Preface

This volume contains the papers presented at LSFA 2017: the 12th Workshop on Logical and Semantic Frameworks, with Applications held on 23-24 September 2017 in Brasília. LSFA 2017 is a satellite event of the 26th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux 2017), the 11th International Symposium on Frontiers of Combining Systems (FroCoS 2017) and the 8th International Conference on Interactive Theorem Proving (ITP 2017).

LSFA 2017 aims to be a forum for presenting and discussing work in progress, and therefore to provide feedback to authors on their preliminary research. The proceedings are produced after the meeting, so that authors can incorporate this feedback in the published papers. Logical and semantic frameworks are formal languages used to represent logics, languages and systems. These frameworks provide foundations for formal specification of systems and programming languages, supporting tool development and reasoning.

There were 23 submissions for LSFA 2017, each of which was reviewed by 3 program committee members. The committee decided to accept 18 papers, out of which 16 were submited by the authors for these proceedings.

Many people helped to make LSFA 2017 a success. First, we wish to thank the support we had from the organisers of the Tableaux+FroCoS+ITP conferences: Cláudia Nalon, Daniele Nantes, Elaine Pimentel and João Marcos. We thank the CNPq and FAPDF for their financial support. We would also like to thank Easychair for making the management of this event very smooth and easy. We are indebted to the program committee members and the external referees for their careful and efficient work in the reviewing process. Finally we are grateful to the authors for submitting their work to LSFA 2017.

September 12, 2017

Sandra Alves
Renata Wassermann

Program Committee

Sandra Alves	University of Porto
Carlos Areces	FaMAF - Universidad Nacional de Córdoba
Mauricio Ayala-Rincon	Universidade de Brasilia
Veronica Becher	Universidad de Buenos Aires
Mario Benevides	Programa de Engenharia de Sistemas e Computação COPPE/Sistemas; Departamento de Ciência da Computação DCC / Instituto de Matemática IM; Universidade Federal do Rio de Janeiro UFRJ
Walter Carnielli	Centre for Logic, Epistemology and the History of Science – CLE
Carlos Castro	Professor
Kaustuv Chaudhuri	INRIA
Marcelo Coniglio	University of Campinas
Flavio L. C. De Moura	Universidade de Brasilia
Valeria De Paiva	University of Birmingham
Santiago Escobar	Universitat Politècnica de València
Amy Felty	University of Ottawa
Maribel Fernandez	KCL
Marcelo Finger	Universidade de Sao Paulo
Ichiro Hasuo	National Institute of Informatics
Edward Hermann Haeusler	PUC-Rio
Delia Kesner	Université Paris-Diderot
Bjoern Lellmann	TU Vienna
Vivek Nigam	Universidade Federal da Paraíba
Jorge Petrucio Viana	Universidade Federal Fluminense
Elaine Pimentel	UFRN
Jorge A. Pérez	University of Groningen
Giselle Reis	Carnegie Mellon University - Qatar
Camilo Rocha	Department of Electronics and Computer Science, Pontificia Universidad Javeriana Cali
Simona Ronchi Della Rocca	Universita' di Torino - dipartimento di Informatica
Alvaro Tasistro	Universidad ORT Uruguay
Christian Urban	King's College London
Renata Wassermann	University of São Paulo

Additional Reviewers

Abriola, Sergio
Cano, Mauricio
Machado, Vitor
Pinto, Darllan
Rodriguez, Ricardo Oscar
Testa, Rafael
Valencia, Frank
Ziliani, Beta

Model-Theoretic Conservative Extension for Definitional Theories

Arve Gengelbach[a,1] Tjark Weber[a,2]

[a] *Department of Information Technology*
Uppsala University
Uppsala, Sweden

Abstract

Many logical frameworks allow extensions, i. e. the introduction of new symbols, by definitions. Different from asserting arbitrary non-logical axioms, extensions by definitions are expected to be conservative: they should entail no new theorems in the original language. The popular theorem prover Isabelle implements a variant of higher-order logic that allows *ad hoc* overloading of constants. In 2015, Kunčar and Popescu introduced *definitional theories*, which impose a non-circularity condition on constant and type definitions in this logic, and showed that this condition is sufficient for definitional extensions to preserve consistency. We strengthen and generalise this result by showing that extensions of definitional theories are model-theoretic conservative, i. e. every model of the original theory can be expanded to a model of the extended theory.

Keywords: higher-order logic, conservative theory extension, model-theoretic conservativity, definitional theories, Isabelle

1 Introduction

Among the many different mechanisms for extending theories by definitions, particularly constant and type definitions [3,12], the one used by the theorem prover Isabelle [9] had flaws. Isabelle implements polymorphic higher-order logic with *ad hoc* overloading. Users extend a theory incrementally by defining constants and types (or in the case of overloaded constants, by defining constant instances for previously defined types). A key strength of overloading is the separation of the declaration of a constant from its instance definitions. However, by combining type definitions with *ad hoc* overloading of constants, it was possible to introduce an inconsistent extension of a theory [6]. For a simple example, consider a theory that declares a polymorphic constant c_α. Define a type τ by $\tau \equiv \{\mathsf{True}, c_{\mathsf{bool}}\}$. Next, define the constant instance c_{bool} by $c_{\mathsf{bool}} \equiv \neg(\forall x_\tau, y_\tau.\ x_\tau \doteq y_\tau)$. It follows that $c_{\mathsf{bool}} \doteq \mathsf{True}$ iff τ is a singleton iff $c_{\mathsf{bool}} \doteq \mathsf{False}$, a contradiction.

[1] Email: arve.gengelbach@it.uu.se
[2] Email: tjark.weber@it.uu.se

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

To address this issue, Kunčar and Popescu [6] in 2015 introduced *definitional theories* for Isabelle. In the previous example, τ (through its definition) depends on c_{bool}, and c_{bool} depends on τ. Definitional theories disallow circular dependencies between constant and type definitions. Using a novel semantics for higher-order logic that interprets polymorphic types as macros for families of ground types, Kunčar and Popescu showed that definitional theories preserve consistency, i. e. every well-formed definitional theory has a model. Their acyclicity check has since been integrated into Isabelle [5].

As pointed out in [6], consistency is "a crucial, but rather weak property." It merely ensures that definitional theories do not prove False. Extensions by definitions are generally expected to satisfy a much stronger property known as *conservativity* [1,13]. Conservativity comes in a proof-theoretic (syntactic) and in a model-theoretic (semantic) flavour. In this paper, we are primarily concerned with the latter. Roughly, an extension D' of a theory D is *model-theoretic conservative* if every model $\mathcal{M}$ of D can be expanded to a model $\mathcal{M}'$ of D'. (We give a more precise definition in Section 3.) Note that model-theoretic conservativity immediately implies consistency: if D has a model, then so does D'.

We observe that extensions of definitional theories are *not* model-theoretic conservative if we require that the expanded model leaves the interpretation of all constants that are defined in D unchanged. For a simple counterexample, assume constants c_{bool} and d_{bool}, let $D := \{c_{\mathsf{bool}} \equiv d_{\mathsf{bool}}\}$ and $D' := D \cup \{d_{\mathsf{bool}} \equiv \mathsf{True}\}$. Then any model of D that interprets c_{bool} as false cannot be expanded (in the above sense) to a model of D'. The issue here is again that definitions may be provided separately from declarations: c_{bool} depends on d_{bool}, but the definition of d_{bool} is only provided in D'. In general, an extension by definitions in this logic does not imply an extension of the signature, i. e. the introduction of new symbols.

This example motivates a modified (more permissive) definition of model expansion. Our main contributions in this paper are:

(i) We define a notion of model expansion that is suitable for definitional theories, where definitions may be provided separately from declarations; and

(ii) we show that extensions of definitional theories are model-theoretic conservative with respect to this notion of model expansion.

This strengthens and generalises the consistency result previously obtained by Kunčar and Popescu [6].

The rest of the paper is structured as follows. We introduce the language of polymorphic higher-order logic and definitional theories in Section 2. Our main results, including a suitable notion of model expansion and a proof of model-theoretic conservativity, are presented in Section 3. We give an overview of related work in Section 4 and conclude with a discussion of future work, in particular with regard to proof-theoretic conservativity, in Section 5.

2 Background

This section introduces the language of polymorphic higher-order logic (HOL), definitional theories and their semantics. We follow the notation and naming conven-

tions of [6], to which we refer the reader for additional motivation and examples. We do not describe the deductive system of HOL [10], since it is not relevant in the context of this paper.

2.1 The Language of Polymorphic HOL

The syntax of polymorphic HOL is that of the simply-typed lambda calculus, enriched with a first-order language of types. We fix an infinite set TVar of *type variables*, ranged over by α, β, and an infinite set Var of *(term) variables*, ranged over by x, y.

A *signature* is a four-tuple $(K, \mathrm{arOf}, \mathsf{Const}, \mathrm{tpOf})$, where K is a countable set of symbols called *type constructors*, and Const is a countable set of symbols called *constants*. Each type constructor has an associated *arity* that is given by the function $\mathrm{arOf}\colon K \to \mathbb{N}$. Each constant has an associated *type* that is given by the function $\mathrm{tpOf}\colon \mathsf{Const} \to \mathsf{Type}$, where the set Type, ranged over by σ, τ, is defined inductively as the smallest set such that

- $\mathsf{TVar} \subseteq \mathsf{Type}$, and
- $(\sigma_1, \ldots, \sigma_n)k \in \mathsf{Type}$ whenever $k \in K$, $\mathrm{arOf}(k) = n$ and $\sigma_1, \ldots, \sigma_n \in \mathsf{Type}$.

For the remainder of this paper, we will assume a fixed signature. Moreover, we assume that K contains the following *built-in* type constructors:

- bool of arity 0,
- ind of arity 0,
- $\Rightarrow$ a right-associative type constructor of arity 2.

We also assume that Const contains the following *built-in* constants:

- $\to$ of type $\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$,
- $\doteq$ of type $\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$,[3]
- some of type $(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha$,
- zero of type ind,
- succ of type $\mathsf{ind} \Rightarrow \mathsf{ind}$.

We say that a type σ is *built-in* if either $\sigma = \mathsf{bool}$ or $\sigma = \mathsf{ind}$ or if there exist types $\sigma_1, \sigma_2 \in \mathsf{Type}$ such that $\sigma = \sigma_1 \Rightarrow \sigma_2$. Thus any type is built-in whose type constructor is either $()\mathsf{bool}$, $()\mathsf{ind}$ or $(\sigma_1, \sigma_2) \Rightarrow$. For a set of types $T \subseteq \mathsf{Type}$ let $\mathsf{Cl}(T)$ denote the *built-in closure* of T, defined inductively as the smallest set such that

- $T \cup \{\mathsf{bool}, \mathsf{ind}\} \subseteq \mathsf{Cl}(T)$, and
- $\sigma_1 \Rightarrow \sigma_2 \in \mathsf{Cl}(T)$ whenever $\sigma_1, \sigma_2 \in \mathsf{Cl}(T)$.

A *type substitution* is a function $\rho\colon \mathsf{TVar} \to \mathsf{Type}$ that replaces type variables by types. We denote the set of type substitutions by TSubst. Generally we extend a type substitution ρ to a function on Type by defining, for each type constructor

[3] We use $\doteq$ rather than $=$ to avoid confusion with equality in our meta language.

$k \in K$,

$$\rho\left((\sigma_1, \ldots, \sigma_{\mathrm{arOf}(k)})k\right) := \left(\rho(\sigma_1), \ldots, \rho(\sigma_{\mathrm{arOf}(k)})\right) k$$

Let σ, $\sigma' \in \mathsf{Type}$ be two types. If there exists a type substitution $\rho \in \mathsf{TSubst}$ such that $\rho(\sigma) = \sigma'$ we write $\sigma' \leq \sigma$ and say that σ' is an *instance* of σ.

The set of *typed constants* CInst is a subset of the cartesian product $\mathsf{Const} \times \mathsf{Type}$, where $(c, \sigma) \in \mathsf{CInst}$ if and only if σ is an instance of $\mathrm{tpOf}(c)$. We use c_σ as shorthand notation for the tuple (c, σ).

The *terms* of our language are given by the following grammar, for $x \in \mathsf{Var}$, σ, $\sigma' \in \mathsf{Type}$, $c_\sigma \in \mathsf{CInst}$:

$$t \quad ::= \quad x_\sigma \quad | \quad c_\sigma \quad | \quad (t_{\sigma' \Rightarrow \sigma}\ t'_{\sigma'})_\sigma \quad | \quad (\lambda x_{\sigma'}.t_\sigma)_{\sigma' \Rightarrow \sigma}$$

We may write t for t_σ when there is no risk of ambiguity. We require all terms to be *well-typed*, i. e. in $t\ t'$ the type of t' has to be the same as the argument type of t. Equality of terms is considered modulo α-equivalence. The set of all terms is denoted by Term.

We extend tpOf to terms by defining $\mathrm{tpOf}(t_\sigma) := \sigma$. We say that a term is *built-in* if its type is built-in. For a set M of terms (or types), we define M^* to mean the subset of all non-built-in terms (or types). For example Type^* are all built-in types and CInst^* are all built-in constant instances, i.e. all constant instances whose type is built-in.

We extend type substitutions to terms by defining $\rho(t_\sigma) := t_{\rho(\sigma)}$. We say that $t_{\sigma'}$ is an *instance* of t_σ, written $t_{\sigma'} \leq t_\sigma$, if σ' is an instance of σ.

We define the function $\mathrm{types}\colon \mathsf{Term} \to \mathcal{P}(\mathsf{Type})$ to collect all types that syntactically occur in a given term. For instance, if K contains a type constructor list of arity 1 (written in postfix notation), $\mathrm{types}(c_{\mathsf{bool\ list}}) = \{\mathsf{bool}, \mathsf{bool\ list}\}$. Likewise, we define the function $\mathrm{consts}\colon \mathsf{Term} \to \mathcal{P}(\mathsf{CInst})$ to collect all constant instances that syntactically occur in a given term.

We say that a type is *ground* if it contains no type variables. The set of ground types is denoted by GType. We denote the set of type substitutions that map type variables to ground types only by $\mathsf{GTSubst}$. A constant instance $c_\sigma \in \mathsf{CInst}$ is *ground* if its type σ is ground. We denote the set of ground constant instances by GCInst.

2.2 Definitional Theories

Definitional theories are theories that consist of definitions $a \equiv b$, with a defining term b on the right-hand side and the name of the introduced constant or type a on the left-hand side.

As the semantics of definitional theories use the logical constants '$\forall$' and '$\exists$', we require that the underlying theory provides axioms for these constants, e. g.

$$\forall \doteq \lambda P_{\alpha \to \mathsf{bool}}.(P \doteq (\lambda x.\mathsf{True})),$$

in a logically equivalent manner like the theory *LOG* extends the minimal theory of HOL [10, Section 2.4.2].

We write $\mathrm{TV}(\sigma)$ for the set of type variables that syntactically occur in the type σ and extend this to terms: for a term t we abbreviate $\mathrm{TV}(t)$ for $\mathrm{TV}(\mathrm{tpOf}(t))$.

Definition 2.1 [Definitional Theory] We call a finite set where each element has the shape

- $c_\tau \equiv t$ with $t \in \mathsf{Term}$ a closed term and $\mathrm{TV}(t) \subseteq \mathrm{TV}(\tau)$, or
- $\tau \equiv t$ with $\mathrm{tpOf}(t) = \sigma \Rightarrow \mathsf{bool}$ (for a $\sigma \in \mathsf{Type}$)

(for c_τ constant, τ a type and t a term) a *definitional theory*. We say *constant instance definition* for $c_\tau \equiv t$ and *type definition* for $\tau \equiv t$.

