrow{e} & T^\sharp(X+Y) \\ & \overline{e} \searrow & \uparrow \\ & & T_\lambda(X+Y) \end{array}$$

for some $\lambda < \aleph_\infty$, and the solution of e is obtained by solving $\overline{e}$ in $\mathcal{K}$. In fact, the following triangle

$$\begin{array}{ccc} X & \xrightarrow{e^\dagger} & T^\sharp Y \\ & \overline{e}^\dagger \searrow & \uparrow \\ & & T_\lambda Y \end{array}$$

commutes.

References

[AAV] P. Aczel, J. Adámek and J. Velebil, A Coalgebraic View of Infinite Trees and Iteration, *Electronic Notes in Theoretical Computer Science* 44.1 (2001).

[AAMV] P. Aczel, J. Adámek, S. Milius and J. Velebil, Infinite Trees and Completely Iterative Theories: A Coalgebraic View, accepted in *Theoretical Computer Science*.

[AM] P. Aczel and P. F. Mendler, A Final Coalgebra Theorem, in: *Category Theory and Computer Science*, D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, A. Poigné (eds.), LNCS 389, Springer-Verlag, 1989, 357–365

[AK] J. Adámek and V. Koubek, On the greatest fixed point of a set functor, *Theoretical Computer Science* 150 (1995), 57–75.

[AMV] J. Adámek, S. Milius and J. Velebil, Final Coalgebras and A Solution Theorem for Arbitrary Endofunctors, to appear in *Electronic Lecture Notes in Computer Science*, 2002.

[B] M. Barr, Terminal Coalgebras in Well-Founded Set Theory, *Theoretical Computer Science* 124 (1994), 182–192.

[ADJ] J. A. Goguen, S. W. Thatcher, E. G. Wagner and J. B. Wright, Initial Algebra Semantics and Continuous Algebras, *Journal of the ACM* 24 (1977), 68–95.

[E] C. C. Elgot, Monadic Computation and Iterative Algebraic Theories, in: *Logic Colloquium '73* (eds: H. E. Rose and J. C. Shepherdson), North-Holland Publishers, Amsterdam, 1975.

[EBT] C. C. Elgot, S. L. Bloom, R. Tindell, On the Algebraic Structure of Rooted Trees, *J. Comp. Syst. Sciences* 16, (1978), 361–399.

[L] J. Lambek, A Fixpoint Theorem for Complete Categories, *Math. Z.* 103 (1968), 151–161.

[M] S. Milius, On Iteratable Endofunctors, to appear in *Electronic Lecture Notes in Computer Science*, 2002.

[Mo] L. Moss, Parametric Corecursion, *Theoretical Computer Science* 260 (2001), no. 1–2, 139–163.

Institute of Theoretical Computer Science, Technical University, Braunschweig, Germany
E-mail address: {adamek,milius,velebil}@iti.cs.tu-bs.de

Coalgebraic Approaches to Algebraic Terms

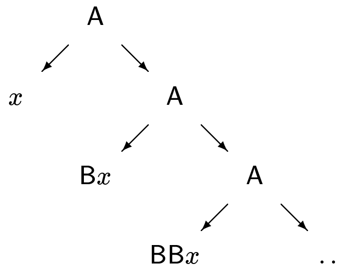
Neil Ghani, Christoph Lüth and Federico De Marchi

June 13, 2002

What are Algebraic Terms? If Σ is a signature, a Σ-algebraic system of equations is of the form

$$\begin{array}{rcl} \phi_1(x_1,\ldots,x_{n_1}) & = & t_1(x_1,\ldots,x_{n_1}) \\ \phi_2(x_1,\ldots,x_{n_2}) & = & t_2(x_1,\ldots,x_{n_2}) \\ & \vdots & \\ \phi_m(x_1,\ldots,x_{n_m}) & = & t_m(x_1,\ldots,x_{n_m}) \end{array} \tag{1}$$

where we regard the ϕ_i as the constructors of a signature Ω and the terms t_i, where $i \in \{1,\ldots,n\}$, are built from the signature $\Sigma \cup \{\phi_1,\ldots,\phi_n\}$. Courcelle [3] proves that algebraic systems in *Greibach Normal Form*, ie where the right-hand sides of the equations all start with a Σ-constructor, have unique solutions in the set of infinite trees over Σ. The proof uses a least fixed point construction relying on the set of infinite trees forming a complete metric space. For example, if Σ contains a binary symbol A and a unary symbol B while Ω contains only a unary symbol ϕ, then the Σ-algebraic equation $\phi(x) = \mathsf{A}(x, \phi(\mathsf{B}(x)))$ has the following solution



Note that nested recursion is permitted, eg in the equation $\phi(x) = \mathsf{A}(x, \phi(\mathsf{B}(\phi(x))))$. Infinite terms which arise as the solutions of algebraic systems of equations are called algebraic terms. *Rational terms* [3, 6, 4] arise as a special case of algebraic terms where, intuitively, the recursive calls of the function symbols ϕ_i do not involve any change in the parameters, eg in the equation $\phi'(x) = \mathsf{A}(x, \phi'(x))$.

Generalising Algebraic Terms: This abstract generalises algebraic terms as follows:

- The definition of signature etc is highly correlated to working over the category of **Set**. We show how algebraic terms can be defined for other categories and discuss applications.

- Σ-algebraic systems of equations can be described as recursion over the initial Σ-algebra since the right hand side of each equation belongs to the initial Σ-algebra over some set of variables. We generalise algebraic terms to other algebras and, again, briefly discuss applications.

Related Work: Recently Moss [8], Adamek et al. [1] and ourselves [4] have been looking at categorical, specifically coalgebraic, approaches to recursion. The general aim has been the same as those given above, namely to take standard concepts such as rational and algebraic terms and generalise them to a wider setting. For some time, it has been realised that infinite terms form a final coalgebra in much the same way that

finite terms form an initial algebra. The fact that one can solve generalised forms of rational equations using coalgebra was proven in the above papers in the order given. Moss also has an unpublished approach [7] to solving algebraic equations using coalgebra and we are investigating whether his approach generalises in the directions proposed within this abstract. We are grateful to him for sharing his ideas with us.

A Categorical Reformulation: If $\mathbb{N}$ is the discrete category whose objects are the natural numbers thought of as sets with that cardinality, a signature Σ is a functor $\Sigma : \mathbb{N} \to \mathbf{Set}$ mapping a number to the set of operations of that arity. This generalises to a *locally finitely presentable* category [2, 5] $\mathcal{C}$ where one defines a signature to be a functor $\Sigma : \mathcal{N} \to \mathcal{C}$ where $\mathcal{N}$ is the discrete subcategory formed by the set of generators of the category. ¿From a signature Σ one usually constructs an associated functor $F_\Sigma : \mathbf{Set} \to \mathbf{Set}$ which can be thought of as mapping a set of variables X to the set $F_\Sigma(X)$ of Σ-terms of depth 1 built from variables X. Formally, F_Σ is the left Kan extension of Σ along the inclusion $J : \mathbb{N} \to \mathbf{Set}$, ie $F_\Sigma = \mathsf{Lan}_J\Sigma$. Finally, if T_Σ is the free monad on F_Σ, then $T_\Sigma(X)$ is the usual initial algebra of terms built from variables X. The multiplication and unit of the monad provide an abstract model of the key concepts of substitution and variables which underly the ability to solve equations. Within this terminology, a *guarded Σ-algebraic system of equations* is a natural transformation

$$\Omega \to (1 + F_\Sigma T_{\Sigma+\Omega}) \circ J \tag{2}$$

where Ω is some signature. Note that we have allowed the mild generalisation in that the signature may declare an infinite number of operators (all with finite arities) and the right-hand side of an equation may be a variable (the 1+ part). These generalisations actually make the mathematics easier and hence their inclusion. Also note how the condition that the right hand side of an equation has a root symbol from Σ is modelled by applying the endofunctor F_Σ after the monad calculating terms built from the coproduct of Σ and Ω.

Since the left Kan extension is left adjoint to precomposition, (2) corresponds to a natural transformation

$$F_\Omega = \mathsf{Lan}_J\Omega \to 1 + F_\Sigma T_{\Sigma+\Omega} \tag{3}$$

$1 + F_\Sigma T_{\Sigma+\Omega}$ is a monad because we can embed F_Σ in $T_{\Sigma+\Omega}$ and hence (3) corresponds to a monad morphism

$$T_\Omega \to 1 + F_\Sigma T_{\Sigma+\Omega}$$

This achieves our first goal of defining algebraic equations over categories more general than **Set**. To allow right hand sides of equations to come from arbitrary terms algebras, we need to replace the initial algebras from the monad $T_{\Sigma+\Omega}$ with something more general. Now, by abstract reasoning [5], $T_{\Sigma+\Omega} = T_\Sigma + T_\Omega$ where the latter coproduct is in the category of monads. It turns out that T_Ω can be replaced by any monad whatsoever! For T_Σ however, we cannot take any monad since the elements of the generalisation of T_Σ must be interpreted in the monad T_Σ^∞ of infinite terms. We use the notion of a *coalgebraic monad* we introduced recently [4].

Coalgebraic Monads: The free monad T_F^μ on an endofunctor F satisfies the isomorphism $T_F^\mu \cong 1 + FT_F^\mu$, ie T_F^μ is a fixed point of the $1 + F \circ _$ endofunctor on the category of endofunctors. The monad of finite and infinite terms T_F^ν (which is pointwise the final $X + F \circ _$ coalgebra) is also a fixed point of $1 + F \circ _$. The same holds for the term graph monad and the rational monad [4]. Abstracting from these examples, [4] defines:

Definition: *Let F be an endofunctor on a category $\mathcal{C}$. An F-*coalgebraic *monad on $\mathcal{C}$ is a 4-tuple (T, η, μ, τ) such that (T, η, μ) is a monad on $\mathcal{C}$ and τ is a natural transformation between F and T for which the monad morphism $[\eta, \mu.\tau T] : 1 + FT \longrightarrow T$ is an isomorphism.*

Given a functor F, [4] proves that the free monad T_F^μ is the initial F-coalgebraic monad while T_F^ν is the final F-coalgebraic monad. Coalgebraic monads provide suitable right hand sides for algebraic equations:

Definition: *Let H be an F-coalgebraic monad. An* algebraic system *over H consists of a monad E and a monad morphism $e : E \to 1 + F(H + E)$. A solution for e is a monad morphism $e^\dagger : E \to T_F^\nu$ making the*

following commute ($!_H$ is the unique monad morphism from H to the final F-coalgebraic monad T^ν_F.)

$$\begin{array}{ccc} E & \xrightarrow{\quad e^\dagger \quad} & T^\nu{}_F \\ {\scriptstyle e}\downarrow & & \cong \\ 1+F(H+E) & \xrightarrow[1+F[!_H, e^\dagger]]{} & 1+FT^\nu{}_F \end{array}$$

Lemma: *If $e : E \to 1 + F(H + E)$ is an algebraic system over H, then e has a unique solution*

Space constraints prevents anything but a proof sketch: First, $1+F(H+E)$ is a monad since the coalgebraicity of H allows us to map F into H. Next, there is a monad morphism $1+F\mathsf{inl} : H \cong 1+FH \to 1+F(H+E)$ and hence a monad morphism $[1+F\mathsf{inl}, e] : H+E \to 1+F(H+E)$. This map endows $H+E$ with a $1+F\circ_$-coalgebra structure and hence there is a monad morphism $H + E \to T^\nu_F$. Precomposition with the second inclusion gives our candidate solution which is easily seen to make the relevant diagram commute.

We believe the elegance of the categorical formulation of an algebraic equation is matched by the simplicity of the proof of the existence of a unique solution — in a full paper the details could be fleshed out but the above sketch illustrates our viewpoint. Having obtained solutions for our monadic reformulation of algebraic systems, we can specialise these results back to functors and to signatures. Let $G : \mathcal{C} \to \mathcal{C}$ be a functor and H an F-coalgebraic monad. A *functorial algebraic system* over H consists of a natural transformation $e : G \to 1 + F(H + T^\mu_G)$. The notion of a solution of a functorial algebraic system is as one would expect although space prevents us from giving it. Solutions for a functorial algebraic system $e : G \to 1 + F(H + T^\mu_G)$ then arise as solutions of the algebraic system $e^* : T^\mu_G \to 1+F(H+T^\mu_G)$. Finally, if $\Sigma : \mathcal{N} \to \mathcal{C}$ is a signature and H an F-coalgebraic monad, then a *algebraic system of equations* over H consists of a natural transformation $e : \Sigma \to (1 + F(H + T^\mu{}_\Sigma)) \circ J$. Again, solutions are defined as expected and are generated as solutions of the functorial algebraic system $\mathsf{Lan}_J e : F_\Sigma \to 1 + F(H + T^\mu_\Sigma)$.

Applications: We have a number of reasons for solving generalised forms of algebraic equations. Firstly we want to extend the categorical semantics of rewriting to cover infinite rewriting. One possibility is to consider rewriting over algebraic terms which naturally leads to the idea of infinite rewrites being defined by algebraic equations which entails working on something like the category **Pre** of preorders. Secondly, we wish to understand compilation of functional programming languages such as Haskell where terms are stored as term graphs so as to increase efficiency by the sharing of common subterms. In particular, recursive equations are actually modelled as recursively defined term graphs which motivates algebraic equations whose right hand sides are more general than terms. Finally, and more speculatively, we are interested in a coalgebraic treatment of recursively defined geometric objects such as Sierpinskis Triangle and other fractal like objects.

References

[1] P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. 2001.

[2] J. Adámek and J. Rosicky. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Notes*. Cambridge University Press, 1994.

[3] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[4] N. Ghani, C. Lüth, and F. De Marchi. Coalgebraic monads. In L.M. Moss, editor, *Proceedings CMCS'02*, 2002.

[5] G.M. Kelly and J. Power. Adjunctions whose counits are equalizers, and presentations of finitary monads. *Journal of Pure and Applied Algebra*, (89):163–179, 1993.

[6] Stefan Milius. Free iterative theories: a coalgebraic view (extended abstract). presented at FICS 2001.

[7] L. Moss. The coalgebraic treatment of second-order substitution and uniinterpreted recursive program schemes. privately circulated manuscript.

[8] L. Moss. Parametric corecursion. preprint, available at `http://math.indiana.edu/home/moss/parametric.ps`.

Generalizing Substitution (Extended Abstract)

Tarmo Uustalu

Inst. of Cybernetics, Tallinn Techn. Univ.
Akadeemia tee 21, EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

It is well known that, given an endofunctor H on a category $\mathcal{C}$, the initial $(A + H-)$-algebras, i.e., the algebras of wellfounded H-terms over different sets-of-variables A (if existing) give rise to a monad with substitution as the extension operation (the free monad generated by H). That a remarkably similar monad, which even enjoys the additional property of having "iterations" for all guarded substitution rules, arises from the inverses of the final $(A + H-)$-coalgebras, i.e., the algebras of non-wellfounded H-terms over different A (if existing) was only recently pointed out by Moss [4] and Aczel, Adámek, Velebil [1], although the "down-to-earth" universal-algebra case of H a polynomial endofunctor on **Set** was settled in the 1970s by Elgot and colleagues, see e.g., [2]. We consider the following generalization: if T' is a endobifunctor on $\mathcal{C}$ such that the functors $T'(-, X)$ uniformly carry a monad structure, then the initial $T'(A, -)$-algebras (if existing) give rise to a monad and the inverses of the final $T'(A, -)$-coalgebras (if existing) yield an "iterative" monad. Besides wellfounded and non-wellfounded terms, examples include Krstić, Launchbury and Pavlović hyperfunctions [3] and finitely or potentially infinitely branching wellfounded or non-wellfounded node-labelled trees. The following is a compressed technical summary.

"Iterative" monads Roughly following [1], we say that a monad $(T, \eta, -^\star)$ on $\mathcal{C}$ is "iterative" for a morphism $f : A \to T(A + B)$, if there is a unique morphism $f' : A \to TB$, denoted $\bar{f}$, such that

$$f' = [\, f', \eta_B \,]^\star \circ f$$

It is useful to note that this is equivalent to the unique existence of a morphism $h : T(A+B) \to TB$, denoted $\tilde{f}$, such that

$$h = [\, h \circ f, \eta_B \,]^\star$$

The conversions are: $\tilde{f} = [\, \bar{f}, \eta_B \,]^\star$, $\bar{f} = \tilde{f} \circ f = \tilde{f} \circ \eta_{A+B} \circ \mathsf{inl}_{A,B}$.

Substitution in wellfounded and non-wellfounded term algebras Given any endofunctor H on a category $\mathcal{C}$ with finite coproducts. Then, given any assignment of some $(A+H-)$-algebra (TA, α_A) to every object A, we also have, for every object A, morphisms $\eta_A : A \to TA$, $\tau_A : HTA \to TA$ defined by $\eta_A = \alpha_A \circ \mathsf{inl}_{A,HTA}$, $\tau_A = \alpha_A \circ \mathsf{inr}_{A,HTA}$, with the property that $[\, \eta_A, \tau_A \,] = \alpha_A$. Say that (T, α) is substitution-carrying, if, for every morphism $f : A \to TB$, there exists a unique morphism $h : TA \to TB$, denoted $f^\star$, satisfying

$$\begin{array}{ccc} A + HTA & \xrightarrow{[\, \eta_A, \tau_A \,]} & TA \\ {\scriptstyle \mathsf{id}_A + Hh} \big\downarrow & & \big\downarrow {\scriptstyle h} \\ A + HTB & \xrightarrow{[\, f, \tau_B \,]} & TB \end{array}$$

If (T, α) is substitution-carrying, then, immediately, $(T, \eta, -^\star)$ is a monad. Two examples concern us. The first constitutes the starting point of the category-theoretic approach to universal algebra. If the initial $(A+H-)$-algebra (the free H-algebra over A, or the algebra of usual, wellfounded H-terms over A) exists for every A, then $(T, \alpha) = (\mu(A + H-), \mathsf{in}_{A+H-})$ is substitution-carrying. The reason is that the characterizing equation of $f^\star$ is, for any given morphism $f : A \to TB$, trivially equivalent to the characterizing equation of the iterative extension of the $(A + H-)$-algebra $(TB, [\, f, \tau_B \,])$. The resulting monad is called the free monad generated by H.

The other example is that of Moss [4] and Aczel, Adámek, Velebil [1]. Their "substitution theorem" states that, if the final $(A+H-)$-coalgebra (the coalgebra of non-wellfounded H-terms over A) exists for every A, then (T,α) defined by $(TA,\alpha_A) = (\nu(A+H-), \mathsf{out}^{-1}_{A+H-})$ is substitution-carrying. This is smoothly proved by noting the equivalence for any $f : A \to TB$ of the characterizing equation of the primitive corecursive extension of the $(B + H(- + TB))$-coalgebra

$$(TA, [\,(\mathsf{id}_B + H\mathsf{inr}_{TA,TB}) \circ \alpha_B^{-1} \circ f, \mathsf{inr}_{B,H(TA+TB)}\,] \circ (\mathsf{id}_A + \mathsf{inl}_{TA,TB}) \circ \alpha_A^{-1})$$

to that of $f^\star$. The "solution theorem" states that the resulting monad $(T, \eta, -^\star)$ is "iterative" for all morphisms $f : A \to T(A + B)$ which factor in the canonical way

$$f = [\,\eta_{A+B} \circ \mathsf{inr}_{A,B}, \tau_{A+B}\,] \circ \varphi$$

through some morphism $\varphi : A \to B + HT(A + B)$ (which intuitively means that all guarded substitution rules can be iterated). For a proof, one can show, e.g., the equivalence for any such f of the characterizing equation of the coiterative extension of the $(B + H-)$-coalgebra

$$(T(A + B), [\,[\,\varphi, \mathsf{inl}_{B,HT(A+B)}\,], \mathsf{inr}_{B,HT(A+B)}\,] \circ \alpha^{-1}_{A+B})$$

to the characterizing equation of $\tilde{f}$.

The generalization of this paper The monads resulting from substitution-carrying assignments of an $(A+H-)$-algebra to every object A are, by the explicitly defining and characterizing equations of their constituent data, very similar to the monads that the object mappings $T'(-,X)$ given by $T'(A,X) = A + HX$ carry uniformly in X. Indeed, setting $\eta'_{A,X} = \mathsf{inl}_{A,HX}$, $\tau_{A,X} = \mathsf{inr}_{A,HX}$ and $f^\circ = [\,f, \tau_B\,]$ $(f : A \to T'(B,X))$, we get that $(T'(-,X), \eta'_{-,X}, -^\circ)$ is a monad for every X. This observation leads to the questions: Is this a hint about some "causality"? Can it be used to modularize the proofs of the statements above and to generalize them? The answer is: yes.

Say that a parametrized monad on a category $\mathcal{C}$ is a mapping $T' : |\mathcal{C}| \times \mathcal{C} \to \mathcal{C}$ functorial in the 2nd argument together with a $|\mathcal{C}| \times |\mathcal{C}|$-indexed family η' of morphisms $\eta'_{A,X} : A \to T'(A,X)$ and an operation $-^\circ$ taking every morphism $f : A \to T'(B,X)$ to a morphism $f^\circ : T'(A,X) \to T'(B,X)$ such that (i) $f^\circ \circ \eta'_{A,X} = f$ $(f : A \to T'(B,X))$, (ii) $\eta'_{A,X}{}^\circ = \mathsf{id}_{T'(A,X)}$, (iii) $(g^\circ \circ f)^\circ = g^\circ \circ f^\circ$ $(f : A \to T'(B,X), g : B \to T'(C,X))$, (iv) $T'(A,\xi) \circ \eta'_{A,X} = \eta'_{A,Y}$ $(\xi : X \to Y)$, (v) $T'(B,\xi) \circ f^\circ = (T'(B,\xi) \circ f)^\circ \circ T'(A,\xi)$ $(f : A \to T'(B,X), \xi : X \to Y)$. (A parametrized monad is essentially just a functor from $\mathcal{C}$ to $\mathbf{Monad}(\mathcal{C})$, but it is more convenient to use an "uncurried" equivalent notion here.)

Given now any parametrized monad $(T', \eta', -^\circ)$ on any category $\mathcal{C}$. Then, given any assignment of some $T'(A,-)$-algebra (TA, α_A) with α_A iso[1] to every object A, we also have, for every object A, a morphism $\eta_A : A \to TA$ defined by $\eta_A = \alpha_A \circ \eta'_{A,TA}$, and, for every morphism $f : A \to TB$, a morphism $\{f\} : T'(A,TB) \to TB$ defined by $\{f\} = \alpha_B \circ (\alpha_B^{-1} \circ f)^\circ$, so that $\{\eta_A\} = \alpha_A$. We choose to say that (T,α) is substitution-carrying, if, for every $f : A \to TB$, there is a unique $h : TA \to TB$, denoted $f^\star$, such that

$$\begin{array}{ccc} T'(A,TA) & \xrightarrow{\{\eta_A\}} & TA \\ {\scriptstyle T'(A,h)}\downarrow & & \downarrow{\scriptstyle h} \\ T'(A,TB) & \xrightarrow{\{f\}} & TB \end{array}$$

Clearly, the previous definition for the special case $T'(A,X) = A + HX$ is an instance of this new more general definition. But, pleasantly, everything we stated for the special concept generalizes. Moreover, the proofs become simpler, since the presence of a parametrized monad structure on T' is used consciously, not proven over and over again without noticing.

[1] The iso requirement can be avoided at the cost of switching to a somewhat less intuitive setup of definitions and statements.

Firstly and foremostly, if (T, α) is substitution-carrying, then $(T, \eta, -^\star)$ is a monad. Further, if the initial $T'(A, -)$-algebra exists for every A, then (T, α) defined by $(TA, \alpha_A) = (\mu(T'(A, -)), \mathsf{in}_{T'(A,-)})$ is substitution-carrying, with $f^\star$ constructed, for any given morphism $f : A \to TB$, as the iterative extension of the $T'(A, -)$-algebra $(TB, \{f\})$. If the final $T'(A, -)$-coalgebra exists for every A, then (T, α) defined by $(TA, \alpha_A) = (\nu(T'(A, -)), \mathsf{out}^{-1}_{T'(A,-)})$ is substitution-carrying, with $f^\star$ equal for any $f : A \to TB$ to the primitive corecursive extension of the $T'(B, - + TB)$-coalgebra

$$(TA, (T'(B, \mathsf{inr}_{TA,TB}) \circ \alpha_B^{-1} \circ f)^\circ \circ T'(A, \mathsf{inl}_{TA,TB}) \circ \alpha_A^{-1})$$

and, finally, the resulting monad $(T, \eta, -^\star)$ is "iterative" for all morphisms $f : A \to T(A + B)$ which factor in the canonical way

$$f = \{\eta_{A+B} \circ \mathsf{inr}_{A,B}\} \circ \varphi$$

through some morphism $\varphi : A \to T'(B, T(A+B))$, with $\tilde{f}$ constructible as the coiterative extension of the $T'(B, -)$-coalgebra

$$(T(A + B), [\varphi, \eta'_{B,T(A+B)}]^\circ \circ \alpha^{-1}_{A+B})$$

Examples

1. If $\mathcal{C}$ has finite coproducts, then the parametrized monad structure on T' given by $T'(A, X) = A + HX$ where $H : \mathcal{C} \to \mathcal{C}$ yields a monad structure on T given by $TA = \mu(A + H-)$ and $TA = \nu(A + H-)$, meaning that TA is the object of wellfounded resp. non-wellfounded H-terms over A).
2. If $\mathcal{C}$ is cartesian closed, then the parametrized monad structure on T' given by $T'(A, X) = HX \Rightarrow A$ where $H : \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$ yields a monad structure on T given by $TA = \mu(H- \Rightarrow A)$ and $TA = \nu(H- \Rightarrow A)$. In the special case $HX = X \Rightarrow E$ where E is a $\mathcal{C}$-object, TA is the inductive resp. coinductive object of hyperfunctions from E to A in the sense of Krstić, Launchbury and Pavlović [3].
3. If $\mathcal{C}$ has finite products, then the parametrized monad structure on T' given by $T'(A, X) = A \times U(HX)$ where $H : \mathcal{C} \to \mathbf{Mon}(\mathcal{C})$ yields a monad structure on T given by $TA = \mu(A \times U(H-))$ and $TA = \nu(A \times U(H-))$. If $\mathcal{C}$ has (finite) list objects, then, in the special case $HX = (\mathsf{List}X, \mathsf{nil}_X, \mathsf{append}_X)$, TA is the object of wellfounded resp. non-wellfounded A-labelled finitely branching trees, which may, e.g., be thought of as Böhm trees corresponding to lambda-terms without lambdas (purely applicative terms). The substitution operation delivered by our construction agrees with substitution as defined for Böhm trees.

Acknowledgements This work was done during the author's postdoctoral leave to Univ. do Minho, Braga, and was supported by the Portuguese Foundation for Science and Technology under grant No. PRAXIS XXI/C/EEI/14172/98 and the Estonian Science Foundation under grant No. 4155. He is also grateful to Graduiertenkolleg "Logik in der Informatik" for an invitation to LMU München, where he had the opportunity to first present it at their Colloquium in Jan. 2002.

References

1. P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. In A. Corradini, M. Lenisa, and U. Montanari, eds., *Proc. of 4th Wksh. on Coalgebraic Methods in Computer Science, CMCS'01 (Genova, Apr. 2001)*, vol. 44(1) of *Electr. Notes in Theor. Computer Science*. Elsevier, 2001.
2. C. C. Elgot, S. L. Bloom, and R. Tindell. On the algebraic structure of rooted trees. *Journal of Computer and System Sciences*, 16(3):362–399, 1978.
3. S. Krstić, J. Launchbury, and D. Pavlović. Categories of processes enriched in final coalgebras. In F. Honsell and M. Miculan, eds., *Proc. of 4th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS'01 (Genova, Apr. 2001)*, vol. 2030 of *Lecture Notes in Computer Science*, pp. 303–317. Springer-Verlag, 2001.
4. L. S. Moss. Parametric corecursion. *Theoretical Computer Science*, 260(1–2):139–163, 2001.

Traced Premonoidal Categories (Extended Abstract)

Nick Benton
Microsoft Research
Cambridge, UK
nick@microsoft.com

Martin Hyland
University of Cambridge
Department of Pure Mathematics
and Mathematical Statistics
M.Hyland@dpmms.cam.ac.uk

Abstract

Motivated by some examples from functional programming, we propose a generalisation of the notion of trace to symmetric premonoidal categories and of Conway operators to Freyd categories. We show that, in a Freyd category, these notions are equivalent, generalising a well-known theorem of Hasegawa and Hyland.

1 Introduction

Monads were introduced into computer science by Moggi [18] as a structuring device in denotational semantics and soon became a popular abstraction for writing actual programs, particularly for expressing and controlling side-effects in 'pure' functional programming languages such as Haskell [25, 17]. Power and Robinson subsequently introduced *premonoidal categories* as a generalisation of Moggi's computational models [21], whilst Hughes developed *arrows*, which are the equivalent programming abstraction [12].

Some uses of monads in functional programming seem to call for a kind of recursion operator on computations for which, informally, the recursion 'only takes place over the values'. For example, the Haskell Prelude defines the (internally implemented) `ST` and `IO` monads for, respectively, potentially state-manipulating and input/output-performing computations. These come equipped with polymorphic functions

```
fixST :: (a -> ST a) -> ST a
fixIO :: (a -> IO a) -> IO a
```

which allow computations to be recursively defined in terms of the values they produce. For example, the following program uses `fixIO`[1] to extend a cunning cyclic programming trick due to Bird [1] to the case of side-effecting computations. `replacemin` computes a tree in which every leaf of the argument has been replaced by the minimum of all the leaves. It does this in a single pass over the input and prints out each leaf as it encounters it:[2]

[1] We should note that `fixIO` does not *actually* satisfy the axioms we will propose. However, the basic pattern would remain the same, though the code would be a little longer, if we had performed side-effects involving state instead.