Definitional theories contain definitions for constant instances which are defined by a term and definitions of types which are defined by a predicate. The semantics of $\equiv$ are defined later on. First introduced by [6], certain definitional theories guarantee consistency. These are the so called *well-formed* definitional theories, which rely on *orthogonality*:

Definition 2.2 [Orthogonal Types and Constant Instances] Let $\tau \in \mathsf{Type}$ and $\sigma \in \mathsf{Type}$ be two types.

$$\tau \# \sigma \quad :\Longleftrightarrow \quad \forall \gamma \in \mathsf{Type}, \theta \in \mathsf{TSubst} : \neg(\theta(\tau) = \gamma \text{ and } \theta(\sigma) = \gamma)$$

We say the types τ and σ are *orthogonal*, $\tau \# \sigma$, if they have no common type instance and extend this notion to constant instances: Let $c_\tau, d_\sigma \in \mathsf{CInst}$.

$$c_\tau \# d_\sigma \quad :\Longleftrightarrow \quad c \neq d \text{ or } \tau \# \sigma$$

Likewise, we say c_τ and d_σ are *orthogonal*.
We say a definitional theory D is orthogonal if for all distinct elements $u \equiv t, u' \equiv t' \in D$ either one is a type definition and the other is a constant definition, or both are of the same kind and u and u' are orthogonal.

We introduce a binary relation on constant instances and types, to represent the types and constant instances that a definition is depending on, i. e. the types and constant instances occurring within the right hand side of a definition.

Definition 2.3 [Dependency Relation] Let D be a definitional theory and $u, v \in (\mathsf{CInst} \cup \mathsf{Type})^*$ types or non-built-in constant instances. We define $u \rightsquigarrow_D v$, if one of the following two conditions hold.

- $\exists t \in \mathsf{Term}, u \equiv t \in D$ such that $v \in (\mathsf{CInst} \cup \mathsf{Type})^*$ and v occurs in the term t, or
- $\exists c_\sigma \in \mathsf{CInst}^*$ such that $v \in \mathsf{Type}^*$, v occurs in σ and $u = c_\sigma$

We say u *depends on* v.

The relation $\rightsquigarrow_D$ is subject to the definitional theory D which we omit when it is implied from the context. We give small examples for the relation: Trivially $c_\sigma \rightsquigarrow \sigma$ holds for all constant instances $c_\sigma \in \mathsf{CInst}^*$. Assume a theory that defines an unary type list and a constant $\mathsf{map}_{(\sigma\Rightarrow\tau)\Rightarrow\tau\mathsf{list}\Rightarrow\sigma\mathsf{list}}$ using a constant fold to define map. Both, $\mathsf{map} \rightsquigarrow \sigma\mathsf{list}$ and $\mathsf{map} \rightsquigarrow \mathsf{fold}$ hold.

Definition 2.4 [Type Substitutive Closure] Let R be a binary relation on $(\mathsf{CInst} \cup \mathsf{Type})^*$. We define the type substitutive closure $R^\downarrow$ which extends the rela-

tion R to instances of types. Let $t, s \in (\mathsf{CInst} \cup \mathsf{Type})^*$.

$$s\ R^{\downarrow}\ t \ :\Longleftrightarrow \quad \exists \rho \in \mathsf{TSubst}, s', t' \in (\mathsf{CInst} \cup \mathsf{Type})^* : \rho(s') = s, \rho(t') = t \text{ and } s'\ R\ t'$$

The previous definitions allow us to introduce well-formed definitional theories.

Definition 2.5 [Well-formed Definitional Theory] For a definitional theory D we say it is *well-formed* if it is orthogonal and there is no infinite sequence $(a_i)_{i\in\mathbb{N}} \subseteq (\mathsf{Type} \cup \mathsf{Const})^*$ such that each two subsequent elements a_i and a_{i+1} are in the substitutive closure of the dependency relation $a_i \rightsquigarrow_D^{\downarrow} a_{i+1}$ (for all $i \in \mathbb{N}$).

For a binary relation R, let R^+ denote its transitive closure.

2.3 Models of Definitional Theories

We now define the semantics of definitions in definitional theories and the semantics of terms to introduce valuations that allow us to evaluate terms to truths.

The meaning of $\equiv$ depends on the kind of the definition, as defined by [6]:

Definition 2.6 [Semantics of $\equiv$] Let $c_\tau \equiv t$ be a constant instance definition, then we define it to stand for the formula $c_\tau \doteq t$. Let $\tau \equiv t$ be a type definition. We define it to be the formula

$$\begin{aligned}&(\exists x_\sigma.\ t\ x_\sigma) \rightarrow \\ &\exists \mathrm{rep}_{\tau\Rightarrow\sigma}\ \exists \mathrm{abs}_{\sigma\Rightarrow\tau} \\ &(\forall x_\tau.\ t(\mathrm{rep}\ x)) \wedge (\forall x_\tau.\ \mathrm{abs}(\mathrm{rep}\ x_\sigma) \doteq x_\sigma) \wedge (\forall y_\sigma.\ t\ y \rightarrow \mathrm{rep}(\mathrm{abs}\ y_\sigma) \doteq y_\sigma).\end{aligned}$$

The interpretation of a constant instance definition is as expected the identification of the constant with the defining term. The interpretation of the type definition states the existence of an isomorphism of the type τ to a subset of σ that is defined by the predicate t, only if the type-defining predicate t yields a non-empty set.

We define fragments, a tuple of constants and the constants types. The interpretation evaluates terms with respect to the fragment and the model is defined by the True-evaluating terms.

Let $T \subseteq \mathsf{GType}^*$ and $C \subseteq \mathsf{GCInst}^*$ be sets of types and constants that are non-built-in such that the types of typed constants are contained in the closure of T. In this case we call the tuple (T, C) a *fragment*. For a fragment F we define an *F-interpretation* of the fragment as a pair of families $\mathcal{I} = (([\tau])_{\tau\in T}, ([c_\sigma])_{c_\sigma\in C})$, that fulfils the two conditions:

(i) For all types $\tau \in T$ the set $[\tau]$ is non-empty.

(ii) Furthermore to formulate a constraint on the constant instances, we extend the domains of types to the closure of built-in types of T such that $[\mathsf{bool}] = \{\mathsf{True}, \mathsf{False}\}$, $[\mathsf{ind}] = \mathbb{N}_0$ and $[\sigma \Rightarrow \tau] = [\sigma] \rightarrow [\tau]$ are satisfied. This gives meaning to extended family $([\tau])_{\tau\in\mathsf{Cl}(T)}$ that extends $[\cdot]$ to built-in types. Each interpretation of a constant instance $c_\tau \in C$ satisfies $[c_\tau] \in [\tau]$.

We expand the interpretation of a fragment $F = (T, C)$ to the ground built-in constant instances: logical implication $[\rightarrow_{\mathsf{bool}\Rightarrow\mathsf{bool}\Rightarrow\mathsf{bool}}]$, equality relation $[\doteq_{\tau\Rightarrow\tau\Rightarrow\mathsf{bool}}]$,

$[\mathsf{zero}_{\mathsf{ind}}]$ as least element in $\mathbb{N}_0$ and successor function $[\mathsf{succ}_{\mathsf{ind}\Rightarrow\mathsf{ind}}]$. We denote by GBI^F the set of ground built-in constants for a fragment. In addition we fix an arbitrary function choice to return one element of the argument set. We define the interpretation of the function some on functions $f : \tau \to \mathsf{bool}$ as

$$[\mathsf{some}_{(\tau\Rightarrow\mathsf{bool})\Rightarrow\tau}](f) = \begin{cases} \mathsf{choice}(A_f) & \text{if } A_f \neq \emptyset \text{ where } A_f := f^{-1}(\{\mathsf{True}\}) \\ \mathsf{choice}([\tau]) & \text{otherwise} \end{cases}.$$

Following this construction we consider the pair of families $\left(([\tau]_{\tau\in\mathsf{Cl}(T)}), ([c_\tau])_{c_\tau\in C\cup\mathsf{GBI}^F}\right)$.

With this setup we can evaluate terms within the fragment F with respect to its interpretation $\mathcal{I}$. We call a function ξ a *valuation for a model* $\mathcal{I}$ if $\xi : \mathsf{Var}_{\mathsf{Type}} \to \cup_{\sigma\in\mathsf{Type}}[\sigma]$ such that each variable of type σ gets assigned a value of type σ, i. e. $\xi(\mathsf{Var}_\sigma) \subseteq [\sigma]$. Let $\mathsf{Val}^{\mathcal{I}}$ denote the set of valuations for $\mathcal{I}$. The interpretation can now be expanded to general terms of fragments. Let $t \in \mathsf{Term}^F$ be a term in the fragment F. We recursively define the function $[t] : \mathsf{Val}^{\mathcal{I}} \to [\mathsf{tpOf}(t)]$ over the structure of terms.

$$\begin{aligned} [x_\sigma](\xi) &= \xi(x_\sigma) \\ [c_\sigma](\xi) &= [c_\sigma] \\ [t_1\ t_2](\xi) &= [t_1](\xi)([t_2](\xi)) \\ [\lambda x_\sigma.t](\xi) : [\sigma] \to [\mathsf{tpOf}(t)], a &\mapsto [t](\xi(x_\sigma \leftarrow a)) \end{aligned}$$

The latter $\xi(x_\sigma \leftarrow a)$ shall denote a function that takes the value a at x_σ and otherwise agrees with ξ. Kunčar and Popescu motivate the correctness of this recursive definition [6, Lemma 8.5].

For closed terms t of a fragment the valuation $[t]$ does not change for different variable assignments and hence the function $[t]$ is constant. We assume w. l. o. g. $[t] \in [\mathsf{tpOf}(t)]$, whenever t is a closed term.

On fragments $F_1 = (T_1, C_1)$ and $F_2 = (T_2, C_2)$ with $\mathcal{I}_1$ and $\mathcal{I}_2$ as their respective interpretations we define an ordering as $(F_1, \mathcal{I}_1) \leq (F_2, \mathcal{I}_2)$, iff $T_1 \subseteq T_2$, $C_1 \subseteq C_2$ and $[\cdot]^{\mathcal{I}_1} = [\cdot]^{\mathcal{I}_2}\big|_{T_1\cup C_1}$, which indeed is a partial ordering. This ordering is bounded and we call its upper bound *total fragment* $\mathsf{T} = (\mathsf{GType}^*, \mathsf{GCInst}^*)$.

We define a model $\mathcal{I}$ to be a model for a formula φ (in symbols $\mathcal{I} \models \varphi$), if the formula φ is valid with respect to the interpretation in $\mathcal{I}$: $[\varphi]^{\mathcal{I}} = \mathsf{True}$. If $\mathcal{I}$ is a T-interpretation and φ a polymorphic formula then we denote $\mathcal{I} \models \varphi$ if for all ground type substitutions $\theta \in \mathsf{GTSubst} : \mathcal{I} \models \theta(\varphi)$. Furthermore for sets of formulas E we define $\mathcal{I} \models E$ as: For all $\varphi \in E$ it holds $\mathcal{I} \models \varphi$.

One main result of Kunčar and Popescu is that each well-formed definitional theory has a model [6, Theorem 11] and consequently each such theory is consistent [6, Theorem 10]. We extend and generalize these results.

3 Results

The counterexample given in Section 1 motivates the need of a finer notion of conservative extension involving the dependency relation to avoid inconsistencies intro-

duced by dependencies.

3.1 Model-theoretic Conservativity

Fix a language L. A theory T' in the language L is a model-theoretic conservative extension of a theory T (in the same language L), if every model for T can be expanded to a model for T'. The expansion notion means that no formula of the theory T changes its validity.

Within the model construction [6, Theorem 11], undefined constant instances appearing in a term are assigned one value (with choice/some) out of the possible values, according to the constant's type. Undefined types, on the other hand, are interpreted as singletons. These choices in the semantics further propagate to types and constant instances that depend on or make use of types or constants that have no definition. These are the interpretations that we want to adjust properly within the extended model. Accordingly, we decompose a theory $D' = D \uplus \{u \equiv t\}$ (with at least one definition) into two disjoint sets.

Definition 3.1 [Depending Part of a Definitional Theory] Let D' be a well-formed definitional theory and Δ a subset $\Delta \subseteq D'$. We define the set

$$D'_{\rightsquigarrow^{\downarrow+}\Delta} := \{(u \equiv t) \in D' |\ \exists (v \equiv s) \in \Delta : u \rightsquigarrow_{D'}{}^{\downarrow+} v\} \cup \Delta$$

and call it the Δ-*depending part of* D'.

When we extend a theory D by a set of definitions Δ such that the resulting theory $D' := D \cup \Delta$ is a well-formed definitional theory, then the Δ-depending part of D' are all definitions of constant instances and types whose interpretations in a model of D' change compared to a model of D.

Furthermore in the previous situation the equality $D' \setminus D'_{\rightsquigarrow^{\downarrow+}\Delta} = D$ is equivalent to stating that no $u \equiv t \in D$ is depending on any definitional term introduced by Δ. It is possible to use types or constant instances that are not defined as we fixed the signature, although the introduction of cycles in the definitions in definitional theories is avoided by the dependency relation.

In case the set Δ is a singleton $\Delta = \{u \equiv t\}$ the Δ-depending part of D' can be formulated simpler:

$$D'_{\rightsquigarrow^{\downarrow+}\{u \equiv t\}} = \{(v \equiv s) \in D' | v \rightsquigarrow^{\downarrow+} u\} \uplus \{u \equiv t\}.$$

Remark that the relation $\rightsquigarrow^{\downarrow+}$ is not reflexive, so this is written as a union (of disjoint sets).

Lemma 3.2 *Let D' be a well-formed definitional theory and Δ be a subset. Let V be the set of types and constant instances $V := \{u | (u \equiv t) \in D'_{\rightsquigarrow^{\downarrow+}\Delta}\}$ and $\mathsf{G}V := \bigcup_{\theta \in \mathsf{GTSubst}} \theta(V)$. Then $F := (\mathsf{GType}^* \setminus \mathsf{G}V, \mathsf{CInst}^* \setminus \mathsf{G}V)$ is a fragment, and it is the largest fragment such that any term over this fragment is not transitively and type-substitutively depending on any of the definitions in Δ.*

Proof. We abbreviate the set of ground types and ground constant instances that

are type instances of the defined term that depend on Δ:

$$\mathsf{G}T := \mathsf{Type} \cap \mathsf{G}V, \quad \mathsf{G}C := \mathsf{CInst} \cap \mathsf{G}V.$$

Let $c_\sigma \in \mathsf{GCInst}^* \setminus \mathsf{G}C$ be a ground constant instance of type σ. For σ to be in $\mathsf{CI}(\mathsf{GType}^* \setminus \mathsf{G}T)$, there must be at least one type among the non-built-in types that occurs in σ, that is in $\mathsf{G}T$. But as $c_\sigma \notin \mathsf{G}C$ any of the types in $\text{types}^*(\sigma)$ can be transitively type-substitutively depending on u. The maximality is immediate. □

Theorem 3.3 (Model-theoretic conservativity) *Let $\mathcal{M}$ be a model for a well-formed definitional theory D, i. e. $\mathcal{M} \models D$. If the extension of D by a definition $D' := D \cup \{u \equiv t\}$ is a well-formed definitional theory, then there exists a model $\mathcal{M}'$ of the extended theory D', such that $\mathcal{M}$ and $\mathcal{M}'$ coincide on the interpretations of all terms that do not contain any of $\{\theta(v) | \theta \in \mathsf{GTSubst}, (v \equiv s) \in D'_{\rightsquigarrow^{\downarrow+}\{u \equiv t\}}\}$.*

Proof. We decompose the set of definitions D' into two parts, first $D'_u := D'_{\rightsquigarrow^{\downarrow+}\{u \equiv t\}}$ which is type-substitutively and transitively depending on u or the definition of u itself, and second its complement in D'. We recursively define the interpretation $[\cdot]^{\mathcal{M}'}$ where $\mathcal{M}'$ will denote the model for D', and therefore introduce abbreviations for those constants and types whose interpretations change in D', by the (transitive) dependency on u. We define ground constant instances and ground types that depend on the definition of u.

$$V_u := \bigcup \{x | x \equiv y \in D'_u\},$$
$$\mathsf{G}C_u := \mathsf{CInst} \cap \bigcup_{\theta \in \mathsf{GTSubst}} \theta(V_u), \qquad \mathsf{G}T_u := \mathsf{Type} \cap \bigcup_{\theta \in \mathsf{GTSubst}} \theta(V_u).$$

By Lemma 3.2, $F_u := (\mathsf{GType}^* \setminus \mathsf{G}T, \mathsf{GCInst}^* \setminus \mathsf{G}C)$ is a fragment. We can equip the fragment F_u with an interpretation that coincides with the one of $\mathcal{M}$, the top-level total fragment over F_u, so that for any constant instance or type t of the fragment F_u we will obtain $[t]^{\mathcal{M}} = [t]^{F_u}$.

We say a definition $(x \equiv r) \in D'$ matches v, if there is a type substitution $\theta \in \mathsf{TSubst}$ such that $v = \theta(x)$, meaning v is a type instance of x. As D' is orthogonal no definition in $D' \setminus D'_u$ matches any of D'_u.

We can follow the argumentation of [6, Theorem 11] to recursively define the interpretation for the types T_u and for the constant instances C_u based on the fragment F_u. Thereby we obtain a model $\mathcal{M}'$ of $(\mathsf{GType} \cup \mathsf{GCInst})^*$ for D' which by construction satisfies the notion of model extension:

We now define the interpretation of all elements in $(\mathsf{GType} \cup \mathsf{GCInst})^*$ (with respect to $\mathcal{M}'$). For $v \in (\mathsf{GType} \setminus \mathsf{G}T_u \cup \mathsf{GCInst} \setminus \mathsf{G}C_u)^*$ we define $[v]^{\mathcal{M}'} := [v]^{\mathcal{M}}$. This defines an interpretation on the fragment F_u as the types of all constant instances of the fragment are in the fragment and the non-emptyness of interpretation of types is inherited from $[\cdot]^{\mathcal{M}}$. This proofs the property of the model, once we have defined the interpretation for the elements in $(\mathsf{G}T_u \cup \mathsf{G}C_u)^*$ and showed that it is a model for D'. By recursion, assume the interpretation $[w]$ has been defined for all $w \in (\mathsf{GType} \cup \mathsf{GCInst})^*$ for which $v \rightsquigarrow^{\downarrow+} w$ holds.

(i) If for v there is no matching definition in D', then we define v accordingly.

$$[v] := \begin{cases} \{*\} & v \in \mathsf{GType}^* \\ \text{choice}([\sigma]) & v = c_\sigma, \text{ so } v \rightsquigarrow^{\downarrow} \sigma \end{cases}$$

(ii) Let $x \equiv r \in D'$ be a (and due to orthogonality thus 'the') definition which v matches, and let $\theta \in \mathsf{TSubst}$ be the corresponding type substitution. We define the interpretation of v based on the interpretation of the term $s := \theta(r)$, the defining term for v. As in [6], let $V_v := \{y | v \rightsquigarrow^{\downarrow +} y\}$ be the set of types and constant instances that v depends on and accordingly we set

$$\mathfrak{T}_v := V_v \cap \mathsf{Type} \quad \text{and} \quad \mathfrak{C}_v := V_v \cap \mathsf{CInst}$$

for the types and constant instances in V_v, respectively. The pair $F_v := (\mathfrak{T}_v, \mathfrak{C}_v)$ clearly is a fragment and s contains only constant instances and types of this fragment. Hence s is a term in the fragment F_v so we can regard its value $[s]^{F_v,\mathcal{I}_v}$ with respect to the interpretation $\mathcal{I}_v := (([q])_{q \in \mathfrak{T}_v}, ([q])_{q \in \mathfrak{C}_v})$.

Let $\sigma \in \mathsf{Type}$ such that $\text{tpOf}(s) = \sigma \Rightarrow \mathsf{bool}$, for both of the cases where v is a type $v \in \mathsf{GType}^*$. Note that $[s]$ is a function, more precisely $[s]^{F_v,\mathcal{I}_v} : \mathsf{Term}_\sigma \to \mathsf{bool}$.

$$[v] := \begin{cases} [s]^{F_v,\mathcal{I}_v} & v \in \mathsf{GCInst}^* \\ \{*\} & v \in \mathsf{GType}^*, \mathsf{True} \notin [s]^{F_v,\mathcal{I}_v}([\sigma]^{F_v,\mathcal{I}_v}) \\ [\sigma]^{F_v,\mathcal{I}_v} \cap ([s]^{F_v,\mathcal{I}_v})^{-1}(\{\mathsf{True}\}) & \text{otherwise} \end{cases}$$

If $v \in \mathsf{GType}^*$ is a type then its valuation is defined as the empty type, in case that for no feasible value in σ the term s can evaluate to True. In the remaining case $[v]$ is the set of all values that fulfil the property s.

Thus by recursion, let $\mathcal{M}'$ be the interpretation for the top element fragment $(\mathsf{GType}^*, \mathsf{GCInst}^*)$ with respect to $[\cdot]$. Analogous to [6] we can show $\mathcal{M}' \models D'$ by orthogonality of D', showing that each of the defined terms is interpreted as desired. □

Our approach extends an existing model $\mathcal{M}$ for a theory D to a model of any well-formed definitional theory extension D'. A model for the extended theory is constructed by recursion over the $\rightsquigarrow^{\downarrow}$ relation basing on interpretations from the model $\mathcal{M}$. The model construction by [6] was carried out by recursion over the $\rightsquigarrow^{\downarrow}$ relation basing on no interpretations.

As a consequence of the previous lemma we obtain the result [6, Theorem 11]:

Corollary 3.4 *Any well-formed definitional theory admits a model.*

Proof. Theorem 3.3 implies the following: Let $\mathcal{M}$ be a model for a well-formed definitional theory T, i. e. $\mathcal{M} \models T$. If the extension of T by a definition is a well-formed definitional theory, then there exists a model $\mathcal{M}'$ of the extended theory T', such that $\mathcal{M}$ and $\mathcal{M}'$ coincide on the interpretations of all terms that do not contain any of

$$\{\theta(v) | \theta \in \mathsf{GTSubst}, (v \equiv s) \in T'_{\rightsquigarrow^{\downarrow +} T' \setminus T}\}.$$

This generalisation is immediate by iteratively applying the theorem over the $\rightsquigarrow^{\downarrow+}$ relation.

Let D' be a well-formed definitional theory. The construction in Theorem 3.3 shows that the empty theory has a model which interprets undefined types as $\{*\}$ and each non-defined constant as some element of its type. Together with the initial remark we gain a model for D' from a model for the empty theory. □

The consistency of well-formed definitional theories [6, Theorem 6] is an immediate consequence.

Corollary 3.5 *Any well-formed definitional theory is consistent.*

4 Related Work

Extensions by definitions have a long history. In 1967, Shoenfield [12, §4.6] discussed two definitional mechanisms for extensions of theories in (untyped) first-order logic by predicate and function symbols. A new symbol is defined by an equivalence or equality whose right-hand side must not contain the new symbol. For function symbols, the definition must be accompanied by a proof that this equality describes a function in the mathematical sense. Both mechanisms are shown to be proof-theoretic conservative. Moreover, each model of the original theory has a *unique* expansion that is a model of the extended theory; this immediately implies model-theoretic conservativity.

In 1997, Wenzel [13] discussed the theoretical foundation for overloaded definitions and type classes in higher-order logic. He "consider[s] syntactic conservativity as a minimum requirement for well-behaved extension mechanisms within purely deductive logical frameworks." Additionally Wenzel introduces *realisability*, which formalises the intuition that constant definitions can be unfolded, and shows that overloaded constant definitions are both conservative and realisable. However, he assumes that all instances of an overloaded constant are defined at once, and he does not consider the interplay of overloading with type definitions (cf. the example in Section 1).

In 2006, Obua [11] noted that to avoid inconsistencies, the process of unfolding definitions must terminate. He shows that for overloaded definitions that recurse through types, termination is not semi-decidable in general. Obua considers both type and constant definitions and gives a proof sketch that overloading in Isabelle is conservative, but he misses that dependencies through types may introduce inconsistencies.

Most closely related is the already mentioned work by Kunčar and Popescu [6], who in 2015 introduced definitional theories for Isabelle and show that they preserve consistency, i. e. every well-formed definitional theory has a model. In [7], the same authors prove syntactic consistency (i. e. False is not derivable) of definitional theories by a proof-theoretic argument. They introduce a richer logic, HOL with comprehension types (HOLC), into which they encode formulas of HOL by unfolding type and constant definitions. This encoding preserves derivability. The consistency of definitional theories then follows from the consistency of HOLC.

In a recent technical report [8], Kunčar and Popescu show a much stronger result:

using a different unfolding approach that relativises formulas involving defined types to a predicate on the host type (and thus stays within the language of HOL), they establish proof-theoretic conservativity of definitional theories over minimal HOL, i. e. relative to an empty theory that contains no axioms. In contrast, the present paper proves model-theoretic conservativity relative to arbitrary (well-formed) definitional theories.

Other theorem provers for higher-order logic, e. g. HOL4 [10, §2.5.2], implement a more general mechanism for *constant specification*. This mechanism, which in its current form was suggested by Arthan [2], allows implicit definitions. It takes as input a theorem of the form $v_1 \doteq t_1, \ldots, v_n \doteq t_n \vdash P$ (where the v_i are variables) and introduces new constants $c_1, \ldots, c_n$ with $P[c_1/v_1, \ldots, c_n/v_n]$ as their defining axiom. Conventional constant definitions $c \equiv t$ are recovered as a special case when P is of the form $v \doteq t$. Constant specification is proof- and model-theoretic conservative [3], and has been formalised and verified using HOL4 by Kumar et al. [4]. However, in contrast to Isabelle, which supports *ad hoc* overloading natively in its logic, other theorem provers for higher-order logic offer support for overloading only as syntactic sugar, through extensions of parsing and pretty-printing.

5 Conclusion

We defined a notion of model expansion that is suitable for definitional theories, and we showed that extensions of definitional theories are model-theoretic conservative with respect to this notion. This strengthens and generalises an earlier consistency result for definitional theories [6]. We have thereby established an important property of the definitional mechanisms that are implemented in the theorem prover Isabelle.

Model-theoretic conservativity has a proof-theoretic (syntactic) counterpart. Roughly, an extension is *proof-theoretic conservative* if it entails no new theorems in the original language. In other words, every formula of the original language that is a theorem in the extension is already provable in the original theory. Adapting this notion to definitional theories, we conjecture that if D' is an extension of D such that $D' \vdash \varphi$, where φ is a formula that does not contain any constant instance or type that depends on definitions in $D' \setminus D$, then $D \vdash \varphi$.

For logics that have a sound and complete deductive system, model-theoretic conservativity implies proof-theoretic conservativity: suppose $D' \vdash \varphi$. By completeness it suffices to show that φ holds in all models of D. Let $\mathcal{M}$ be a model of D. By model-theoretic conservativity, $\mathcal{M}$ can be expanded to a model $\mathcal{M}'$ of D' that agrees with $\mathcal{M}$ on the interpretation of φ. Since $D' \vdash \varphi$, soundness implies that $\mathcal{M}'$ is a model of φ. Hence $\mathcal{M}$ is a model of φ.

Unfortunately, this argument does not immediately apply to higher-order logic, which is not complete with respect to its standard semantics [10, §2.4.5]. However, higher-order logic *is* complete with respect to non-standard (Henkin) semantics [1, §54]. By adapting the completeness proof to the variant of higher-order logic implemented in Isabelle and to the novel semantics of polymorphic types suggested in [6], it may be possible to derive proof-theoretic conservativity for extensions of definitional theories from their model-theoretic conservativity. We leave the details

to future work.

References

[1] Andrews, P. B., "An Introduction to Mathematical Logic and Type Theory: To Truth through Proof," Number 27 in Applied logic series, Kluwer Academic Publishers, Dordrecht ; Boston, 2002, 2nd ed edition.

[2] Arthan, R., *HOL Constant Definition Done Right*, in: *Interactive Theorem Proving* (2014), pp. 531–536.
URL http://dx.doi.org/10.1007/978-3-319-08970-6_34

[3] Arthan, R., *On Definitions of Constants and Types in HOL*, Journal of Automated Reasoning **56** (2016), pp. 205–219.
URL http://dx.doi.org/10.1007/s10817-016-9366-4

[4] Kumar, R., R. Arthan, M. O. Myreen and S. Owens, *HOL with Definitions: Semantics, Soundness, and a Verified Implementation*, in: *Interactive Theorem Proving* (2014), pp. 308–324.
URL http://dx.doi.org/10.1007/978-3-319-08970-6_20

[5] Kuncar, O., *Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants*, in: X. Leroy and A. Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015* (2015), pp. 85–94.
URL http://doi.acm.org/10.1145/2676724.2693175

[6] Kunčar, O. and A. Popescu, *A Consistent Foundation for Isabelle/HOL*, in: C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, number 9236 in Lecture Notes in Computer Science, Springer International Publishing, 2015 pp. 234–252.
URL http://dx.doi.org/10.1007/978-3-319-22102-1_16

[7] Kunčar, O. and A. Popescu, *Comprehending Isabelle/HOL's Consistency*, in: H. Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, Lecture Notes in Computer Science **10201** (2017), pp. 724–749.

[8] Kunčar, O. and A. Popescu, *Safety and Conservativity of Definitions in HOL and Isabelle/HOL*, Technical report (2017).
URL http://andreipopescu.uk/pdf/conserv_HOL_IsabelleHOL.pdf

[9] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.
URL https://doi.org/10.1007/3-540-45949-9

[10] Norrish, M. and K. Slind, *The HOL System LOGIC* (2014).
URL http://downloads.sourceforge.net/project/hol/hol/kananaskis-10/kananaskis-10-logic.pdf

[11] Obua, S., *Checking Conservativity of Overloaded Definitions in Higher-Order Logic*, in: *Term Rewriting and Applications* (2006), pp. 212–226.
URL http://dx.doi.org/10.1007/11805618_16

[12] Shoenfield, J. R., "Mathematical Logic," A.K. Peters, Natick, Mass, 1967.

[13] Wenzel, M., *Type classes and overloading in higher-order logic*, in: E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, number 1275 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1997 pp. 307–322.
URL http://dx.doi.org/10.1007/BFb0028402

The MMT Perspective on Conservativity

Florian Rabe[1,2]

Computer Science
Jacobs University
Bremen, Germany

Abstract

Conservative extensions are one of the most important concepts in formal logic, capturing the intuition when an extension does not substantially change the extended theory. Two conceptually different definitions have emerged in proof and model theory, respectively. Unfortunately these are conceptually very different and not equivalent.
We develop both notions in the MMT framework, which allows for an elegant uniform treatment of proof and model theory. In MMT, the difference between the two definitions becomes less fundamental. Moreover, we see that the existence of two different notions of conservativity is neither a coincidence nor a defect: it becomes a special case of the well-known difference between admissible and derivable reasoning principles. Moreover, we are able to relate conservativity to the completeness of a logic, thus adding another connection between proof and model theory.