[2] The tilde ~ on the last line specifies 'lazy' pattern matching for the pair `(m,r)`. Haskell's tuples are actually lifted products and pattern matching is, by default, strict. Without the tilde the function would diverge.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

f :: Tree Int -> Int -> IO (Int,Tree Int)
f (Leaf n) m = do print n
                  return (n, Leaf m)
f (Branch t1 t2) m =
    do (m1,r1) <- f t1 m
       (m2,r2) <- f t2 m
       return (min m1 m2, Branch r1 r2)

replacemin :: Tree Int -> IO (Int, Tree Int)
-- m is argument to and part of the result of f
replacemin t = fixIO (\ ~(m,r) -> f t m)
```

As another (though still somewhat contrived) example, consider modelling the heap of a fictitious pure Scheme-like language at a fairly low level. One might interpret heap-manipulating computations using a monad `T` which is an instance of a type class something like this

```
class Monad T => HeapMonad T where
  alloc  :: (Int, Int) -> T Int
  lookup :: Int -> T (Int,Int)
  free   :: Int -> T ()
```

The intention is that `alloc` takes two integers and returns a computation which finds a free cons cell in the heap, fills it with those two integers and returns the (strictly positive) address of the allocated cell. `lookup` takes an integer address and returns the contents of that cons cell, whilst `free` marks a particular address as available for future allocations. Since the values in the *car* and *cdr* of cells can be used as the addresses of other cells, we can intepret programs which build data structures such as lists in the heap. What if the language we are interpreting can create cyclic structures (for example, closures for recursive functions)? At the machine level, cyclic structures are created by allocating cells containing dummy values and then 'tying the knot' by overwriting those dummy values with the addresses returned by the allocator. Hence we could just provide destructive update operations

```
setcar :: Int -> Int -> T ()
setcdr :: Int -> Int -> T ()
```

and use those to create cycles. However, if the interpreted language itself does not include destructive assignment, but only creates cycles using higher-level constructs, then adding assignment operations to the monad breaks an abstraction barrier. One solution is to add a recursion operation to the monad

```
fixT :: (a -> T a) -> T a
```

with a definition such that the following code creates a two-element cyclic list (and returns the addresses of both cells):

```
onetwocycle :: T (int,int)
onetwocycle  = fixT (\~(x,y)->
                       do { x' <- alloc(1,y)
                            y' <- alloc(2,x)
                            return (x',y')
                          })
```

Observe that although the computation is recursively defined, it should only perform the two allocation side-effects once.

Many of the real uses of this kind of recursion have the flavour of the previous example: they involve computations which create cyclic structures for which the identity, order of creation or multiplicity of creation of the objects in the structure is significant. An interesting example arises in work on using Haskell to model hardware. Early versions of both Lava [2] and Hawk [16] specified circuits in a monadic style, instantiating the monad differently for different applications (such simulating the circuit, generating a netlist or interfacing with a theorem prover). Cyclic circuits (i.e. those with feedback) were defined in essentially the style used to define `onetwocycle` above. Lava has moved away from that style, in part because it is syntactically awkward.[3] Launchbury et al. [16] also noted that programming in a monadic style with `fixT` is awkward, and suggested extending Haskell's `do` notation to allow recursive bindings. That suggestion was followed up by Launchbury and Erkök, who proposed an axiomatisation of operators like `fixT` (which they call `mfix`) and showed how the `do` notation can be extended to allow recursive bindings in the case that the underlying monad supports such an `mfix` operation [15].

Launchbury and Erkök's axiomatisation of `mfix` is partly in terms of equations and partly in terms of inequations, intended to be interpreted in the 'usual' (slightly informal) concrete domain theoretic model of Haskell. One striking feature of [15] is that it does not appear to build on any of the large body of existing work on axiomatic/categorical treatments of recursion, even those (such as [7]) which consider fixed points in terms of monads. The authors cite some of this work but state, quite correctly, that the non-standard kind of recursion in which they are interested is different from that covered in the literature. Although the presence of a fixpoint object [7], for example, allows an operator with the same type as `mfix` to be defined, it is not of the kind we want.

From a categorical perspective, we seem to want a notion of recursion or feedback on the Kleisli category of a CCC with a strong monad. There is a special case of this situation in which earlier work *does* provide an answer. Although none of Launchbury and Erkök's examples are of commutative monads, in that case the Kleisli category will be symmetric monoidal and Joyal, Street and Verity's notion of *trace* seems to fit the bill [14].

In the general case of a non-commutative monad, however, the Kleisli category will only be symmetric *pre*monoidal. The work described here grew firstly from the natural mathematical question of what the right definition of traced premonoidal category might be, and secondly from wondering whether an answer might provide a sensible categorical semantics for the kind of fixpoint operators described in [15]. We give a natural, straightforward and well-behaved answer to the first question, though it only accounts for a rather special subset of the cases considered by Launchbury and Erkök.

[3]Lava now uses a modified version of Haskell with 'observable sharing': allowing new name generation as an implicit side-effect of every expression and hence changing the equational theory of the language [6].

2 Background

2.1 Premonoidal Categories

For a careful definition of the notion of *(symmetric) premonoidal category* and *(symmetric) premonoidal functor*, see Power and Robinson's paper [21]. Briefly, a premonoidal category is a monoidal category except that the tensor product $\otimes$ need only be a functor in each of the two variables separately. Thus if $f : A \to B$ and $g : A' \to B'$ in a premonoidal category $\mathbb{K}$ then the two evident morphisms $A \otimes A' \to B \otimes B'$

$$\begin{aligned} f \ltimes g &= A \otimes A' \xrightarrow{f \otimes A'} B \otimes A' \xrightarrow{B \otimes g} B \otimes B' \\ f \rtimes g &= A \otimes A' \xrightarrow{A \otimes g} A \otimes B' \xrightarrow{f \otimes B'} B \otimes B' \end{aligned}$$

are not generally equal.

We generally write I for the unit of the tensor in a (pre)monoidal category, σ for the symmetry if there is one, and λ, ρ, α for the natural isomorphisms

$$\begin{aligned} \lambda &: I \otimes A \to A \\ \rho &: A \otimes I \to A \\ \alpha &: (A \otimes B) \otimes C \to A \otimes (B \otimes C) \end{aligned}$$

However, since we have coherence theorems for (symmetric) (pre)monoidal categories [21], we will usually elide the structural isomorphisms.

Definition 2.1. A morphism $f : A \to B$ in a premonoidal category $\mathbb{K}$ is *central* if for all $g : A' \to B'$ in $\mathbb{K}$, $f \ltimes g = f \rtimes g$. If at least one of f and g is central, then we may unambiguously write $f \otimes g$. The *centre* $Z(\mathbb{K})$ of a premonoidal category $\mathbb{K}$ is the monoidal subcategory of $\mathbb{K}$ with the same objects but only the central morphisms.

The inclusion functor $Z(\mathbb{K}) \to \mathbb{K}$ is a strict, identity-on-objects premonoidal functor (and symmetric if $\mathbb{K}$ is). In more recent work Power in particular has stressed the importance from the algebraic point of view in having an explicit choice of centre. That is, one is interested in the situation where one has a functor $J : \mathbb{M} \to \mathbb{K}$ from a specified (symmetric) monoidal subcategory of a (symmetric) premonoidal $\mathbb{K}$; J factors through $Z(\mathbb{K})$, so this amounts to specifying a particular subcategory of central morphisms. (For many results J does not even need to be faithful, but we do not consider that generality here.) We call a $J : \mathbb{M} \to \mathbb{K}$ as above a *centred premonoidal category*, but since this is our preferred notion we usually drop the 'centred'. In this context, by *central* morphisms we shall mean the morphisms of $\mathbb{M}$. One should think of $\mathbb{M}$ as a category of *values* and $\mathbb{K}$ as a category of possibly-effectful *computations*. An important special case is the following:

Definition 2.2. A *Freyd category* [22] is specified by a cartesian category $\mathbb{C}$, a symmetric premonoidal category $\mathbb{K}$ and an identity-on-objects strict symmetric premonoidal functor $J : \mathbb{C} \to \mathbb{K}$.

Note that morphisms in the specified centre of a Freyd category are 'pure' not merely in the sense of commuting with arbitrary effectful computations, but also in being copyable and discardable.

Example 2.1. If T is a strong monad on a symmetric monoidal category $\mathbb{M}$, then the Kleisli category $\mathbb{M}_T$ is symmetric premonoidal and the canonical functor from $\mathbb{M}$ to $\mathbb{M}_T$ is strict symmetric premonoidal. Thus in the case that $\mathbb{M}$ is cartesian, we have a Freyd category. If the monad is commutative, then $\mathbb{M}_T$ is symmetric monoidal and $J : \mathbb{M} \to \mathbb{M}_T$ is strict symmetric monoidal.

2.2 Traces and Fixpoints

The notion of traced monoidal category was introduced in [14]. The use of traces to interpret recursion in programming languages and the relationship between traces and fixpoints have attracted much attention in recent years, beginning with Hasegawa's thesis [11]. Categorical axiomatisations of fixpoint operators have been extensively studied, see [7, 19, 4] for example; a particularly crisp and up-to-date account appears in [24].

Definition 2.3. A *trace* on a symmetric monoidal category $(\mathbb{M}, \otimes, I, \lambda, \rho, \alpha, \sigma)$ is a family of functions

$$\mathrm{tr}^U_{A,B} : \mathbb{M}(A \otimes U, B \otimes U) \to \mathbb{M}(A, B)$$

satisfying the following conditions

- Naturality in A (Left Tightening).
  If $f : A' \otimes U \to B \otimes U$, $g : A \to A'$ then
  $$\mathrm{tr}^U_{A,B}((g \otimes U); f) = g; \mathrm{tr}^U_{A',B}(f) : A \to B$$
- Naturality in B (Right Tightening).
  If $f : A \otimes U \to B' \otimes U$, $g : B' \to B$ then
  $$\mathrm{tr}^U_{A,B}(f; (g \otimes U)) = \mathrm{tr}^U_{A,B'}(f); g : A \to B$$
- Dinaturality (Sliding).
  If $f : A \otimes U \to B \otimes V$, $g : V \to U$ then
  $$\mathrm{tr}^U_{A,B}(f; (B \otimes g)) = \mathrm{tr}^V_{A,B}((A \otimes g); f) : A \to B$$
- Action (Vanishing). If $f : A \to B$ then
  $$\mathrm{tr}^I_{A,B}(\rho; f; \rho^{-1}) = f : A \to B$$
  and if $f : A \otimes (U \otimes V) \to B \otimes (U \otimes V)$ then
  $$\mathrm{tr}^{U \otimes V}_{A,B}(f) = \mathrm{tr}^U_{A,B}(\mathrm{tr}^V_{A \otimes U, B \otimes U}(\alpha; f; \alpha^{-1}))$$
- Superposing. If $f : A \otimes U \to B \otimes U$ then
  $$\begin{aligned}&\mathrm{tr}^U_{C \otimes A, C \otimes B}(\alpha; C \otimes f; \alpha^{-1}) \\ &\quad = C \otimes \mathrm{tr}^U_{A,B}(f) \ : \ C \otimes A \to C \otimes B\end{aligned}$$
- Yanking. For all U, $\mathrm{tr}^U_{U,U}(\sigma_{U,U}) = U : U \to U$.

Monoidal categories provide a formal basis for reasoning about many of the graphical 'boxes and wires' notations used in computer science. The trace axioms are presented graphically in Figure 1, though we do not consider the formal semantics of such diagrams here.

Definition 2.4. A *parameterized fixpoint operator* on a cartesian category $\mathbb{C}$ is a family of functions

$$(\cdot)^\dagger : \mathbb{C}(A \times U, U) \to \mathbb{C}(A, U)$$

satisfying

- Naturality. If $f : B \times U \to U$ and $g : A \to B$ then
  $$g; f^\dagger = ((g \times U); f)^\dagger : A \to U$$
- Fixed point property. If $f : A \times U \to U$ then
  $$\langle A, f^\dagger \rangle; f = f^\dagger : A \to U$$

The above definition is rather weak. Well-behaved fixpoint operators typically satisfy other interesting conditions.

Definition 2.5. A *Conway operator* is a parameterized fixpoint operator which additionally satisfies

- Parameterized Dinaturality. If $f : A \times V \to U$ and $g : A \times U \to V$ then
  $$\langle A, (\langle \pi_1, f \rangle; g)^\dagger \rangle; f = (\langle \pi_1, g \rangle; f)^\dagger : A \to U$$
- Diagonal Property. If $f : A \times U \times U \to U$ then
  $$((A \times \Delta); f)^\dagger = (f^\dagger)^\dagger : A \to U$$

Parameterized dinaturality is easily seen to imply the parameterized fixed point property and, in some concrete categories of domains, is sufficient to characterize the least fixed point operator uniquely [23]. Conway operators satisfy various other useful identities, including the 'Bekič property', which allows simultaneous fixed points to be reduced to sequential ones.

There are also further 'uniformity' properties which a fixpoint operator may have [24], but we shall not consider those in the present paper.

An important theorem about traces and fixpoints is the following, which is due (independently) to Hasegawa and to Hyland, though its essential combinatorial content had been observed earlier in a slightly different context [3, 5]:

Theorem 2.1 (Hasegawa, Hyland). *To give a trace on a cartesian category $\mathbb{C}$ is to give a Conway operator on $\mathbb{C}$.* □

3 Traces and Fixpoint Operators on Premonoidal Categories

So, what is an appropriate generalisation of the notions of trace and fixpoint operator to the premonoidal case? We want definitions which make sense, have useful concrete instances, give the monoidal versions as special cases and lead to a generalisation of Theorem 2.1.

3.1 Symmetric Premonoidal Traces

We start by trying to generalise the definition of trace to a centred symmetric premonoidal category $J : \mathbb{M} \to \mathbb{K}$. Although none of the conditions in Definition 2.3 are expressed in terms of tensoring arbitrary morphisms (in which case we'd certainly have to reexamine them), we cannot simply leave the definition unchanged:

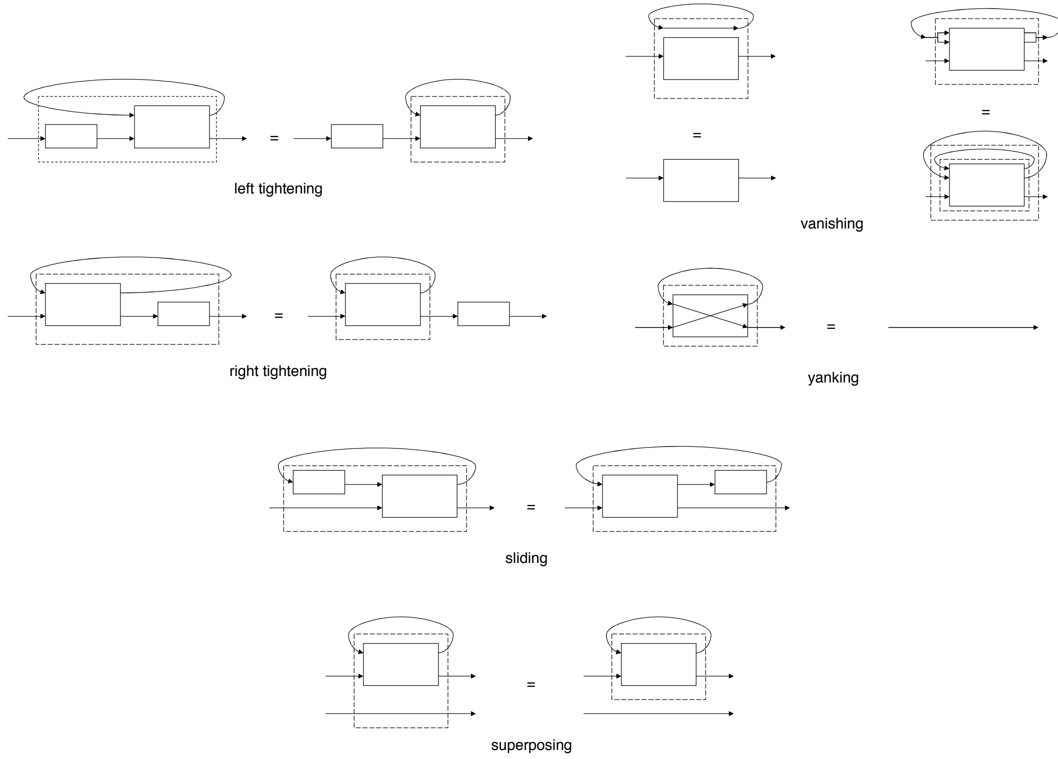


Figure 1: Trace Axioms

Proposition 3.0.1. *A symmetric premonoidal category with a trace as defined in Definition 2.3 is actually monoidal.* □

The key step in the proof of the previous Proposition uses the Sliding axiom to commute the side-effects of two computations. This observation motivates the following definition:

Definition 3.1. A *trace* on a centred symmetric premonoidal category $J : \mathbb{M} \to \mathbb{K}$ is is a family of functions

$$\mathrm{tr}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

satisfying the same conditions given in Definition 2.3 *except* that the Sliding axiom is replaced by

- Premonoidal Sliding. If $f : A \otimes U \to B \otimes V$ and if $g : V \to U$ is a central morphism then

$$\mathrm{tr}^U_{A,B}(f; (B \otimes g)) = \mathrm{tr}^V_{A,B}((A \otimes g); f) : A \to B$$

and we impose the further requirement

- Centre Preservation. If $f : A \otimes U \to B \otimes U$ is central then so is $\mathrm{tr}^U_{A,B}\, f : A \to B$.

Clearly, if $J : \mathbb{M} \to \mathbb{K}$ has a premonoidal trace on $\mathbb{K}$, the restriction of that trace to $\mathbb{M}$ is a trace operator in the traditional sense of Definition 2.3. In particular, Definition 3.1 really is a generalisation of Definition 2.3.

Requiring the trace to preserve the distinguished centre $\mathbb{M}$ is largely a matter of taste: we prefer to keep our equations algebraic. Even without the condition it is still easy to see that the trace preserves $Z(\mathbb{K})$:

Proposition 3.0.2. *If $f : A \otimes U \to B \otimes U$ is in $Z(\mathbb{K})$ and $g : C \to D$ then $g \ltimes \mathrm{tr}^U_{A,B}(f) = g \rtimes \mathrm{tr}^U_{A,B}(f)$.* □

It might also be remarked that the premonoidal sliding condition appears somewhat asymmetric, since it requires that g, rather than one of f and g, be central. However, a little calculation shows that the symmetric case is a consequence:

Proposition 3.0.3. *Assume $f : A \otimes U \to B \otimes V$ is central and $g : V \to U$, then $\mathrm{tr}^V_{A,B}((A \otimes g); f) = \mathrm{tr}^U_{A,B}(f; B \otimes g)$.* □

3.2 Symmetric Premonoidal Fixpoints

We now turn to generalising the notion of fixpoint operator to the premonoidal case. Since some of the axioms involve duplication and discarding, we will assume that we are working in a Freyd category $J : \mathbb{C} \to \mathbb{K}$. We also use Δ, π_1, $\langle \cdot, \cdot \rangle$, etc. as shorthand notation for the lifting of the appropriate operations from $\mathbb{C}$ to $\mathbb{K}$ (i.e. we elide uses of J). The notation $\langle f, g \rangle$ is ambiguous unless we specify the order in which the components are computed, but we shall only use it in the case one of the maps is central.

Definition 3.2. A *parameterized fixpoint operator* on a Freyd category $J : \mathbb{C} \to \mathbb{K}$ is a family of functions

$$(\cdot)^* : \mathbb{K}(A \otimes U, U) \to \mathbb{K}(A, U)$$

which satisfies

- Centre Preservation. If $f : A \otimes U \to U$ is central then so is $f^* : A \to U$.

- Naturality. If $f : B \otimes U \to U$ and $g : A \to B$ then

$$g; f^* = ((g \otimes U); f)^* : A \to U$$

- Central Fixed Point Property. If $f : A \otimes U \to U$ is central, then

$$\langle A, f^* \rangle; f = f^* : A \to U$$

Just as in the cartesian case, this is the bare minimum one might require of a fixpoint operator. We are interested in rather stronger conditions, and propose the following as an appropriate generalisation of Conway operators on cartesian categories:

Definition 3.3. A parameterized fixpoint operator $(\cdot)^*$ on a Freyd category is a *Conway operator* if it satisfies the following conditions:

1. *Parallel Property.* If $f : A \otimes U \to U$ and $g : B \otimes V \to V$ with one of f and g central then

$$(A \otimes \sigma \otimes V\,;\, f \otimes g)^* = f^* \otimes g^* \;:\; A \otimes B \to U \otimes V$$

2. *Withering Property.* If $f : A \otimes U \to B \otimes U$ and $g : B \to C$ then

$$(\langle \pi_1, \pi_3 \rangle; f; g \otimes U)^* = (\langle \pi_1, \pi_3 \rangle; f)^*; g \otimes U \;:\; A \to C \otimes U$$

3. *Diagonal Property.* If $f : A \otimes U \otimes U \to U$ then

$$((A \otimes \Delta); f)^* = (f^*)^* : A \to U$$

The axioms of a premonoidal Conway operator are shown graphically in Figure 3.2, where we follow Jeffrey [13] in using a heavy line to indicate the sequencing of effects in $\mathbb{K}$ (and that line runs outside those boxes intended to represent central morphisms). The diagonal property is essentially the same as in the cartesian case, but the parallel and withering properties are more unusual.

There *is* a natural generalization of the dinaturality condition to Freyd categories:

Definition 3.4. A parameterized fixpoint operator $(\cdot)^*$ on a Freyd category satisfies *parameterized central dinaturality* if, given $f : A \otimes U \to V$ and $g : A \otimes V \to U$ with g central

$$(\langle \pi_1, f \rangle; g)^* = \langle A, (\langle \pi_1, g \rangle; f)^* \rangle; g$$

As in the cartesian case, parameterized central dinaturality clearly implies the central fixed point property. But in the case of Freyd categories, the dinaturality condition does not seem sufficient (along with the diagonal property) to establish the equivalence between traces and Conway operators, which is what motivated our parallel and withering axioms. These do imply dinaturality, however:

Proposition 3.0.4. *A Conway operator on a Freyd category satisfies parameterized central dinaturality.* □

Furthermore, our definition of a Conway operator on a Freyd category does generalise the standard one:

Proposition 3.0.5. *Definition 3.3 is equivalent to Definition 2.5 in the case that the category is cartesian.* □

3.3 Relating Fixpoints and Traces in Freyd Categories

We now show our main result: in a Freyd category, to give a premonoidal trace is equivalent to giving a premonoidal Conway operator.

Theorem 3.1. *Let $J : \mathbb{C} \to \mathbb{K}$ be a Freyd category such that $\mathbb{K}$ is traced, as in Definition 3.1. Then the operation*

$$(\cdot)^* : \mathbb{K}(A \otimes U, U) \to \mathbb{K}(A, U)$$

defined by, for $f : A \otimes U \to U$:

$$f^* = \mathrm{tr}^U_{A,U}(f; \Delta)$$

is a Conway operator in the sense of Definition 3.3. □

Remark 3.1. Hasegawa has also given a construction for a fixpoint operator from a trace in the special case of the Kleisli category of a commutative strong monad on a cartesian category (a case in which the premonoidal structure is monoidal) [11, Theorem 7.2.1]. However, restricting our construction to this special case does not generally give the same fixpoint operator. Hasegawa's construction uses the adjunction in an essential way and repeats side-effects.

Theorem 3.2. *Let $J : \mathbb{C} \to \mathbb{K}$ be a Freyd category where $\mathbb{K}$ has a Conway operator $(\cdot)^*$ as defined in Definition 3.3. Then the operation*

$$\mathrm{tr}^U_{A,B}(\cdot) : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by, for $f : A \otimes U \to B \otimes U$

$$\mathrm{tr}^U_{A,B}(f) = (\langle \pi_1, \pi_3 \rangle; f)^*; \pi_1 \;:\; A \to B$$

is a premonoidal trace in the sense of Definition 3.1. □

Proposition 3.2.1. *The constructions of trace from Conway operator and of Conway operator from trace given in Theorems 3.2 and 3.1 respectively are mutually inverse.* □

Thus we have succeeded in establishing a premonoidal generalization of Theorem 2.1.

Remark 3.2. Starting from a fixpoint operator, there is another candidate for the definition of a trace, viz

$$\mathrm{tr}'(f) = \langle A, (f; \pi_2)^* \rangle; f; \pi_1 : A \to B$$

where $f : A \otimes U \to B \otimes U$. If $\mathbb{K}$ is monoidal this is the same as the construction used in Theorem 3.2, but in the general premonoidal case they are different, and tr' does not seem to have useful properties.

4 Examples

4.1 Monoids

Let $\mathbb{M}$ be a traced symmetric monoidal category as in Definition 2.3 and $(M, \mu : M \otimes M \to M, \eta : I \to M)$ be a monoid in $\mathbb{M}$. Let $\mathbb{K}$ be the Kleisli category of the monad $TA = M \otimes A$ on $\mathbb{M}$, so $\mathbb{K}(A, B) = \mathbb{M}(A, M \otimes B)$. Then the tensor on $\mathbb{M}$ lifts so that $J : \mathbb{M} \to \mathbb{K}$ is a centred symmetric premonoidal category (it is monoidal iff M is a commutative monoid).

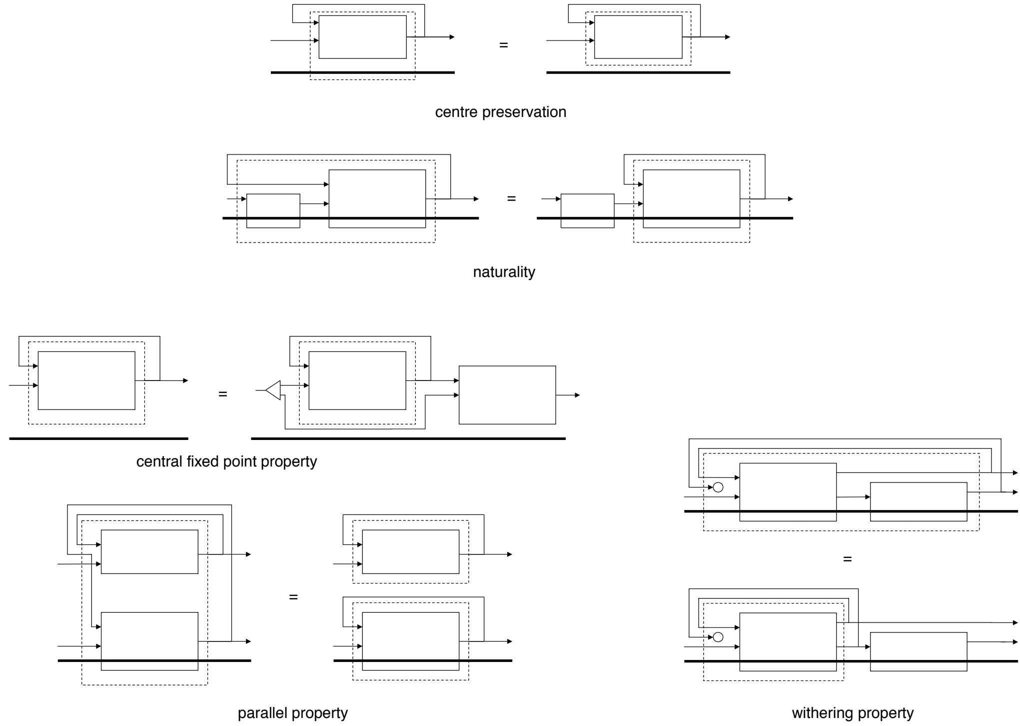


Figure 2: Premonoidal Conway Axioms

Proposition 4.0.2. *In the above situation, the operation*

$$\hat{\mathrm{tr}}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by $\hat{\mathrm{tr}}^U_{A,B}(f) = \mathrm{tr}^U_{A,M\otimes B}(f)$ *is a premonoidal trace.* □

Notions of computation based on monoids are fairly common. Commutative monoids such as the natural numbers under addition can be used for modelling resource usage (e.g. timed computations) whereas non-commutative monoids model, for example, side-effecting output. In Haskell syntax, the signature could look like this:

```
class Monoid m where
   mult :: (m,m) -> m
   unit :: m

newtype Cross m a = Cross (m,a) deriving Show

instance Monoid m => Monad (Cross m) where
   return a = Cross (unit,a)
   Cross(m,a) >>= f = let Cross(m',b) = f a
                      in Cross(mult (m,m'), b)

instance Monoid [a] where
   mult (s,t) = s ++ t
   unit = []

-- command which writes to the output
output s = Cross(s,())
```

If we then apply our construction of a premonoidal Conway operator from the trace defined in Proposition 4.0.2 then we end up with an `mfix` operation of the type described by Launchbury and Erkök:

```
instance Monoid m => MonadRec (Cross m) where
   mfix f = let Cross(m,a) = f a
            in Cross(m,a)
```

And this does have the expected behaviour:

```
nats_output =
   mfix (\ys -> do output "first "
                   output "second."
                   return (0 : map succ ys))

> nats_output
Cross ("first second.",[0,1,2,3,4,5,6,7,8,9,...
```

The two side effects have happened once only and in the order specified.

4.2 State

Let $\mathbb{M}$ be a traced symmetric monoidal category, S be an object of $\mathbb{M}$ and $\mathbb{K}$ be the category with the same objects as $\mathbb{M}$ and $\mathbb{K}(A,B) = \mathbb{M}(S \otimes A, S \otimes B)$ with the evident composition. If $\mathbb{M}$ is closed then $\mathbb{K}$ is equivalent to the Kleisli category of the state monad $TA = S \multimap S \otimes A$. Then $J : \mathbb{M} \to \mathbb{K}$ is premonoidal.

Proposition 4.0.3. *In the above situation, the operation*

$$\hat{\mathrm{tr}}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by $\hat{\mathrm{tr}}^U_{A,B}(f) = \mathrm{tr}^U_{S\otimes A,S\otimes B}(f)$ *is a premonoidal trace.* □

Again, the derived fixed point operator on the Kleisli category is easily defined in Haskell:

```
newtype State s a = State (s -> (s,a))

instance Monad (State s) where
  return a = State (\s ->(s,a))
  State m >>= f = State (\s -> let (s',a) = m s
                                   State m' = f a
                               in m' s')

instance MonadRec (State s) where
  mfix f = State (\s -> let State m = f a
                            (s',a) = m s
                        in (s',a))
```

Note how the final value, `a` is recursively defined, but the final state `s'` is not – operationally, each time we go around the loop, the initial state is 'snapped back' to `s`.

5 Related Work

Compared with Launchbury and Erkök's work on axiomatising `mfix`, our definitions and results are in a rather more general setting, but account for rather fewer concrete examples. The axioms in [15] are almost identical to our definition of a premonoidal Conway operator except that they weaken some of our equations to inequations (interpreted in a concrete category of domains), add a strictness condition on one and regard some as additional properties which may hold in some cases (i.e. not part of the basic definition of what they call a 'recursive monad'). These weaker conditions admit definitions of `mfix` for monads such as `Maybe` $(1 + (\cdot))$, lazy lists and Haskell's `IO` monad [8] which do not have Conway operators satisfying our conditions.

Paterson has designed a convenient syntax for programming with Hughes's arrows (just as Haskell adds `do` to simplify programming with monads). Paterson's recent paper [20] gives axioms for an `ArrowLoop` operation which are the same as our definition of a premonoidal trace; our results thus prove an equivalence between `ArrowLoop` and a particular (newly identified) class of `mfix`s.

Jeffrey [13] has also considered a variant of traces in a premonoidal setting, though his construction is rather different from ours: he considers a *partial* trace (only applicable to certain maps) on the category of values rather than on that of computations.

Friedman and Sabry have also recently looked at defining an `mfix` operation [9], though their approach is rather different from the axiomatic one which we and the others cited here have taken. They view the ability to define computations recursively as an additional effect and give a 'monad transformer' which adds a state-based updating implementation of recursion to an arbitrary monad. Lifting the operations of the underlying monad to the new one is left to the programmer (and can generally be done in different ways).

6 Conclusions and Further Work

We have formulated and proved a natural generalisation of the theorem relating traces and Conway operators to the case of premonoidal categories. This has applications to the semantics of some non-standard recursion and feedback operations on computations which have been found useful in functional programming.

It would be interesting to see if one could explain Launchbury and Erkök's weaker axiomatisation in a more general setting. The natural thing to try here is to be more explicit about the presence of an abstract lifting monad, along the lines of [10]. This may also help establish a connection with the partial trace operation used by Jeffrey [13]. We would also like to have some more examples.

We are in the process of investigating the premonoidal version of the 'geometry of interaction' construction, which traditionally embeds a traced monoidal category into a compact closed one. This is interesting from a mathematical view and may also have some connection to the building of layered protocol stacks from stateful components.

References

[1] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, 1998.

[3] S. L. Bloom and Z. Esik. Axiomatizing schemes and their behaviors. *Journal of Computer and System Sciences*, 31(3):375–393, 1985.

[4] S. L. Bloom and Z. Esik. *Iteration Theories.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[5] V. E. Cazanescu and Gh. Stefanescu. A general result on abstract flowchart schemes with applications to the study of accessibility, reduction and minimization. *Theoretical Computer Science*, 99:1–63, 1992.

[6] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, 1999.

[7] R. L. Crole and A. M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

[8] L. Erkök, J. Launchbury, and A. Moran. Semantics of `fixio`. In *Proceedings of the Workshop on Fixed Points in Computer Science (FICS'01), Firenze, Italy*, September 2001.

[9] D. P. Friedman and A. Sabry. Recursion is a computational effect. Technical Report 546, Computer Science Department, Indiana University, December 2000.

[10] C. Fuhrmann, A. Bucalo, and A. Simpson. An equational notion of lifting monad. *Theoretical Computer Science*, 200? To appear.

[11] M. Hasegawa. *Models of Sharing Graphs (A Categorical Semantics of Let and Letrec).* Distinguished Dissertations in Computer Science. Springer-Verlag, 1999.

[12] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–112, 2000.

[13] A. Jeffrey. Premondoidal categories and a graphical view of programs. `http://www.cogs.susx.ac.uk/users/alanje/premon/`, June 1998.

[14] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3), 1996.

[15] J. Launchbury and L. Erkök. Recursive monadic bindings. In *International Conference on Functional Programming*, 2000.

[16] J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *International Conference on Functional Programming*, 1999.

[17] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] P. S. Mulry. Strong monads, algebras and fixed points. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in LMS Lecture Notes, pages 202–216, 1992.

[20] R. Paterson. A new notation for arrows. In *Proceedings of the International Conference on Functional Programming*. ACM Press, September 2001.

[21] A. J. Power and E. P. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997.

[22] A. J. Power and H. Thielecke. Closed Freyd and κ-categories. In *International Conference on Automata, Languages and Programming*, 1999.

[23] A. K. Simpson. A characterization of the least-fixed-point operator by dinaturality. *Theoretical Computer Science*, 118(2):301–314, 1993.

[24] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of 15th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, 2000.

[25] P. Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*. ACM, 1992.

Kleene Through the Process Algebraic Glass

Luca Aceto
BRICS*
Department of Computer Science, Aalborg University,
Fr. Bajersvej 7E, 9220 Aalborg Ø, Denmark.
Email: luca@cs.auc.dk

Process algebras are prototype specification languages for reactive systems. These languages usually consist of a collection of operators to build new processes from existing ones, and of facilities for the description of recursive behaviours, e.g., terms of the form $\mu x.\ t$ to denote some distinguished solution of the equation $x = t$. Alternatively, one may use variations on the Kleene star operation—first introduced in [4]—to define iterative processes in a purely algebraic syntax. The latter approach has been advocated by Bergstra, Bethke and Ponse in [1, 2], and subsequently followed by several authors, notably Fokkink and his co-workers.

My aim in this talk will be to present a survey of some of the results that I have contributed with Wan Fokkink and Anna Ingólfsdóttir to the study of semantic theories of process algebras incorporating variations on the Kleene star operation. In particular, I shall focus on variations on the language of Basic Process Algebra [3] with the original binary Kleene star. Starting from basic principles, I shall review results, both of a positive and a negative nature, on the equational axiomatization of behavioural equivalences over variations on this language, and mention some small, but hopefully interesting, results on the expressive power of variations on the binary Kleene star modulo bisimulation equivalence [5]—the prime example in the menagerie of behavioural equivalences between reactive systems that have been considered in the literature.

References

[1] J. Bergstra, I. Bethke, and A. Ponse, *Process algebra with iteration*, Report CS-R9314, Programming Research Group, University of Amsterdam, 1993.

[2] ——, *Process algebra with iteration and nesting*, Computer Journal, 37 (1994), pp. 243–258.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation

[3] J. Bergstra and J. Klop, *Fixed point semantics in process algebras*, Report IW 206, Mathematisch Centrum, Amsterdam, 1982.

[4] S. Kleene, *Representation of events in nerve nets and finite automata*, in Automata Studies, C. Shannon and J. McCarthy, eds., Princeton University Press, 1956, pp. 3–41.

[5] D. Park, *Concurrency and automata on infinite sequences*, in 5th GI Conference, Karlsruhe, Germany, P. Deussen, ed., vol. 104 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 167–183.

A NOTE ON GLOBAL INDUCTION MECHANISMS IN A μ-CALCULUS WITH EXPLICIT APPROXIMATIONS

CHRISTOPH SPRENGER AND MADS DAM

1. Introduction

The first-order μ-calculus [7] provides a useful setting for semi-automatic program verification. It is expressive enough to encode, from the bottom up, a range of program logics (e.g. LTL, CTL, CTL*, Hoare Logic) as well as process calculi and programming languages including their data types and operational semantics. A framework based on this idea is described in [4] and has been applied to a substantial part of a real concurrent programming language Erlang in the Erlang Verification Tool [1].

A key aspect in the design of such a framework is proof search, in particular the handling of fixed point formulas. The standard approach, Park's fixed point induction rule (cf. [5]), is not suitable for proof search in practice. An alternative is to admit cyclic proof structures (cf. [6, 10, 3]) and look for sound discharge conditions which will ensure the well-foundedness of the inductive reasoning. In this paper, we study such discharge conditions in the context of a Gentzen-style proof system for the first-order μ-calculus, which is a variant of previous systems [3, 4, 8, 9]. In particular, it shares with [3, 4, 8] the technique, first proposed for the modal μ-calculus in [3], of introducing explicit approximation ordinal variables and ordering constraints between them into the proof system. Discharge conditions then rely on these ordering constraints. The use of approximation ordinals considerably simplifies earlier treatments (cf. [2]). A simple semantic condition was proposed in [3] expressing in a natural way the requirement of well-foundedness of all inductive reasoning. Due to its semantical nature it is not suitable for the purpose of practical proof. However, it serves as a useful reference to which other, more syntactical conditions may be compared.

Previous syntactic discharge conditions [3, 4, 8] turn out to be strictly stronger than the semantic condition. The main contribution of the present paper is the formulation of a syntactic condition that precisely matches the semantic one. We introduce *traces* which are non-increasing (w.r.t. the ordering constraints) sequences of dependent ordinal variables running along a path in the proof structure. They can be seen as a generalisation of the μ- and ν-traces in [6] to the setting of explicit approximants. The characterisation result is then obtained by introducing the notion of progress for traces and by establishing its close correspondence with the semantic notion of well-foundedness. For practical application we then give an automata-theoretic formulation of our discharge condition in terms of an inclusion of the languages recognised by two Büchi automata.

Finally, we remark that this work is part of a larger programme aiming at clarifying the relation between the global, lazy induction mechanism used here and an eager induction discipline as implemented in a local proof rule for well-founded induction on the ordinals.

2. Logic

The logic we consider augments first-order logic with two fixed point operators, a standard and an approximated one parametrised by an ordinal variable. Formulas ϕ and abstractions Φ_X of the μ-calculus over a first-order signature Σ are inductively defined as follows

$$\phi ::= t = t' \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists x.\phi \mid \Phi_X(\overline{t}) \qquad \text{and} \qquad \Phi_X ::= X \mid \mu X(\overline{x}).\phi \mid \mu^\kappa X(\overline{x}).\phi$$

where t ranges over Σ-terms, x over individual variables, κ over ordinal variables and X over predicate variables. Each abstraction Φ_X has the arity of X which is required to match the lengths of the vectors $\overline{x}$ and $\overline{t}$ of individual variables and terms, respectively. Fixed point abstraction formation is subject to the usual syntactic monotonicity condition.

A *model* $\mathcal{M}$ is a pair $(\mathcal{A}, \rho)$ where $\mathcal{A}$ is a first-order Σ-structure with support set A and ρ is an $\mathcal{A}$-*environment* interpreting each variable according to its type. Formulas are interpreted as elements of the two-point lattice $\mathbf{2} = \{0, 1\}$ and n-ary abstractions as elements of $\mathsf{Pred}(A^n) = \mathbf{2}^{A^n}$, the lattice n-ary predicates over A under the pointwise ordering. For first-order formulas the semantics is as expected. For abstractions and their application we have

$$\|X\|_\rho^\mathcal{A} = \rho(X) \quad \|\mu X(\overline{x}).\phi\|_\rho^\mathcal{A} = \mu\Psi \quad \|\mu^\kappa X(\overline{x}).\phi\|_\rho^\mathcal{A} = \mu^{\rho(\kappa)}\Psi \quad \|\Phi_X(\overline{t})\|_\rho^\mathcal{A} = \|\Phi_X\|_\rho^\mathcal{A}(\|\overline{t}\|_\rho^\mathcal{A})$$

where $\Psi = \lambda P.\lambda\overline{a}.\|\phi\|^\mathcal{A}_{\rho[P/X,\overline{a}/\overline{x}]} : \mathsf{Pred}(A^n) \to \mathsf{Pred}(A^n)$, $\mu\Psi$ is the least fixed point of the (monotone) function Ψ and $\mu^{\rho(\kappa)}\Psi$ is the approximation $\rho(\kappa)$ of $\mu\Psi$, both defined as usual.

3. Proof Rules

Sequents of the proof system are of the form $\Gamma \vdash_\mathcal{O} \Delta$, where Γ and Δ are finite multi-sets of μ-calculus formulas and $\mathcal{O} = (|\mathcal{O}|, <_\mathcal{O})$ is a strict partial order on a finite set $|\mathcal{O}|$ of ordinal variables. Following [8] the latter is used to record constraints between ordinal variables. An environment ρ *respects* $\mathcal{O}$ if $\rho(\iota) < \rho(\kappa)$ whenever $\iota <_\mathcal{O} \kappa$. A sequent $\Gamma \vdash_\mathcal{O} \Delta$ is *valid* if the formula $\bigwedge\Gamma \to \bigvee\Delta$ is satisfied in all models $\mathcal{M} = (\mathcal{A}, \rho)$, where ρ respects $\mathcal{O}$.

The proof system extends the standard Gentzen-style rules for first-order logic with the following rules for the fixed point operators (we omit an ordinal constraint strengthening rule [8] for brevity):

$$(\mu\text{-L}) \quad \frac{\Gamma, (\mu^\kappa X(\overline{x}).\phi)(\overline{t}) \vdash_{\mathcal{O}'} \Delta}{\Gamma, (\mu X(\overline{x}).\phi)(\overline{t}) \vdash_\mathcal{O} \Delta} \quad \kappa \notin |\mathcal{O}|,\ \mathcal{O}' = (|\mathcal{O}| \cup \{\kappa\}, <_\mathcal{O})$$

$$(\mu\text{-R}) \quad \frac{\Gamma \vdash_\mathcal{O} \phi[\mu X(\overline{x}).\phi/X,\ \overline{t}/\overline{x}], \Delta}{\Gamma \vdash_\mathcal{O} (\mu X(\overline{x}).\phi)(\overline{t}), \Delta}$$

$$(\mu^\kappa\text{-L}) \quad \frac{\Gamma, \phi[\mu^{\kappa'} X(\overline{x}).\phi/X,\ \overline{t}/\overline{x}] \vdash_{\mathcal{O}'} \Delta}{\Gamma, (\mu^\kappa X(\overline{x}).\phi)(\overline{t}) \vdash_\mathcal{O} \Delta} \quad \kappa' \notin |\mathcal{O}|,\ \mathcal{O}' = (|\mathcal{O}| \cup \{\kappa'\}, (<_\mathcal{O} \cup \{(\kappa', \kappa)\})^+)$$

$$(\mu^\kappa\text{-R}) \quad \frac{\Gamma \vdash_\mathcal{O} \phi[\mu^{\kappa'} X(\overline{x}).\phi/X,\ \overline{t}/\overline{x}], \Delta}{\Gamma \vdash_\mathcal{O} (\mu^\kappa X(\overline{x}).\phi)(\overline{t}), \Delta} \quad \kappa' <_\mathcal{O} \kappa$$

4. Discharge Conditions

A derivation may be stopped at a leaf node N if N is a substitution instance of some node M in the derivation tree. It is worth noting that M need not lie on the path from the root node to N. We write $N(\Gamma \vdash_\mathcal{O} \Delta)$ to mean that N is labeled by the sequent $\Gamma \vdash_\mathcal{O} \Delta$. Given a node $M(\Gamma \vdash_\mathcal{O} \Delta)$ and a leaf $N(\Gamma' \vdash_{\mathcal{O}'} \Delta')$ of a derivation tree and a substitution σ, we say that $R = (M, N, \sigma)$ is a *repeat*, if $\Gamma\sigma \subseteq \Gamma'$, $\Delta\sigma \subseteq \Delta'$ and $\mathcal{O}\sigma \subseteq \mathcal{O}'$, where σ is order-preserving in the latter inclusion. We assume that σ maps predicate variables to predicate variables. N is called a *repeat node* and M its *companion*. A *pre-proof* $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ for $\Gamma \vdash_\mathcal{O} \Delta$ is a derivation tree $\mathcal{D}$ whose root node is labeled by $\Gamma \vdash_\mathcal{O} \Delta$ together with a set of repeats $\mathcal{R}$ such that each non-axiom leaf of $\mathcal{D}$ belongs to exactly one repeat in $\mathcal{R}$. A *path* $\pi = N_0 \cdots N_i \cdots$ of $\mathcal{P}$ is a (finite or infinite) sequence of nodes of $\mathcal{D}$ starting in the root N_0 of $\mathcal{D}$ such that for any two successive nodes N_i and N_{i+1} either (N_i, N_{i+1}) is an edge of $\mathcal{D}$ or there is a substitution σ such that $(N_{i+1}, N_i, \sigma) \in \mathcal{R}$ is a repeat.

Intuitively, each repeat should correspond to the application of an induction hypothesis, but we need to make sure that the inductive reasoning embodied in each individual repeat and their combinations is indeed well-founded. This requires a (necessarily global) *discharge condition* that identifies the legal proofs. We present three such conditions, the semantical one from [3], a new syntactical condition as well as its automata-theoretic formulation and establish their equivalence.

Let $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ be a pre-proof, $\mathcal{A}$ a Σ-structure and $\Pi = (N_0, \rho_0) \cdots (N_i, \rho_i) \cdots$ a (finite or infinite) sequence of pairs of nodes $N_i(\Gamma_i \vdash_{\mathcal{O}_i} \Delta_i)$ of $\mathcal{D}$ and $\mathcal{A}$-environments ρ_i. Then Π is called a *run* of $\mathcal{P}$ if N_0 is the root of $\mathcal{D}$, each ρ_i respects $\mathcal{O}_i$ and for each pair (N_i, N_{i+1}) either

(1) (N_i, N_{i+1}) is an edge of $\mathcal{D}$ and ρ_{i+1}, ρ_i agree on free variables common to N_i and N_{i+1}, or
(2) $(N_{i+1}, N_i, \sigma) \in \mathcal{R}$ is a repeat and $\rho_{i+1} = \rho_i \circ \sigma$.

The run Π is said to follow the path $\pi = N_0 \cdots N_i \cdots$. A pre-proof $\mathcal{P}$ for $\Gamma \vdash_\mathcal{O} \Delta$ satisfies *discharge condition (R-DC)* if all runs of $\mathcal{P}$ are finite, in which case $\mathcal{P}$ is called a *proof* for $\Gamma \vdash_\mathcal{O} \Delta$.

Theorem 1. **(Soundness)** *If there is a proof for* $\Gamma \vdash_{\mathcal{O}} \Delta$ *then* $\Gamma \vdash_{\mathcal{O}} \Delta$ *is valid.*

As condition (R-DC) captures the well-foundedness of the reasoning in a pre-proof in a very natural way, it serves as our reference discharge condition. Due to its semantical nature it is, however, hardly usable in practical proofs and we therefore introduce an alternative, purely syntactical, discharge condition.

Let $\tau = (N_0, w_0) \cdots (N_i, w_i) \cdots$ be a (finite or infinite) sequence of pairs of nodes $N_i(\Gamma_i \vdash_{\mathcal{O}_i} \Delta_i)$ of $\mathcal{D}$ and non-empty words w_i over the alphabet of ordinal variables. Then τ is called a *trace* of $\mathcal{P}$ if $w_i(j) \in |\mathcal{O}_i|$ and $w_i(j+1) <_{\mathcal{O}_i} w_i(j)$ for all i and j and for each pair (N_i, N_{i+1}) either

(1) (N_i, N_{i+1}) is an edge of $\mathcal{D}$ and $w_i(*) = w_{i+1}(0)$, or
(2) (N_{i+1}, N_i, σ) is a repeat in $\mathcal{R}$ and $w_i(*) = \sigma(w_{i+1}(0))$,

where $w_i(*)$ denotes the last letter of w_i. Intuitively, a trace τ records a sequence of vertical dependencies (w_i is a descending chain in $\mathcal{O}_i$) and horizontal dependencies (linking $\mathcal{O}_i$ and $\mathcal{O}_{i+1}$ as in (1), (2)) between ordinal variables. So τ roughly associates a non-increasing chain of ordinal variables with (a suffix of) a path. We say that a trace τ *progresses* at position i if $|w_i| > 1$ and that τ is *progressive* if it progresses at infinitely many positions. A path π of $\mathcal{P}$ is said to be *progressive* if there is a progressive trace $\tau = (N_0, w_0)(N_1, w_1) \cdots$ such that $N_0 N_1 \cdots$ is a suffix of π. Our syntactic *discharge condition (T-DC)* requires that all infinite paths of $\mathcal{P}$ are progressive. If an infinite path π is progressive, as witnessed by some τ, then it cannot be followed by an infinite run, as respecting the dependencies in τ would lead to an infinite decreasing chain of ordinals. Conversely, π can be extended to an infinite run if there is no progressive trace following π. Thus,

Theorem 2. *A pre-proof satisfies (R-DC) if and only if it satisfies (T-DC).*

We now turn to an automata-theoretic formulation of these conditions, which is more suitable for the practical application in a proof tool. Let $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ be a pre-proof. Since $\mathcal{O}_i \subseteq \mathcal{O}_{i+1}$ whenever (N_i, N_{i+1}) is an edge of $\mathcal{D}$, we may without loss of generality for condition (T-DC) restrict our attention to the *normal* traces of $\mathcal{P}$ making progress at most at repeat nodes. Based on this observation we construct two Büchi automata over the alphabet $\mathcal{R}$. Automaton B_1 recognises sequences of repeats that are traversed on paths of $\mathcal{P}$. Automaton B_2 recognises sequences of repeats that are from some point on connected through the ordinal variables they preserve. More precisely, let $s = R_0 R_1 \cdots$ be a sequence of repeats with $R_i = (M_i, N_i, \sigma_i)$ and $N_i(\Gamma_i \vdash_{\mathcal{O}_i} \Delta_i)$. Then $r = \bullet^j(\kappa_j, R_j, \kappa_{j+1})(\kappa_{j+1}, R_{j+1}, \kappa_{j+2})(\kappa_{j+2}, R_{j+2}, \kappa_{j+3}) \cdots$ is a run of B_2 on s if $j \geq 0$ and $\sigma_i(\kappa_{i+1}) \leq_{\mathcal{O}_i} \kappa_i$ for all $i \geq j$. It is accepting if $\sigma_i(\kappa_{i+1}) <_{\mathcal{O}_i} \kappa_i$ for infinitely many i. Note that if $u \in L(B_1)$ then r induces a normal trace, which is progressive precisely if r is accepting. Hence,

Theorem 3. *A pre-proof satisfies (T-DC) if and only if* $L(B_1) \subseteq L(B_2)$.

References

[1] T. Arts, M. Dam, L. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 38–41, 1998.
[2] M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140:95–114, 1998.
[3] M. Dam and D. Gurov. μ-calculus with explicit points and approximations. In *Fixed Points in Computer Science, FICS 2000*, 2000.
[4] L. Fredlund. *A Framework for Reasoning about Erlang Code.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.
[5] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
[6] D. Niwiński and I. Walukiewicz. Games for the μ-calculus. *Theoretical Computer Science*, 163:99–116, 1997.
[7] D. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976.
[8] U. Schöpp. Formal verification of processes. Master's thesis, University of Edinburgh, 2001.
[9] U. Schöpp and A. Simpson. Verifying temporal properties using explicit approximants: Completeness for context-free processes. In *FOSSACS '02*, Lecture Notes in Computer Science. Springer-Verlag, 2002. to appear.
[10] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

Christoph Sprenger, Swedish Institute of Computer Science (SICS), Kista, Sweden
E-mail address: sprenger@sics.se

Mads Dam, Royal Institute of Technology (KTH), Kista, Sweden
E-mail address: mfd@kth.se

Model Checking Knowledge and Fixpoints

N. V. Shilov, N. O. Garanina

Institute of Informatics Systems, Russian Academy of Sciences, Siberian Division,
6, Lavrentiev ave., 630090, Novosibirsk, Russia,
shilov@iis.nsk.su

Abstract. We study the model checking problem for finitely-generated one-way infinite synchronous/asynchronous traces for fusion of propositional logics of knowledge, common knowledge, branching time and fixpoint calculus.

1 Introduction and Motivation

Program logics are modal logics for reasoning about programs and systems. Traditionally they comprise dynamic logics, temporal logics and their variants with explicit fixpoints [14, 8]. A more recent addition to the family of the program logics is logic of knowledge [9].

At the same time the computer science community begins to understand better the importance of study of combined propositional program logics on traces. For example, several logics of this kind have been studied in the static analysis context [7]. The most powerful logic of these new logics is a so called reversible fixpoint calculus ($\overleftarrow{\mu}$C). This calculus can be considered as a "fusion" of a propositional fixpoint logic and a propositional logic of knowledge for the single agent. It is defined on two-way backward/forward (past/future) infinite traces.

We study the model checking problem for fusion of propositional logics of knowledge with propositional dynamic and state-based temporal logics extended by fixpoints. But (in contrast to $\overleftarrow{\mu}$C) we exploit one-way (future) traces. Nevertheless we hope, that our research, suggested technique, and presented results are a step towards a full-scale study of the reversible fixpoint calculus $\overleftarrow{\mu}$C.

In particular we examine the model checking problem for combinations of

(1) Elementary Propositional Dynamic Logic (EPDL),
(2) branching time Computation Tree Logic extended by actions (*Act*-CTL),
(3) the propositional μ-Calculus (μC)

with

(a) Propositional Logic of Knowledge for n agents (PLK$_n$),
(b) Propositional Logic of Common Knowledge for n agents (PLC$_n$).

We are especially interested in two extreme cases: Forgetful Asynchronous finite systems (FAS) vs. Synchronous finite systems with Perfect Recall (PRS). "Trace-based" means that possible worlds incorporate sequences of states, actions, etc. "Perfect recall" means that in every possible the history of the world is represented, while "forgetful" means that the information of this kind is not available in any world. "Synchronous" means that traces of different lengths can be distinguished, while "asynchronous" means that some traces with different lengths can be indistinguishable.

An importance of study combined logics in a framework of trace-based semantics in synchronous perfect recall settings rely upon characteristic of them as logics of knowledge acquisition. Let us illustrate this by the following False Coin Puzzle[1] $FCP(N, M)$:

> A set of coins consists of $(N+1)$ enumerated coins. The last coin is valid. A single coin with a number in $[1..N]$ is false, all other coins with numbers in $[1..N]$ are valid. All valid coins have one and the same weight while the false coin has a different weight. Is it possible to identify a false coin by balancing coins M times at most?

Let us refer a person who has to solve the problem as an agent. The agent knows neither a number in $[1..N]$ of a false coin, nor whether it is lighter or heavier than valid coins. The agent can make balancing queries and read balancing results after every query. A balancing query $b_{(L,R)}$ is an action. It consists in balancing two disjoint sets L and R of coins in $[1..N+1]$ on the left and on the right pan. There are three possible balancing results: $<$, $>$, and $=$, which mean that the left pan is lighter, heavier than or equal to the right pan respectively. Of course, there are initial states (marked by ini) which represent a situation in which no query has been done.

Let us summarize. The agent acts in a space $[1..N] \times \{l, h\} \times \{<, >, =, ini\}$. His/her admissible actions for a moving between states are all $b_{(L,R)}$ for disjoint $L, R \subset [1..N+1]$ with $|L| = |R|$. The agent has information about the last balancing only. The agent should acquire knowledge about the number of the false coin from a sequence which begins from the initial state and then consists of M queries and M corresponding results. The combination of Propositional Dynamic Logic and Propositional Logic of Knowledge seems to be a very natural framework for expressing this quest:

$$\underbrace{\langle \cup_{\dots} b_{(L,R)} \rangle \dots \langle \cup b_{(L,R)} \rangle}_{M \text{ times}} \Big(\bigvee_{f \in [1..N]} K(\text{a false coin number is } f) \Big),$$

where all non-deterministic choices $\cup_{\dots}$ range over all $L, R \subset [1..N+1]$ such that $L \cap R = \emptyset$ and $|L| = |R|$.

It is natural to assume that the agent remembers a sequence of balancing queries and a sequence of corresponding balancing results. Moreover, the agent

[1] A knowledge-based analysis of muddy children puzzle, synchronous attack and Byzantine agreement is very popular in literature on logic of knowledge, ex., [9].

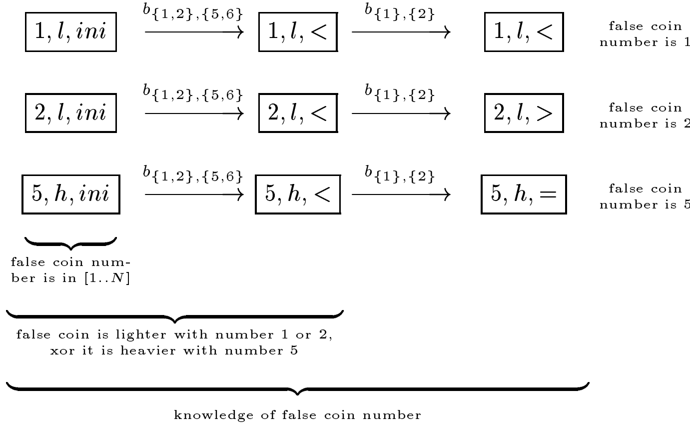


Fig. 1. Evolving knowledge in synchronous system with perfect recall