Keywords: conservative extension, logical framework, admissible, derivable, completeness

1 Introduction

Motivation

Conservativity is at the heart of mathematics and logics. Most abstractly, it means to extend a formal system in a way that effectively does not change it. Because this is what happens when adding definitions and theorems, it has received substantial attention in the area of formal logic.

Not surprisingly, different concrete definitions of the abstract intuition have been studied. Two notions have been used most widely, one based on proof theory and one based on model theory.[3] Due to the different philosophical backgrounds, it is not surprising that these notions are conceptually very different. This is not unusual: for example, the notion of *theorem* also has two very different definitions in proof and model theory. However, contrary to theorems, the two definitions of conservativity are not equivalent even in the presence of a sound and complete calculus. (The

[1] This work was supported by DFG under grant RA-18723-1.

[2] f.rabe@jacobs-university.de

[3] The author has traced the model-theoretical definition back to [7] but could not locate the first use of the (older) proof theoretical definition.

model theoretical one implies the proof theoretical one.) Consequently, this may lead to confusion and possibly even contention among researchers. [4]

Contribution

This paper contributes to the discussion by developing both definitions in the context of the author's MMT framework [13,12]. MMT uses logical frameworks [10] as formal meta-logics in which to represent logics. In doing so, it integrates frameworks with a proof theoretical background such as LF [6] with model theoretical frameworks such as institutions [4]. This lets MMT elegantly represent both proof and model theoretical concepts uniformly [8,1,12]. Moreover, MMT is systematically designed to be logic-independent. Thus we can use it to give rigorous definitions of logical concepts (such as proofs and models or logic translations) in full generality. These properties make MMT well-suited to study the two notions of conservativity.

Our most important result is that we are able to relate the two notions of conservativity to the concepts of admissible and derivable rules. At the MMT-level, proof and model theoretical conservativity end up being special cases of admissibility and derivability, respectively. This provides an elegant understanding of both the similarities and the differences of the two competing definitions.

MMT also lets us elegantly capture the subtlety that the model theoretic definition actually constitutes a family of different definitions—one each for every model theory that is used. We naturally encounter a certain syntactic model theory that induces the strictest reasonable notion of conservativity, with every refinement of the model theory leading to a more lenient notion. Taking these refinements to the extreme, we find the proof theoretic definition as the most lenient reasonable notion.

Furthermore, we are able to cast completeness of a logic as a special case of admissibility as well, thus creating an appealing connection between conservativity and completeness.

Overview

Sect. 2 recalls the existing definitions of proof and model theoretical conservativity, and we summarize the necessary preliminaries about MMT in Sect. 3. Sect. 4 develops our definitions and establishes their properties.

2 Existing Definitions of Conservativity

Conservativity can be defined for an arbitrary logic. To state the definitions in full generality, we can use a framework like institutions [4]. However, the precise abstract definition is not essential for our purposes. Therefore, we only assume a very lightweight definition to make the paper more accessible. A logic consists of

- a category of theories,
- a set of sentences $\mathbf{Sen}(\Sigma)$ for each theory Σ and a sentence translation $v(-) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ for every theory morphism $v : \Sigma \to \Sigma'$,

[4] In fact, this paper was motivated by the author's impression that this seems to be the case.

- a provability predicate giving the provable subset of $\mathbf{Sen}(\Sigma)$,
- a class of models $\mathbf{Mod}(\Sigma)$ for each theory Σ and a model reduction function $\mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ for every theory morphism $v : \Sigma \to \Sigma'$,
- a satisfaction relation between models in $\mathbf{Mod}(\Sigma)$ and sentences in $\mathbf{Sen}(\Sigma)$,

with some coherence conditions between them. Detailed definitions based on institutions are given in, e.g., in [3,11].

Relative to such a logic, soundness and completeness can be defined in the usual way by relating the provable sentences to those that are satisfied by all models. Moreover, we can state the usual definitions of conservativity:

Definition 2.1 [Proof-Theoretically Conservative] A morphism $v : \Sigma \to \Sigma'$ is conservative if every Σ-sentence F is provable iff the Σ'-sentence $v(F)$ is provable.

For the special case of an extension $v : \Sigma \hookrightarrow \Sigma'$, this says that the larger theory does not prove any Σ-sentences that were not already provable in Σ.

Definition 2.2 [Model-Theoretically Conservative] A morphism $v : \Sigma \to \Sigma'$ is conservative if for every Σ-model m there is a Σ'-model m' that reduces to m via v.

For the special case of an extension, this says that every model of the smaller theory can be extended to a model of the larger theory.

As indicated before, these definitions are not equivalent:

Theorem 2.3 *If v is conservative in the model-theoretical sense and the logic is sound and complete, then v is conservative in the proof-theoretical sense.*

The converse is not true in general.

3 Logics in the MMT-Framework

This section is a summary of the basic definitions of logics in MMT as given in [12]. [12] uses an arbitrary logical framework that is itself defined in MMT. All our results in this paper generalize easily to this general case. However, to make this paper more accessible (and shorter), we use a single, fixed logical framework. Concretely, we use LF.

3.1 Logical Framework

The Logical Framework LF

LF [6] is a dependent type theory using the following concepts and notations:

- a universe `type` of types
- dependent function types $\Pi_{x:A}\, B$, which are written $A \to B$ if x does not occur free in B
- dependent function abstraction $\lambda_{x:A}\, t$,
- function application $f\, a$,
- β-reduction and η-conversion.

We omit the well-known typing rules.

Theories

LF-theories Σ are sets of declarations, which are one of the following

- type declarations $c : \Pi_{x_1:A_1} \dots \Pi_{x_n:A_n}$ `type`
- term declarations $c : A$ for some type A

Relative to a theory Σ, we have the set of all types A and the set of all terms t. These are subject to the typing judgment $t : A$ and the equality judgments $t \equiv t'$ and $A \equiv A'$. (The latter equality judgment is needed because terms may appear in types.) Typing and equality of terms and types are decidable. We say that a type A is **inhabited** if there is a term $t : A$.

Example 3.1 [Syntax of First-Order Logic] We define first-order logic FOL as the following LF theory *FOL*:

$$
\begin{array}{lcl}
o & : & \texttt{type} \\
i & : & \texttt{type} \\
\mathit{thm} & : & o \to \texttt{type} \\
\neg & : & o \to o \\
\wedge & : & o \to o \to o \\
\Rightarrow & : & o \to o \to o
\end{array}
\qquad
\begin{array}{lcl}
\doteq & : & i \to i \to o \\
\forall & : & (i \to o) \to o \\
\exists & : & (i \to o) \to o \\
\mathit{mp} & : & \Pi_{F:o}\, \Pi_{G:o}\, \mathit{thm}\,(F \Rightarrow G) \to \mathit{thm}\, F \to \mathit{thm}\, G \\
\vdots & &
\end{array}
$$

FOL-terms and sentences are represented as LF-terms over the theory *FOL* of type i and o, respectively. We use currying to represent functions with multiple arguments as unary functions. We will use the usual infix notations where applicable, e.g. the term $(\wedge\, F)\, G$ represents the sentence $F \wedge G$. Binders are represented using higher-order abstract syntax: the term $\forall(\lambda x : i.F(x))$ represents the sentence $\forall x.F(x)$. In future examples, we will use the usual notations instead of the ones technically prescribed by our encoding in LF.

The type constructor *thm* serves as the truth judgment. Proof-theoretically, we use a Curry-Howard representation of proofs as terms, i.e., the terms $p : \mathit{thm}\, F$ are the proofs of F; model-theoretically, we will below treat F as true if the type $\mathit{thm}\, F$ is inhabited.

We only give a single proof rule as an example: The modus ponens rule *mp* takes two formulas F and G, a proof of $F \Rightarrow G$ and a proof of F and returns a proof of G. All natural deduction rules of first-order logic can be written as LF declarations in this style.

Theory Morphisms

LF-theory morphisms $\sigma : \Sigma \to \Sigma'$ are sets of assignments, one for each declaration in Σ:

- for every Σ-type declaration $c : \Pi_{x_1:A_1} \dots \Pi_{x_n:A_n}$ `type`, a type assignment

$$a \mapsto \lambda_{x_1:\sigma(A_1)} \dots \lambda_{x_n:\sigma(A_n)}\, B$$

  for some Σ'-type B with free variables $x_1, \dots, x_n$.

- for every Σ-term declaration $c : A$, a term assignment $c \mapsto t$ for a Σ'-term $t : \sigma(A)$

where $\sigma(-)$ is the homomorphic extension of σ defined below.

Every theory morphism σ extends to a homomorphic translation $\sigma(-)$, which maps Σ-terms and types to Σ'-terms and types. $\sigma(-)$ preserves typing and equality, e.g., if $t : A$ holds over Σ, then $\sigma(t) : \sigma(A)$ holds over Σ'. If particular, if A is inhabited over Σ, then $\sigma(A)$ is inhabited over Σ'.

Example 3.2 [Semantics of First-Order Logic] We sketch a morphism *FOLZF* from *FOL* a theory *ZF* for axiomatic set theory. The intuition behind *FOLZF* is that it is the interpretation function that maps *FOL*-terms and *FOL*-formulas to their denotations.

ZF is an extension of *FOL* that declares the binary predicate $\in$: $i \to i \to o$ and adds the axioms of set theory. Besides the usual set theoretical operations, *ZF* defines in particular the 2-element set $bool : i$ of Booleans containing the elements $0 : i$ and $1 : i$. Moreover, we add a type constructor $Elem : i \to \texttt{type}$ such that terms of type $Elem\, A$ represent the elements of the set $A : i$. The complete definition of *ZF* can be found in [9].

We extend *ZF* with a theory Δ that axiomatizes a basic FOL-model: Δ contains the declarations $univ : i$ and $nonempty : thm\,(\exists x.x \in univ)$ which describe a non-empty set.

Then we define the semantics as the morphism $FOLZF : FOL \to ZF, \Delta$, which maps in particular

- $FOLZF(i) = Elem\, univ$, i.e., *univ* is an arbitrary non-empty set representing the universe of the model and terms are interpreted as elements of *univ*,
- $FOLZF(o) = Elem\, bool$, i.e., every formula is interpreted as a boolean truth value,
- $FOLZF(thm) = \lambda x : Elem\, bool.thm(x \doteq 1)$, i.e., $FOLZF(thm\, F)$ is inhabited iff $FOLZF(F)$ is provably equal to the truth value 1.

All connectives can now be mapped in the usual way. For example, $FOLZF(\wedge)$ is the binary conjunction of Booleans. All proof rules can be mapped as well—each assignment of a proof rule represents a case in the soundness proof.

Ultimately, the typing preservation of LF-morphism guarantees soundness: every *FOL*-proof $P : thm\, F$ gives rise to a *ZF*-proof $FOLZF(P) : thm\,(FOLZF(F) \doteq 1)$, i.e., the usual soundness theorem.

Pushouts

LF theories and theory morphisms form a category. Moreover, this category has pushouts along inclusions, which we write as

$$\begin{array}{ccc} Syn, \Sigma & \xrightarrow{sem^{\Sigma}} & Sem, sem(\Sigma) \\ \uparrow & & \uparrow \\ Syn & \xrightarrow{sem} & Sem \end{array}$$

$sem(\Sigma)$ can be seen as the homomorphic translation of Σ: It contains the decla-

ration $c : sem^{\Sigma}(A)$ for every declaration $c : A$ in Σ. Here sem^{Σ} maps Syn-constants like sem, and it maps each $c : A$ in Σ to the corresponding c in $sem(\Sigma)$.

For a morphism $v : Syn, \Sigma \to Syn, \Sigma'$ that is the identity on Syn, we write $sem(v) : Sem, sem(\Sigma) \to Sem, sem(\Sigma')$ for the unique factorization through the pushout. Thus, $sem(-)$ is a functor from extensions of Syn to extensions of Sem.

Technically, there are some subtleties here because the above construction of the pushout is not always well-defined—there is a problem if the same name is declared in both Sem and Σ. This is discussed in [12] and not essential for the results in this paper.

3.2 Logics

In MMT, we can define logics easily by abstracting from the intuitions presented in Ex. 3.1 and 3.2:

Definition 3.3 [Logical Theories] A **logical theory** Syn is an LF-theory with distinguished declarations $o : \texttt{type}$ and $thm : o \to \texttt{type}$.

Consider two logical theories Syn (with o and thm) and Syn' (with o' and thm'). A **logical morphism** is an LF-morphism $l : Syn \to Syn'$ such that $l(thm\, x) = thm'(k\, x)$ for some expression $k : l(o) \to o'$. (k is uniquely determined if it exists.)

Definition 3.4 [Logic] A logic is a 4-tuple forming a logical morphism $sem : Syn \to Sem, \Delta$.

Example 3.5 [First-Order Logic] FOL from Ex. 3.1 is a logical theory where o and thm are the distinguished declarations.

$FOLZF : FOL \to ZF, \Delta$ from Ex. 3.2 is a logical morphism with $k\, x = x \doteq 1$.

Every logic in the sense of Def. 3.4 induces a logic in the sense of Sect. 2. Here the intuitions behind Syn, Sem, Δ, and sem are as follows:

- The logical theory Syn represents the syntax and proof theory: sentences are the terms of type o, and proofs are the terms of type $thm\, F$.
- The logical theory Sem represents the semantic foundation, e.g., an ambient set theory like ZF.
- Δ extends Sem with the axiomatization of a basic model. For FOL, Δ is very simple—the theory of a non-empty set. But Δ can be arbitrarily complex, e.g., the theory of a category for categorical models or the theory of a Kripke frame for Kripke models.
- The logical morphism sem describes the interpretation of the syntax and proofs in an arbitrary model.

In the remainder of this section, we make these intuitions precise for a fixed logic $sem : Syn \to Sem, \Delta$.

Definition 3.6 [Abstract Negation] For a type A in a logical theory, we abbreviate $\overline{A} := A \to \Pi_{F:o} F$.

A logical theory is *classical* if it has a term of type $classical : \Pi_{F:o} \overline{\overline{thm\, F}} \to thm\, F$.

The type $\overline{A}$ represents a negation of A in the sense that if A is inhabited, the theory is inconsistent because every formula is provable. Thus, classical logics are the one that have double-negation elimination.

Definition 3.7 [Syntax and Proofs] A *Syn*-**theory** is an extension $Syn \hookrightarrow Syn, \Sigma$ of Syn.

A *Syn*-**theory morphism** is a morphism $\sigma : Syn, \Sigma \to Syn, \Sigma'$ satisfying $\sigma|_{Syn} = id_{Syn}$.

Consider a *Syn*-theory Σ:

- A Σ-**sentence** is a Σ-term $F : o$.
- A Σ-**proof** of F is a Σ-term $p : thm\, F$.
- A Σ-**disproof** of F is a Σ-term of type $\overline{thm\, F}$.
- F is **(dis)provable** if there is a (dis)proof of F.

A *Syn*-theory morphism $\sigma : \Sigma \to \Sigma'$ translates sentences and proofs over Σ to Σ' by applying the homomorphic extension $\sigma(-)$.

Definition 3.8 [Semantics and Models] Consider a *Syn*-theory Σ, and a Σ-sentence F. Then:

(i) A Σ-**model** via *sem* is a *Sem*-theory morphism $m : Sem, \Delta, sem(\Sigma) \to Sem, M$ such that $m|_{Sem} = id_{Sem}$.

(ii) F is **true** resp. **false** in such a model m if the type $m(sem^{\Sigma}(thm\, F))$ resp. $\overline{m(sem^{\Sigma}(thm\, F))}$ is inhabited in the theory in Sem, M.

A *Syn*-theory morphism $\sigma : \Sigma \to \Sigma'$ reduces a model m of Σ' to the model $m \circ sem(\sigma)$ of Σ.

$$\begin{array}{ccccc} Syn, \Sigma & \xrightarrow{sem^{\Sigma}} & Sem, \Delta, sem(\Sigma) & \xrightarrow{m} & Sem, M \\ \uparrow & & \uparrow & & \uparrow \\ Syn & \xrightarrow{sem} & Sem, \Delta & \hookleftarrow & Sem \end{array}$$

Note that a model m must be the identity on *Sem* and interpret the declarations in Δ and in $sem(\Sigma)$ as values in *Sem*. That is the reason why we need two different theories for *Sem* and Δ when defining a logic.

Definition 3.9 [Consistency] An *Syn*-theory Σ is **consistent** if there is no Σ-sentence that is both provable and disprovable.

A logical morphism $sem : Syn \to Sem$ **preserves consistency** if $sem(\Sigma)$ is consistent whenever Σ has at least one sentence and is consistent.

One of the main theorems of [12] is the following:

Theorem 3.10 (Soundness/Completeness) *Every logic $sem : Syn \to Sem, \Delta$ is sound in the sense that provable Σ-sentences are true in all Σ-models via sem.*

Moreover, if Syn is classical and sem preserves consistency, the logic is also complete in the dual sense.

4 Conservative Morphisms

We will now develop definitions of conservativity for arbitrary logics defined in the MMT framework. A key insight is to define general notions of derivability and admissibility first.

4.1 Derivable and Admissible Rules

Derivable and *admissible* are well-known concepts in the study of inference systems. Both mean that a rule can be added to an inference system without changing its essence. In MMT, we can give a very general precise definition:

Definition 4.1 [Admissible, Derivable] Consider a theory Syn.

For a set J of Syn-types, a type R is called J-**admissible** if every type in J is inhabited over Syn iff it is inhabited over $Syn, r : R$.

A type R is called **derivable** if it is admissible for the set of all Syn-types.

For J-admissibility, the left-to-right implication always holds: If there is a Syn-term $t : A$, then t is also a $Syn, r : R$-term. Only the right-to-left implication is special.

Theorem 4.2 *A Syn-type R is derivable iff there is a Syn-term $P : R$.*

In particular, we have a morphism $id_{Syn}, r \mapsto P$ from $Syn, r : R$ to Syn.

Proof. Assume there is a term $P : R$. We can always replace r with P in any term over $Syn, r : R$, thus proving J-admissibility.

Assume admissibility for every type T. We obtain P by instantiating with $T = R$. □

Thus, if a rule (represented by the type R) is derivable, we have a derivation for it (represented by the term P), and adding it to the inference system (the declaration $r : R$) just adds an abbreviation (the name r) for a derivation that already existed. For example, let $Syn = FOL$ and $R = \Pi_{F,G,H}\, thm\,(F \Rightarrow G \Rightarrow H) \to thm\,F \to thm\,G \to thm\,H$. R is a derivable rule: We can derive it applying modus ponens twice, and the term $P : R$ is given by $\lambda_{F,G,H}\, \lambda_{p,q,r}\, mp\,G\,H\,(mp\,F\,(G \Rightarrow H)\,p\,q)\,r$.

A J-admissible rule may create substantively new derivations as long as it does not create new J-terms. The most important special case arises when Syn is a logical theory and J is the set of all types of the form $thm\,F$: then J-admissibility of a rule R means that no new formula F becomes provable if R is added to Syn.

Derivable rules are admissible but not vice versa. Admissible-but-not-derivable rules are of great importance in the meta-theory of logics. Examples are the deduction theorem in Hilbert calculi and the cut rule in sequent calculi.

Proofs of derivability are usually straightforward: we just have to exhibit the derivation P. Proofs of admissibility, however, are usually very involved, often requiring an induction over all Syn-terms. Moreover, derivability is the more robust notion: Extending Syn in any way can never break the derivability of R (because $P : R$ remains a well-formed term), but it can break admissibility (because it adds a new case that has to be considered in the inductive proof).

In the sequel, we often need to talk about admissibility where the set J a certain form. Therefore, we define:

Definition 4.3 For a logical theory and a type constructor $thm : o \to \texttt{type}$, we say thm-**admissible** if we mean admissible for the set of all types of the form $thm\, F$.

4.2 Derivable and Admissible Morphisms

The previous section formalized the existing concepts of derivable and admissible in MMT. Now we use the MMT formalism to generalize the definitions to arbitrary theory extensions.

Definition 4.4 [Admissible/Derivable Extensions] Consider a theory extension $Syn \hookrightarrow Syn'$.

It is called J-**admissible** if every type in J is inhabited over Syn iff it is inhabited over Syn'.

It is called **derivable** if it is admissible for the set of all Syn-types.

Theorem 4.5 *An extension $Syn \hookrightarrow Syn'$ is derivable iff it has a retraction, i.e., if there is a morphism $P : Syn' \to Syn$ such that $P|_{Syn} = id_{Syn}$.*

Proof. Let $Syn' = Syn, \Sigma$.

Assume there is a morphism P. We can always replace any Σ-symbol c with $P(c)$ in any Syn'-term, thus proving J-admissibility.

Assume admissibility for every type T. We build the morphism P by induction on Σ. Assume we have $P : Syn, \Sigma_0 \to Syn$. For the next Σ-declaration $c : A$, we instantiate admissibility with $T = P(A)$ to obtain a Syn-term p. Then $P, c \mapsto p$ is a morphism $Syn, \Sigma_0, c : A \to Syn$. □

Theory extensions are the most important special case when studying conservativity. But it is easy to generalize the concepts to arbitrary morphisms. This has practical importance because it allows considering, e.g., renamings or isomorphisms in addition to extensions:

Definition 4.6 [Admissible/Derivable Morphisms] Consider a theory morphism $v : Syn \to Syn'$.

v is called J-**admissible** if every type A in J is inhabited over Syn iff $v(A)$ is inhabited over Syn'.

v is called **derivable** if it has a retraction, i.e., if there is a morphism $P : Syn' \to Syn$ such that $P \circ v = id_{Syn}$.

Theorem 4.7 *If $v : Syn \to Syn'$ is derivable, then it is admissible for all Syn-types.*

Proof. Applying the retraction of v yields a Syn-term for every Syn'-term. □

For arbitrary morphisms, admissibility for all Syn-types is not the same anymore as having a retraction. The problem is that quantifying over all Syn-types A is not strong enough to quantify over all Syn'-types because not every Syn'-type is of the form $v(A)$. Therefore, we have to choose one of the two notions as the appropriate

generalization of derivability. In Def. 4.6, we choose the retraction property because it better captures the intuition that all Syn'-declarations can be derived in Syn.

Finally we establish some basic closure properties.

Theorem 4.8 (Closure Properties) *Derivable and admissible morphisms have the following closure properties:*

- *Identity*
  - *The identity morphism is derivable.*
  - *The identity morphism is J-admissible for any J.*
- *Composition*
  - *If v and w are derivable, then so is $w \circ v$.*
  - *If v is J-admissible and w is $v(J)$-admissible, then $w \circ v$ is J-admissible.*
- *Decomposition*
  - *If $w \circ v$ is derivable, then so is v.*
  - *If $w \circ v$ is J-admissible, then so is v.*

Additionally, derivable morphisms have the following closure properties:

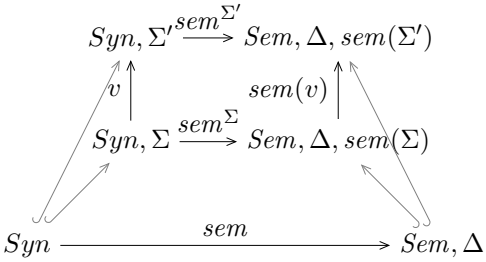
- *Union: The union $\Sigma, \Sigma_0, \Sigma_1$ of derivable extensions Σ, Σ_0 and Σ, Σ_1 is derivable.*
- *Pushout: The pushout of a derivable morphism is derivable.*

Proof. All proofs are straightforward. □

The closure under pushouts formally captures the robustness of derivability: It is preserved under translations of the domain theory. Admissibility of morphisms, however, is brittle: It can be broken by relatively minor changes to the domain theory.

4.3 Conservative Morphisms

In this section, we fix a logic $sem : Syn \to Sem, \Delta$. We want to study the conservativity of a Syn morphism $v : \Sigma \to \Sigma'$. The following diagram describes our basic situation.



Definition 4.9 [Proof-Theoretically Conservative] A Syn-morphism $v : \Sigma \to \Sigma'$ is called **proof-conservative** via sem if $sem(v)$ is $sem^{\Sigma}(thm)$-admissible.

v is simply called **proof-conservative** if it is proof-conservative via id_{Syn}.

Intuitively, v is proof-conservative via sem if semantic proofs (i.e. proofs carried out in the ambient foundation Sem that talk about truth in an arbitrary model) exhibit the typical conservativity property when moving along v.

For every logical theory Syn, we have the trivial semantics id_{Syn}. The special case of being proof-conservative via id_{Syn} immediately yields the usual proof-theoretical notion from Def. 2.1:

Theorem 4.10 *$v : \Sigma \to \Sigma'$ is proof-conservative iff the following holds: every Σ-sentence F is Σ-provable iff $v(F)$ is Σ'-provable.*

In particular, an extension $\Sigma \hookrightarrow \Sigma'$ is proof-conservative if every Σ-sentence is Σ-provable iff it is Σ'-provable.

Proof. This follows easily because if $sem = id_{Syn}$, then $sem(-)$ is the identity, and in particular $sem(v) = v$. □

Definition 4.11 [Model-Theoretically Conservative] A Syn-morphism $v : \Sigma \to \Sigma'$ is **model-conservative** via sem if $sem(v)$ is derivable, i.e., if there is a retraction r as in the commutative diagram below.

$$\begin{array}{ccccc} Syn, \Sigma' & \xrightarrow{sem^{\Sigma'}} & Sem, \Delta, sem(\Sigma') & \overset{r}{>} & Sem, \Delta, sem(\Sigma) \\ {\scriptstyle v}\uparrow & & {\scriptstyle sem(v)}\uparrow & \nearrow_{id} & \\ Syn, \Sigma & \xrightarrow{sem^{\Sigma}} & Sem, \Delta, sem(\Sigma) & & \end{array}$$

v is simply called **model-conservative** if it is model-conservative via id_{Syn}, i.e., if v has a retraction.

The question whether our notion of model-conservativity is equivalent to the usual one from Def. 2.2, is tricky. We establish one direction first:

Theorem 4.12 *If $v : \Sigma \to \Sigma'$ is model-conservative via sem, then every Σ-model m via sem can be expanded to a Σ'-model m' via sem that reduces to m.*

Proof. We put $m' = m \circ r$. The reduct of m' via v is $m' \circ sem(v)$, which is equal to m because $r \circ sem(v) = id$.

$$\begin{array}{ccccc} Syn, \Sigma' & \xrightarrow{sem^{\Sigma'}} & Sem, \Delta, sem(\Sigma') & \overset{r}{>} & Sem, \Delta, sem(\Sigma) \\ {\scriptstyle v}\uparrow & & {\scriptstyle sem(v)}\uparrow & \nearrow_{id} & \downarrow{\scriptstyle m} \\ Syn, \Sigma & \xrightarrow{sem^{\Sigma}} & Sem, \Delta, sem(\Sigma) & & Sem, M \end{array}$$

□

The other direction of the equivalence depends on subtle properties of the theory Sem. For example, consider the case where sem and Sem formalize the usual set-theoretical semantics in some ambient set theory. If we formalize Def. 2.2, we obtain something like the Sem-sentence U given by

$$\forall m \in \mathbf{Mod}(\Sigma).\, \exists m' \in \mathbf{Mod}(\Sigma').\, reduct(v, m') = m$$

Our definition, on the other hand, is equivalent to exhibiting a function f such that

$$\forall m \in \mathbf{Mod}(\Sigma).\, f(m) \in \mathbf{Mod}(\Sigma') \wedge reduct(v, f(m)) = m$$

This is subtly stronger because it requires actually giving f, which in particular requires f to be definable as a *Sem*-expression.

For example, consider the most direct formalization of set theory using a *FOL*-theory *ZF* with a single predicate symbol $\in$. This *FOL*-theory has no function symbols at all and therefore cannot give any function f even if it can prove U and has the axiom of choice. However, assume a variant of *ZF* that has a choice operator $\varepsilon : \Pi_{F:i\to\texttt{prop}} \to (thm\,\exists x.F(x)) \to i$, which chooses some element that satisfies F provided that such an element exists. Then we can define f as the LF-expression

$$\lambda_m \,.\, \varepsilon\left(\lambda_{m'}\ m' \in \mathbf{Mod}(\Sigma') \wedge \mathit{reduct}(v, m') = m\right)\left(\forall_E\, P\, m\right)$$

where $\forall_E$ is the elimination rule that instantiates $P : thm\,U$ with m to show the existence of the needed m'.

Note that the axiom of choice would not be sufficient here. It would only allow proving the existence of f but not choose a term for it.

Such choice operators are relatively strong features of axiomatic set theories. However, in practical foundations of mathematics that are used in proof assistants, they are very common. Examples include higher-order logic [5] and the set theory underlying Mizar [15]. For constructive foundations such as the calculus of constructions underlying Coq [2], the choice operator is trivial because the only way to prove U in the first place is to exhibit f.

We summarize the above analysis in the following theorem:

Theorem 4.13 *Assume that Sem adequately formalizes the ambient foundation of mathematics that is implicitly used in Def. 2.2.*

Moreover, assume that whenever Sem can prove a statement of the form "for all m exists m' such that $F(m, m')$", it can also define an LF-function f such that $F(m, f(m))$.

Then $v : \Sigma \to \Sigma'$ is model-conservative via sem iff it is conservative in sense of Def. 2.2.

Proof. The left-to-right direction is proved by Thm. 4.12. For the right-to-left direction, assume that for every Σ-model m there is a Σ'-model m' that reduces to m. Using the assumptions about *Sem*, that yields a function f that maps Σ-models to Σ'-models.

We construct the needed retraction r of v as follows. First we package the declarations in $\Delta, \mathit{sem}(\Sigma)$ into a *Sem*-term m. The details of this packaging depend on how *Sem* defines models. For example, if *Sem* defines models using record types, the packaging just constructs a record.

Second, r maps every symbol c of $\mathit{sem}(\Sigma')$ to the term that selects the component c from $f(m)$. Again the details of this selection depend on *Sem*. For example, if *Sem* defines models using record types, the selection is just the projection of the field c. □

4.4 Relating the Notions of Conservativity

For a logic $sem : Syn \to Sem, \Delta$, Def. 4.9 and 4.11 yield four different notions of conservativity. We fix a Syn-morphism $v : \Sigma \to \Sigma'$ and abbreviate as follows:

(PS) v is proof-conservative via sem.

(P) v is proof-conservative.

(MS) v is model-conservative via sem.

(M) v model-conservative.

By instantiating Thm. 4.8, we immediately obtain several closure properties for all four notions.

Recall that (M) means that v has a retraction, and that (P) and (MS) correspond to the notions of conservativity from Def. 2.1 and 2.2

The following theorem shows how the four properties relate to each other:

Theorem 4.14 *Let* C *be the assumption that Syn is classical and that sem preserves consistency. Then we have the following graph of implications where $A \xrightarrow{L} B$ means that L and A imply B:*

$$\begin{array}{ccc} \mathrm{M} & \longrightarrow & \mathrm{MS} \\ \downarrow & & \downarrow \\ \mathrm{P} & \overset{\mathrm{C}}{\longleftrightarrow} & \mathrm{PS} \end{array}$$

Proof. (MS) implies (PS): This is a special case of Thm. 4.5.

(M) implies (P): This is the special case of (MS) implies (PS) for $sem = id_{Syn}$.