can not distinguish two sequences of states iff he/she made equal sequences of queries and read equal results along these sequences. We would like to refer this property as a synchronous perfect recall hypothesis. In general, a synchronous perfect recall hypothesis is a ability of an agent to remember sequences of executed actions and corresponding distinguishing items of information for intermediate states, and to distinguish sequences of states in accordance with these "memories". An agent knows some fact iff this fact holds in all indistinguishable situations. Fig. 1 illustrates knowledge evolution and acquisition under an assumption of synchronous prefect recall for $FCP(5,2)$.

The rest of the paper is organized as follows. A survey of background logics is presented in the section 2. Combined logics are introduced in the section 3. The expressive power, the model checking problem and decidability for these logics are examined (in the general settings) in the section 3 too. In the section 4 forgetful asynchronous systems are defined and algorithmic properties of the combined logics are investigated in forgetful asynchronous settings. In the section 5 we determine synchronous systems with perfect recall and study simulation power of synchronous perfect recall semantics. The model checking problem in synchronous perfect recall settings is explored in the section 6. In the section 7 we refine time bounds for the model checking problem for branching time logic of knowledge with actions (*Act*-CTL-K) in the synchronous environments with perfect recall, using techniques with k-trees. Related papers and research topics are discussed in the concluding section 8.

2 Background Logics

All logics which we are going to discuss are some propositional polymodal logics. Let $\{true, false\}$ be boolean constants, *Prp* and *Rlt* be disjoint finite alphabets of propositional variables and relational symbols. Formulae of our logics are

constructed from boolean constants, propositional variables, and connectives[2] $\neg$, $\wedge$, $\vee$ and necessity/eventuality modalities associated with relation symbols: if $r \in Rlt$ and ϕ is a formula then $([r]\phi)$ and $(\langle r\rangle\phi)$ are formulae[3]. Semantics of logics is defined in Kripke structures. A model M is a triple (D_M, I_M, V_M) where the domain D_M is a nonempty set of possible worlds, the interpretation I_M maps relation symbols into binary relations on D_M, and the valuation V_M maps propositional variables into subsets of D_M. For every model M and every formula ϕ, $M(\phi)$ is the set of all possible worlds which satisfy the formula ϕ in the model M and let us write $w \models_M \phi$ iff $w \in M(\phi)$. For propositional variables we have: $w \models_M p$ iff $w \in V_M(p)$. For boolean constants and connectives $\models_M$ is defined in the standard way. Semantics of modalities is:

- $w \models_M (\langle r\rangle\phi)$ iff $(w, w') \in I_M(r)$ and $w' \models_M \phi$ for some w',
- $w \models_M ([r]\phi)$ iff $(w, w') \in I_M(r)$ implies $w' \models_M \phi$ for every w'.

A very useful general notion is abstraction for polymodal logics. Let Φ be a set of formulae, $M_1 = (I_1, D_1)$ and $M_2 = (I_2, D_2)$ be two models and $g : D_1 \to D_2$ be a mapping. The model M_2 is called an abstraction [4] of the model M_1 with respect to formulae in Φ iff for every formula $\phi \in \Phi$ and every state $s \in D_1$ the following fact holds: $s \models_1 \phi \Leftrightarrow g(s) \models_2 \phi$.

A particular example of propositional polymodal logics is Propositional Logic of Knowledge for n agents (PLK$_n$, $n \geq 1$) [9]. In this framework a special terminology, a notation and models are used. The alphabet of relational symbols consists of agents $[1..n]$ (an integer number $n > 0$). Another notation for modalities is adopted: if $i \in [1..n]$ and ϕ is a formula then $(K_i\phi)$ and $(S_i\phi)$ are formulae, read as "agent i knows" and "agent i supposes", instead of $([i]\,\phi)$ and $(\langle i\rangle\,\phi)$. Agents are interpreted in models by equivalence relations: $I_M(i)$ is a symmetric, reflexive, and transitive binary relation for every $i \in [1..n]$ and every model $M = (D_M, I_M, V_M)$. Every model M is denoted as $(D_M, \overset{1}{\sim}, .. \overset{n}{\sim}, V_M)$ instead of (D_M, I_E, V_M) with $I_M(i) = \overset{i}{\sim}$ for every $i \in [1..n]$.

Common knowledge of several agents is a very important notion for distributed systems [9]. Let us define Propositional Logic with Common knowledge for n agents (PLC$_n$, $n \geq 1$). This time the alphabet of relational symbols consists of sets of agents $2^{[1..n]}$ (an integer number $n > 0$). Another notation for modalities is adopted: if $G \subseteq [1..n]$ and ϕ is a formula then $(C_G\phi)$ and $(J_G\phi)$ are formulae, read as "ϕ is common knowledge of agents in G" and "ϕ is joint hypothesis of agents in G", instead of $[G]\phi$ and $\langle G\rangle\phi$. Agents are interpreted in models by equivalence relations $\overset{1}{\sim}, .. \overset{n}{\sim}$ (as in PLK$_n$) and groups of agents are also interpreted by equivalence relations: $I_M(G)$ is a reflexive-transitive closure $\left(\bigcup_{i\in G} I_M(i)\right)^*$ for every $G \subseteq [1..n]$ and every model $M = (D_M, I_M, V_M)$. Now every model M is denoted as $(D_M, \overset{1}{\sim}, .. \overset{n}{\sim}, V_M)$ instead of (D_M, I_E, V_M) with $I_M(i) = \overset{i}{\sim}$ for every $i \in [1..n]$.

[2] $\to$ and $\leftrightarrow$ are admissible too but as standard abbreviations only

[3] which are read as "box/diamond r ϕ" or "after r always/sometimes ϕ" respectively

[4] g is called an abstraction mapping in this case.

Elementary Propositional Dynamic Logic (EPDL) [11] is another well-known particular propositional polymodal logic. This time the alphabet of relational symbols consists of action symbols *Act*. Another notation for modalities is adopted: if $a \in Act$ and ϕ is a formula then $([a]\phi)$ and $(\langle a\rangle\phi)$ are formulae, read as "*after a always* ϕ" and "*after a sometimes* ϕ", instead of $(\boxed{a}\phi)$ and $(\diamond\phi)$ respectively. But in contrast to PLK, no restriction on relations in models is imposed.

Another propositional polymodal logic, which we would like to define, is the basic propositional branching time temporal logic Computational Tree Logic (CTL) [8, 3, 5] extended by action symbols, which is referred as CTL with actions (*Act*-CTL). Syntax of *Act*-CTL exploits special constructors associated with action symbols: if $a \in Act$, ϕ and ψ are formulae, then $(\mathbf{AX}^a\phi)$, $(\mathbf{EX}^a\phi)$, $(\mathbf{AG}^a\phi)$, $(\mathbf{AF}^a\phi)$, $(\mathbf{EG}^a\phi)$, $(\mathbf{EF}^a\phi)$, $(\mathbf{A}\phi\mathbf{U}^a\psi)$, and $(\mathbf{E}\phi\mathbf{U}^a\psi)$[5] are formulae too. Semantics of these formulae coincides with semantics of CTL formulae, but a-trace are used instead of *next*-trace.

The propositional μ-Calculus (μC) [13, 14] is an extension of EPDL by fixpoints. Syntax of μC expands EPDL syntax: if $p \in Prp$ and ϕ is a formula with positive instances[6] of p, then $(\mu p.\phi)$ and $(\nu p.\phi)$ are formulae, read as "mu/nu p ϕ" and "the least/greatest fixpoint p of ϕ" respectively. Semantics of μ and ν is defined in the following way. For every model M, every set $S \subseteq D_M$, every $p \in Prp$ and every formula ϕ without negative instances of p, $M_{S/p}$ is a model which agrees with M everywhere but $V_M(p) = S$ and $\lambda S.\ M_{S/p}(\phi)\ :\ S \mapsto M_{S/p}(\phi)$ is a monotonous non-decreasing mapping on 2^{D_M}. In accordance with Tarski-Knaster theorem [20], this function has the least and the greatest fixed points (with respect to set inclusion $\subseteq$): $\mu(\lambda S.\ M_{S/p}(\phi))$ and $\nu(\lambda S.\ M_{S/p}(\phi))$. In these settings

- $w \models_M (\mu p.\phi)$ iff $w \in \mu(\lambda S.\ M_{S/p}(\phi))$ (or iff $w \in S$ for every $S \subseteq M_{S/p}(\phi)$),
- $w \models_M (\nu p.\phi)$ iff $w \in \nu(\lambda S.\ M_{S/p}(\phi))$ (or iff $w \in S$ for some $S \supseteq M_{S/p}(\phi)$).

3 Algorithmic Problems for Combined Logics

We are going to define a combined Propositional Logic with fixpoints and Common knowledge (μPLC). Let $[1..n]$ be a set of agents ($n > 0$), and *Act* be a finite alphabet of action symbols. Syntax of this logic admits all knowledge modalities C_G, and J_G for $G \subseteq [1..n]$, all action modalities $[a]$ and $\langle a\rangle$ for $a \in Act$, all fixpoints μp and νp for $p \in Prp$ (which are applicable to formulae with non-negative instances of p). Semantics is defined in terms of satisfiability and is a combination of formal semantics for μC and PLC. Models for this combined logic are called environments (it is the single terminology specifics). An environment is a tuple $E = (D_E, \overset{1}{\sim}, .. \overset{n}{\sim}, I_E, V_E)$ such that $(D_E, \overset{1}{\sim}, .. \overset{n}{\sim}, V_E)$ is a model for

[5] **A** is read as "for all futures", **E** – as "for some futures", **X** – as "next time", **G** – as "always", **F** – as "sometime", **U** – as "until", and a sup-index ^{a} is read as "in a-run"

[6] i.e., in range of even amount of negations (otherwise it is negative instance)

PLC$_n$ and (D_E, I_E, V_E) is a model for μC. For every environment E and every formula ϕ, $E(\phi)$ is the set of all possible worlds which satisfy formula ϕ in E. For every environment E, every possible world w and every formula ϕ let us write $w \models_E \phi$ iff $w \in E(\phi)$. Other combined logics EPDL-K, EPDL-C, *Act*-CTL-K, *Act*-CTL-C and μPLK can be defined in the same way.

Proposition 1. *All expressibilities between listed logics are presented below:*

$n = 1$:

$$\begin{array}{ccccc} EPDL\text{-}C_n & < & Act\text{-}CTL\text{-}C_n & < & \mu PLC_n \\ \| & & \| & & \| \\ EPDL\text{-}K_n & < & Act\text{-}CTL\text{-}K_n & < & \mu PLK_n \end{array}$$

$n > 1$:

$$\begin{array}{ccccc} EPDL\text{-}C_n & < & Act\text{-}CTL\text{-}C_n & < & \mu PLC_n \\ \vee & & \vee & & \| \\ EPDL\text{-}K_n & < & Act\text{-}CTL\text{-}K_n & < & \mu PLK_n \end{array}$$

All expressibilities have linear complexity. All non-expressibilities can be justified in finite environments representing finite games of two players.

The model checking is a problem to check whether $w \models_E \phi$ for an input world w, a finite environment E, and a formula ϕ of combined logics EPDL-K$_n$, EPDL-C$_n$, ACT-CTL-K$_n$, ACT-CTL-C$_n$, μPLK$_n$, or μPLC$_n$. Let m_E be the overall complexity $(d_E + r_E)$ of the finite environment E, presented as a finite graph, where d_E and r_E be the amount of nodes and the amount of edges (including knowledge). If ϕ is a formula then let f_ϕ be a size of ϕ. if ϕ is a formula then let an alternating fixpoint depth a_ϕ be 1 plus the amount of alternations in nesting μ and ν with respect to the syntactical dependence and positive/negative instances.

Proposition 2. *There exists a model checking algorithm for worlds of finite environments which runs*

- *in the linear time $O(m \times f)$ for formulae of fixpoint-free logics with (common) knowledge EPDL-K$_n$, EPDL-C$_n$, Act-CTL-K$_n$, and Act-CTL-C$_n$;*
- *in the exponential time $O(m \times f) \times (\frac{d \times f}{a})^{a-1}$. for formulae of logics with fixpoints and (common) knowledge μPLK$_n$ and μPLC$_n$.*

The above proposition is proved by reducing the model checking problem for all listed logics in finite models to the model checking problem for μC in finite models.

The decidability problem is to check whether there exist an environment E and a world w such that $w \models_E \phi$ for an input formula ϕ of a combined logic EPDL-K$_n$, EPDL-C$_n$, ACT-CTL-K$_n$, ACT-CTL-C$_n$, μPLK$_n$, or μPLC$_n$.

Proposition 3. *(1) EPDL-K$_n$ is $PSPACE$-complete, but (2) EPDL-C$_n$, Act-CTL-K$_n$, Act-CTL-C$_n$, μPLK$_n$, and μPLC$_n$ are $EXPTIME$-complete.*

The first part of the proposition is proved by reduction of EPDL-K$_n$ to PLK$_m$ (where $m = (n + 2|Act|)$) which is in $PSPACE$ (PLK$_m$ can be reduced to the polymodal **S5** in linear time). The second part follows from reduction of

the listed logics to the propositional μ-Calculus with converse (μC^-) which is $EXPTIME$-complete [22].

Let us summarize propositions 1, 2, and 3:

Theorem 1. *The expressive power (E), model checking upper bounds (M) and decidability complexities (D) of the combined logics EPDL-K, EPDL-C, Act-CTL-K, Act-CTL-C, μPLK and μPLC in general settings enjoy the following properties:*

$$\overbrace{\overset{linear\ M}{EPDL\text{-}C_n} \overset{E}{<} \overset{linear\ M}{Act\text{-}CTL\text{-}C_n} \overset{E}{<} \overset{exp.\ M}{\mu PLC_n}}^{EXPTIME\text{-}complete\ D}$$

$$\begin{array}{ccc} n{=}1 \quad n{>}1 & n{=}1 \quad n{>}1 & \\ \parallel \quad \vee & \parallel \quad \vee & \parallel \end{array}$$

$$\underbrace{\overset{linear\ M}{EPDL\text{-}K_n}}_{PSPACE\text{-}complete\ D} \overset{E}{<} \underbrace{\overset{linear\ M}{Act\text{-}CTL\text{-}K_n} \overset{E}{<} \overset{exp.\ M}{\mu PLK_n}}_{EXPTIME\text{-}complete\ D}$$

4 Forgetful Asynchronous Systems

In this section we examine trace-based forgetful asynchronous environments generated from background environments. Let E be an environment $(D_E, \overset{1}{\sim}, .. \overset{n}{\sim}, I_E, V_E)$. A trace-based Forgetful Asynchronous environment generated by E is another environment $FAS(E) = (D_{FAS(E)}, \overset{1}{\underset{fas}{\approx}}, .. \overset{n}{\underset{fas}{\approx}}, I_{FAS(E)}, V_{FAS(E)})$, where

- $D_{FAS(E)}$ is D_E^+, i.e. the set of all non-empty sequences of states;
- for every $i \in [1..n]$ and for all $wrld', wrld'' \in D_{FAS(E)}$,
  $wrld' \overset{i}{\underset{fas}{\approx}} wrld''$ iff $w' \overset{i}{\sim} w''$,
  where w' and w'' are the last elements in $wrld'$ and $wrld''$ respectively;
- for every $a \in Act$ and for all $wrld', wrld'' \in D_{FAS(E)}$,
  $(wrld', wrld'') \in I_{FAS(E)}(a)$ iff[7] $wrld'' = wrld'^\wedge w''$, $(w', w'') \in I_E(a)$, where w' and w'' are the last elements in $wrld'$ and $wrld''$ respectively;
- for every $p \in Prp$ and for every $wrld \in D_{FAS(E)}$,
  $wrld \in V_{FAS(E)}(p)$ iff $wrld_{|wrld|} \in V_E(p)$.

Let us exploit some special notation in study of forgetful asynchronous systems. Let D be a set of elements. For this D let

- *last-D* $: D^+ \to D$ be a function which maps every non-empty finite sequence of elements of D to the last element of this sequence;
- *D-past* $: D \to 2^{D^+}$ be a function which maps every element $d \in D$ to the set of all finite sequences with this last element d.

Both functions can be extended on power-sets in the standard manner:

- *last-D* $: 2^{D^+} \to 2^D$ be a function which maps every set of non-empty finite sequences to the set of last elements of these sequences;

[7] where $^\wedge$ is concatenation of words

– $D\text{-}past : 2^D \to 2^{D^+}$ be a function which maps every set of elements to the set of all finite sequences with these last elements.

We do not distinguish these functions and their extensions.

Proposition 4. *For every formula ϕ of the combined Propositional Logic with fixpoints and Knowledge μPLK, for every environment E the following holds:*

– $FAS(E)(\phi) = D_E\text{-}past(E(\phi))$,
– $E(\phi) = last\text{-}D_E(FAS(E)(\phi))$

(i.e., the formula ϕ holds on a non-empty finite sequence of worlds in the corresponding forgetful asynchronous environment $FAS(E)$ iff ϕ holds on the last world of the sequence in the background environment E).

For "simple" formulae[8] a proof of the proposition follows from definitions, but for fixpoints the theory of an abstract interpretation [6, 7] is used. This proposition implies the next one.

Proposition 5. *Every environment E is an abstraction of the corresponding forgetful asynchronous environment $FAS(E)$ with respect to formulae of the combined Propositional Logic with fixpoints and Common knowledge μPLC. The corresponding abstraction function maps every non-empty finite sequence of states to the last element of the sequence.*

This proposition together and the theorem 1 imply the following theorem.

Theorem 2. *The expressive power, the model checking problem and decidability of the combined logics EPDL-K_n, EPDL-C_n, Act-CTL-K_n, Act-CTL-C_n, μPLK_n and μPLC_n in forgetful asynchronous settings are equivalent to the expressive power, the model checking problem and decidability of listed logics in general settings.*

Let us remark, that if agents do not distinguish two traces iff their last states are indistinguishable (as in the original settings) and both traces are generated by some (may be different) sequences of actions, propositions 4, 5 and theorem 2 hold also.

5 Synchronous Systems with Perfect Recall

We are especially interested in trace-based perfect recall synchronous environments generated from background finite environments. Let E be an environment $(D_E, \overset{1}{\sim}, .. \overset{n}{\sim}, I_E, V_E)$. A trace-based Perfect Recall Synchronous environment generated by E is another environment $PRS(E) = (D_{PRS(E)}, \overset{1}{\sim}_{prs}, .. \overset{n}{\sim}_{prs}, I_{PRS(E)}, V_{PRS(E)})$, where

[8] i.e., propositional variables and combinations, formulae which begin with modalities

- $D_{PRS(E)}$ is the set of all pairs $(wrld, acts)$ where $wrld \in D_E^+$, $acts \in Act^*$, $|wrld| = |acts| + 1$, and $(wrld_j, wrld_{j+1}) \in I_E(acts_j)$ for every $j \in [1..|acts|]$;
- for every $i \in [1..n]$ and for all $(wrld', acts')$, $(wrld'', acts'') \in D_{PRS(E)}$, $(wrld', acts') \overset{i}{\underset{prs}{\sim}} (wrld'', acts'')$ iff $acts' = acts''$ and $wrld'_j \overset{i}{\sim} wrld''_j$ for every component j;
- for every $a \in Act$ and for all $(wrld', acts')$, $(wrld'', acts'') \in D_{PRS(E)}$, $((wrld', acts'), (wrld'', acts'')) \in I_{PRS(E)}(a)$ iff[9] $acts'^\wedge a = acts''$, and $wrld'' = wrld'^\wedge w''$, $(w', w'') \in I_E(a)$, where w' and w'' are last elements in $wrld'$ and $wrld''$ respectively;
- for every $p \in Prp$ and for every $(wrld, acts) \in D_{PRS(E)}$, $(wrld, acts) \in V_{PRS(E)}(p)$ iff $wrld_{|wrld|} \in V_E(p)$.

Simulation power of the class of synchronous environments with perfect recall generated by finite environments is very high (as follows from the propositions 6 and 7 below). We utilize these propositions for study of the model checking problem in perfect recall settings.

The following proposition is inspirited by [15].

Proposition 6.
Let 'next' be a fixed action symbol. There exists a PLC_2 formula ϕ which enjoys the following property: for every machine $\mathcal{M} \in \mathcal{CL}$ there exists a finite environment E such that for every $m \geq 0$, and every input α for $\mathcal{M}$ there exists a sequence of worlds $wrld$ with $|wrld| = m$ and

$$\mathcal{M} \text{ halts on } \alpha \text{ utilizing } m \text{ cells} \Leftrightarrow (wrld, next^{(m-1)}) \models_{PRS(E)} \phi.$$

The formula ϕ can be constructed in the constant time, the environment E – in time $O(|\mathcal{M}|)$, the sequence $wrld$ – in time $O(|\alpha|)$.

The Weak Second-Order logic of 1 Successor $WS(1)S$ can be defined in different manners [2, 18, 17, 19, 1]. The following proposition is inspirited by [16]:

Proposition 7.
Let 'next' be a fixed action symbol. For every formula ϕ of $WS(1)S$, there exist a finite environment E and a set of worlds T such that for every $m \geq 0$
(1) *there is a one-to-one correspondence vt between evaluations in $[0..(m-1)]$ of variables in ϕ and next-traces of the length m in E that finish in T,*
(2) *there is a translation tr of subformulae of ϕ into formulae of Act-CTL-K_2 with the single action symbol 'next'*
such that:
for every evaluation ev in $[0..m-1]$ of variables occurring in ϕ and every subformula ψ of ϕ the following holds:

$$ev \models \psi \Leftrightarrow (vt(ev), next^{(m-1)}) \models_{PRS(E)} tr(\psi).$$

The environment E and the set T can be constructed in time $exp(|\phi|)$, the correspondence vt – in time $O(m \times |\phi|)$, and the translation tr – in time $O(|\phi|)$.

[9] where $^\wedge$ is concatenation of words

6 Model Checking Knowledge Acquisition in Synchronous Systems with Perfect Recall

In this section we examine the model checking problem for combined logics EPDL-K$_n$, EPDL-C$_n$, *Act*-CTL-K$_n$, *Act*-CTL-C$_n$, μPLK$_n$, and μPLC$_n$ in perfect recall synchronous environments generated from finite environments. Namely we study the complexity of the set $CHECK(L) \equiv \{(E\ ,\ (wrld, acts)\ ,\ \phi)$: E is a finite environment, $(wrld, acts) \in D_{PRS(E)}$, ϕ is a formula of L, and $(wrld, acts) \models_{PRS(E)} \phi\}$, where L is a particular combined logic.

Proposition 8.
For all $n > 1$ and $Act \neq \emptyset$, $CHECK(EPDL\text{-}C_n)$ is $PSPACE$-complete.

In accordance with the proposition 6, every polynomial computation of every Turing machine $\mathcal{M}$ can be represented as the model checking problem for some fixed formula ϕ of PLC$_2$ in the environment E (which can be constructed in time $O(|\mathcal{M}|)$) on some $(wrld, acts) \in D_{PRS(E)}$ (of a polynomial length). It implies that $CHECK$(EPDL-C$_n$) is $PSPACE$-hard. $CHECK$(EPDL-C$_n$) is in $PSPACE$ due to the opportunity to check formulae of EPDL-C$_n$ in PRS environments by alternating Turing machines with the polynomial space and bounded amount of alternations, and the generalised Savich's theorem for PSPACE-alternating computations[4].

If $E = (D_E, \overset{1}{\sim}, \dots \overset{n}{\sim}, I_E, V_E)$ is a finite background environment presented as a finite graph, then let m_E be an overall complexity $(d_E + r_E)$, where d_E and r_E be amount of worlds in D_E and total amount of pairs of worlds in I_E. If $(wrld, acts) \in D_{PRS(E)}$ then let $l_{(wrld,acts)}$ be $|wrld| = |acts| + 1$. If ϕ is a formula then let f_ϕ be the size of ϕ. A complexity measure for triples $(E\ ,\ (wrld, acts)\ ,\ \phi)$, where E is a finite environment, $(wrld, acts) \in D_{PRS(E)}$, and ϕ is a formula, is $(m_E + l_{(wrld,acts)} + f_\phi)$.

Proposition 9.
For all $n > 1$ and $Act \neq \emptyset$, $CHECK(Act\text{-}CTL\text{-}K_n)$ is decidable with the upper and the lower bounds $\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}^{O(t)}$, where t is the overall complexity of the input triple.

It is known that the Weak Second-Order logic of 1 Successor $WS(1)S$ is decidable with a non-elementary lower bound $\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}^{O(f)}$, where f is the size of the input formula [17] (see also [18, 19, 1]). In accordance with the proposition 7, $WS(1)S$ can be encoded in the model checking problem for CTL-K$_2$ in perfect recall synchronous settings. Complexity of this encoding is exponential. These arguments imply a non-elementary lower bound for $CHECK$(*Act*-CTL-K$_n$).
A principle decidability of $CHECK$(*Act*-CTL-K$_n$) with a non-elementary upper bound $\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}^{O(t)}$ is based on reduction of *Act*-CTL-K$_n$ to the Chain Logic with Equal-length predicate CLE [21].

Proposition 10.
CHECK(Act-CTL-C$_n$), CHECK(μPLK$_n$), and CHECK(μPLC$_n$) are undecidable for all $n > 1$ and $Act \neq \emptyset$.

Proof. In accordance with the proposition 6, every computation of every Turing machine $\mathcal{M}$ which exploits a fixed space can be represented as the model checking problem for some fixed formula ϕ of PLC$_2$. Hence the halting problem for Turing machines can be represented as the model checking problem for the CTL-C$_2$ formula $\mathbf{EF}\phi$. It implies undecidability of $CHECK(Act$-CTL-C$_n)$. In accordance with the proposition 1, it implies undecidability of $CHECK(\mu$PLK$_n)$ and $CHECK(\mu$PLC$_n)$. The proof is **over**.

7 Formulae with a bounded knowledge depth

Let us fix a finite environment $E = (D_E, \stackrel{1}{\sim}, .. \stackrel{n}{\sim}, I_E, V_E)$. Thus the corresponding synchronous environment with perfect recall $PRS(E) = (D_{PRS(E)}, \stackrel{1}{\sim}_{\text{prs}}, .. \stackrel{n}{\sim}_{\text{prs}}, I_{PRS(E)}, V_{PRS(E)})$ is also fixed. Let us call it PRS. The knowledge depth of a formula is the maximal nesting of knowledge operators in that formula. Let Act-CTL-K$_n^k$ be a sublogics of Act-CTL-K$_n$ with a bounded knowledge depth $k \geq 0$. Naturally, Act-CTL-K$_n = \bigcup_{k \geq 0} Act$-CTL-K$_n^k$.

For every integer $k \geq 0$ we define by a mutual recursion the set $\mathcal{T}_k$ of *k-trees over E*, and the set $\mathcal{F}_k$ of *forests of k-trees over E*. Let $\mathcal{T}_0$ be the set of all tuples of the form $(w, \emptyset, ...\emptyset)$ where w is a world and the number of copies of the empty set $\emptyset$ is equal to the number of agents n. Once $\mathcal{T}_k$ has been defined, let $\mathcal{F}_k$ be the set of all subsets of $\mathcal{T}_k$. Let $\mathcal{T}_{k+1}$ be the set of all tuples of the form $(w, U_1, ...U_n)$, where w is a world and $U_i \neq \emptyset$ is in $\mathcal{F}_k$ for each $i \in \{1..n\}$. Let us denote $\bigcup_{k \geq 0} \mathcal{T}_k$ by $\mathcal{T}$. Let $exp(a, b)$ be the following function: $exp(a, 0) = a$, $exp(a, b) = a \times 2^{exp(a,b-1)}$ when $b > 0$.

Proposition 11. *Let $k \geq 0$ be an integer and E be a finite environment for n agents with d states. The number of k-trees over E C_k is $\leq \frac{exp(n \times d, k)}{n}$ and the number of nodes in every $(k+1)$-tree over E is $< C_k^2$ (for $n < d$).*

Let $(wrld, acts)$ be a world of $PRS(E)$. Knowledge available in this world can be represented as an infinite sequence $tree_0(wrld, acts)$ ... $tree_k(wrld, acts)$ ... where each $tree_k(wrld, acts)$, $k \geq 0$, is a k-tree which is defined as follows. Let $tree_0(wrld, acts)$ be $(wrld_{|wrld|}, \emptyset, ...\emptyset)$ and for every $k \geq 0$ let $tree_{k+1}(wrld, acts)$ be

$$\Big(wrld_{|wrld|},\ \{tree_k(wrld', acts') : (wrld', acts') \stackrel{1}{\sim}_{\text{prs}} (wrld, acts)\},\ ...\ \{tree_k(wrld', acts') : (wrld', acts') \stackrel{n}{\sim}_{\text{prs}} (wrld, acts)\}\Big).$$

Let us define knowledge update functions for k-trees. The similar functions has been used in [15] to provide an algorithm for the model checking problem for formulae of PLK_n in synchronous environments with perfect recall. Let D_E and I_E be a domain and an interpretation of relation symbols from $Acts$ in the environment E. For every number $k \geq 0$, $act \in Acts$ and $i \in \{1..n\}$ the functions $G_k^{act} : \mathcal{T}_k \times D_E \to \mathcal{T}_k$ and $H_{k,i}^{act} : \mathcal{F}_k \times D_E \to \mathcal{F}_k$, are defined

by induction on k and mutual recursion. Let $G_0^{act}(tr, wrld) = (wrld, \emptyset, ...\emptyset)$ iff $(root(tr), wrld) \in I_E(act)$. Once G_k^{act} has been defined, we can define for each $i \in \{1..n\}$ the function $H_{k,i}^{act}$ by setting $H_{k,i}^{act}(U, wrld)$, to be the set of k-trees $G_k^{act}(tr, wrld')$ where $tr \in U$ and $wrld' \overset{i}{\sim} wrld$. Using the functions $H_{k,i}^{act}$, $i \in \{1..n\}$, we can define G_{k+1}^{act} by setting $G_{k+1}^{act}((wrld, U_1, ...U_n),\ wrld')$ to be $(\ wrld'\ , H_{k,1}^{act}[U_1, wrld'],\ ...\ H_{k,n}^{act}[U_n, wrld']\ )$ iff $(wrld, wrld') \in I_E(act)$.

The following proposition is inspirited by [15]:

Proposition 12. *For every $k \geq 0$, every $act \in Act$, every finite environment E, every $(wrlds, acts) \in D_{PRS(E)}$, every $wrld \in D_E$, and every $act \in Act$, the following incremental knowledge update property holds*[10]*:*

$$tree_k((wrlds, acts)^\wedge\{(wrld, act)\}) \ = \ G_k^{act}(tree_k(wrlds, acts),\ wrld).$$

Let Act be an alphabet of action symbols and $[1..n]$ be agents. Let Act^{+n} be Act extended by new action symbols associated with agents $[1..n]$.

A natural translation of formulae of Act-CTL-K$_n$ to the formulae of Act^{+n}-CTL is simple: just replace every instance of K_i and S_i by corresponding $\mathbf{AX}^i$ and $\mathbf{EX}^i$ respectively ($i \in [1..n]$). For every formula ϕ of Act-CTL-K$_n$ let us denote by ϕ^{+n} the resulting formula of Act^{+n}-CTL. This translation is supported by the corresponding transformation of environments for Act-CTL-K$_n$ to the models for Act^{+n}-CTL: the environment $E = (D_E, \overset{1}{\sim}, ...\overset{n}{\sim}, I_E, V_E)$ can be presented as the model $E^{+n} = (D_E, I_E^{+n}, V_E)$, where I_E^{+n} is equal to I_E on action symbols, but $I_E^{+n}(i) = \overset{i}{\sim}$ for every agent $i \in [1..n]$. The following proposition is straightforward.

Proposition 13. *$E(\phi) = E^{+n}(\phi^{+n})$ for every environment E and every formula ϕ of Act-CTL-K$_n$. In particular, $PRS(E)(\phi) = (PRS(E))^{+n}(\phi^{+n})$ for every environment E and every formula ϕ of Act-CTL-K$_n$.*

Now we define a class of associated models based on k-trees. For every $k \geq 0$ let $TR_k(E)$ be the following model $(D_{TR_k(E)}, I_{TR_k(E)}, V_{TR_k(E)})$ for Act^{+n}-CTL:

- $D_{TR_k(E)}$ is the set of all 0-,... k-trees over E for n agents;
-
  - for every $act \in Act$, $I_{TR_k(E)}(act) = \{(tr', tr'') \in D_{TR_k(E)}$: $tr'' = G_j^{act}(tr', wrld)$ for some $j \in [0..k]$ and some $wrld \in D_E\ \}$;
  - for every $i \in [1..n]$, $I_{TR_k(E)}(i) = \{(tr', tr'') \in D_{TR_k(E)}$: $tr'' \in U_i$ and $tr' = (wrld, U_1, ...U_n)$ for some $wrld \in D_E\ \}$;
- $V_{TR_k(E)}(p) = \{tr : root(tr) \in V_E(p)\}$ for every $p \in Prp$.