(M) implies (MS): (M) yields a retraction $r : \Sigma' \to \Sigma$. We obtain the retraction r^+ that establishes (MS) as $sem(r)$.

$$\begin{array}{ccccc} Syn, \Sigma & \xleftarrow{r} & Syn, \Sigma' & \xrightarrow{sem^{\Sigma'}} & Sem, \Delta, sem(\Sigma') \xrightarrow{r^+} Sem, \Delta, sem(\Sigma) \\ & \nwarrow^{id} & \uparrow{\scriptstyle v} & & \uparrow{\scriptstyle sem(v)} \quad \nearrow_{id} \\ & & Syn, \Sigma & \xrightarrow{sem^{\Sigma}} & Sem, \Delta, sem(\Sigma) \end{array}$$

(PS) implies (P) if (C): Consider a Σ-sentence F such that there is a Σ'-proof $P' : v(thm\, F)$. We need to exhibit a Σ-proof $P : thm\, F$.

First, applying (PS) to the term $sem^{\Sigma'}(P')$, whose type is

$$sem^{\Sigma'}(v(thm\, F)) = sem(v)(sem^{\Sigma}(thm\, F)),$$

yields a $sem(\Sigma)$-term Q of type $sem^{\Sigma}(thm\, F)$.

The remainder of the proof uses (C) to obtain P from Q. If Σ is inconsistent, P exists trivially. So assume it is consistent. Consider $\Sigma^* = \Sigma, a : \overline{thm\, F}$. If Σ^* is inconsistent, we obtain a Σ-term of type $\overline{\overline{thm\, F}}$ and classicality of Syn yields the needed term P.

We conclude the proof by showing that Σ^* is indeed inconsistent. Because sem

preserves consistency, it suffices to show that $Sem, \Delta, sem(\Sigma^*)$ is inconsistent. That follows from the terms $a : sem^{\Sigma}(\overline{thm\, F})$ and Q.

(P) implies (PS) if (C): Consider a Σ-sentence F such that there is a $sem(\Sigma')$-proof $Q' : sem(v)(sem^{\Sigma}(thm\, F))$. We need to exhibit a $sem(\Sigma)$-proof $Q : sem^{\Sigma}(thm\, F)$.

If Σ is inconsistent, this is trivial. So assume it is consistent. Then (P) implies that Σ' is consistent, too. Now as in the case (PS) implies (P), we use the consistency of Σ' and (C) to obtain from Q' a Σ'-term $P' : v(thm\, F)$. Then (P) yields a Σ-term $P : thm\, F$, and we can put $Q = sem^{\Sigma}(P)$. □

(M) captures the situation where the syntax of the logic itself can express the proof of conservativity: as a theory morphism that retracts v. Therefore, if v satisfies (M), v satisfies any other reasonable definition of conservativity, i.e., (M) is the minimal/strongest reasonable definition. In particular, the retraction of v yields both a proof transformation from Σ' to Σ, which shows that (M) implies (P), and a model reduction from Σ to Σ', which helps showing that (M) implies (MS). [5]

One might think that (M) is too strong in practice. But it is actually very common as we see in Ex. 4.15 below.

(P) can be seen as a dual to (M)—not satisfying (P) captures the situation where the syntax of the logic itself can express a counter-example to conservativity: as a Σ-sentence that is a Σ'-theorem but not a Σ-theorem. Therefore, if v satisfies any reasonable definition of conservativity, it should satisfy (P), i.e., (P) is the maximal/weakest reasonable choice.

(MS) sits in between (M) and (P). It captures the situation where the semantics (but not necessarily the syntax) of the logic can express the proof of conservativity: as a model transformation that expands Σ-models to Σ'-models. Because the semantics is usually more expressive than the syntax, it is not surprising that (MS) is weaker than (M). Moreover, because there may be multiple different ways to give the semantics of a logic, it is not surprising that (MS) is less canonical than (M) or (P), i.e., that (MS) depends on the choice of the model theory.

Example 4.15 (M) subsumes a wide variety of important morphisms including:

- Isomorphisms.
- Extension with a definable constant. If we add a constant $c : A$ as an abbreviation for some Σ-term $t : A$, then $\Sigma, c : A$ is derivable using the morphism that maps c to t.
- Extension with a provable theorem. Adding an axiom $a : thm\, F$ if F has a proof P is just a special case of the previous case.
- Extension with a new type. Let Syn have a declaration tp : `type` such that terms $A : tp$ represent types of the logic. Adding a new type $c : tp$ is almost always derivable because we just have to map c to some existing Σ-type. The only exception is when Σ is the empty theory of a logic without any built-in types. In that case, (P) and (MS) hold but not (M).
- Extension with a new predicate symbol $p : A_1 \to \ldots \to A_n \to o$. This is essentially always derivable because we can map p to $\lambda_{x_1,\ldots,x_n} F$ for some formula F. The only exception is contrived, namely when Σ has no sentences,

[5] The corresponding observation for the framework of institutions was previously made in [14].

in which case (P) and (MS) hold but not (M).
- Extension with a new function symbol $f : A_1 \to \ldots \to A_n \to A$. This is derivable whenever Σ has a term of type A in context $x_1 : A_1, \ldots, x_n : A_n$, which is often the case. If there is no such term, (M) fails; (P) and (MS) still hold unless Syn allows empty types.
- Compositions, unions, and pushouts of morphisms that satisfy (M).

[12] describes how logic translations and semantics become formally the same thing in MMT. For example, we can have logical morphisms $Syn \xrightarrow{t} Syn' \xrightarrow{s} Sem \xrightarrow{r} Sem'$ representing a logic translation t, a semantics s, and a refinement r of the semantic foundation. Note that if $r \circ s \circ t$ preserves consistency, so do $s \circ t$ and t. Let us write (M), (Mt), (Mts), (Mtsr) for model-conservativity via id_{Syn}, t, $s \circ t$, and $r \circ s \circ t$, respectively. Then we have (M) implies (Mt) implies (Mts) implies (Mtsr) implies (P).

Thus, the more we refine Syn, the weaker model-conservativity becomes. Eventually, if we refine further and further, model-conservative and proof-conservative may eventually coincide.[6] Thus, we can think of (P) as an extreme case of (MS).

4.5 Conservativity and Completeness

Via admissibility, we can unify the concepts of conservative morphism and complete semantics:

Theorem 4.16 *The logic $sem : Syn \to Sem, \Delta$ is complete iff all sem^Σ are thm-admissible.*

Proof. [12] already proves that a sentence F holds in all Σ-models iff $Sem, \Delta, sem(\Sigma)$ has a term of type $sem^\Sigma(thm\, F)$. Due to admissibility, such a term exists iff Σ has a term of type $thm\, F$. □

Of course, a logic is also complete if the morphisms are derivable. However, because Sem is usually stronger than Syn, they are virtually never derivable in practice.

One might hope for a stronger theorem where completeness already holds whenever sem is admissible. For that, we have to ask if the admissibility of sem implies the admissibility of sem^Σ. This is not always the case, and we develop a sufficient criterion now:

Definition 4.17 We say that Syn can abstract over the declaration $c : A$ if for every for $\Sigma, c : A$-sentence F, there is a Σ-sentence $\forall_{c:A} F$ such that $\forall_{c:A} F$ is Σ-provable iff F is $\Sigma, c : A$-provable.

We write $\forall_\Gamma F$ when we iterate this construction for all declarations in Γ that occur in a Σ, Γ-sentence F.

We speak of abstracting over theories when we can abstract over every declaration that is allowed in a theory.

The intuition behind $\forall_\Gamma F$ is to universally quantify over the declarations in Γ. Thus, abstracting over theories means that Syn has universal quantification over

[6] This happens, e.g., if we use maximal consistent sets of sentences as the models.

all concepts that may be declared in theories. Note that most logics can quantify over axioms by using implication, e.g., in *FOL* we can put $\forall_{a:thm\,G} F := G \Rightarrow F$.

Example 4.18 *FOL*-theories may declare function and predicate symbols. But *FOL* can only universally quantify over variables. Therefore, it cannot abstract over theories.

Higher-order logic (HOL) with a single base type can declare typed constants. Because HOL can quantify over variables of all types, it can abstract over theories. However, the variant of HOL that allows theories to introduce additional base types cannot abstract over theories because HOL cannot quantify over type variables. For the same reason typed FOL cannot abstract over theories.

Type theories with universe hierarchies (such as the calculus of constructions) can usually quantify over all types. Therefore, they can abstract over theories.

First-order set theory allows its theories to declare sets and elements of sets. It can quantify over both and thus over theories.

Languages that allow axiom schemata, e.g., polymorphic axioms in HOL, usually cannot abstract over them.

Theorem 4.19 *Assume a logic where sem is thm-admissible.*

If Syn can abstract over Σ, then sem^{Σ} is thm-admissible. In particular, the logic is complete if Syn can abstract over all theories.

Proof. The proofs are straightforward. □

The requirement that *Syn* can abstract over theories is needed because admissibility of *sem* is a very weak notion: it talks only about sentences over the empty *Syn*-theory. Abstracting over theories makes sure that every relevant statement can be coded as a sentence over the empty theory. As a counter-example, consider FOL without equality and without constants for truth and falsity: then the empty theory happens to have no sentences at all so that any morphism out of *Syn* is already proof-conservative.

5 Conclusion

We investigated the various notions of conservativity in the MMT framework. We were able to recover the existing notions of model-theoretical (MC) and proof-theoretical (PC) conservativity as special cases of admissibility and derivability.

We saw that these two notions should always be discussed together with a third one, the retractability of an extension (R). Using MMT, it naturally emerged that (R) and (PC) are the strongest and weakest extremes, whereas (MC) sits anywhere in between depending on how far the model theory refines the syntax. Moreover, (R) arises as the special case where the initial model theory is used (where the models are theory morphisms), and (P) arises as the special cases where a maximally refined model theory is used (e.g., where the models are maximal consistent sets of sentences). This harmonically resolves the tension between the two competing notions of conservativity.

In a second result, we showed how the conservativity of a model theory (seen as a translation of the syntax) corresponds to the completeness of the logic.

References

[1] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.

[2] Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

[3] R. Diaconescu. Proof systems for institutional logic. *Journal of Logic and Computation*, 16(3):339–357, 2006.

[4] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[5] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[7] W. Hodges. *Model Theory*. Cambridge University Press, 1993.

[8] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011.

[9] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

[10] F. Pfenning. Logical frameworks. In J. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.

[11] F. Rabe. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*, 23(5):945–1001, 2013.

[12] F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 2014. doi:10.1093/logcom/exu079.

[13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[14] L. Schröder, T. Mossakowski, and C. Lüth. Type Class Polymorphism in an Institutional Framework. In J. Fiadeiro, P. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques*, pages 234–251. Springer, 2004.

[15] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.

Formalization of Universal Algebra in Agda

Emmanuel Gunther [1] Alejandro Gadea[2] Miguel Pagano[3]

FaMAF, UNC – Córdoba, Argentina

Abstract

In this work we present a novel formalization of universal algebra in Agda. We show that heterogeneous signatures can be elegantly modelled in type-theory using sets indexed by arities to represent operations. We prove elementary results of heterogeneous algebras, including the proof that the term algebra is initial and the proofs of the three isomorphism theorems. We further formalize equational theory and prove soundness and completeness. At the end, we define (derived) signature morphisms, from which we get the contra-variant functor between algebras; moreover, we also proved that, under some restrictions, the translation of a theory induces a contra-variant functor between models.

Keywords: universal algebra, formalization of mathematics, equational logic

1 Introduction

Universal algebra [2] is the study of different types of algebraic structures at an abstract level, thus revealing common results which are valid for all of them and also allowing for a unified definition of constructions (for example, products, sub-algebra, congruences). Universal algebra has played a relevant role in computer science since its earliest days, in particular the seminal paper of Birkhoff [3] features regular languages as a prominent example; shortly before, Burstall [5] had proved properties of programs using structural induction, by conceiving the language as an initial algebra. The ADJ group [10] promoted multi-sorted algebras as a key theoretical tool for specifying data types [15], semantics [16], and compilers [26]. More recently, institutions [11], a generalization of universal algebra, has been used as a foundation of methodologies and frameworks for software specification and development [23].

In spite of the rich mathematical theory of heterogeneous algebras (mostly inherited from the monosorted setting, but not always [25]), there are few publicly available formalizations in type theory (which we discuss in the conclusion). This situation is to be contrasted with impressive advances in mechanization of particular

[1] Email:gunther@famaf.unc.edu.ar Partially supported by a CONICET scholarship.
[2] Email:gadea@famaf.unc.edu.ar Partially supported by a CONICET scholarship.
[3] Email:pagano@famaf.unc.edu.ar

algebraic structures as witnessed, for example, by the proof of the Feit-Thompson theorem in Coq by Gonthier and his team [17].

In this work we present an Agda library of multi-sorted universal algebra aiming both a reader with a background in the area of algebraic specifications and also the community of type theory. For the former, we try to explain enough Agda in order to keep the paper self-contained; we will recall the most important definitions of universal algebra. The main contributions of this paper are: (i) the first formalization of basic universal algebra in Agda; (ii) the first, to our knowledge, formalization in type theory of derived signature morphisms and the reduct algebras induced by them; (iii) a novel representation of heterogeneous signatures in type theory, where operations are modelled using sets indexed by arities; and (iv) an independent library of heterogeneous vectors. We formalized the proof that the term algebra is initial and also the proofs of the three isomorphism theorems; moreover we also define a deduction system for conditional equational logic and prove its soundness and completeness with respect to Goguen and Meseguer semantics [13]. We also showed that the translations of theories arising from derived signature morphisms induces a contravariant functor between models. In the complete development, which is available at `https://cs.famaf.unc.edu.ar/~mpagano/universal-algebra/`, we include several examples featuring both the use of equational reasoning and the preservation of models by signature morphisms.

Outline. In Sec. 2 we introduce the basic concepts of Universal Algebra: signature, algebras and homomorphisms, congruences, quotients and subalgebras, the proofs of three isomorphisms theorems, and the proof of the initiality of the term algebra. In Sec. 3 we define an equational calculus, introducing concepts of equations, theories, satisfiability and provability, ending with the Birkhoff proofs of soundness and completeness. In Sec. 4 we introduce a new representation of (derived) signature morphisms and reduct algebras (and homomorphisms), and we explore translation and implication of theories. Finally, we conclude in Sec. 5, discussing the work done, and pointing out possible future directions.

2 Universal Algebra

In this section we present our formalization in Agda of the core concepts of heterogeneous universal algebra; in the next two sections we focus respectively on equational logic and signature morphisms. Meinke' and Tucker's chapter [20] is our reference for heterogeneous universal algebra; we will recall some definitions and state all the results we formalized. Bove et al. [4] offer a gentle introduction to Agda; we expect the reader to be familiar with Haskell or some other functional language.

2.1 Signature, algebra, and homomorphism

Signature

A *signature* is a pair of sets (S, F), called *sorts* and *operations* (or *function symbols*) respectively; each operation is a triple $(f, [s_1, \ldots, s_n], s)$ consisting of a *name*, its *arity*, and the *target sort* (we also use the notation $f : [s_1, ..., s_n] \Rightarrow s$).

In Agda we use dependent records to represent signatures; in dependent records

the type of some field may depend on the value of a previous one or parameters of the record. Type-theoretically one can take operations (of a signature) as a family of sets indexed by the arity and target sort (an indexed family of sets can also be thought as predicates over the index set, an index satisfies the predicate if its family is inhabited):

```
record Signature : Set₁ where
  field
    sorts : Set
    ops   : List sorts × sorts → Set
```

A × B corresponds to the non-dependent cartesian product of A and B.

In order to declare a concrete signature one first declares the set of sorts and the set of operations, which are then bundled together in a record. For example, the mono-sorted signature of monoids has an unique sort, so we use the unit type ⊤ with its sole constructor tt. We define a family indexed on List ⊤ x ⊤, with two constructors, corresponding with the operations: a 0-ary operation e, and a binary operation • (note that constructors can start with a lower-case letter or any symbol):

```
data monoid-op : List ⊤ × ⊤ → Set where
  e : monoid-op ([ ] , tt)
  • : monoid-op ([ tt , tt ] , tt)
monoid-sig : Signature
monoid-sig = record {sorts = ⊤; ops = monoid-op}
```

The signature of monoid actions has two sorts, one for the monoid and the other for the set on which the monoid acts.

```
data actMon_s : Set where
  mon : actMon_s
  set : actMon_s
data actMon_o : List actMon_s × actMon_s → Set where
  e : actMon_o ([ ] , mon)
  * : actMon_o ([ mon , mon ] , mon)
  • : actMon_o ([ mon , set ] , set)
actMon-sig : Signature
actMon-sig = record {sorts = actMon_s; ops = actMon_o}
```

Defining operations as a family indexed by arities and target sorts carries some benefits in the use of the library: as in the above examples, the names of operations are constructors of a family of datatypes and so it is possible to perform pattern matching on them. Notice also that infinitary signatures can be represented in our setting; in fact, all the results are valid for any signature, be it finite or infinite.

Algebra

An *algebra* $\mathcal{A}$ for the signature Σ consists of a family of sets indexed by the sorts of Σ and a family of functions indexed by the operations of Σ. We use $\mathcal{A}_s$ for the

interpretation or the *carrier* of the sort s; given an operation $f : [s_1, ..., s_n] \Rightarrow s$, the interpretation of f is a total function $f_{\mathcal{A}} : \mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n} \to \mathcal{A}_s$. We formalize the product $\mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n}$ as *heterogeneous vectors*. The type of heterogeneous vectors is parameterized by a set I and a family of sets indexed by I; and is indexed over a list of I:

```
data HVec {I : Set} (A : I → Set) : List I → Set where
  ⟨⟩    : HVec A []
  _▷_ : ∀ {i is} → A i → HVec A is → HVec A (i :: is)
```

The first parameter I is implicit (written in braces), which means that Agda will infer it by unification; notices that the constructor _▷_ also takes two implicit arguments (we use the notation ∀ to skip their types). Let Σ be a signature and A : sorts Σ → Set, then the product $\mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n}$ is formalized as HVec A [s_1,...,s_n].

We need one more ingredient to give the formal notion of algebras: the mathematical definition of carriers assumes an underlying notion of equality. In type theory one makes it apparent by using setoids (i.e. sets paired with an equivalence relation), which were thoroughly studied by Barthe et al. [1]. Setoids are defined in the the standard library [7] of Agda [4] as a record with three fields.

```
record Setoid : Set₁ where
  field
    Carrier : Set
    _≈_     : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _≈_
```

The relation is given as a family of types indexed over a pair of elements of the carrier (a b : Carrier are related if the type a ≈ b is inhabited); IsEquivalence _≈_ is again a record whose fields correspond to the proofs of reflexivity, symmetry, and transitivity.

The finest equivalence relation over any set is given by the *propositional equality* which only equates each element with itself, thus we can endow any set with a setoid structure with the function setoid : Set → Setoid of standard library; vice versa, there is a forgetful functor ‖_‖ : Setoid → Set which returns the carrier.

Setoid morphisms are functions which preserve the equality:

```
record _⟶_ (A B : Setoid) : Set where
  field
    _⟨$⟩_ : ‖ A ‖ → ‖ B ‖
    cong : ∀ {a a'} → _≈_ A a a' → _≈_ B (_⟨$⟩ a) (_⟨$⟩ a')
```

Notice that _⟶_ is a record parameterized on two setoids. The first field is the function, by declaring it mixfix one can write f ⟨$⟩ a when f : A ⟶ B and a : ‖ A ‖; the second field is given by a function mapping equivalence proofs on the source setoid to equivalence proofs on the target. Setoid morphisms will be used to give the interpretation of operations.

[4] Our formalization is based on several concepts defined in the standard library.

Let A : I → Set be a family of sets and P : {i : I} → A i → Set a family of predicates, we let P * : ∀ {is} → HVec A is → Set be the point-wise extension of P over heterogeneous vectors. We also use the point-wise extension to define the setoid of heterogeneous vectors given a family of setoids A : I → Setoid and write A ∗ is for the setoid of heterogeneous vectors with index is. Algebras are formalized as records parameterized on the signature.

```
record Algebra (Σ : Signature) : Set₁ where
  field
    _⟦_⟧s : sorts Σ → Setoid
    _⟦_⟧o : ∀ {ar s} → (f : ops Σ (ar , s)) → _⟦_⟧s ∗ ar ⟶≈ _⟦_⟧s s
```

If A is an algebra for the signature monoid-sig, then A ⟦ tt ⟧s is the carrier, A ⟦ e ⟧o and A ⟦ • ⟧o are the interpretations of the operations. We invite the interested reader to browse the examples to see algebras for the signatures we have shown, which cannot be given here for lack of space.

Homomorphism

Let Σ be a signature and let $\mathcal{A}$ and $\mathcal{B}$ be algebras for Σ. A *homomorphism* h from $\mathcal{A}$ to $\mathcal{B}$ is a family of functions indexed by the sorts $h_s : \mathcal{A}_s \to \mathcal{B}_s$, such that for each operation $f : [s_1, ..., s_n] \Rightarrow s$, the following holds:

$$h_s(f_{\mathcal{A}}(a_1, ..., a_n)) = f_{\mathcal{B}}(h_{s_1}\, a_1, ..., h_{s_n}\, a_n) \tag{1}$$

Notice that this is a condition over the family of functions.

In order to formalize homomorphisms we first introduce a notation for families of setoid morphisms indexed over sorts:

```
_⇝_ : ∀ {Σ} → Algebra Σ → Algebra Σ → Set
_⇝_ {Σ} A B = (s : sorts Σ) → A ⟦ s ⟧s ⟶≈ B ⟦ s ⟧s
```

We make explicit the implicit parameter Σ because otherwise sorts Σ does not make sense. [5] To enforce (1) we also define a predicate over families of setoids morphisms:

```
homCond : ∀ {Σ} {A B} → A ⇝ B → Set
homCond {Σ} {A} {B} h = ∀ {ar s} (f : ops Σ (ar , s)) (as : ‖ A ⟦_⟧s ∗ ar ‖) →
  h s ⟨$⟩ (A ⟦ f ⟧o ⟨$⟩ as) ≈s B ⟦ f ⟧o ⟨$⟩ map h as
```

where _≈s_ is the equivalence relation of the setoid B ⟦ s ⟧s and map h is the obvious extension of h over vectors. A homomorphism is a record parameterized by the source and target algebras

```
record Homo {Σ} (A B : Algebra Σ) : Set where
  field
    ′_′ : A ⇝ B
    cond : homCond ′_′
```

[5] In the library we use modules in order to avoid the repetition of the parameters Σ, A, and B.

As expected, we have the identity homomorphism $\mathsf{Id}_h\ \mathsf{A}\ :\ \mathsf{Homo\ A\ A}$ and the composition $\mathsf{G} \circ_h \mathsf{F} : \mathsf{Homo\ A\ C}$ of homomorphisms $\mathsf{F} : \mathsf{Homo\ A\ B}$ and $\mathsf{G} : \mathsf{Homo\ B\ C}$. It is also expected that $\mathsf{F} \circ_h \mathsf{Id}_h\ \mathsf{A}$ and F are equal in some sense. Since Agda is based on an intensional type theory, we cannot take the definitional equality (which distinguishes id from $\lambda\ \mathsf{n} \to \mathsf{n} + 0$ as functions on naturals); instead, we equate setoid morphisms whenever their function parts are extensionally equal:

$$\begin{aligned}&\_ \approx_{ext} \_\ :\ (\mathsf{f\ g}\ :\ \mathsf{A} \xrightarrow{\approx} \mathsf{B}) \to \mathsf{Set}\\ &\mathsf{f} \approx_{ext} \mathsf{g}\ =\ \forall\ (\mathsf{a}\ :\ \|\ \mathsf{A}\ \|) \to (\mathsf{f}\ \langle\$\rangle\ \mathsf{a})\ \approx_B\ (\mathsf{g}\ \langle\$\rangle\ \mathsf{a})\end{aligned}$$

Two homomorphisms are equal when their corresponding setoid morphisms are extensionally equal:

$$\begin{aligned}&\_ \approx_h \_\ :\ \forall\ \{\Sigma\}\ \{\mathsf{A\ B}\} \to \mathsf{Homo\ A\ B} \to \mathsf{Homo\ A\ B} \to \mathsf{Set}\\ &\mathsf{F} \approx_h \mathsf{F'}\ =\ (\mathsf{s}\ :\ \mathsf{sorts}\ \Sigma) \to {}'\,\mathsf{F}\,'\,\mathsf{s}\ \approx_{ext}\ {}'\,\mathsf{F'}\,'\,\mathsf{s}\end{aligned}$$

With respect to this equality, it is straightforward to prove the associativity of the composition $\_ \circ_h \_$ and that Id_h is the identity for the composition.

2.2 Quotient and subalgebras

In order to prove the more basic results of universal algebra, we need to formalize subalgebras, congruence relations, and quotients.

Subalgebra

A subalgebra $\mathcal{B}$ of an algebra $\mathcal{A}$ consists of a family of subsets $\mathcal{B}_s \subseteq \mathcal{A}_s$, that are closed under the interpretation of operations; that is, for every $f : [s_1, \ldots, s_n] \Rightarrow s$ the following condition holds

$$(a_1, \ldots, a_n) \in \mathcal{B}_{s_1} \times \cdots \times \mathcal{B}_{s_n} \text{ implies } f_{\mathcal{A}}(a_1, \ldots, a_n) \in \mathcal{B}_s\ . \tag{2}$$

As shown by Salvesen and Smith [22], subsets cannot be added as a construction in intensional type theory because they lack desirable properties. If $\mathsf{A}\ :\ \mathsf{Set}$ and $\mathsf{P}\ :\ \mathsf{A} \to \mathsf{Set}$ is a predicate over A, then one can represent the subset containing the elements on A that satisfy P as the dependent sum [6] $\Sigma[\ \mathsf{a} \in \mathsf{A}\ ]\ \mathsf{P}$ whose inhabitants are pairs $(\mathsf{a}\ ,\ \mathsf{p})$ where $\mathsf{a}\ :\ \mathsf{A}$ and $\mathsf{p}\ :\ \mathsf{P\ a}$. Let us consider a setoid A and a predicate on its carrier $\mathsf{P}\ :\ \|\ \mathsf{A}\ \| \to \mathsf{Set}$; first notice that we can lift the subset construction to setoids, defining the equivalence relation $(\mathsf{a}\ ,\ \mathsf{q}) \approx (\mathsf{a'}\ ,\ \mathsf{q'})$ iff $\mathsf{a} \approx \mathsf{a'}$. Moreover, we might assume that P is *well-defined*, which means that $\mathsf{a}\ \approx_A\ \mathsf{a'}$ and $\mathsf{P\ a}$ imply $\mathsf{P\ a'}$.

$$\begin{aligned}&\mathsf{WellDef}\ :\ (\mathsf{A}\ :\ \mathsf{Setoid}) \to (\mathsf{P}\ :\ \|\ \mathsf{A}\ \| \to \mathsf{Set}) \to \mathsf{Set}\\ &\mathsf{WellDef\ A\ P}\ =\ \forall\ \{\mathsf{a\ a'}\} \to \mathsf{a}\ \approx_A\ \mathsf{a'} \to \mathsf{P\ a} \to \mathsf{P\ a'}\end{aligned}$$

A family of well-defined predicates will induce a subalgebra; but we still need to formalize the condition (2). Let Σ be a signature and A be an algebra for Σ.

[6] Do not confuse the syntax $\Sigma[\_\in\_]\_$ of dependent sum, with a variable $\Sigma\ :\ \mathsf{Signature}$

```
opClosed : (P : (s : sorts Σ) → ∥ A ⟦ s ⟧s∥ → Set) → Set
opClosed P = ∀ {ar s} (f : ops Σ (ar , s)) → (P * ⟨→⟩ P s) (A ⟦ f ⟧o ⟨$⟩_)
```

(Q ⟨→⟩ R) f can be read as the pre-condition Q implies post-condition R after applying f; so opClosed P f asserts that if a vector a* satisfies the predicate P, then the application of the interpretation A ⟦ f ⟧o to a* satisfies P, according to Eq. (2). In summary, given an algebra A for the signature Σ and a family P of predicates, such that P s is well-defined for every sort s and P is opClosed, we can define the SubAlgebra A P

```
SubAlgebra : ∀ {Σ} A P → WellDef P → opClosed P → Algebra Σ
```

In the subalgebra, an operation f is interpreted by applying the interpretation of f in A to the first components of the argument (and use the fact that P is op-closed to show that the resulting value satisfies the predicate of the target sort).

Congruence and Quotients

A *congruence* on a Σ-algebra $\mathcal{A}$ is a family Q of equivalence relations indexed by sorts, and each of them is closed under the operations of the algebra. This condition is called *substitutivity* and can be formalized using the point-wise extension of Q over vectors: for every operation $f : [s_1, \ldots, s_n] \Rightarrow s$

$$(\boldsymbol{a}, \boldsymbol{b}) \in Q_{s_1} \times \cdots \times Q_{s_n} \text{ implies } (f_{\mathcal{A}}(\boldsymbol{a}), f_{\mathcal{A}}(\boldsymbol{b})) \in Q_s \tag{3}$$

As with predicates, we say that a binary relation over a setoid is well-defined if it is preserved by the setoid equality; this notion can be extended over families of relations in the obvious way. In our formalization, a congruence on an algebra A is a family Q of well-defined, equivalence relations. The substitutivity condition (3) is aptly captured by the generalized containment operator _=[_]⇒_ of the standard library, where P =[f]⇒ Q if, for all a,b ∈ A, (a,b) ∈ P implies (f a, f b) ∈ Q.

```
record Congruence (A : Algebra Σ) : Set where
  field
    rel : (s : sorts Σ) → (∥ A ⟦ s ⟧s ∥ → ∥ A ⟦ s ⟧s ∥ → Set)
    welldef : (s : sorts Σ) → WellDefBin (rel s)
    cequiv : (s : sorts Σ) → IsEquivalence (rel s)
    csubst : ∀ {ar s} → (f : ops Σ (ar , s)) → rel * =[ A ⟦ f ⟧o ⟨$⟩_ ]⇒ rel s
```

Given a congruence Q over the algebra $\mathcal{A}$, we can obtain a new algebra, the *quotient algebra*, by interpreting the sort s as the set of equivalence classes $\mathcal{A}_s/Q$; the condition (3) ensures that the operation $f : [s_1, \ldots, s_n] \Rightarrow s$ can be interpreted as the function mapping the vector $([a_1], \ldots, [a_n])$ of equivalence classes into the class $[f_{\mathcal{A}}(a_1, \ldots, a_n)]$. In Agda, we take the same carriers from A and use Q s as the equivalence relation over ∥ A ⟦ s ⟧s ∥; operations are interpreted just as in A and the congruence proof is given by csubst Q.

Isomorphism Theorems

The definitions of subalgebras, quotients, and epimorphisms (surjective homomorphisms) are related by the three isomorphims theorems. Although there is some small overhead by the coding of subalgebras, the proofs follow very close what one would do in paper. For proving these results we also defined the *kernel* and the *homomorphic* image of homomorphisms.