The following proposition can be proved by induction on formulae structure with help of the proposition 12.

Proposition 14. *For all integers $k \geq 0$ and $n \geq 1$, for every environment E, the model $TR_k(E)$ is an abstraction of the model $(PRS(E))^{+n}$ with respect to formulae of Act^{+n}-CTL which correspond to the formulae of Act-CTL-K$_n$ with knowledge depth k at most. The corresponding abstraction function maps every trace to the k-tree of this trace.*

[10] where $^\wedge$ is concatenation of words

In accordance with the proposition 2, the complexity of model checking of an *Act*-CTL formula in a finite model is $O(m \times f)$, where m is the overall model complexity and f is the complexity of a formula. But the cited proposition has been proved under the assumption that all worlds of the model have some constant complexity. This assumption does not hold for models where worlds are k-trees, since (in accordance with the proposition 11) the complexity of these trees is a non-elementary function of k and n. We should add the corresponding world-complexity factor to complexity bound. In these settings the above propositions 2, 11 and 14 lead to the following proposition.

Proposition 15. *For every integers $k \geq 1$ and $n \geq 1$, for every finite environment E, for every formula ϕ of Act-CTL-K–n with knowledge depth k at most, the model checking problem for synchronous environment with perfect recall $PRS(E)$ and the formula ϕ is decidable with the upper bound*

$$O\Big(f \times \frac{exp(n \times d, k) \times (exp(n \times d, k-1))^2}{n^3}\Big),$$

where f is the size of the formula, d is the amount of states in D_E, and the function $exp(a, b)$ is defined by induction as follows: $exp(a, 0) = a$ and $exp(a, b+1) = a \times 2^{exp(a,b)}$.

The above propositions 8, 9, 10 and 15 lead to the following theorem.

Theorem 3.
For all $n > 1$ and $Act \neq \emptyset$, the model checking problem for synchronous finitely generated environments with perfect recall for

- *EPDL-C_n is $PSPACE$-complete;*
- *Act-CTL-K_n is decidable with the non-elementary upper and lower bounds (which linearly depend on a formula size and non-elementary depend on amount of states, agents and a knowledge depth);*
- *Act-CTL-C_n, μPLK_n, and μPLC_n are undecidable.*

8 Related papers and Conclusion

In the paper algorithmic problems for several combinations of propositional program logics have been studied from the theoretical point of view. The focus of the paper is model checking problem in synchronous perfect recall settings for logics, which combine knowledge, actions, and fixpoints. The paper contributes to model checking researches for logics of knowledge acquisition and extends the results of the paper [16], where the model checking problem in synchronous perfect recall settings has been examined for fusion of logics PLK and PLC with the Propositional Logic of Linear Time (PLLT). It has been proven in the cited paper that the problem is
(1) undecidable for PLLT-C_n;
(2) non-elementary decidable for PLLT-K_n;
(3) $PSPACE$-complete for UNTIL-free PLLT-C_n.

A tree-like data structure for the model checking linear time and knowledge with bounded nesting has been suggested in the paper [16]. This data structure is very convenient for presentation of knowledge evolution and update. It comprises the trees which depth is equal to knowledge nesting. The paper [16]demonstrated that the model checking problem for PLLT-K$_n$ in synchronous perfect recall semantics can be reduced to the emptiness of a Büchi automata, whose inputs are infinite sequences of these trees. We develop a similar data structure but exploit an abstraction and a reduction to the model checking problem for the variant of CTL and models where states are trees.

A natural related problem is: whether an automatic model checking is feasible for PLLT-K$_n$ and *Act*-CTL-K$_n$? Some experience with tree-like data structures (which are similar to the data structure mentioned in the previous paragraph) is reported in [10]. In the cited experimental research, **(1)** input finite environments are specified on a Multi Agent System Language, **(2)** input PLLT-K$_n$ formulae should not have negative/positive instances of knowledge modalities K_i/S_i for any agent $i \in [1..n]$, **(3)** a model checking engine is finite-state PLLT model checker SMV [3, 5].

Another related paper is [12]. It has studied the decidability problem for combinations of temporal logics PLLT and CTL with logics PLK and PLC in synchronous perfect recall and forgetful settings. In particular, it has demonstrated completeness of the problem in the following complexity classes

	PRS	FAS
CTL-C$_n$, $n \geq 2$	Π^1_1	$EXPTIME$
CTL-K$_n$, $n \geq 2$	nonelementary time	$EXPTIME$
CTL-K$_1$ = CTL-C$_1$	doubly-exponentially time	$EXPTIME$

Our paper extends the above table on some other combined logics in forgetful asynchronous settings.

References

1. Börger E., Grädel E., Gurevich Yu. *The Classical Decision Problem.* Springer, 1997.
2. Büchi J.R. *On a decision method in restricted second order arithmetic.* Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science, Stanford Univ. Press, Stanford, 1960, p.1-11.
3. Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J. *Symbolic Model Checking: 10^{20} states and beyond.* Information and Computation, v.98, n.2, 1992, p.142-170.
4. Chandra A.K., Kozen D.C., Stockmeyer L.J. *Alternation.* Journal of the ACM, v.28, n.1, 1981, p.114-133.
5. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.
6. Cousot P., Cousot R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* Proceedings of 4-th PoPL, ACM Press, 1977, p.238-252.
7. Cousot P., Cousot R. *Temporal Abstract Interpretation.* Proceedings of 27-th PoPL, ACM Press, 2000, p.12-25.

8. Emerson E.A. *Temporal and Modal Logic. Handbook of Theoretical Computer Science*, v.B, Elsevier and The MIT Press, 1990, 995-1072.
9. Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995.
10. Garanina N.O. *Experiments with Model Checking Knowledge and Time.* Ms. Thesis. Mechanics and Mathematics Department of Novosibirsk State University, 2001 (in Russian).
11. Harel D. *First-Order Dynamic Logic.* Lecture Notes in Computer Science, v.68, 1979.
12. Halpern J.Y., Vardi M.Y. The complexity of Reasoning About Knowledge and Time. Proc. of Symp. on STOC, 1986, p.304-315.
13. Kozen D. *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.
14. Kozen D., Tiuryn J. *Logics of Programs.* Handbook of Theoretical Computer Science, v.B, Elsevier and The MIT Press, 1990, 789-840.
15. van der Meyden R. *Common Knowledge and Update in Finite Environments.* Information and Computation, v.140, n.2, 1998, p.115-157.
16. van der Meyden R., Shilov N.V. *Model Checking Knowledge and Time in Systems with Perfect Recall.* Lecture Notes in Computer Science, v.1738, 1999, p.432-445.
17. Meyer A.R. *The inherent complexity of theories of ordered sets.* Proc. of the Int. Congress of Math., Vancouver, v.2, Canadian Mathematical Congress, 1974, 477-482
18. Rabin M.O. *Decidability of second order theories and automata on infinite trees.* Trans. Amer. Math. Soc., v.141, 1969, p.1-35.
19. Rabin M.O. *Decidable Theories.* in Handbook of Mathematical Logic, ed. Barwise J. and Keisler H.J., North-Holland Pub. Co., 1977, 595-630.
20. Tarski A. *A lattice-theoretical fixpoint theorem and its applications.* Pacific J. Math., v.5, 1955, p.285-309.
21. Thomas W. *Infinite trees and automaton-definable relations over ω-words.* Theoretical Computer Science, v.103, 1992, p.143-159.
22. Vardi M.Y. *Reasoning about the past with two-way automata'.* Lecture Notes in Computer Science, v.1443, 1998, p.628-641.

Strong Next-Time Operators for Multiple-Valued μ-Calculus

Benet Devereux
Department of Computer Science,
University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: benet@cs.toronto.edu

April 20, 2002

1 Introduction

Multiple-valued logics [2] provide an interesting alternative to classical boolean logic for modeling and reasoning about systems. By allowing additional truth values, they support the explicit modeling of uncertainty and disagreement.

In order to do temporal reasoning over multiple-valued systems, we must extend a classical temporal logic to the multiple-valued case. For instance, the branching-time temporal logic CTL is a fragment of the modal μ-calculus [5]: it has conjunction, disjunction, negation, two next-time operators, weak (EX) and strong (AX) [4], and several additional temporal operators described as least or greatest fixpoints. For example, the property EFφ, "eventually φ may become true", is the least fixpoint:

$$\mathrm{EF}\varphi \triangleq \mu Z.\,(\varphi \vee (\mathrm{EX}Z))$$

and AGφ, "φ holds everywhere", is the greatest fixpoint:

$$\mathrm{AG}\varphi \triangleq \nu Z.(\varphi \wedge (\mathrm{AX}Z))$$

The necessary conditions for finite-time convergence of fixpoint computations are the finiteness of the state-space, the finiteness of the set of state valuations, and the monotone increasing property of EX and AX.

The intuitive idea of the weak next-time operator EXφ is "there exists a successor state where φ holds". If S is the (finite) state-space, and R the system's transition relation, then for any state s:

$$(\mathrm{EX}\varphi)(s) \triangleq \exists s' \in S \cdot (R(s, s') \wedge \varphi(s'))$$

The sense of AXφ, however, involves causation: "if there is a transition to state s', then φ holds there", and is defined using implication:

$$(\mathrm{AX}\varphi)(s) \triangleq \forall s' \in S \cdot (R(s, s') \to \varphi(s'))$$

In this work, we focus on possible multiple-valued generalizations of AX. We begin with a review of multiple-valued model-checking, described in greater detail elsewhere [3]. We choose to use finite sets of truth values, in order to guarantee finite-time convergence of fixpoints. Following that, we state the conditions for implication operators which both correspond to intuition, and allow a monotone increasing AX to be defined, which is needed to prove the existence of fixpoints. As an example, we choose three implications for a particular multiple-valued logic, and discuss the relationships between them, and with EX. Finally we show a structural condition on both multiple-valued models and implications that allows us to guarantee that AX is stronger than EX, while still permitting flexibility in modelling partial systems.

2 Background

Let $\mathcal{L}$ be a finite set of logic values, partially ordered by truth degree. We use sets of truth values that have the structure of a *De Morgan algebra* [6] $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \neg)$: a distributive lattice with an antimonotonic and involute negation operator $\neg$: if $x \sqsubseteq y$, then $\neg y \sqsubseteq \neg x$, and $x = \neg\neg x$. The least element of the lattice is denoted $\bot$, and the highest element $\top$. Figure 1(a) shows the truth

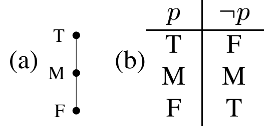


Figure 1: (a) Lattice for 3-valued logic, and (b) table for its De Morgan negation.

ordering for a 3-valued logic, and Figure 1(b) defines the De Morgan negation. Here $\top$=T and $\bot$=F.

A χKripke structure [3] is a tuple (S, s_0, R, I, A, L) where:

1. S is a finite state space, s_0 an initial state;
2. $L = (\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \neg)$ is a De Morgan algebra;
3. $R : S \times S \to \mathcal{L}$ is a (valued) transition relation, assigning degrees of truth to transitions between states;
4. A is a finite set of state variables;
5. $I : S \times A \to \mathcal{L}$ is the interpretation function assigning values to variables in states.

Some examples of χKripke structures are shown in Figure 2. The form of R is constrained to ensure that states have successors. In classical Kripke structures, this totality condition is formalized as $\forall s \cdot \exists s' \cdot R(s, s')$. There are three possible ways to generalize this condition for χKripke structures:

$$\begin{array}{ll} \forall s \cdot \exists s' \cdot R(s, s') = \top & \text{(strong totality)} \\ \forall s \cdot (\bigsqcup_{s' \in S} R(s, s')) = \top & \text{(join totality)} \\ \forall s \cdot \exists s' \cdot R(s, s') \neq \bot & \text{(weak totality)} \end{array}$$

All of these definitions collapse to ordinary totality in the classical case; and in the three-valued case, strong totality and join totality coincide. For example, the χKripke structures of Figure 2 satisfy weak totality, but not strong.

A temporal logic formula is interpreted in a given χKripke structure as a map $S \to \mathcal{L}$. For $p \in A$, and formulas φ and ψ, some of the connectives are defined as follows:

$$\begin{array}{rcl} p(s) & \triangleq & I(s, p) \\ (\varphi \wedge \psi)(s) & \triangleq & \varphi(s) \sqcap \psi(s) \\ (\mathrm{EX}\varphi)(s) & \triangleq & \bigsqcup_{s' \in S}(R(s, s') \wedge \varphi(s)) \end{array}$$

For example, EXφ in state s of Figure 2(b) is computed as $(R(s,t) \sqcap \varphi(t)) \sqcup (R(s,u) \sqcap \varphi(u)) = (\mathrm{M} \sqcap \mathrm{M}) \sqcup (\mathrm{M} \sqcap \mathrm{F}) = \mathrm{M} \sqcup \mathrm{F} = \mathrm{M}$. The other connectives are defined similarly, and fixpoints have the standard definition.

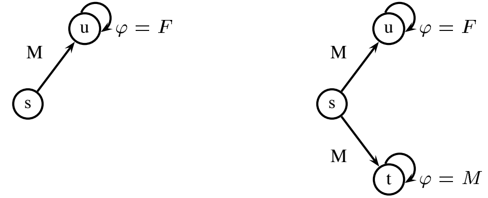


Figure 2: Example multiple-valued Kripke structures.

3 Multi-Valued Implications

A common approach to defining implication for multiple-valued logics is through residuation of a monoid operation on the logic values [1]. We take a simpler approach: defining the criteria for an implication, and then exploring the space of candidates for applicability.

We propose the following criteria for a useful implication operator $\to$, for all $x, y, z \in \mathcal{L}$:

$$\begin{array}{ll} (\bot \to x) = \top & \text{(vacuity)} \\ \text{if } x \sqsubseteq y \text{ then } (z \to x) \sqsubseteq (z \to y) & \text{(monotonicity)} \\ (\top \to x) \sqsubseteq x & \text{(sub-identity))} \\ (\top \to \top) = \top, (\bot \to \top) = \top & \text{(reduction to classical)} \\ (\bot \to \bot) = \top, (\top \to \bot) = \bot & \end{array}$$

The reason for the vacuity condition is as follows: suppose, for some state $s \in S$, AXφ has a computed value. Now we adjoin some new state t to S, with no transition from s to t ($R(s, t) = \bot$). The value of AXφ in s should not change!

Monotonicity guarantees that AX, as defined using $\to$, is also monotonic, which is required for the existence of a fixpoint. The sub-identity rule formalizes the intuition that the statement $\top \to x$ indicates that "under any conditions, x holds", and it does not make sense for this expression to be any *more* true than the truth value of x. Finally, we want the implication to behave classically when only classical truth-values are used, since if we define a system using only $\top$ and $\bot$ as values, then analysis should yield the same results as classical model-checking.

Under these conditions, a 3-valued logic yields 33 possible implication operators, so we restrict our attention to three: *material implication*, defined as $x \to_M y = \neg x \sqcup y$; *Gödel implication*, where $x \to_G y = \top$ if $x \sqsubseteq y$, and y

$\to_M$	T	M	F
T	T	M	F
M	T	M	M
F	T	T	T

$\to_G$	T	M	F
T	T	M	F
M	T	T	F
F	T	T	T

$\to_L$	T	M	F
T	T	M	F
M	T	T	M
F	T	T	T

Figure 3: Material ($\to_M$), Gödel ($\to_G$), and Łukasiewicz ($\to_L$) implication tables for the 3-valued logic of Figure 1.

otherwise; and *Łukasiewicz implication*, which is defined on an n-valued, totally ordered logic by mapping truth values into the fractions k/n. The tables for these implications are shown in Figure 3.

4 Strong Next-time Operators

In this section, we define three different AX operators using multi-valued implications, and give some of their properties. We also discuss the ordering between these operators: for one operator to be stronger than another corresponds to the intuitive notion that it is a more conservative definition of "in all next states".

For an implication operator $\to$, we define a strong next-time operator as follows:

$$(\mathrm{AX}\varphi)(s) \triangleq \prod_{s' \in S} (R(s,s') \to \varphi(s'))$$

Using the three implication operators, we define three next-time operators, AX_M, AX_G, and AX_L. Since the monotonicity rule holds for all three implications, the AX operators are monotonic as well:

Proposition 1 *If* $\varphi \sqsubseteq \psi$, *then* $\mathrm{AX}_A\varphi \sqsubseteq \mathrm{AX}_A\psi$, *for* $A \in \{M, G, L\}$.

Any ordering between implication operations is inherited by the AX operators defined using them:

Proposition 2 *For any* $\to_A, \to_B, \mathrm{AX}_A, \mathrm{AX}_B$, *with* $A, B \in \{M, G, L\}$, *and any formula* φ:

$$\text{if } \forall x, y \cdot (x \to_A y) \sqsubseteq (x \to_B y) \text{ then } \mathrm{AX}_A\varphi \sqsubseteq \mathrm{AX}_B\varphi$$

By inspection, it is easy to see that the following relationship holds:

$$\mathrm{AX}_M\varphi \sqcup \mathrm{AX}_G\varphi = \mathrm{AX}_L\varphi \qquad (1)$$

In general, for n-valued logics, Equation 1 does *not* hold; a simple counterexample in the 5-valued totally-ordered logic can be found.

5 Relationships Between Next-time Operators

We are interested in how AX relates to EX, and also to its dual $\neg\mathrm{EX}\neg$. Under strong totality, $\mathrm{AX}\varphi \sqsubseteq \mathrm{EX}\varphi$; under weak totality, this ordering does not necessarily hold. In the 3-valued case, join totality is equivalent to strong totality, but whether join totality guarantees the correct ordering remains an open question in the general case.

We state the result for strong totality more formally:

Proposition 3 *For all states* s, *if strong totality holds, then* $\mathrm{AX}\varphi \sqsubseteq \mathrm{EX}\varphi$.

Proof:
By strong totality, for any state s, there is $s' \in S$ such that $R(s,s') = \top$. Then:

$$(\mathrm{AX}\varphi)(s) \sqsubseteq (R(s,s') \to \varphi(s')) \sqsubseteq \varphi(s')$$

by sub-identity. In turn:

$$\varphi(s') = R(s,s') \sqcap \varphi(s') \sqsubseteq \mathrm{EX}\varphi$$

and thus the desired property holds:

□

Looking at the proof of Proposition 3, we can see that a weaker condition is possible. In order to define this in a simple manner, we need to state that if our next-time operators are correctly ordered in all propositional variables p and their negations $\neg p$, then they are also correctly ordered for *any* temporal logic formula:

Lemma 1 *Let* EX *and* AX *be monotonic operators. If, for all* $p \in A$, $\mathrm{AX}p \sqsubseteq \mathrm{EX}p$, *and* $\mathrm{AX}\neg p \sqsubseteq \mathrm{EX}\neg p$, *then:*

$$\mathrm{AX}\varphi \sqsubseteq \mathrm{EX}\varphi$$

for all temporal logic formulas φ.

We can also define a weaker condition ensuring $\mathrm{AX}\varphi \sqsubseteq \mathrm{EX}\varphi$. We call this condition *sufficient* totality, and it is defined relative to the AX operator being used.

Theorem 1 *If for all $p \in A$, and $s \in S$:*

$$\exists t, u \in S \cdot R(s,t) \rightarrow p(t) \sqsubseteq R(s,u) \sqcap p(u)$$

with $t \neq u$, and

$$\exists t, u \in S \cdot R(s,t) \rightarrow \neg p(t) \sqsubseteq R(s,u) \sqcap \neg p(u)$$

then $\mathrm{AX}\varphi \sqsubseteq \mathrm{EX}\varphi$ in all states.

Proof:
Direct, and by Lemma 1. □

In Figure 2(a), we have a χKripke structure which is only weakly total. It is sufficiently total for AX_G, but not for either of the other strong next-times: here $\mathrm{EX}\varphi = \mathrm{F}$ and $\mathrm{AX}_M\varphi = \mathrm{AX}_L\varphi = \mathrm{M}$. In this case, our analysis seems to say that, on the one hand, there is no transition to a state where φ holds; but, on the other hand, maybe φ holds in any next state! By contrast, the structure in Figure 2(b) is sufficiently total for all three AX operators. In this case, $\mathrm{AX}_M\varphi = \mathrm{AX}_L\varphi = \mathrm{EX}\varphi = \mathrm{M}$, and $\mathrm{AX}_G\varphi = \mathrm{F}$.

We conclude with observations about the connection between $\neg\mathrm{EX}\neg$ and AX operators. By definition, $\mathrm{AX}_M\varphi = \neg\mathrm{EX}\neg\varphi$, and thus:

Proposition 4 *$\neg EX\neg$ is stronger than AX_L:*

$$\neg\mathrm{EX}\neg\varphi \sqsubseteq \mathrm{AX}_L\varphi$$

and incomparable with AX_G.

We might say that AX_L is the most "optimistic" strong next-time operator.

6 Summary and Future Work

This work reports on our first investigations of the space of strong next-time operators for multiple-valued temporal logic. We have stated axioms for implications that can be used to define an AX operator, and discussed the relationships among candidate operations and their relationship to EX. Finally, we have given a weaker totality condition for χKripke structures which is parameterized in the choice of AX.

In the future, we plan to generalize this work to any (finite) multiple-valued logic, and discover more general relationships between classes of multiple-valued implications, and thus between the AX operators they define. As well, we hope to find applications in verification for the different AX operators, and incorporate them into our verification framework [3].

7 Acknowledgements

We would like to thank Marsha Chechik for helping improve the presentation of results in this paper, and also Wendy MacCaull and Arie Gurfinkel for advice and discussions.

References

[1] T.S. Blyth and M.F. Janowitz. *Residuation Theory*. Pergamon Press, Oxford, 1972.

[2] L. Bolc and P. Borowik. *Many-Valued Logics*. Springer-Verlag, 1992.

[3] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking ". CSRG Technical Report 448, Department of Computer Science, University of Toronto, 1997.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] D. Kozen. "Results on the Propositional Mu-Calculus". *Theoretical Computer Science*, pages 333–354, Dec. 1983.

[6] H. Rasiowa. *An Algebraic Approach to Non-Classical Logics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1978.

On Two Letters versus Three

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
kozen@cs.cornell.edu

Abstract

If A is a context-free language over a two-letter alphabet, then the set of all words obtained by sorting words in A and the set of all permutations of words in A are context-free. This is false over alphabets of three or more letters. Thus these problems illustrate a difference in behavior between two- and three-letter alphabets.

The following problem appeared on a recent exam at Cornell:

> Let Σ be a finite alphabet with a fixed total ordering on the letters. For a string $x \in \Sigma^*$, let $\mathsf{sort}\ x$ be the string obtained by sorting the letters in increasing order. For example, if $a < b < c$, then $\mathsf{sort}\ abacbaa = aaaabbc$. For $A \subseteq \Sigma^*$, let $\mathsf{sort}\, A = \{\mathsf{sort}\, x \mid x \in A\}$. Of the following three statements, two are false and one is true. Give counterexamples for the two false ones and a proof of the true one.
>
> (i) If A is regular, then so is $\mathsf{sort}\ A$.
>
> (ii) If A is context-free, then so is $\mathsf{sort}\ A$.
>
> (iii) If A is context-sensitive, then so is $\mathsf{sort}\ A$.

One might also ask the same questions about $\mathsf{perm}\ A$, the set of all permutations of words in A.

Of course, it is (i) and (ii) that are false, since

$$\mathsf{sort}\ (abc)^* = \mathsf{perm}\ (abc)^* \cap a^*b^*c^* = \{a^nb^nc^n \mid n \geq 0\}.$$

Interestingly, (ii) is true for both sort and perm over a two-letter alphabet. This is quite surprising: whereas a two-letter alphabet is exponentially more succinct than

a one-letter alphabet, one does not normally think of a break in behavior between two- and three-letter alphabets. In many applications, three letters (or for that matter any fixed finite number of letters) can be coded into two with only a linear loss of efficiency. Not so, apparently, in this case.

In this short note we give an elementary proof of these facts. The proof for sort is a fairly straightforward construction relying on Parikh's theorem and Pilling normal form, but the proof for perm is somewhat more involved, requiring a bit of linear algebra over integer lattices.

Let $\Sigma = \{a_1, \ldots, a_d\}$, and let $\pi : \Sigma^* \to \mathbb{N}^d$ be the Parikh map

$$\pi(x) \stackrel{\text{def}}{=} (\#a_1(x), \ldots, \#a_d(x)),$$

where $\#a(x)$ is the number of a's in x. Define

$$\begin{aligned} \pi(A) &\stackrel{\text{def}}{=} \{\pi(x) \mid x \in A\} \\ \mathsf{perm}\ A &\stackrel{\text{def}}{=} \pi^{-1}(\pi(A)) \\ \mathsf{sort}\ A &\stackrel{\text{def}}{=} \mathsf{perm}\ A \cap a_1^* \cdots a_d^*. \end{aligned}$$

Theorem 1 *For $d \leq 2$, if A is a context-free language, then so are* perm A *and* sort A.

This is trivial for $d = 1$ and false for $d \geq 3$. The interesting case is $d = 2$.

Lemma 1 *It suffices to prove Theorem 1 for A regular. When manipulating regular expressions, we can also use the commutativity axiom $xy = yx$.*

Proof. This is a consequence of Parikh's theorem (the commutative image of any context-free language is the commutative image of some regular set), observing that the definitions of perm A and sort A depend only on the commutative image $\pi(A)$ of A. □

Lemma 2 *It suffices to prove Theorem 1 for A of the form $xy_1^* \cdots y_k^*$, where $x, y_1, \ldots, y_k \in \Sigma^*$.*

Proof. Under commutativity, every regular expression is equivalent to a sum of expressions of this form. This is known as Pilling normal form (see [1]). □

Here is a direct construction for sort A. This result will also follow from the result for perm A by intersecting with a^*b^*, but the proof for perm A is somewhat harder.

Without loss of generality, assume A is of the form of Lemma 2. Let $m = \#a(x)$, $n = \#b(x)$, $m_i = \#a(y_i)$, and $n_i = \#b(y_i)$, $1 \le i \le k$. A context-free grammar for sort A is

$$\begin{array}{rcl} S & \to & a^m T_1 b^n \\ T_i & \to & a^{m_i} T_i b^{n_i} \mid T_{i+1}, \quad 1 \le i \le k-1 \\ T_k & \to & a^{m_k} T_k b^{n_k} \mid \varepsilon. \end{array}$$

For perm A, we will need to use some linear algebra on integer lattices.

Lemma 3 *Let $y_1, \ldots, y_n$ be nontrivial. The following are equivalent:*

(i) $\pi(y_1), \ldots, \pi(y_n)$ *are linearly dependent over* $\mathbb{Q}$.

(ii) $\pi(y_1), \ldots, \pi(y_n)$ *are linearly dependent over* $\mathbb{Z}$.

(iii) *There exists a partition of $y_1, \ldots, y_n$ into two nonempty disjoint sets $y_1, \ldots, y_k$ and $y_{k+1}, \ldots, y_n$ (renumbering if necessary) and coefficients $a_i \in \mathbb{N}$, $1 \le i \le n$, such that not all $a_i = 0$, $1 \le i \le k$, not all $a_i = 0$, $k+1 \le i \le n$, and $\prod_{i=1}^{k} y_i^{a_i} = \prod_{i=k+1}^{n} y_i^{a_i}$.*

The property in (iii) regarding the vanishing of the coefficients follows from the observation that we cannot have $\prod_{i=1}^{k} y_i^{a_i} = 1$ with $a_i \in \mathbb{N}$ unless all $a_i = 0$.

The following lemma gives a stronger version of Pilling normal form.

Lemma 4 (Conway [1, Theorem 2, p. 92]) *Any regular subset of $\mathbb{N}^d$ can be written as a sum of terms of the form $xy_1^* \cdots y_n^*$ with $\pi(y_1), \ldots, \pi(y_n)$ linearly independent over $\mathbb{Q}$.*

Proof. Suppose $\pi(y_1), \ldots, \pi(y_n)$ are linearly dependent. Let $\prod_{i=1}^{k} y_i^{a_i} = \prod_{i=k+1}^{n} y_i^{a_i}$ with $a_i \in \mathbb{N}$, $1 \le i \le n$, not all $a_1, \ldots, a_k = 0$ and not all $a_{k+1}, \ldots, a_n = 0$. Using the Kleene algebra identities

$$\begin{array}{rcl} y^* & = & (\sum_{i=0}^{n-1} y^i)(y^n)^* \\ x_1^* \cdots x_n^* & = & (x_1 \cdots x_n)^* (\sum_{i=1}^{k} \prod_{\substack{1 \le j \le k \\ j \ne i}} x_j^*) \end{array}$$

(the second one requires commutativity), rewrite $y_1^* \cdots y_k^*$ as $\alpha (y_1^{a_1})^* \cdots (y_k^{a_k})^*$, where

$$\alpha = \prod_{i=1}^{k} \sum_{j=0}^{a_i - 1} y_i^j,$$

and then $(y_1^{a_1})^* \cdots (y_k^{a_k})^*$ as

$$(y_1^{a_1} \cdots y_k^{a_k})^* (\sum_{i=1}^{k} \beta_i),$$

where

$$\beta_i = \prod_{j \neq i} (y_j^{a_j})^*, \quad 1 \leq i \leq k.$$

Note α contains no starred terms, so it can be expressed as a finite sum of products of the y_i. Then $y_1^* \cdots y_n^*$ can be written as a sum of terms of the form

$$u(y_1^{a_1} \cdots y_k^{a_k})^* \beta_i y_{k+1}^* \cdots y_n^*.$$

Now we can replace $\prod_{i=1}^{k} y_i^{a_i}$ with $\prod_{i=k+1}^{n} y_i^{a_i}$ to get

$$u(y_{k+1}^{a_{k+1}} \cdots y_n^{a_n})^* \beta_i y_{k+1}^* \cdots y_n^*.$$

Since this is contained in $y_1^* \cdots y_n^*$, we have $u \in y_1^* \cdots y_n^*$, thus

$$u(y_{k+1}^{a_{k+1}} \cdots y_n^{a_n})^* \beta_i y_{k+1}^* \cdots y_n^* \quad \subseteq \quad u \beta_i' y_{k+1}^* \cdots y_n^*,$$

where

$$\beta_i' = \prod_{j \neq i} y_j^*, \quad 1 \leq i \leq k.$$

Thus the original term $xy_1^* \cdots y_n^*$ can be written as a sum of terms of the same form but with one fewer starred y_i.

We can continue decreasing the number of starred terms inductively until the y_i are linearly independent. □

By this lemma, to prove Theorem 1 for the case $\mathsf{perm}\ A$, it suffices to consider A of the form xu^* or xu^*v^*, where $\pi(u)$ and $\pi(v)$ are linearly independent. Note that the dimension is at most two since we are over a two-letter alphabet. We can get rid of the x without loss of generality by $|x|$ applications of the following lemma:

Lemma 5 *Let $a \in \Sigma$. If A is context-free, then so is $\{xay \mid xy \in A\}$. It follows that if* $\mathsf{perm}\ A$ *is context-free, then so is* $\mathsf{perm}\ aA$*, since* $\mathsf{perm}\ aA = \{xay \mid xy \in \mathsf{perm}\ A\}$.

Proof. Consider a Chomsky normal form grammar for $\mathsf{perm}\ A$. For every nonterminal X, add a new nonterminal X_a, which is meant to generate all the strings that X generates but with an extra a somewhere. For every production $X \to YZ$, add the productions $X_a \to Y_aZ \mid YZ_a$. For every production $X \to b$, add the productions $X_a \to ba \mid ab$. For every production $X \to \varepsilon$, add the production $X_a \to a$. The new start symbol is S_a, where S was the old start symbol. □

Now we show that $\mathsf{perm}\ u^*v^*$ is context-free. (We leave the easier case, $\mathsf{perm}\ u^*$, as an exercise for the interested reader.) Suppose $\#a(u) = u_1$, $\#b(u) = u_2$, $\#a(v) = v_1$, $\#a(v) = v_2$; thus $\pi(u) = (u_1, u_2)$ and $\pi(v) = (v_1, v_2)$. Arrange $\pi(u)$ and $\pi(v)$ in a 2×2 matrix

$$A \stackrel{\text{def}}{=} \begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix}$$

with positive determinant $\Delta = u_1v_2 - u_2v_1 > 0$. (The sign of the determinant is determined by the orientation of u and v; exchange if necessary to make it positive.) The adjoint (pseudo-inverse) of A is

$$A' \stackrel{\text{def}}{=} \begin{bmatrix} v_2 & -v_1 \\ -u_2 & u_1 \end{bmatrix}$$

and satisfies the property

$$AA' \;=\; A'A \;=\; \begin{bmatrix} \Delta & 0 \\ 0 & \Delta \end{bmatrix}.$$

Now we give a nondeterministic one-way automaton with an integer counter accepting $\mathsf{perm}\ u^*v^*$. The machine actually keeps three counters, c_1, c_2, c_3, but the counters c_1 and c_3 hold only finitely many values and can be stored in the finite control. The counter c_2 holds an integer. We can simulate this with a pushdown automaton with a single-letter stack, keeping the sign in the finite control.

The automaton starts in the state $c_1 = c_2 = c_3 = 0$ and takes the following actions on each input symbol: on input a,

$$\begin{aligned} c_1 &:= (c_1 + v_2) \bmod \Delta \\ c_2 &:= c_2 - u_2 \\ c_3 &:= \min(c_3 + 1, v_1) \end{aligned}$$

and on input b,

$$\begin{aligned} c_1 &:= (c_1 - v_1) \bmod \Delta \\ c_2 &:= c_2 + u_1. \end{aligned}$$

In addition, it may nondeterministically choose to take the following *reset step* whenever $c_3 = v_1$ without reading an input symbol.

$$\begin{aligned} c_2 &:= c_2 - \Delta \\ c_3 &:= 0. \end{aligned}$$

Thus after scanning a prefix y of the input string,

$$\begin{aligned} c_1 &= (v_2\#a(y) - v_1\#b(y)) \bmod \Delta \\ c_2 &= -u_2\#a(y) + u_1\#b(y) - \Delta q, \end{aligned} \tag{1}$$

where q is the number of resets that have occurred, and c_3 contains the number of a's seen since the last reset, up to a maximum of v_1. The automaton accepts if $c_1 = c_2 = 0$.

Now we show that the automaton accepts $\mathsf{perm}\ u^*v^*$. For $s, t \in \mathbb{Z}^2$, note that $As = t$ iff $A't = \Delta s$. Applying this with $s = (p, q)$ and $t = (\#a(x), \#b(x))$, we have

$$\begin{aligned} \#a(x) &= u_1 p + v_1 q \\ \#b(x) &= u_2 p + v_2 q \end{aligned} \tag{2}$$

iff

$$\begin{aligned} v_2\#a(x) - v_1\#b(x) &= \Delta p \\ -u_2\#a(x) + u_1\#b(x) &= \Delta q. \end{aligned} \tag{3}$$

This implies that the following are equivalent:

(i) $x \in \mathsf{perm}\ u^*v^*$

(ii) there exist $p, q \in \mathbb{N}$ such that $x \in \mathsf{perm}\ u^p v^q$

(iii) there exist $p, q \in \mathbb{N}$ satisfying either of the equivalent conditions (2) or (3).

Now suppose $x \in \mathsf{perm}\ u^*v^*$ and condition (iii) holds with $p, q \in \mathbb{N}$. Let the automaton choose to perform the reset step at its earliest opportunity while scanning x (i.e., as soon as the counter c_3 reaches v_1), but only q times. It has the opportunity to perform a reset at least q times, since by (2), $\#a(x) \geq v_1 q$. By (1), the final values of c_1 and c_2 are

$$\begin{aligned} (v_2\#a(x) - v_1\#b(x)) \bmod \Delta &= 0 \\ -u_2\#a(x) + u_1\#b(x) - \Delta q &= 0, \end{aligned}$$

respectively, so the machine accepts.

Conversely, suppose the machine accepts. Let q be the number of times the reset occurred. By (1), there exists $p \in \mathbb{Z}$ such that (3) holds, and we need only show that $p \geq 0$. Since the reset occurred q times, we have $\#a(x) \geq v_1 q$. Then

$$\begin{aligned} u_1 v_2 p &= \Delta p + u_2 v_1 p \\ &= v_2 \#a(x) - v_1 \#b(x) + u_2 v_1 p \\ &\geq v_2 v_1 q - v_1(u_2 p + v_2 q) + u_2 v_1 p \\ &= 0. \end{aligned}$$

But $u_1 v_2 = \Delta + u_2 v_1 > 0$, therefore $p \geq 0$.

Acknowledgements

Thanks to Juris Hartmanis, Jon Kleinberg, and Lillian Lee for valuable comments. This work was supported in part by NSF grant CCR-0105586 and by ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

[1] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

Kleenean Semimodules and Linear Languages

Hans Leiß

Centrum für Informations-
und Sprachverarbeitung
Universität München
Oettingenstr. 67
D-80538 München, Germany
leiss@cis.uni-muenchen.de

Abstract. Viewing a Kleenean algebra K as an idempotent semiring with an iteration * as axiomatized by D. Kozen[4], we consider left semimodules $(A, +, 0)$ over K. Kleenean semimodules are those where each linear equation $x = a + r : x \leq x$ has a least solution, where : is the product from $K \times A$ to A. The linear context-free languages can be viewed as a Kleenean semimodule A over a Kleenean algebra R of binary regular word relations. Thus, the simultaneous linear fixed-point operator μ on languages can be reduced to iteration * on R and the scalar product :.

Definition 1. Let $\mathcal{K} = (K, +, \cdot, ^*, 0, 1)$ be a Kleenean algebra and $\mathcal{A} = (A, +, 0)$ an idempotent commutative semigroup. $\mathcal{M} = (\mathcal{K}, \mathcal{A}, :)$ is a *left semimodule* over $\mathcal{K}$, if : is a mapping from $K \times A$ to A, such that for all $r, s \in K$ and $a, b \in A$:

$$r : 0 = 0 \tag{1}$$
$$r : (a + b) = r : a + r : b, \tag{2}$$
$$0 : a = 0, \tag{3}$$
$$(r + s) : a = r : a + s : a, \tag{4}$$
$$1 : a = a, \tag{5}$$
$$(r \cdot s) : a = r : (s : a), \tag{6}$$

The semimodule $\mathcal{M}$ is a *Kleenean semimodule*, if we also have

$$r : b + a \leq b \;\Rightarrow\; r^* : a \leq b \tag{7}$$

for all $r \in K$ and $a, b \in A$, using $x \leq y :\iff x + y = y$. We often omit $\mathcal{K}$ from $\mathcal{M}$ and call $(\mathcal{A}, :)$ or $(A, +, 0, :)$ a $\mathcal{K}$-semimodule.

The distributivity properties imply that the scalar multiplication : is monotone in both arguments. In particular, one has $r^n : a \leq r^* : a$. The *dynamic algebras* of Kozen[3] are like semimodules over a *-continuous Kleenean algebra: instead of (7) one there has $r^* : a = \bigsqcup\{ r^n : a \mid n \in \mathbb{N} \}$, and $\mathcal{A}$ is a Boolean algebra.

Proposition 1. *In any Kleenean semimodule $(\mathcal{K}, \mathcal{A}, :)$, for any $r \in K$ and $a \in A$ the element $r^* : a$ is the least solution of the right-linear equation $x = r : x + a$.*

Example 1. If $\mathcal{K}$ is a Kleenean algebra, then $\mathcal{M} = (\mathcal{K}, \mathcal{A}, \cdot)$ with $\mathcal{A} = (K, +, 0)$ is a Kleenean semimodule over $\mathcal{K}$, where 0, $+$ and $\cdot$ are the corresponding operations from $\mathcal{K}$.

In particular, let $\mathcal{L}^n_\Sigma$ be the Kleenean algebra of all n-ary relations between words over the alphabet Σ, where $+$ is union and $\cdot$ is the lifting of componentwise concatenation to relations. Then $\mathcal{L}^n_\Sigma$ is a Kleenean module over itself. For $n = 3$ and $A = \{(a, b, c)\}$, the least solution of $X = A : X + 1$ is $A^* = \{ (a^m, b^m, c^m) \mid m \in I\!N \}$.

By $Reg(\mathcal{L}^n_\Sigma)$, the n-ary *regular word relations over* Σ, we mean the Kleenean subalgebra of $\mathcal{L}^n_\Sigma$ generated by the finite (or: singleton) elements of $\mathcal{L}^n_\Sigma$.

Definition 2. The *linear* and the *regular* expressions over the finite sets Γ, Δ of constants are defined by the grammar

$$p, q \;:=\; 0 \mid c \mid (p + q) \mid r : p \tag{8}$$
$$r, s \;:=\; 0 \mid 1 \mid d \mid (r + s) \mid (r \cdot s) \mid r^*, \tag{9}$$

where c ranges over Γ and d ranges over Δ.

In a semimodule $\mathcal{M} = (\mathcal{K}, \mathcal{A}, :)$ with given elements $c^\mathcal{A} \in A$, $d^\mathcal{K} \in K$ for each $c \in \Gamma$ and $d \in \Delta$ these expressions are interpreted as elements $p^\mathcal{A} \in A$ and $r^\mathcal{K} \in K$, using the corresponding operations of $\mathcal{M}$; in particular $(r : p)^\mathcal{A} = r^\mathcal{K} : p^\mathcal{A}$.

Example 2. The following two *standard interpretations* are actually continuous Kleenean modules. (They are also reducts of Boolean modules in the sense of Brink e.a.[1]). We only give $\cdot$ and $:$, since then * follows from $R^* = \bigcup\{ R^n \mid n \in I\!N \}$.

(i) (C. S. Peirce, ~1870) Let $\mathcal{K} = (2^{M\times M}, \cup, \circ, ^*, \emptyset, 1_M)$ be the algebra of binary relations on a set $M \neq \emptyset$, $\mathcal{A} = (2^M, \cup, \emptyset)$ the algebra of subsets of M, and $\cdot$ and $:$ be the relation composition and *inverse image* of a set under a relation:

$$R \circ S = \{ (m, n) \mid \exists k \in M \; (R(m, k) \wedge S(k, n)) \} \tag{10}$$
$$R : A = \{ m \mid \exists a \in A \; R(m, a) \}. \tag{11}$$

(ii) (J. Gruska, 1971) Let $\mathcal{K} = (2^{\Sigma^*\times\Sigma^*}, \cup, \cdot, ^*, \emptyset, \{(\varepsilon, \varepsilon)\})$ consist of the binary word-relations over the alphabet Σ and $\mathcal{A} = (2^{\Sigma^*}, \cup, \emptyset)$ of the word sets (languages), and let $\cdot$ and $:$ be the (pointwise) *infixation* of a word relation (resp. a set) into a word relation:

$$R \cdot S = \{ (v_1 w_1, w_2 v_2) \mid (v_1, v_2) \in R, (w_1, w_2) \in S \} \tag{12}$$
$$R : A = \{ v_1 w v_2 \mid (v_1, v_2) \in R, w \in A \}. \tag{13}$$

Example 3. Not every semimodule over a Kleenean algebra is a Kleenean semimodule.

Let $\mathcal{M}_{n,m}(\mathcal{A})$ be the $n \times m$ matrices with entries from $\mathcal{A}$.

Theorem 1. *Let $(\mathcal{A}, :)$ be a (Kleenean) semimodule over the Kleenean algebra $\mathcal{K}$. Then $\mathcal{M}_{n,1}(\mathcal{A})$ is a (Kleenean) semimodule over the Kleenean algebra $\mathcal{M}_{n,n}(\mathcal{K})$, under the scalar multiplication*

$$\begin{pmatrix} r_{1,1} & \cdots & r_{1,n} \\ \vdots & \ddots & \vdots \\ r_{1,1} & \cdots & r_{1,n} \end{pmatrix} : \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \Sigma_{j=1}^n r_{1,j} : a_j \\ \vdots \\ \Sigma_{j=1}^n r_{n,j} : a_j \end{pmatrix}.$$

Definition 3. Let $\mathcal{A} = (A, +, 0, :)$ be a Kleenean semimodule over $\mathcal{K}$. The *submodule of $\mathcal{A}$ generated by $B \subseteq A$* consists of the set

$$\langle B \rangle^{\mathcal{A}} := \bigcap \{\, M \mid B \subseteq M \subseteq A,\ M + M \subseteq M,\ K : M \subseteq M \,\}$$

together with the restrictions of the operations from $\mathcal{A}$ to $\langle B \rangle^{\mathcal{A}}$.

Clearly, $\langle B \rangle^{\Sigma}$ consists of the *linear combinations* $\Sigma_{i=1}^{m}\ r_i : b_i$ of elements $b_i \in B$ with coefficients $r_i \in K$.

Definition 4. A *system of linear equations* (or a *linear context-free grammar*) over the alphabet Σ is an equation system

$$\begin{array}{ccc} x_1 & = & r_1(x_1, \ldots, x_m) \\ \vdots & \vdots & \vdots \\ x_m & = & r_m(x_1, \ldots, x_m) \end{array} \tag{14}$$

where each r_i is a sum of words w or uxv in which $u, v, w \in \Sigma^+ \cup \{0, 1\}$ and x is one of the recursion variables $x_1, \ldots, x_m$. A language $L \subseteq \Sigma^*$ is *linear* if it is a component of the least solution (in the language interpretation) of a linear equation system over Σ.

A summand ux_jv of $r_i(x_1, \ldots, x_n)$ can be written as an insertion $(u, v) : x_j$ of the variable x_j into the 'context' (u, v). Hence it is natural to relate linear languages to Example 2 (ii). Let $\mathcal{A} = (2^{\Sigma^*}, \cup, \emptyset, :)$ be the Kleenean $Reg(\mathcal{L}_{\Sigma}^2)$-semimodule of languages over Σ, where $R : A$ is the infixation (13) of a language $A \subseteq \Sigma^*$ into a regular(!) word relation $R \subseteq \Sigma^* \times \Sigma^*$. Let $\Gamma_\Sigma := \Sigma \cup \{\varepsilon\}$ and $\Delta_\Sigma := \{\, (\epsilon, a) \mid a \in \Sigma \,\} \cup \{\, (a, \epsilon) \mid a \in \Sigma \,\}$. The linear and regular expressions over $\Gamma_\Sigma, \Delta_\Sigma$ can be interpreted in $\mathcal{A}$, using the corresponding singleton sets of A resp. K as interpretation of the constants of Γ_Σ resp. Δ_Σ.
Note that the languages in the submodule $\langle \Sigma \cup \{\varepsilon\} \rangle^{\mathcal{A}}$ are those that can be generated from $\emptyset$, $\{\varepsilon\}$ and $\{a\}$, for $a \in \Sigma$, by (finite) union and infixation into regular word relations. We obtain the following result, essentially due to Gruska[2]:

Theorem 2. *Suppose $L \subseteq \Sigma^*$. The following conditions are equivalent:*

(i) $L \in \langle \Sigma \cup \{\varepsilon\} \rangle^{\mathcal{A}}$,
(ii) $L = p^{\mathcal{A}}$ *for some linear expression p over $\Gamma_\Sigma, \Delta_\Sigma$,*
(iii) *L is a component of the least solution of a linear equation system over Σ.*

Proof. (Sketch) Obviously, (i) and (ii) are equivalent. Claim (ii) $\Rightarrow$ (iii) is shown by induction on the linear expressions. For (iii) $\Rightarrow$ (i), the equation system for L is considered as a matrix equation $X = Q : X + B$ in the semimodule $\mathcal{M}_{n,1}(\mathcal{A})$. By Proposition 1, its least solution is

$$\mu X(Q : X + B) = Q^* : B.$$

The entries of Q^* are regular expressions in the entries of Q, hence give regular word relations over Σ, while the components of B are finite languages over Σ. This implies (i). □

References

1. C. Brink. Boolean modules. *Journal of Algebra*, 71:291–313, 1981.
2. J. Gruska. A characterization of context-free languages. *Journal of Computer and System Sciences*, 5:353 – 364, 1971.
3. D. Kozen. On induction vs. *-continuity. In D.Kozen, editor, *Proc. Workshop on Logics of Programs 1981*, LNCS 131, pages 167–176. Springer Verlag, 1981.
4. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *6th Annual Symposium on Logic in Computer Science*. Computer Society Press, 1991.

Decidable fragments of domain mu-calculus: an automata-theoretic perspective

Guo-Qiang Zhang

Department of EECS, Case Western Reserve University
Cleveland, Ohio 44022, U.S.A.
gqz@eecs.cwru.edu

Abstract

Domain mu-calculus. Propositional domain logic (a.k.a. Abramsky logic), based on the view of types as topological spaces, properties as open sets, and computational processes as points (Smyth 83), provides a smooth integration among three relatively independent approaches to programming semantics: operational, denotational, and axiomatic. In addition to proof systems for higher-order strict-analysis and concurrent processes, it has also been adapted to reasoning about imperative parallel programs (Brookes 86, Zhang 91). The beauty of this approach is that one can pass from the denotation of a computational process to its properties, with harmony guaranteed by Stone-style-duality. Moreover, higher-order objects are treated exactly the same way as first-order objects.

The domain μ-calculus (Zhang 91) is a natural least fixed-point extension of propositional domain logic. This extension is a necessary step to increase the expressive power of propositional domain logic, since propositional formulas represent *compact* Scott open sets only. The domain logic consists of three syntactic categories: a language of types, a language of formulas, together with equational proof rules indexed over the types. In the domain μ-calculus, every closed type expression determines a canonical domain, and hence a topological space of Scott open sets. The semantics of a μ-formula is a *fixed* Scott open set of the corresponding domain (some standard restrictions are necessary for function space to work properly). The equational proof system is intended to capture the containment of Scott open sets; it uses Park's rules (Park 81) for least fixed-point induction at each closed type. It is said to be *sound* and *complete* if "theorems" of the form $\varphi \leq \psi$ coincide with their semantic counterparts $[\![\varphi]\!] \subseteq [\![\psi]\!]$. It is important to note that for each closed type there is a corresponding μ-calculus; therefore, by "domain μ-calculus" we refer to *a spectrum of μ-calculi* which may or may not share the same properties, such as completeness and decidability.

Modal mu-calculus and domain mu-calculus. Many properties of hardware and software systems can be expressed concisely in the propositional modal μ-calculus (Kozen 83). The modal μ-calculus has attracted a great deal of attention in the last decade. Although decidability and finite-model properties have been established early on, the difficult completeness problem was settled only recently

(Walukiewicz 00). Classification of expressive power of the μ-calculus has been obtained (Bradfield 98), establishing the strictness of the alternation hierarchy based on rather sophisticated results of definability (Lubarsky 93).

Domain μ-calculus and modal μ-calculus share at least two common ideas. One is that they are intended to capture *infinitary* behavior of a system. The other is that fixed-point formulas serve as a uniform way to *approximate* ideal infinitary properties in the end. However, what domain μ-calculus provides that modal μ-calculus does not is the integration of types, higher-order objects, and denotational semantics and program logics, all in the same framework. This offers a uniform and yet highly flexible set of logical tools.

While much progress has been made for the modal mu-calculus, not much is known about the domain mu-calculus. The following table provides a picture of the situation. It also gives an indication that reducing domain μ-calculus to modal μ-calculus may be hard, if not impossible, due to the lack of finite-model property for the domain μ-calculus. (The reason boils down to the fact that Scott open sets are not necessarily compact.)

	Modal μ-calculus	Domain μ-calculus
"fragment" means	restriction on formulas	restriction on types
finite model property	yes	no
decidability	yes	open
completeness	yes	open
duality results	yes	open

Main results. The main idea of this work is to establish a relationship between domain logic and automata theory in order to gain insights into decidability properties of the domain μ-calculus. The idea may work in two directions. If a (type) fragment of μ-formulas can be encoded as a class of formal languages, and this class of formal languages is decidable, then the domain μ-calculus is decidable (for emptiness, containment). If, on the other hand, an undecidable class of formal languages (such as context-free languages) can be emdeded as μ-formulas of a specific type, then the μ-calculus for that type (and any type more expressive than it) is undecidable. The results reported here are of the first kind. We show that (here $+$ is for coalesced-sum, and $\times$ is for smash-product)

1. the domain μ-calculus for $N = \mathbf{1}_\perp + N$, the domain of natural numbers, is decidable. It is equivalent in expressive power to *Presburger arithmetic*.
2. the domain μ-calculus for $P = \Sigma_\perp + \Sigma_\perp \times P$ is decidable, where Σ is a non-empty, finite set, and $\Sigma_\perp$ the corresponding flat domain. It is equivalent in expressive power to *regular languages* not containing the empty string.
3. the domain μ-calculus for $Q = \Sigma_\perp + (\Sigma_\perp \times Q \times \Sigma_\perp)$ is decidable. It is equivalent in expressive power to *even linear languages* which do not contain the empty string.

Automata theory and domain mu-calculus. The expressiveness result for natural numbers has been given in (Zhang 91). For the second type, consider

$P = \{0,1\}_\perp + \{0,1\}_\perp \times P$, to be specific. We first show that each $\wedge$-free μ-formula can be encoded as a regular expression over $\{0,1\}$ as illustrated by the following table, assuming some standard notational conventions.

μ-Formula	Regular expression
$0 \times 1 \vee 1 \times 0$	$01 + 10$
$\mu x.(0 \vee 0 \times x)$	0^+
$\mu x.(0 \vee 1) \vee (0 \vee 1) \times x$	$(0+1)^+$

Since regular languages are closed under intersection, we know that *every* μ-formula can be encoded as a regular language. The interpretation of μ-formulas ensures that $[\![\psi]\!] \subseteq [\![\varphi]\!]$ if and only if set containment in the corresponding regular languages holds. Therefore, entailment of the form $[\![\psi]\!] \subseteq [\![\varphi]\!]$ is decidable, by the classical result of Rabin and Scott.

The situation with respect to $Q = \Sigma_\perp + (\Sigma_\perp \times Q \times \Sigma_\perp)$ is more intriguing. A simple-minded analysis shows that the μ-formulas here should be encoded as the standard *linear* grammars (which determine linear languages). However, containment between linear languages is undecidable, by a text-book style reduction of PCP (Post's Correspondence Problem) to it. Our solution involves (a rediscovery of) the so-called *even linear grammars* of Amar and Putzolu (64, 65). An even linear grammar is a context-free grammar, each production of which is of the form $A \to w$ with $w \in \Sigma^\star$, or $A \to w_1 B w_2$, with $w_1, w_2 \in \Sigma^\star$ and w_1, w_2 being of *equal length*. The class of languages determined by even linear grammars strictly contains the class of regular languages (e.g. $\{a^n b c^n \mid n \geq 1\}$). However, containment of even linear languages is decidable, by a technique similar to Nerode's right-invariant relation. For each $L \subseteq \Sigma^*$ and $x, y \in \Sigma^*$, define $x\, E_L\, y$ if for each $a, b \in \Sigma$, $axb \in L$ iff $ayb \in L$. The equivalence relation E_L has finitely many equivalence classes exactly when L is an even linear language. The fact that even linear languages form a Kleene algebra with respect to a *non-standard* monoidal product and the fact that they are closed under set union and intersection hold the remaining missing pieces for this reduction to work.

Concluding remarks. One can use a notion of *proportional* linear grammar to provide decidability results for a larger fragment of domain μ-calculus, with small variations on the type definitions. Our results so far remain somewhat limited, although similar ideas and techniques may work for a bigger segment. Reduction to tree languages of certain kind seems to be a plausible way to perhaps completely resolve the decidability issue. However, we feel that the current restricted sense of achievement is due in large part to the inherent combinatorial nature of the problem. It suffices to note that the restricted star-height problem had been open for more than two decades, and the *generalized star-height problem* remains open even today. Completeness issues are expected to be harder.

For related work, connections between fixed-point logics and languages have been explored by many researchers in various contexts in the past. The special character of our work is that it brings the notions of types and topology into the picture, enriching the interplay among logic, topology, and languages.

References

[Abramsky] S. Abramsky. Domain theory in logical form. *Ann. of Pure and Applied Logic* 51:1–77, 1991.

[Amar64] V. Amar and G. Putzolu. On a family of linear grammars. *Info. and Contr.* 7:283-291, 1964.

[Ama65] V. Amar and G. Putzolu. Generalizations of regular events. *Info. and Contr.* 8:56-63, 1965.

[Bradfield98] J.C. Bradfield. Simplifying the modal mu-calculus alternation hierarchy. *Lecture Notes in Computer Science* 1373:39-49, 1998.

[Brookes86] S. Brookes. A semantically based proof system for partial correctness and deadlock in CSP. In Proceedings, *Symposium on Logic in Computer Science*, pages 58-65, Cambridge, Massachusetts, 1986.

[Kozen83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27: 333–354, 1983.

[Lubarsky93] R.S. Lubarsky. μ-definable sets of integers. *Journal of Symbolic Logic*, 58, 291-313, 1993.

[Park81] D. Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science* 154:561-572, 1981.

[Smyth83] M. Smyth. Powerdomains and predicate transformers: a topological view. *Lecture Notes in Computer Science* 154:662-675, 1983.

[Walukiewicz00] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. *Information and Computation* 157:142–182, 2000.

[Zhang91] G.-Q. Zhang. *Logic of Domains*. Birkhauser, Boston, 1991.

Fixed Points on Abstract Structures without the Equality Test

M. V. Korovina

BRICS*
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

The aim of the paper is to present a study of definability properties of fixed points of effective operators on abstract structures without the equality test, in particular, on the real numbers. The question of definability of fixed points of Σ-operators on abstract structures with equality was first studied in [1, 3, 2]. One of the most fundamental theorems in the area is Gandy theorem which states that the least fixed point of any positive Σ-operator is Σ-definable. This theorem allows us to treat recursive definitions using Σ-formulas. It turns out that in some cases it is natural to consider languages without equality, for example, in computable analysis [4, 5, 10]. Indeed, in all effective approaches to exact real number computation via concrete representations, the equality test is undecidable. This is not surprising, because infinite amount of information must be checked in order to decide that two given numbers are equal.

Until now there have been no Gandy-type theorem for such structures. We prove that Gandy theorem holds for abstract structures without the equality test.

The concept of Σ-definability is closely related to the generalised computability over abstract structures [1, 3, 9, 11], in particular over the real numbers [7, 8, 11].

Notions of Σ-definable sets or relations on abstract structures generalise those of computable enumerable sets of natural numbers, and play leading role in the specification theory that is used in the higher order computation theory over abstract structures. Considering an abstract structure A without the equality test, we investigate properties of Σ-operators defined on the set of subsets of A^n. Let us consider an arbitrary structure $\mathcal{U} = (A, \sigma)$, where σ is a finite language without equality. Clearly, very little computability theory can be developed within a completely arbitrary structure $\mathcal{U}$.

In order to do any kind of computation or computability theory one has to work within a structure rich enough for information to be coded and stored. For this purpose we extend the structure $\mathcal{U}$ by the set of hereditarily finite sets $\mathrm{HF}(A)$. Note that such extensions of structures with equality are rather well studied in the theory of admissible sets [1] and used in the theory of abstract state machines [6]. We will construct the set of hereditarily finite sets over structures

* Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

without equality. The hereditarily finite sets permit us to define the natural numbers, and to code and store information via formulas.

We construct the set of hereditarily finite sets, $\mathrm{HF}(A)$, as follows:

1. $\mathrm{S}_0(A) \rightleftharpoons A$, $\mathrm{S_{n+1}}(A) \rightleftharpoons \mathcal{P}_\omega(\mathrm{S_n}(A)) \cup \mathrm{S_n}(A)$, where $n \in \omega$ and for every set B, $\mathcal{P}_\omega(B)$ is the set of all finite subsets of B.
2. $\mathrm{HF}(A) = \bigcup_{n\in\omega} \mathrm{S}_n(A)$.

We define $\mathbf{HF}(A)$ as the following structure:

$$\mathbf{HF}(A) \rightleftharpoons \left\langle \mathrm{HF}(A), A, \sigma, \emptyset_{\mathbf{HF}(A)}, \in_{\mathbf{HF}(A)} \right\rangle,$$

where the constant $\emptyset$ stands for the empty set and the binary predicate symbol $\in_{\mathbf{HF(M)}}$ has the set-theoretic interpretation.

The notions of terms and atomic formulas can be introduced in the standard manner.

The set of Δ_0-*formulas* is the closure of the set of atomic formulas under $\wedge, \vee, \neg$, and bounded quantifiers $(\exists x \in s)$, $(\forall x \in s)$, where $(\exists x \in s)\ \varphi$ denotes $\exists x(x \in s \wedge\ \varphi)$ and $(\forall x \in s)\ \varphi$ denotes $\forall x(x \in s \rightarrow \varphi)$.

The set of Σ-*formulas* is the closure of the set of Δ_0 formulas under $\wedge, \vee$, $(\exists x \in s)$, $(\forall x \in s)$, and $\exists$.

Definition 1. *A set $B \subseteq A^n$ is Σ-definable , if there exists a Σ-formula $\Phi(x)$ such that $x \in B \leftrightarrow \mathbf{HF}(A) \models \Phi(x)$.*

Let $\Phi(\mathbf{x}, P)$ be a Σ-formula, where the arity of the new predicate symbol P coincides with the length of $\mathbf{x}$, P occurs positively in Φ and $\mathbf{x}$ ranges over A^n.

We think of Φ as defining a positive Σ-operator $\Gamma : \mathcal{P}(A^n) \rightarrow \mathcal{P}(A^n)$ given by

$$\Gamma(S) = \{\mathbf{x} |\, (\mathbf{HF}(A), S) \models \Phi(\mathbf{x}, P)\}.$$

Theorem 1. *Let us consider a structure A in a finite language without the equality test. Then the least fixed-point of any positive Σ-operator is Σ-definable.*

Let us note, that if we consider the real numbers in the language of strictly ordered rings then Gandy theorem can be used to derive Σ-definability of the truth of Σ-sentences and to obtain a universal Σ-relation. It leads to a topological characterisation of Σ-relations on $\mathbb{R}$.

Corollary 1. *Let $\mathbb{R}$ be the real numbers in the language of strictly ordering rings. The sets $B \subseteq \mathbb{R}$ that are Σ-definable in $\mathbf{HF}(\mathbb{R})$ are exactly the effective unions of open semialgebraic sets.*

References

1. J. Barwise, Admissible sets and structures, Berlin, Springer–Verlag, 1975.
2. R. Gandy, Inductive definitions, Generalized Recursion Theory, Amsterdam: Noeth-Holland, 1974, pages 265–300.

3. Yu. L. Ershov, Definability and computability, Plenum, New York, 1996.
4. H. Freedman and K. Ko, Computational complexity of real functions, Theoret. Comput. Sci. , v. 20, 1982, pages 323–352.
5. A. Grzegorczyk, On the definitions of computable real continuous functions, Fund. Math., N 44, 1957, pages 61–71.
6. A. Blass, Y. Gurevich, Background, Reserve and Gandy Machines, Proceedings of CSL'2000, LNCS N 1862, 2000, pages 1–17.
7. M. Korovina, O. Kudinov, Characteristic Properties of Majorant-Computability over the Reals, Proc. of CSL'98, LNCS, 1584, 1999, pages 188–204.
8. M. Korovina, O. Kudinov, A Logical approach to Specifications of Hybrid Systems, Proc. of PSI'99, LNCS 1755, 2000, pages 10–16.
9. Y. N. Moschovakis, Abstract first order computability, Trans. Amer. Math. Soc., v. 138, 1969, pages 427–464.
10. M. B. Pour-El, J. I. Richards, Computability in Analysis and Physics, Springer-Verlag, 1988.
11. J. V. Tucker, J. I. Zucker, Computable functions and semicomputable sets on many-sorted algebras, Handbook of Logic in Computer Science, v. 5, Oxford University Press, Oxford, 2000, pages 397–525.

Recursion in the Call-by-Value λ-Calculus (*)

Gérard Boudol and *Pascal Zimmer*

INRIA Sophia Antipolis
BP 93 – 06902 Sophia Antipolis Cedex, France

Abstract

We propose an abstract machine to run the call-by-value λ-calculus extended with a call-by-value fixed-point, and we show that this provides us with a correct implementation of our calculus.

1. Introduction

It is a well-known fact that the call-by-value λ-calculus is the functional core of programming languages like Scheme or ML, and that the ability to use fixed-points in this calculus is essential for the expressivity of such programming languages. However, in a language like ML, recursion is usually syntactically restricted to

$$(\mathsf{let\ rec}\ f = \lambda x M\ \mathsf{in}\ \cdots)$$

so that in particular one can only define functions recursively. A reason for this is that it would be dangerous to use an unrestricted recursion construct ($\mathsf{let\ rec}\ f = M\ \mathsf{in}\ \cdots$), where one evaluates the expression M before binding (recursively) its value to f. Indeed, the evaluation of this construct gets stuck if in evaluating M one needs the value of f. An example is

$$(\mathsf{let\ rec}\ f = \mathsf{F}(f\,\mathsf{I})\ \mathsf{in}\ \cdots) \qquad \text{where} \quad \mathsf{F} = \lambda x \lambda y\, y \quad \text{and} \quad \mathsf{I} = \lambda z\, z$$

Nevertheless, one may wish to have the ability to define recursive *non-functional* values, like mutually recursive modules (*cf.* [3, 4]), or objects as recursive records (*cf.* [1, 2]), that is

$$(\mathsf{let\ rec}\ o = (\lambda \mathit{self}\, R)o\ \mathsf{in}\ \cdots)$$

where R is a record of fields and methods. In [1] we have shown that one can determine, by a static analysis – namely, a type system –, that some unrestricted recursions are actually safe. However, the question remained of how to implement this unrestricted recursion, in a more effective (and provably correct) way than what is done in the interpreter built by the second author. For instance, since in an implementation a λ-calculus variable is not a value, one may wonder how we can compute $(\lambda \mathit{self}\, R)o$ without having a value for o. In this paper we introduce an abstract machine for such unrestricted – but typable in our system – recursion, and we show that this implementation is correct. For simplicity reasons, we do this for a minimal calculus, the call-by-value λ-calculus with (call-by-value) recursion, and we do not consider non-functional values like pairs, records or modules. Indeed, we have shown in [1] that the difficulties with recursion lie in this functional core, and that adding more types and related constructs does not cause any particular trouble.

(*) Work partially supported by the CTI "Objets Migrants: Modélisation et Vérification", France Télécom R&D

2. The Calculus

Assuming that a denumerable set $\mathcal{V}ar$ of variables, ranged over by x, y, $z\ldots$, is given, the syntax of our calculus is the following:

$$\begin{array}{rcll} M,\, N\ldots & ::= & V \mid (MN) \mid \mu x M & \textit{expressions} \\ V & ::= & x \mid \lambda x M & \textit{values} \end{array}$$

where $\mu x M$ is the standard fixed-point construct, here computed in a call-by-value regime. Another possible syntax would be (let rec $x = N$ in M), which stands for (let $x = \mu x N$ in M) – that is, as far as polymorphism is not concerned, $(\lambda x M)(\mu x N)$ –, since $\mu x M$ may be read (let rec $x = M$ in x). We could also consider as a primitive a combinator fix (which is $\lambda f \mu x(fx)$), since $\mu x M$ could then be defined as $\mathsf{fix}(\lambda x M)$, but, as we said in the Introduction, it would not suit our purposes to use the call-by-value fixed-point combinator $\mathsf{Y_v}$ (*cf.* [7]), defined by

$$\mathsf{Y_v} \;=\; (G_\mathsf{v} G_\mathsf{v}) \qquad \text{where} \quad G_\mathsf{v} = \lambda g f.f(\lambda x.(gg)fx)$$

Denoting by $\{x \mapsto N\}M$ the capture avoiding substitution of x by N in M, the reduction – or better, evaluation – relation is given by

$$\begin{array}{rcl} (\lambda x M V) & \rightarrow & \{x \mapsto V\}M \\ \mu x V & \rightarrow & \{x \mapsto \mu x V\}V \\ M \rightarrow M' & \Rightarrow & \mathbf{E}[M] \rightarrow \mathbf{E}[M'] \end{array}$$

where the evaluation contexts $\mathbf{E}$ are

$$\mathbf{E} \;::=\; [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid \mu x \mathbf{E}$$

As we said in the Introduction, we use a type discipline to rule out unsafe recursion. This is briefly presented in the Appendix. Our type discipline here is more generous than the one of [1], in the sense that it actually does not impose any restriction on λ-terms – without recursion:

REMARK 2.1. *If M is a λ-term, that is an expression written without using μ, then M is typable.*

In [1], we proved that the following holds:

THEOREM (TYPE SAFETY) 2.2. *Let M be a closed expression. If M is typable, then its evaluation either diverges, or ends with a value.*

Moreover, we have characterized the behaviour of the recursive variables, which can only be "passed around", as argument of functions, or be put inside a value:

LEMMA 2.3. *If $\mu x M$ is typable and $M \xrightarrow{*} \mathbf{E}[x]$, where x is not bound by $\mathbf{E}$, then $\mathbf{E} = \mathbf{E}'[(\lambda y N\,[])]$.*

This is the key property that allows us to design a correct abstract machine for our calculus: we know that if a recursive variable comes to be evaluated, that is if we have to evaluate $\mu x \mathbf{E}[x]$, we actually do not need its value, since we can continue with $\mu x \mathbf{E}'[\{y \mapsto x\}N]$.

3. The Abstract Machine

Our machine builds upon standard stack machines for evaluating λ-expressions. As such, it has a *control stack* S, to record the current evaluation context, and an *environment* part σ, to hold the values of the free variables occurring in the *code*, that is, the sub-expression currently evaluated. The stack is made up of *frames*, that are basic evaluation contexts (*cf.* [6]). As usual, substitution of terms for variables is not performed; instead, the machine deals with *closures* [5], that are pairs of an environment and an expression, that we denote σM. As regards recursion, we use the standard

trick of Landin's imperative fixpoint: to compute $\mu x M$, allocate a new address ℓ, evaluate M in an environment where x is bound to ℓ, and assign to ℓ the result of this evaluation. As noted in [5], this works correctly if M is a function, that is $M = \lambda y N$, though in that case we may have a more direct implementation, binding (circularly) x to $\lambda y N$ in the environment. Then, assuming that a set $\mathcal{L}$ of *locations* is given, our machine has an extra component ξ, which is a memory to store the evaluated recursion's bodies. A configuration of the machine is therefore a tuple

$$\mathcal{M} \;=\; (S, \sigma, M, \xi)$$

where the control stack and environment are built according to the following syntax:

$$\begin{array}{rcl} S & ::= & \varepsilon \;\mid\; S :: \mathbf{F} \\ \mathbf{F} & ::= & ([]\,\sigma N) \;\mid\; (\sigma A\,[]) \;\mid\; \mu\ell\,[] \\ \sigma,\ \rho\ldots & ::= & \varepsilon \;\mid\; \sigma :: \{x \mapsto \rho A\} \;\mid\; \sigma :: \{x \mapsto A\} \;\mid\; \sigma :: \{x \mapsto \ell\} \\ A & ::= & \lambda x M \end{array}$$

In $\sigma :: \{x \mapsto A\}$, the abstraction A is the value of the recursive variable x, and is in the scope of σ (except for what regards x). In a concrete implementation the store ξ simply is a partial mapping assigning values to locations; however, for the correctness proof it is more convenient to have a syntactic description of it, as follows:

$$\xi \;::=\; \emptyset \;\mid\; \xi \uplus \{\ell \mapsto \bullet\} \;\mid\; \xi[\ell := \sigma A]$$

The domain of a store ξ is defined in the obvious way:

$$\begin{array}{rcl} \mathsf{dom}(\emptyset) & = & \emptyset \\ \mathsf{dom}(\xi \uplus \{\ell \mapsto \bullet\}) & = & \mathsf{dom}(\xi) \cup \{\ell\} \\ \mathsf{dom}(\xi[\ell := \sigma A]) & = & \mathsf{dom}(\xi) \end{array}$$

The operation $\xi \uplus \{\ell \mapsto \bullet\}$ is the extension of ξ with a new location ℓ to which is assigned a "dummy" value, while $\xi[\ell := \sigma A]$ is the updating operation. Then a store ξ represents a partial function, defined as:

$$(\xi \uplus \{\ell \mapsto \bullet\})(\ell') \;=\; \begin{cases} \bullet & \text{if } \ell' = \ell \\ \xi(\ell') & \text{otherwise} \end{cases}$$

$$(\xi[\ell := \sigma A])(\ell') \;=\; \begin{cases} \sigma A & \text{if } \ell' = \ell \\ \xi(\ell') & \text{otherwise} \end{cases}$$

To evaluate a closed expression M, we start from the configuration $(\varepsilon, \varepsilon, M, \emptyset)$. Now we describe the behaviour of the machine, that is the transition relation $\mathcal{M} \to \mathcal{M}'$ between configurations. The first three rules for evaluating an application (MN) are standard: first, we record the closure of the argument N on the control stack and evaluate the function M. Then we record (the closure of) its value $A = \lambda x M'$ on the stack, and turn to the evaluation of the argument. When a value for it is computed, we update the environment of the function with a binding for x to this value, and we compute M'. There is a special case when the argument is a recursive variable, whose value is not yet computed: this is dealt with by a fourth rule for application, which is new. Here are the rules:

$$\begin{array}{rcll} (S, \sigma, (MN), \xi) & \to & (S :: ([]\,\sigma N), \sigma, M, \xi) & \\ (S :: ([]\,\rho N), \sigma, A, \xi) & \to & (S :: (\sigma A\,[]), \rho, N, \xi) & \\ (S :: (\sigma \lambda x M\,[]), \rho, A, \xi) & \to & (S, \sigma :: \{x \mapsto \rho A\}, M, \xi) & \\ (S :: (\sigma \lambda y M\,[]), \rho :: \{x \mapsto \ell\}, x, \xi) & \to & (S, \sigma :: \{y \mapsto \ell\}, M, \xi) & \xi(\ell) = \bullet \end{array}$$

As we said, to evaluate a recursion we use Landin's imperative fixpoint technique, except that we make a special case when the body of the recursion is known to be a function:

$$\begin{array}{rcll} (S,\sigma,\mu x A,\xi) & \to & (S,\sigma :: \{x \mapsto A\}, A, \xi) & \\ (S,\sigma,\mu x M,\xi) & \to & (S :: \mu\ell\,[], \sigma :: \{x \mapsto \ell\}, M, \xi \uplus \{\ell \mapsto \bullet\}) & M \neq \lambda y N,\ \ell \notin \mathsf{dom}(\xi) \\ (S :: \mu\ell\,[], \sigma, A, \xi) & \to & (S,\sigma,A,\xi[\ell := \sigma A]) & \end{array}$$

Finally we have rules to fetch the value of a variable from the environment σ. These rules are standard: one discards the bindings that do not concern the given variable, and then, depending on the value of the variable in the environment, one fetches the appropriate closure, which is unravelled, or the abstraction that is the recursive value of the variable.

$$\begin{array}{rcll} (S,\sigma :: \{y \mapsto Z\}, x, \xi) & \to & (S,\sigma,x,\xi) & x \neq y \\ (S,\sigma :: \{x \mapsto \rho A\}, x, \xi) & \to & (S,\rho,A,\xi) & \\ (S,\sigma :: \{x \mapsto A\}, x, \xi) & \to & (S,\sigma :: \{x \mapsto A\}, A, \xi) & \\ (S,\sigma :: \{x \mapsto \ell\}, x, \xi) & \to & (S,\rho,A,\xi) & \xi(\ell) = \rho A \end{array}$$

For instance, the reader can check that if $M = \mu z(\mathsf{K}\, z)$ where $\mathsf{K} = \lambda x \lambda y\, x$, then

$$\begin{array}{rcll} (\varepsilon,\varepsilon,M,\emptyset) & \xrightarrow{*} & (\varepsilon :: \mu\ell\,[] :: (\rho\mathsf{K}\,[]), \rho, z, \xi) & \text{where } \rho = \varepsilon :: \{z \mapsto \ell\},\ \xi = \emptyset \uplus \{\ell \mapsto \bullet\} \\ & \to & (\varepsilon :: \mu\ell\,[], \sigma, \lambda y\, x, \xi) & \text{where } \sigma = \rho :: \{x \mapsto \ell\} \\ & \to & (\varepsilon, \sigma, \lambda y\, x, \xi[\ell := \sigma\lambda y\, x]) & \end{array}$$

A final state of the machine is a configuration of the form $\mathcal{M} = (\varepsilon,\sigma,A,\xi)$, and we may define the result of the computation as the expression $\mathcal{V}al(\mathcal{M}) = \langle\xi\rangle \circ \langle\sigma\rangle A$ associated with this configuration, where $\langle\sigma\rangle$ is a simultaneous substitution for the variables, defined in such a way that it satisfies in particular:

$$\begin{array}{rcl} \langle\sigma :: \{x \mapsto \rho A\}\rangle x & = & \langle\rho\rangle A \\ \langle\sigma :: \{x \mapsto A\}\rangle x & = & \langle\sigma\rangle \mu x A \\ \langle\sigma :: \{x \mapsto \ell\}\rangle x & = & \ell \end{array}$$

where we assume for convenience that the language is extended so that locations are a new kind of variables, and $\langle\xi\rangle$ is a substitution for locations, such that

$$\langle\xi[\ell := \rho A]\rangle\ell \;=\; \mu x\{\ell \mapsto x\}(\langle\xi\rangle \circ \langle\rho\rangle)A$$

where x is fresh. Then we define a partial function $\mathcal{E}val$ from closed expressions to closed values as follows: for any closed expression M, $\mathcal{E}val(M)$ is defined if there exists a (unique, since the machine is deterministic) final configuration $\mathcal{M}$ such that $(\varepsilon,\varepsilon,M,\emptyset) \xrightarrow{*} \mathcal{M}$, and in this case $\mathcal{E}val(M) = \mathcal{V}al(\mathcal{M})$. Then our main result is:

Theorem (Correctness) 3.1. *Let M be a closed expression. Then*

(i) *if $\mathcal{E}val(M)$ is defined then $M \xrightarrow{*} \mathcal{E}val(M)$;*

(ii) *if M is typable and $M \xrightarrow{*} A$ then $\mathcal{E}val(M)$ is defined and $\mathcal{E}val(M) = A$.*

To see that the typing is needed, one can check for instance that $\mu x(\mathsf{K}(\mu z\, x))$, which is not typable, reduces to $\lambda y \mu x \lambda y\, x$, while the evaluation of this expression in the machine gets stuck in the configuration

$$(\varepsilon :: \mu\ell\,[] :: (\rho\mathsf{K}\,[]) :: \mu\ell'\,[], \rho, x, \emptyset \uplus \{\ell \mapsto \bullet\} \uplus \{\ell' \mapsto \bullet\}) \qquad \text{where} \quad \rho = \varepsilon :: \{x \mapsto \ell\}$$

The abstract machine we presented, being based on named variables, is well-suited for establishing a correspondance with the calculus. However, it should not be difficult to built a more concrete, nameless version of the machine, based on de Bruijn's indices.

References

[1] G. Boudol, *The recursive record semantics of objects revisited*, to appear (2002). Available from the web page of the author.

[2] L. Cardelli, *A semantics of multiple inheritance*, Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol. 76 (1988).

[3] K. Crary, R. Harper, S. Puri, *What is a recursive module?*, PLDI'99 (1999) 50-63.

[4] T. Hirschowitz, X. Leroy, *Mixin modules in a call-by-value setting*, ESOP'02, Lecture Notes in Comput. Sci. 2305 (2002) 6-20.

[5] P. J. Landin, *The mechanical evaluation of expressions*, Computer Journal Vol. 6 (1964) 308-320.

[6] A. Pitts, *Operational semantics and program equivalence*, Lectures given at the International Summer School on Applied Semantics, APPSEM'00 (2000). Available from the web page of the author.

[7] G. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theoret. Comput. Sci. 1 (1975) 125-159.

Appendix: The Type System

Our notion of type is a refinement of the usual one, featuring boolean *degrees* 0 or 1, to discriminate which variables are certainly safe (having degree 1) or potentially unsafe (degree 0) for recursion. For lack of space, we shall not comment in detail here on our type system. The interested reader is referred to [1, 4]. The degrees and types are respectively given by

$$\begin{array}{rcl} \alpha,\ \beta \ldots & := & 0 \mid 1 \mid (\alpha \wedge \beta) \\ \tau,\ \theta \ldots & ::= & * \mid (\theta^{\alpha} \to \tau) \end{array}$$

We consider types up to an equality asserting that $*$ is, as a data type, Scott's D_∞ domain of functions:

$$* = (*^0 \to *)$$

Then our type system will accept any λ-term. We also consider types up to equality of degrees $\alpha = \beta$, defined as $\alpha \leqslant \beta \;\&\; \beta \leqslant \alpha$, where the preorder $\leqslant$ satisfies the following laws:

$$\begin{array}{c} 0 \leqslant \alpha \leqslant 1 \\ \alpha \leqslant \alpha \wedge \beta \qquad \text{and} \qquad \alpha \leqslant \beta \wedge \alpha \\ \alpha \leqslant \alpha_0 \;\&\; \alpha \leqslant \alpha_1 \;\Rightarrow\; \alpha \leqslant \alpha_0 \wedge \alpha_1 \end{array}$$

The typing judgements are $\Gamma^\gamma \vdash M : \tau$ where Γ is a typing assumption, that is a mapping from a finite set of variables to types, and γ is a degree assumption, from the same set of variables to degrees. We use the following notations:

- we abusively denote by 0 and 1 the degree assumptions (of any appropriate domain) that uniformly assign respectively 0 and 1 to any variable.
- for any degree α and expression N, we denote by α_N the degree assumption defined by $\alpha_N(x) = \alpha$ if $x \in \mathsf{fv}(N)$, and $\alpha_N(x) = 1$ otherwise.
- $\delta \leqslant \gamma$ on X, where δ and γ are degree assumptions with the same domain and X is a set of variables, means that $\delta(x) \leqslant \gamma(x)$ for all $x \in X$.
- $\delta \wedge \gamma$ is the degree assumption pointwise defined by $(\delta \wedge \gamma)(x) = \delta(x) \wedge \gamma(x)$.

The typing rules are as follows:

$$\frac{\Gamma^\gamma \vdash M : \tau \qquad \delta \leqslant \gamma \text{ on } \mathsf{fv}(M)}{\Gamma^\delta \vdash M : \tau} \qquad \frac{}{\Gamma^1, x : \tau^0 \vdash x : \tau} \qquad \frac{\Gamma^\gamma, x : \theta^\alpha \vdash M : \tau}{\Gamma^1 \vdash \lambda x M : (\theta^\alpha \to \tau)}$$

$$\frac{\Gamma^\gamma \vdash M : \theta^\alpha \to \tau \qquad \Gamma^\gamma \vdash N : \theta}{\Gamma^{0_M \wedge \delta} \vdash (MN) : \tau} \quad \text{where} \quad \delta(x) = \begin{cases} \alpha & \text{if } N = x \\ (\alpha_N \wedge \gamma)(x) & \text{otherwise} \end{cases}$$

$$\frac{\Gamma^\gamma, x : \tau^1 \vdash M : \tau}{\Gamma^\gamma \vdash \mu x M : \tau}$$

The only constraint that may not be satisfied in typing a term is that any recursive variable must have degree 1 in the typing context for its body. There are three ways in which a variable may get degree 1 in the typing context for an expression M: either it does not occur in M, or x occurs either within abstractions in M or in arguments of “protective” functions, that are expressions of type $\theta^1 \to \tau$. For instance the function $\mathsf{K} = \lambda x \lambda y\, x$ is protective, since it puts its argument within an abstraction, and therefore $\mu z(\mathsf{K}z)$ is typable.

Kleene's (unary) star in nondeterministic context

Anna Labella
U Rome I (La Sapienza)
Italy
(joint work with Rocco De Nicola)

Trees represent w.r.t. non deterministic processes what languages are w.r.t. automata. This statement can be quite easily formalized in enriched category theory. Hence it seems natural and important to investigate the behaviour of the (unary) Kleene star operator in this context. From an algebraic point of view it amounts to drop idempotency of sum and left distributivity from the usual properties that hold for Kleene algebras. It can be proved that star-regular trees (i.e. trees corresponding to regular expressions), differently from regular languages, are finitely equationally axiomatizable, provided that a sort of non empty word property is required. On the other hand one can finitely present every such a tree as minimal solution (fixed point) of a hierarchical right-linear equation system. This property holds also in the case where the non empty word property above is not satisfied, i.e. when in the equation system there are variables essentially non guarded. We can prove that in the category of star regular trees, Tree*, we can define a "natural numbers object" (and, therefore, recursion) as minimal solution of a very simple equational system. The purpose is achieved by introducing a simple method of representing morphisms by hierarchical right-linear equational systems as well. The same method can be used to prove many properties of the category Tree*, in particular the existence of a subobject classifier. It must be remarked that the natural numbers object, as well as the subobject classifier trivially collapse into the terminal object when translated in the category of regular languages.

References

1. Bloom,S.L., Ésik,Z.: Iteration Algebras of Finite State Process Behaviors. Manuscript.

2. Bloom,S.L., Ésik,Z., Taubner,D.: Iteration theories of synchronization trees. *Information and Computation* **102**, pp.1-55, 1993.

3. Bosscher, D.: Grammars Modulo Bisimulation, Ph.D. Thesis, University of Amsterdam, 1997.

4. Corradini,F., De Nicola,R., Labella,A.: A Finite Axiomatization of Nondeterministic Regular Expressions. *Theoretical Informatics and Applications* **33**, pp. 447-465, 1999.

5. De Nicola,R., Labella,A.: Nondeterministic Regular Expressions as Solutions of Equational Systems, to appear

6. Milner,R.: A complete inference system for a class of regular behaviors. *Journal of Computer and System Sciences* **28** (3), pp. 439-466, 1984.

7. Salomaa,A.: Theory of Automata. Pergamon Press, 1969.

Iteration schemes for fixed point computation*

Thomas Jensen Florimond Ployette Olivier Ridoux

IRISA
Campus de Beaulieu
F- 35042 Rennes
France

Abstract

We report on work whose overall aim is to obtain a template fixed point algorithm that can be used to compare existing algorithms such as work-set algorithms, top-down solvers, local solvers *etc.* for calculating solutions of monotone systems of equations over lattices. The approach taken here is to focus on the dependency graph between the variables of the system and exploit this graph to schedule the iteration of the system. The resulting iteration scheme reduces the number of expressions to be re-evaluated both in theory and in practice (up to 50 % in certain cases).

1 Introduction

Data flow analysis usually involves a phase of fixed point computation whose algorithmics is largely independent of the particular property analysed for [4, 9]. This has lead to the search for generic fixed point algorithms that can serve as "back ends" in program analyzers. A common feature of these algorithms is a dependency relation that describes how the value of one variable depends on the value of another variable. However, the algorithms are often expressed in a manner that makes it difficult to compare *how* the dependency relation is exploited. The aim of the work reported here is

1. to make clear how the dependency relation can be exploited to implement and optimise a fixed point computation,
2. to provide a framework (in the form of a template algorithm plus accompanying data structures) that enables a comparative study of iteration-based fixed point algorithms.

The paper is organised as follows. After introducing the basic notation we define a theoretical iteration scheme (Section 3) based on the notion of *minimal elements* of a dependency relation. The theoretical algorithm defines an ideal iteration scheme but is defined in terms of an exact dependency relation which makes it unfeasible in practice. Section 4 shows how to calculate a safe approximation to the exact dependency relation based on a syntactic notion of dependency. This leads to a basic iteration scheme for which we then present benchmarks comparing it with other iteration schemes in terms of space, time and number of iterations. Section 6 draws some conclusions and outlines further issues to be investigated.

*This work was partially supported by the IST FET/Open project "Secsafe".

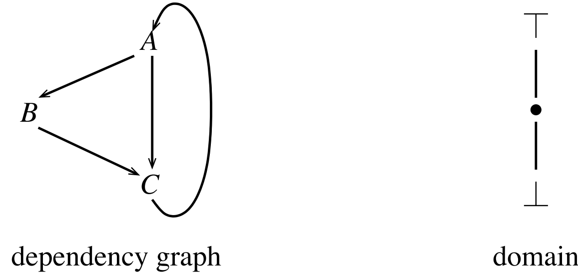


Figure 1: A toy dependency graph and domain

2 Notation

In this section we briefly define the notation to be used in the paper. For a much more comprehensive introduction to fixed points, see the recent textbook by Arnold and Niwinski [1]. In the following, i, j range over the integers $1, \ldots, n$. Let $E_1, \ldots, E_n$ be finite lattices and let e_i be an expression satisfying that all free variables in e_i are included in $\{f_1, \ldots, f_n\}$, that e_i defines a monotone function in all variables, and that evaluating e_i in an environment ϕ satisfying $\phi(f_j) \in E_j$ yields a value in E_i. We write $e_i\ \phi$ for the value of e_i in environment ϕ. A solution to a system of equations of the form

$$\{f_1 = e_1, \ldots, f_n = e_n\}$$

can be characterised as a fixed point of the function

$$e : (E_1 \times \ldots \times E_n) \rightarrow (E_1 \times \ldots \times E_n)$$

induced by the e_i and defined by $e\phi = (e_1\ \phi, \ldots,\ e_n\ \phi)$ The least solution to the system of equations can be found as the limit of the *ascending Kleene chain* (AKC)

$$\{e^{\,i}(\bot_{E_1}, \ldots, \bot_{E_n})\}_{i=0}^{\infty} \tag{1}$$

We do not put any restrictions on the type of the expressions e_i. In particular, these expressions can denote functions.

Example 1 (A toy example) *We will use as a running example the following equation system*

$$\begin{aligned} B &= incr(A) \\ C &= max(B, A) \\ A &= C \end{aligned}$$

where the variable domain is as in Figure 1 and incr is the mapping $\{\bot \mapsto \bullet, \bullet \mapsto \top, \top \mapsto \top\}$*. The syntactic dependency graph (to be defined in Section 4 of this equation system is as in Figure 1. Figure 2 shows the AKC for this example.*

A	B	C
$\bot$	$\bot$	$\bot$
$\bot$	$\bullet$	$\bot$
$\bot$	$\bullet$	$\bullet$
$\bullet$	$\bullet$	$\bullet$
$\bullet$	$\top$	$\bullet$
$\bullet$	$\top$	$\top$
$\top$	$\top$	$\top$
$\top$	$\top$	$\top$

Figure 2: The AKC for Example 1: 21 evaluations

3 Scheduling for an ideal fixed point algorithm

A somewhat less naïve method for calculating the limit of an AKC consists in iterating all equations in order (the *round-robin* algorithm). However, this is still an inefficient approach and is in general only feasible for the simplest dataflow problems [7]. The inefficiency stems from the possibility that an equation may be iterated even though the value of its defining expression will not change. This may happen either because there have been no changes in the variables occurring in the expression or because these changes do not have any impact on its value. Thus, only expressions that are guaranteed to change value should be re-evaluated during an iteration. This set of equations depends on the current approximation ϕ and is given by

$$W_\phi = \{f_i \mid \phi_i \neq e_i\,\phi\}$$

Not all equations in W_ϕ should necessarily be scheduled for iteration. If the value of f_i depends on the value of f_j then it is usually wasted effort to re-evaluate f_i before having found the new value of f_j. The only exception to this is when f_i and f_j are mutually dependent. The *dynamic dependency relation* is defined relative to the current values of variables ϕ by

$$f_j \xrightarrow{\phi} f_i \equiv e_i\,\phi \sqsubset e_i\,\phi[f_j \mapsto e_j\,\phi].$$

The notation "$f_j \xrightarrow{\phi} f_i$" should be read as "$f_i$ depends on f_j under the valuation ϕ".

3.1 Minimality

The equations in W_ϕ that should be iterated can now be characterised as the *minimal* elements of W_ϕ with respect to the dynamic dependency relation. We define for a set $S \subseteq \{1,\ldots,n\}$ and relation $\rightarrow$ its set of "minimal" elements as follows:

$$min(S,\rightarrow) \equiv \{j \in S : \textit{if}\ \exists i \in S. i \rightarrow^+ j\ \textit{then}\ j \rightarrow^+ i\}.$$

where $\rightarrow^+$ is the transitive closure of $\rightarrow$.

The notion of minimality corresponds to the intuitive notion of minimality when there are no cycles in the graph induced by $\rightarrow$. However, in case of a cycle in S, all elements of the cycle will belong to the *min*-set. In that case, Algorithm 1 can be further improved by only iterating *one* (arbitrarily chosen) element from each cycle. We leave it as an open problem to

A	B	C	I	W_ϕ
$\bot$	$\bot$	$\bot$	$\{B\}$	$\{B\}$
	$\bullet$		$\{C\}$	$\{C\}$
		$\bullet$	$\{A\}$	$\{A\}$
$\bullet$			$\{B\}$	$\{B\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\{A\}$	$\{A\}$
$\top$			$\emptyset$	$\emptyset$

Figure 3: Trace of the ideal algorithm (Example 1): 6 evaluations

find a better definition of minimality that takes cycles into account—perhaps as an iterative definition starting from elements without predecessors.

3.2 Theoretical algorithm

The above considerations lead to the following iterative algorithm for calculating the limit of the sequence (1).

Algorithm 1

forall $i \in \{1, \ldots, n\} \phi_i := \bot$;
repeat
  $I := min(W_\phi, \xrightarrow{\phi})$;
  forall $i \in I$
    $\phi_i := e_i\ \phi$;
until $I = \emptyset$

Theorem 3.1 *Algorithm 1 terminates with ϕ a fixed point of e*

PROOF: If $I = \emptyset$ implies that $W_\phi = \emptyset$ and hence, by definition of W_ϕ, we have $\phi = e(\phi)$. □

This algorithm is mainly of theoretical interest since it is in general not possible to calculate W_ϕ and $\xrightarrow{\phi}$ exactly without evaluating the defining expression of each equation and thus no work is saved. Hence approximations of these entities are needed to obtain practical algorithms.

Figure 3 shows the trace of the ideal algorithm applied to the toy example from Figure 1. We assume that every time a set of variables is selected for evaluation (I in the sequel) the corresponding expressions are evaluated in parallel.

A	B	C	WS
$\bot$	$\bot$	$\bot$	$\{A, B, C\}$
$\bot$	$\bullet$	$\bot$	$\{C\}$
		$\bullet$	$\{A\}$
$\bullet$			$\{B, C\}$
	$\top$	$\bullet$	$\{C\}$
		$\top$	$\{A\}$
$\top$			$\{B, C\}$
	$\top$	$\top$	$\emptyset$

Figure 4: Trace of the workset algorithm (Example 1): 11 evaluations

4 Scheduling for work-set algorithms

A common approach to calculating the limit of an AKC is to employ a *work-set* algorithm [8] that keeps a work-set W of equations to be iterated next. The algorithm continues to select a set of equations from the work-set for iteration according to some strategy and adds elements that are liable to have changed value to the work-set after each such iteration. The process ends when the work-set is empty. The following is a template work-set algorithm, parameterised on the strategies for selecting the next equation to iterate and for modifying the work-set.

Algorithm 2 (The template work-set algorithm)

forall $i \in \{1, \ldots, n\}$ $\phi_i, \tau_i := \bot$*;*
$W := init_W$*;*
repeat
  $I :=$ strategy(ϕ, W)*;*
  forall $i \in I$
    $\tau_i := e_i\ \phi$ *;*
  $W :=$ modify(W, ϕ, τ)*;*
  $\phi := \tau$*;*
until $I = \emptyset$

The naïve method mentioned above is obtained by always keeping the whole set of equations in the work-set W and using a strategy that after evaluation of equation i selects equation $i + 1 \bmod n$. The theoretical iteration algorithm is obtained by setting

$$\text{strategy}(\phi, W) = min(W_\phi, \stackrel{\phi}{\to}),$$

replacing τ with ϕ and removing the last two assignments to W and ϕ since these no longer play any rôle in the algorithm.

A standard approximation, dating back at least to Kildall [8], of the set W_ϕ of equations that will change value amounts to replacing it with the set of equations in which a variable has changed value [8]. More precisely, the system of equations defines a syntactic *dependency*

A	B	C	I	WS
$\perp$	$\perp$	$\perp$	$\{A,B,C\}$	$\{A,B,C\}$
$\perp$	$\bullet$	$\perp$	$\{C\}$	$\{C\}$
		$\bullet$	$\{A\}$	$\{A\}$
$\bullet$			$\{B\}$	$\{B,C\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\{A\}$	$\{A\}$
$\top$			$\{B\}$	$\{B,C\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\emptyset$	$\emptyset$

Figure 5: Trace of the basic algorithm (Example 1): 10 evaluations

relation, denoted by $\rightarrow$, between the variables: variable f_i depends on variable f_j if f_j occurs in the expression e_i:

$$f_j \rightarrow f_i \equiv f_j \text{ occurs textually in } e_i.$$

From the $\rightarrow$ relation we define the "*image-under-*$\rightarrow$" operation !

$$I! \equiv \{j \mid \exists i \in I : f_i \rightarrow f_j\}$$

that yields the set of indices of the equations containing an x_i with $i \in I$. This is used to define "modify", as shown in Algorithm 3 below. Figure 4 shows the trace of applying a work-set algorithm to the toy example from Figure 1.

The set $I!$ over-estimates the set of variables that change value as a result of changes in the f_i, and it is this set that is added to the work-set after each iteration. As noticed in Section 3, it is possible that there are dependencies between the elements in a set $I!$ of equations to be iterated and only the minimal elements (with respect to $\rightarrow$) are scheduled for evaluation. Instantiating the work-set algorithm above, we arrive at the *basic iteration scheme*:

Algorithm 3 (Basic iteration scheme)

forall $i \in \{1,\ldots,n\}\phi_i, \tau_i := \perp$;
$W := \{1,\ldots,n\}$
repeat
  $I := min(W, \rightarrow)$; % strategy
  forall $i \in I$
    $\tau_i := e_i\ \phi$;
  $W := (W \setminus I) \cup \{i \in I : \phi_i \neq \tau_i\}!$ % modify
  $\phi := \tau$;
until $I = \emptyset$

Proposition 4.1 *If $W_\phi \subseteq W$ holds on entry to the loop then it also holds on exit.*

Theorem 4.2 *Algorithm 3 terminates with a fixed point of e*

PROOF: Since $W_\phi \subseteq W$ trivially holds initially, it also holds when the algorithm terminates. Then $I = \emptyset$ implies $W = \emptyset$ hence $W_\phi = \emptyset$ which implies that ϕ is a fixed point. □

In Figure 5 we show the trace of the basic iteration scheme applied to the example from Figure 1.

5 Experiments

The Basic Iteration Scheme has been used to implement a series of program analyses ranging from pointer analysis of C programs [6] over polyhedral analysis of synchronous SIGNAL programs [2] to class analysis of Java programs. We here report some experimental figures to show that the number of equations evaluated in the different iteration schemes do indeed decrease as we exploit the dependency relation better. For each program analysed we give the number of equations evaluated with the three different iteration schemes presented in the paper.

Analysis	No. of eqns.	Naive	Workset	Basic
C pointer analysis	42 eqs	1050	120	60
Java class analysis	49 eqs	147	75	44
Java class analysis	188 eqs	1692	272	158

As can be seen from the table, the basic algorithm obtained a 40–50 % reduction in the number of expressions that were evaluated during iteration, compared to a standard work-set algorithm. The impact of avoiding superfluous evaluations on the overall running time varies with the type of analysis. We observed significant savings only for the C pointer analysis. This is due to the fact that the expression in the C pointer analysis are all rather complex matrix manipulations whereas the expressions in the Java class analysis are simple set manipulations. Another point is that the computation of the transitive dependency relation is in itself costly. No attempt has been made to optimise this part of the computation, so for the moment the Basic Iteration Scheme only improves the overall running time of the C pointer analysis.

6 Conclusions

The Basic Iteration Scheme (BIS) constitutes a simple algorithm for implementing a fixed point solver. It relies on the dependency relation between variables to improve the standard work-set algorithm by only scheduling variables that do not depend on other variables ready for evaluation. This theoretical improvement in the number of expressions to evaluate manifests itself in practical experiemnts. There are further related issues that we have not addressed here:

- The BIS exploits the dependency relation $\rightarrow$ to avoid certain useless re-evaluations of expressions and has been used to implement several program analyzers. The practical experiments reported in Section 5 show that the overhead incurred by calculating the sets $min(W, \rightarrow)$ in certain cases exceeds the savings obtained. Thus, it is necessary to experiment with cheaper approximations to these sets.

- The top-down solvers [3, 5] will *suspend* evaluation of an expression if some of its variables need re-evaluation, and instead schedule these variables for evaluation. The top-down solvers use an evaluation stack combined with cycle detection to implement this. We obtain an effect similar to this by calculating the $min(W, \rightarrow)$-sets, by which

we suspend evaluation of a variable if it depends on other variables that migt evolve. A precise comparison between the two algorithms is challenging future work.

Finally, we remark that in certain cases, we are only interested in the value of some of the variables in the system of equations. An algorithm for calculating such a *local fixed point* was given in [10]. This algorithm can be expressed quite naturally as an extension to the BIS. We will show this extension in a longer version of the paper.

References

[1] A. Arnold and D. Niwinski. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2001.

[2] F. Besson, T. Jensen, and J.-P. Talpin. Polyhedral analysis for synchronous languages. In G. Filé, editor, *Proc. of 7th Int. Symp. on Static Analysis*. Springer LNCS vol. 1694, 1999.

[3] B. Le Charlier and P. van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. ACM Trans. on Programmling Languages and Systems, 16(1):35–101, 1994.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of 2. Int. Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod.

[5] C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2-3):137–162, 1999.

[6] P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming, ESOP'96*, volume 1058 of *LNCS*, pages 125–140, Linköping, Sweden, April 1996. Springer-Verlag.

[7] M. S. Hecht. *Flow Analysis of Computer Programs.* Programming Languages series. North-Holland, New York, 1977.

[8] G. Kildall. A unified approach to global program optimization. In *Proc. of ACM Symp. on Principles of Programming Languages*, 1973.

[9] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[10] B. Vergauwen, J. Wauman, and J. Levi. Efficient fixpoint computation. In B. Le Charlier, editor, *Proc. of 1st Int. Static Analysis Symp.*, pages 314–328. Springer LNCS vol. 864, 1994.

Congruences of Modal μ-algebras

Luigi Santocanale*

LaBRI - Université Bordeaux I
santocan@labri.fr

If $f(y)$ is an order preserving endofunction of a closed ordered monoid, then the two inequations

$$f(g(x)) \otimes x \leq g(x) \qquad\qquad g(f(x) \multimap x) \leq x$$

determine an order preserving endofunction $g(x)$ as $\mu_y.(f(y) \otimes x)$, the parameterized least prefixed point of $f(y) \otimes x$. This observation is the core of a method to axiomatize by equations the least prefixed point [7]. When this method is applied to the propositional Boolean modal μ-calculus, it shows that Kozen's axiomatization [4] can be turned into an equivalent equational axiomatization that is complete with respect to Kripke frames by Walukiewicz's theorem [9].

Being able to equationally axiomatize the propositional Boolean modal μ-calculus implies that its algebraic models form a variety of algebras, in the usual sense [3]. We call these models modal μ-algebras. The main characteristic that distinguishes a variety from a quasivariety – i.e. a class of models that can be axiomatized by equational implications – is that such a class is closed under homomorphic images defined from congruences. Thus, in a variety of algebras, there is always a bijection between quotients of an algebra (categorically, strong epimorphisms) and congruences on the algebra.

Modal μ-algebras are therefore closed under homomorphic images; we have investigated congruences of modal μ-algebras and found a characterization suitable for algebraic computations. In this talk we will present this characterization and illustrate its use by lifting some classical results from modal logic and modal algebras [5] to the setting of modal μ-algebras.

The characterization of congruences can also be exploited in the context of verification of systems, as these results can be reformulated in a logical framework. For example, it could be useful to analyze a restricted class of transition systems where given subsets of states are known to satisfy a finite number of algebraic relations. Some propositions of the modal μ-calculus could hold true in these models, and it would be desirable to know whether their truth is a consequence of Kozen's axiomatization and of those algebraic relations. This is a particular instance of a common problem in logic: is a proposition derivable from a finite set of propositions that are assumed as axioms? Among the results that we present is a deduction theorem for the propositional Boolean modal μ-calculus; this theorem allows to reduce derivability from a finite set of axioms to derivability from no axioms. As the latter problem is well known to be decidable for the modal μ-calculus, then derivability from a finite set of propositions becomes decidable as well.

1 Definitions

In the following Σ is a finite non empty alphabet of actions and X is a countable set of variables.

Definition 1.1. *We define by induction a set $\mathcal{T}(Y)$ of μ-terms in context Y, for each finite subset Y of X.*

- *If $x \in Y$, then $x \in \mathcal{T}(Y)$.*
- *$\top \in \mathcal{T}(Y)$, and if $t_1, t_2 \in \mathcal{T}$, then $t_1 \wedge t_2 \in \mathcal{T}(Y)$.*
- *If $t \in \mathcal{T}(Y)$, then $\neg t \in \mathcal{T}(Y)$,*

* This work was developed at the Department of Computer Science, University of Calgary. The author aknowledge financial support from Canada's NSERC, the Pacific Institute for the Mathematical Sciences, and from the European Community through a Marie Curie Individual Fellowship.

- *If $t \in \mathcal{T}(Y)$ and $\sigma \in \Sigma$, then $[\sigma]t \in \mathcal{T}(Y)$.*
- *If $t \in \mathcal{T}(Y)$ and $x \in Y$ is a variable that occurs in t in the scope of an even number of symbols $\neg$, then $\mu_x.t \in \mathcal{T}(Y \setminus \{x\})$.*

Definition 1.2. *A* modal μ-algebra *is a pair $\langle \mathcal{A}, |-| \rangle$, where $\mathcal{A} = \langle A, \top, \wedge, \neg, \{[\sigma]\}_{\sigma\in\Sigma} \rangle$ is a modal algebra, see [5, §2.2.2], and $|-|$ is an interpretation of terms $t \in \mathcal{T}(Y)$ as functions $|t| : A^Y \longrightarrow A$ with the following properties:*

- *$|x|$ is the projection on the x-coordinate.*
- *$|-|$ agrees with the structure of the modal algebra $\mathcal{A}$, that is, $|\top| = \top$, $|t_1 \wedge t_2| = |t_1| \wedge |t_2|$, $|\neg t| = \neg |t|$, and $|[\sigma]t| = [\sigma]|t|$.*
- *The function*
$$|\mu_x.t| : A^{Y\setminus\{x\}} \longrightarrow A$$
is the parameterized least prefixed point of
$$|t| : A^{\{x\}} \times A^{Y\setminus\{x\}} \longrightarrow A\,.$$

Note that if $\mathcal{A}$ is a modal algebra, then there exists at most one interpretation $|-|$ such that $\langle \mathcal{A}, |-| \rangle$ is a modal μ-algebra. Thus we can leave the interpretation $|-|$ in the background and, by saying that $\mathcal{A}$ is a modal μ-algebra, we mean that $\mathcal{A}$ is a modal algebra and that such interpretation $|-|$ can be found. Accordingly, if $t \in \mathcal{T}(Y)$, we simply write t for the function $|t|$. An *homomorphism* of modal μ-algebras from $\mathcal{A}$ to $\mathcal{B}$ is a function f between the underlying sets such that for all $t \in \mathcal{T}(Y)$ $f \circ t = t \circ f^Y$. Other operations on modal μ-algebras can be defined in terms of the given operations, for example:

$$x \rightarrow y = \neg(x \wedge \neg y) \qquad x \leftrightarrow y = (x \rightarrow y) \wedge (y \rightarrow x)$$
$$\langle\sigma\rangle x = \neg[\sigma]\neg x \qquad \nu_x.t(x,y) = \neg\mu_x.\neg t(\neg x, y)\,.$$

Definition 1.3. *A* congruence *on a modal μ-algebra $\mathcal{A}$ is an equivalence relation $\sim \subseteq A \times A$ such that, for every μ-term $t \in \mathcal{T}(Y)$ and every pair $a, b \in A^Y$, if $a_y \sim b_y$ for each $y \in Y$, then $t(a) \sim t(b)$ as well.*

Given a modal μ-algebra $\mathcal{A}$ and a congruence $\sim$ on $\mathcal{A}$, we can define the homomorphic image $\mathcal{A}/\sim$ as follows. Its elements are equivalence classes under $\sim$; we denote by $[a]$ the equivalence class of $a \in A$. If $t \in \mathcal{T}(Y)$ and $\{\,[a_y]\,\}_{y\in Y}$ is a collection of equivalence classes, then we can define

$$t(\{\,[a_y]\,\}_{y\in Y}) = [\,t(\{\,a_y\,\}_{y\in Y})\,]\,.$$

This is well defined, as usual, and $\mathcal{A}/\sim$ is certainly a modal algebra. By the results in [7] this is also a modal μ-algebra.

2 The Characterization

On every modal μ-algebra we define the operation

$$[\Sigma^*]x = \nu_y.(x \wedge \bigwedge_{\sigma\in\Sigma} [\sigma]y)\,.$$

Recall that a *filter* on a modal μ-algebra $\mathcal{A}$ is a subset F of A such that $\top \in F$ and $a \wedge b \in F$ if and only if $a, b \in F$. We say that a filter F is *open* if $f \in F$ implies that $[\Sigma^*]f \in F$.

Theorem 2.1. *There is a bijection between congruences on a modal μ-algebra $\mathcal{A}$ and open filters of $\mathcal{A}$.*

The bijection is as follows: a congruence $\sim \subseteq A \times A$ is sent to the filter $F_\sim$ defined as:

$$x \in F_\sim \text{ iff } x \sim \top .$$

The filter $F_\sim$ is open, since if $x \sim \top$, then $[\Sigma^*]x \sim [\Sigma^*]\top = \top$. Conversely, given an open filter F, we define the relation $\sim_F$ as follows:

$$a \sim_F b \text{ iff } a \leftrightarrow b \in F .$$

This is an equivalence relation, as it is usual in the context of Boolean algebras. To prove that this is also a congruence we use the following lemma:

Lemma 2.2. *For all $c \in A$, $t \in \mathcal{T}(Y)$ and $a, b \in A^Y$, if for all $x \in Y$ $[\Sigma^*]c \leq a_x \leftrightarrow b_x$, then $[\Sigma^*]c \leq t(a) \leftrightarrow t(b)$ as well.*

A filter F is *principal* if it is of the form $\{ b \mid a \leq b \}$ for some a; in this case we say that F is generated by a and write $F = F_a$. Remark that a principal filter F_a is open if and only if it is generated by some element of the form $[\Sigma^*]b$. Indeed, if F_a is open, then $a = [\Sigma^*]a$ since $[\Sigma^*]a \leq a$ and $[\Sigma^*]a \in F_a$. On the other hand, a filter of the form $F_{[\Sigma^*]b}$ is open since $[\Sigma^*]b$ is a fixed point of $[\Sigma^*]$.

Let now $\sim$ be the principal congruence generated by the pair (a, b). From the above discussion, it follows that $c \sim d$ if and only if $c \leftrightarrow d \in F_{[\Sigma^*](a \leftrightarrow b)}$, i.e. if and only if

$$[\Sigma^*](a \leftrightarrow b) \leq c \leftrightarrow d .$$

Thus, we have observed that:

Corollary 2.3. *The variety of modal μ-algebras has equationally definable principal congruences.*

See [1], for a general study of varieties with this property. As noted there, this property corresponds to a deduction theorem for Kozen's axiomatization of the Boolean modal μ-calculus. Indeed, write $\Gamma \vdash p$ if it is possible to derive the proposition p by adding to the set of axioms all the propositions in Γ. Then, from $\Gamma \cup \{q\} \vdash p$, it follows that $\Gamma \vdash [\Sigma^*]q \to p$.

As $F_{[\Sigma^*](a \wedge b)}$ is the least filter containing both $F_{[\Sigma^*]a}$ and $F_{[\Sigma^*]b}$, we can argue that every finitely generated congruence is principal. It follows that if $\sim$ is the congruence generated by the pairs (a_i, b_i), $i = 1, \ldots, n$, then $[c] \leq [d]$ in $\mathcal{A}/\sim$ if and only if $[\Sigma^*](\bigwedge_{i=1,\ldots,n} a_i \leftrightarrow b_i) \leq c \leftrightarrow d$ in $\mathcal{A}$. Thus:

Corollary 2.4. *If $\sim$ is a finitely generated congruence on $\mathcal{A}$, then the order relation of $\mathcal{A}/\sim$ is decidable, provided the one of $\mathcal{A}$ is decidable.*

Since it is easily argued that $[\Sigma^*]\bot = \bot$ and that the greatest open filter contained in both $F_{[\Sigma^*]a}$ and $F_{[\Sigma^*]b}$ is the principal filter generated by $[\Sigma^*]([\Sigma^*]a \vee [\Sigma^*]b)$, it follows that:

Corollary 2.5. *The set of finitely generated congruences of $\mathcal{A}$ is a sublattice of the lattice of congruences on $\mathcal{A}$. Moreover it is a dual Heyting algebra.*

The last statement can be seen as follows. The signature of modal μ-algebras extends the one of Boolean algebras, thus the variety of modal μ-algebras is congruence distributive. Moreover it was shown in [1] that in a variety with equationally definable principal congruences the lattice of finitely generated congruences of an algebra is a dual Brouwerian semilattice.

3 Subdirectly Irreducible Algebras

Recall that an algebra is *subdirectly irreducible* if it has a least proper congruence. Subdirectly irreducible algebras are interesting as any algebra can be represented as a subdirect product of subdirectly irreducible algebras. The following proposition is easily derived:

Proposition 3.1. *A modal μ-algebra $\mathcal{A}$ is subdirectly irreducible if and only if there exists an element $a \in A$, distinct from $\top$, such that if $b \in A$ and $b \neq \top$ then $[\Sigma^*]b \leq a$.*

A frame $\mathcal{F}$ is a pair $\langle X, \{R_\sigma\}_{\sigma \in \Sigma} \rangle$, where X is a set and, for each $\sigma \in \Sigma$, R_σ is a relation on X. The powerset $\mathcal{P}(X)$, with the modal operators

$$[\sigma]S = \{\, x \in X \mid \forall y \in X (x R_\sigma y \Rightarrow y \in S) \,\}\,,$$

is then a modal μ-algebra. The graph of a frame $\mathcal{F}$ is the pair $\langle X, \bigcup_{\sigma \in \Sigma} R_\sigma \rangle$. A *source* in $\mathcal{F}$ is an element $x \in X$ such that every $y \in X$ is reachable from x in the graph of $\mathcal{F}$. A frame is *connected* if every element is a source.

Proposition 3.2. *The algebra of a frame is subdirectly irreducible if and only if the frame has a source. The algebra of a frame is simple if and only if the frame is connected.*

The above statements are easily derived when the characterization of congruences is dualized in terms of ideals. Observe that the analogous statements fail for modal algebras. They are true for modal μ-algebras since reachability is algebraically expressible.

Corollary 3.3. *The variety of modal μ-algebras is not residually small.*

See [8] for the definition of a residually small variety. To prove the corollary it is enough to construct connected frames of arbitrary cardinality.

4 Congruence Extension and Amalgamation

Among the consequences of corollary 2.3 is that the variety of modal μ-algebras has the *congruence extension property.* This property also holds for modal algebras and states that if $(i, \mathcal{B})$ is an extension of a modal μ-algebra $\mathcal{A}$ and $(p, \mathcal{C})$ is an homomorphic image of $\mathcal{A}$, then it is possible to find an extension j of $\mathcal{C}$ and an homomorphic image q of $\mathcal{B}$ such that $q \circ i = j \circ p$:

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{\;i\;} & \mathcal{B} \\ \downarrow{\scriptstyle p} & & \downarrow{\scriptstyle q} \\ \mathcal{C} & \xrightarrow{\;j\;} & \mathcal{D} \end{array}$$

Using the finite model theorem for the modal μ-calculus, the fact that the variety of modal μ-algebras has equationally definable principal congruences, and the congruence extension property, the following fact can easily be proved:

Proposition 4.1. *Let $\mathcal{V}$ be the variety of modal μ-algebras with additional constants subject to a finite number of given algebraic relations. Then $\mathcal{V}$ is residually finite, that is, if an equation holds in every finite member of $\mathcal{V}$, then it holds in every member of $\mathcal{V}$.*

The congruence extension property equivalently states that the pushout of a monomorphism along a quotient is again a monomorphism. As monomorphisms and homomorphic images form a factorization system in any category of algebras, it becomes interesting to investigate the analogous property:

Definition 4.2. *A variety of algebras $\mathcal{V}$ has the* amalgamation property*, if for any algebra $\mathcal{A}$ in $\mathcal{V}$ and any two extensions $(i_1, \mathcal{B}_1)$ and $(i_2, \mathcal{B}_2)$ of $\mathcal{A}$, there exists extensions j_1, j_2 such that*

$j_1 \circ i_1 = j_2 \circ i_2$:

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{i_2} & \mathcal{B}_1 \\ {\scriptstyle i_1}\downarrow & & \vdots\,{\scriptstyle j_2} \\ \mathcal{B}_1 & \xrightarrow{j_1} & \mathcal{C} \end{array}$$

Equivalently, the amalgamation property states that the pushout of a monomorphism along a monomorphism is again a monomorphism. The congruence extension property and the amalgamation property together imply the *transferability property* which states that the pushout of a monomorphism along any homomorphism is again a monomorphism.

The characterization of congruences given here is of help in relating amalgamation to interpolation, a property of a syntactical nature. This has been done for modal algebras and modal logic [6] and can be generalized to modal μ-algebras, as follows.

Definition 4.3. *A variety of modal μ-algebras $\mathcal{V}$ has the* strong amalgamation property *if it has the amalgamation property as in 4.2 and moreover there exists $a \in A$ such that $b_1 \leq i_1(a)$ and $i_2(a) \leq b_2$, whenever $b_1 \in B_1$, $b_2 \in B_2$ and $j_1(b_1) \leq j_2(b_2)$.*

In the following definition $F_{\mathcal{V}}(X)$ denotes the modal μ-algebra in the variety $\mathcal{V}$ freely generated by the set X.

Definition 4.4. *A variety of modal μ-algebras $\mathcal{V}$ has the* interpolation property *if and only if, whenever $p \in F_{\mathcal{V}}(X)$, $q \in F_{\mathcal{V}}(Y)$ and $p \leq q$ in $F_{\mathcal{V}}(X \cup Y)$, then there exists $r \in F_{\mathcal{V}}(X \cap Y)$ such that $p \leq r$ in $F_{\mathcal{V}}(X)$ and $r \leq q$ in $F_{\mathcal{V}}(Y)$.*

Thus:

Proposition 4.5. *A variety of modal μ-algebras has the interpolation property if and only if it has the strong amalgamation property.*

Recently the interpolation property of modal logic has been extended to the propositional Boolean modal μ-calculus [2]. It follows that the variety of modal μ-algebras has the interpolation property, as we have defined it in 4.4, and therefore the amalgamation property.

References

[1] W. J. Blok and D. Pigozzi. On the structure of varieties with equationally definable principal congruences. I. *Algebra Universalis*, 15(2):195–227, 1982.

[2] G. D'Agostino and M. Hollenberg. Logical questions concerning the μ-calculus: interpolation, Lyndon and łoś-Tarski. *J. Symbolic Logic*, 65(1):310–332, 2000.

[3] G. Grätzer. *Universal algebra.* Springer-Verlag, New York, second edition, 1979.

[4] D. Kozen. Results on the propositional μ-calculus. *Theoret. Comput. Sci.*, 27(3):333–354, 1983.

[5] M. Kracht. *Tools and techniques in modal logic.* North-Holland Publishing Co., Amsterdam, 1999.

[6] L. Maksimova. Amalgamation and interpolation in normal modal logic. *Studia Logica*, 50(3-4):457–471, 1991. Algebraic logic.

[7] L. Santocanale. On the equational definition of the least prefixed point. volume 2136 of *LNCS*, pages 645–656, 2001.

[8] W. Taylor. Residually small varieties. *Algebra Universalis*, 2:33–53, 1972.

[9] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. *Inform. and Comput.*, 157(1-2):142–182, 2000.

Recent BRICS Notes Series Publications

NS-02-2 Zoltán Ésik and Anna Ingólfsdóttir, editors. *Preliminary Proceedings of the Workshop on Fixed Points in Computer Science, FICS '02,* (Copenhagen, Denmark, July 20 and 21, 2002), June 2002. iv+81 pp.

NS-02-1 Anders Møller and Michael I. Schwartzbach. *Interactive Web Services with Java: JSP, Servlets, and JWIG.* April 2002. 99 pp.

NS-01-8 Anders Møller and Michael I. Schwartzbach. *The XML Revolution (Revised).* December 2001. 186 pp. This revised and extended report superseeds the earlier BRICS Report NS-00-8.

NS-01-7 Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '01,* (Aalborg, Denmark, August 25, 2001), August 2001. vi+97 pp.

NS-01-6 Luca Aceto and Prakash Panangaden, editors. *Preliminary Proceedings of the 8th International Workshop on Expressiveness in Concurrency, EXPRESS '01,* (Aalborg, Denmark, August 20, 2001), August 2001. vi+139 pp.

NS-01-5 Flavio Corradini and Walter Vogler, editors. *Preliminary Proceedings of the 2nd International Workshop on Models for Time-Critical Systems, MTCS '01,* (Aalborg, Denmark, August 25, 2001), August 2001. vi+ 127pp.

NS-01-4 Ed Brinksma and Jan Tretmans, editors. *Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '01,* (Aalborg, Denmark, August 25, 2001), August 2001. viii+156 pp.

NS-01-3 Martin Hofmann, editor. *Proceedings of the 3rd International Workshop on Implicit Computational Complexity, ICC '01,* (Aarhus, Denmark, May 20–21, 2001), May 2001. vi+144 pp.

NS-01-2 Stephen Brookes and Michael Mislove, editors. *Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01,* (Aarhus, Denmark, May 24–27, 2001), May 2001. viii+279 pp.