Theorem 2.1 (First isomorphism theorem) *If $h : \mathcal{A} \to \mathcal{B}$ is an epimorphism, then $\mathcal{A}/ker\, h \simeq \mathcal{B}$.*

Remember that the quotient $\mathcal{A}/ker\, h$ has the same carrier as $\mathcal{A}$, so h counts as the underlying function and it respects the equivalence relation $ker\, h$ by definition. Clearly h is surjective and its injectivity is obvious.

Theorem 2.2 (Second isomorphism theorem) *If ϕ, ψ are congruences over $\mathcal{A}$, such that $\psi \subseteq \phi$, then $(\mathcal{A}/\phi) \simeq (\mathcal{A}/\psi)/(\phi/\psi)$.*

In order to prove this theorem, we first prove that ϕ/ψ is a congruence over $\mathcal{A}/\psi$: it suffices to prove the well-definedness of ϕ/ψ, i.e. that $(a, c) \in \psi$, $(b, d) \in \psi$, and $(a, b) \in \phi$ imply $(c, d) \in \phi$; an obvious consequence of $\psi \subseteq \phi$. Notice that the underlying carriers are the same in both cases: those of $\mathcal{A}$, so the identity function is the mediating isomorphism and the proof that it satisfies the homomorphism condition is trivial.

Theorem 2.3 (Third isomorphism theorem) *Let $\mathcal{B}$ be a subalgebra of $\mathcal{A}$ and ϕ be a congruence over $\mathcal{A}$. Let $[\mathcal{B}]^\phi = \{K \in A/\phi : K \cap B \neq \emptyset\}$ and let ϕ_B be the restriction of ϕ to $\mathcal{B}$, then (i) ϕ_B is a congruence over $\mathcal{B}$;(ii) $[\mathcal{B}]^\phi$ is a subalgebra of $\mathcal{A}$; and,(iii) $[\mathcal{B}]^\phi \simeq \mathcal{B}/\phi_B$.*

First we define the *trace* of the congruence ϕ on the subalgebra $\mathcal{B}$ as the restriction of ϕ on $\mathcal{B}$; proving that it is a congruence over $\mathcal{B}$ involves some bureaucracy (remember that an element of a subalgebra is a pair (a, p) such that $a \in A$ and p is the proof that a satisfies the predicate defining B). For the second item, we model $[\mathcal{B}]^\phi$ as a predicate over $\mathcal{A}$; it is satisfied by $a \in A$ if there is some $b \in B$ such that $(a, b) \in \phi$. The well-definedness of this predicate is easy (assuming $(a, a') \in \phi$ and $b \in B$ with $(a, b) \in \phi$, one can easily prove that $(a', b) \in \phi$, thus b is also the witness for proving that a' satisfies the predicate). To prove that the predicate is closed under the operations we take a vector of triples (as, bs, ps) consisting of a vector of elements in A, a vector of elements in B, and the proofs ps proving that $(as_i, bs_i) \in \phi$. Let f be an operation, since B is closed we know $f(b_1, \ldots, b_n) \in B$ and because ϕ is also closed we deduce $(f(a_1, \ldots, a_n), f(b_1, \ldots, b_n)) \in \phi$. Finally, the underlying function witnessing the isomorphism $[\mathcal{B}]^\phi \simeq \mathcal{B}/\phi_B$ is given by composing the second projection with the first projection, thus getting an element in B.

2.3 Term algebra is initial

A Σ-algebra $\mathcal{A}$ is called *initial* if for any Σ-algebra $\mathcal{B}$ there exists exactly one homomorphism from $\mathcal{A}$ to $\mathcal{B}$. We give an abstract definition of this universal property, existence of an unique element, for any set A and any relation R

```
hasUnique {A} _≈_ = A × (∀ a a' → a ≈ a')
```

and initiality can be formalized directly:

```
Initial : ∀ {Σ} → Algebra Σ → Set
Initial {Σ} A = ∀ (B : Algebra Σ) → hasUnique (_≈ₕ_ A B)
```

Given a signature Σ we can define the *term algebra* $\mathcal{T}$, whose carriers are sets of well-typed words built up from the function symbols. Sometimes this universe is called the *Herbrand Universe* and is inductively defined:

$$\frac{t_1 \in \mathcal{T}_{s_1} \qquad \cdots \qquad t_n \in \mathcal{T}_{s_n}}{f\,(t_1, ..., t_n) \in \mathcal{T}_s}\; f : [s_1, ..., s_n] \Rightarrow s$$

This inductive definition can be written directly in Agda:

```
data HU {Σ : Signature} : (s : sorts Σ) → Set where
  term : ∀ {ar s} → (f : ops Σ (ar ↦ s)) → HVec HU ar → HU s
```

We use propositional equality to turn each HU_s into a setoid, thus completing the interpretation of sorts. To interpret an operation $f\colon [s_1, \ldots, s_n] \Rightarrow s$ we map the vector $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$: HVec HU $[\mathsf{s}_1, \ldots, \mathsf{s}_n]$ to term f $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$; we omit the proof of cong, which is too long and tedious to be shown.

```
|T| : (Σ : Signature) → Algebra Σ
|T| Σ = record {_⟦_⟧ₛ = setoid ∘ (HU {Σ}); _⟦_⟧ₒ = |_|ₒ}
  where |f|ₒ = record {_⟨$⟩_ = term f; cong = ...}
```

Terms can be interpreted in any algebra $\mathcal{A}$, yielding an homomorphism $h_A\colon \mathcal{T} \to \mathcal{A}$

$$h_A(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}\,(h_A\,t_1, ..., h_A\,t_n)\ .$$

We cannot translate this definition directly in Agda, instead we have to mutually define |h| and its extension over vectors |h*|

```
|h| : ∀ {Σ} → (A : Algebra Σ) → {s : sorts Σ} → HU s → ∥ A ⟦ s ⟧ₛ ∥
|h| A (term f ts) = A ⟦ f ⟧ₒ ⟨$⟩ (|h*| ts)
```

It is straightforward to prove that |h| preserves propositional equality and satisfies the homomorphism condition by construction. To finish the proof that |T| Σ is initial, we prove, by recursion on the structure of terms, that any pair of homomorphisms are extensionally equal.

3 Equational Logic

In this section we introduce the notion of (conditional) equational theories and the corresponding notion of satisfiability of theories by algebras. Moreover we formalize (conditional) equational logic as presented by Goguen and Lin [12] and prove that the deduction system is sound and complete.

3.1 Free algebra with variables

The term algebra we have just defined contained only *ground* terms, i.e. terms without variables. Given a signature Σ and $\mathsf{X} : \mathsf{sorts}\ \Sigma \to \mathsf{Set}$ a family of variables, we define a new signature extending Σ with X by taking the variables as new constants (i.e. , operations with arity []).

$$
\begin{array}{l}
\_(\!|\_|\!) : (\Sigma : \mathsf{Signature}) \to (\mathsf{X} : \mathsf{sorts}\ \Sigma \to \mathsf{Set}) \to \mathsf{Signature} \\
\Sigma\ (\!|\ \mathsf{X}\ |\!) = \mathbf{record}\ \{\mathsf{sorts} = \mathsf{sorts}\ \Sigma; \mathsf{ops} = \mathsf{ops'}\} \\
\quad \mathbf{where}\ \mathsf{ops'}\ ([]\ ,\ \mathsf{s}) = \mathsf{ops}\ \Sigma\ ([]\ ,\ \mathsf{s}) \uplus \mathsf{X}\ \mathsf{s} \\
\qquad\qquad \mathsf{ops'}\ (\mathsf{ar}\ ,\ \mathsf{s}) = \mathsf{ops}\ \Sigma\ (\mathsf{ar}\ ,\ \mathsf{s})
\end{array}
$$

Note that it is easy to refer to constant operations and extend them, because we indexed the set of operations on their arity and target sort.

It is easy to turn the term algebra of the extended signature into an algebra for the original signature:

$$
\begin{array}{l}
|\mathsf{T}|\_(\!|\_|\!) : (\Sigma : \mathsf{Signature}) \to (\mathsf{X} : \mathsf{sorts}\ \Sigma \to \mathsf{Set}) \to \mathsf{Algebra}\ \Sigma \\
|\mathsf{T}|\ \Sigma\ (\!|\ \mathsf{X}\ |\!) = \mathbf{record}\ \{\ \_[\![\_]\!]_s = |\mathsf{T}|\ (\Sigma\ (\!|\ \mathsf{X}\ |\!))\ [\![\_]\!]_s\ ,\ \_[\![\_]\!]_o = \mathsf{io}\} \\
\quad \mathbf{where}\ \mathsf{io}\ \{[]\}\ \mathsf{f} = |\mathsf{T}|\ (\Sigma\ (\!|\ \mathsf{X}\ |\!))\ [\![\ \mathsf{inj}_1\ \mathsf{f}\ ]\!]_o \\
\qquad\qquad \mathsf{io}\ \{\mathsf{ar}\}\ \mathsf{f} = |\mathsf{T}|\ (\Sigma\ (\!|\ \mathsf{X}\ |\!))\ [\![\ \mathsf{f}\ ]\!]_o
\end{array}
$$

The only difference with the algebra of ground terms is that we inject constants from Σ to distinguish them from variables. In order to interpret terms with variables we need *environments* to give meaning to variables.

Let $\mathsf{Env}\ \mathsf{X}\ \mathsf{A} = \forall\ \{\mathsf{s}\} \to \mathsf{X}\ \mathsf{s} \to \|\ \mathsf{A}\ [\![\ \mathsf{s}\ ]\!]_s\ \|$ be the set of environments from X to A. The free algebra $|\mathsf{T}|\ \Sigma\ (\!|\ \mathsf{X}\ |\!)$ has the universal *freeness* property: given $\mathsf{A} : \mathsf{Algebra}\ \Sigma$ and an environment $\theta : \mathsf{Env}\ \mathsf{X}\ \mathsf{A}$, there exists an unique homomorphism $[\![\_]\!]\theta : \mathsf{Homo}\ (|\mathsf{T}|\ \Sigma\ (\!|\ \mathsf{X}\ |\!))\ \mathsf{A}$ such that $[\![\ \mathsf{x}\ ]\!]\theta = \theta\ (\mathsf{x})$ for $\mathsf{x} \in \mathsf{X}$.

3.2 Satisfiability and provability

Equations

In the mono-sorted setting an equation is a pair of terms where all the variables are assumed to be universally quantified and an equational theory is a (finite) set of equations. In a multi-sorted setting both sides of an equation should be terms of the same sort. Moreover we allow quasi-identities which we write as conditional equations:

$$t = t' \ \mathsf{if}\ t_1 = t'_1, \ldots, t_n = t'_n\ .$$

Let Σ be a signature and $\mathsf{X} : \mathsf{sorts}\ \Sigma \to \mathsf{Set}$ be a family of variables for Σ. An identity $\mathsf{e} : \mathsf{Eq}\ \Sigma\ \mathsf{X}\ \mathsf{s}$ is a pair of (open) terms with sort s. A conditional equation is modelled as record with fields for the conclusion and the conditions, modelled as an heterogeneous vector of sorted identities . We declare a constructor to use the lighter notation $\bigwedge \mathsf{eq}\ \mathsf{if}\ (\mathsf{ar}\ ,\ \mathsf{eqs})$ instead of $\mathbf{record}\ \{\mathsf{eq} = \mathsf{e}; \mathsf{cond} = (\mathsf{ar}\ ,\ \mathsf{eqs})\}$.

$$
\begin{array}{l}
\mathbf{record}\ \mathsf{Equation}\ (\Sigma : \mathsf{Signature})\ (\mathsf{X} : \mathsf{sorts}\ \Sigma \to \mathsf{Set})\ (\mathsf{s} : \mathsf{sorts}\ \Sigma) : \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{constructor}\ \bigwedge\_\mathsf{if}\_ \\
\quad \mathbf{field}
\end{array}
$$

```
eq   : Eq Σ X s
cond : Σ[ ar ∈ List (sorts Σ) ] (HVec (Eq Σ X) ar)
```

A *theory* over the signature Σ is given by a vector of conditional equations.

```
Theory : (Σ : Signature) → (X : sorts Σ → Set) → (ar : List (sorts Σ)) → Set
Theory Σ X ar = HVec (Equation Σ X) ar
```

We deviate from Goguen's and Lin's in that we assume that all the equations of a theory share the same set of variables, while they assume that each equation has its own set of quantified variables. Clearly, this simplification is harmless; if we have a theory where each equation has its own set of variables, we can take the union of those sets as the common set. As stressed by Goguen and Meseguer [14], quantifying equations is essential:

> [...] the naive unsorted rules of deduction for equational logic (namely, reflexivity, symmetry, transitivity and substitutivity) are not sound when extended to the many-sorted case in the obvious way; [...] adding variable declarations to these rules yields a rule set that is sound.

Satisfiability

Let Σ be a signature and $\mathcal{A}$ be an algebra for Σ. We say that a conditional equation $t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ is *satisfied* by $\mathcal{A}$ if for any environment $\theta : X \to \mathcal{A}$, $[\![t]\!]\theta = [\![t']\!]\theta$, whenever $[\![t_i]\!]\theta = [\![t'_i]\!]\theta$ for $1 \leqslant i \leqslant n$. In order to formalize satisfiability we first define when an environment models an equation.

```
_⊨ₑ_ : ∀ {Σ X A} → (θ : Env X A) → {s : sorts Σ} → Eq Σ X s → Set
_⊨ₑ_ θ {s} (t , t') = _≈_ (A ⟦ s ⟧ₛ) ([[ t ]] θ) ([[ t' ]] θ)
```

Using the point-wise extension of this relation we can write directly the notion of satisfiability.

```
_⊨_ : ∀ {Σ X} (A : Algebra Σ) → {s : sorts Σ} → Equation Σ X s → Set
A ⊨ (⋀ eq if (_ , eqs)) = ∀ θ → ((θ ⊨ₑ_) * eqs) → θ ⊨ₑ eq
```

We say that $\mathcal{A}$ is a *model* of the theory E if it satisfies each equation in E. As usual an equation is a logical consequence of a theory, if every model of the theory satisfies the equation.

```
_⊨ₘ_ : ∀ {Σ X ar} → (A : Algebra Σ) → (E : Theory Σ X ar) → Set
A ⊨ₘ E = (A ⊨_) * E
_ ⊨Σ _ : ∀ {Σ X ar s} → (E : Theory Σ X ar) → (e : Equation Σ X s) → Set
_ ⊨Σ _ {Σ} E e = (A : Algebra Σ) → A ⊨ₘ E → A ⊨ e
```

Provability

As noticed by Huet and Oppen [18], the definition of a sound deduction system for multi-sorted equality logic is more subtle than expected. We formalize the system

$$\frac{}{E \vdash \forall X,\, t = t} \qquad \frac{E \vdash \forall X,\, t_0 = t_1}{E \vdash \forall X,\, t_1 = t_0} \qquad \frac{E \vdash \forall X,\, t_0 = t_1 \qquad E \vdash \forall X,\, t_1 = t_2}{E \vdash \forall X,\, t_0 = t_2}$$

$$\frac{\forall Y,\, t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \in E \qquad E \vdash \forall X,\, \sigma(t_i) = \sigma(t'_i)}{E \vdash \forall X,\, \sigma(t) = \sigma(t')}\ \sigma \colon Y \to T_\Sigma(X)$$

$$\frac{E \vdash \forall X,\, t_1 = t'_1 \qquad \cdots \qquad E \vdash \forall X,\, t_n = t'_n}{E \vdash \forall X,\, f\,(t_1, \ldots, t_n) = f\,(t'_1, \ldots, t'_n)}\ f : [s_1, ..., s_n] \Rightarrow_\Sigma s$$

Fig. 1. Deduction system

presented in [12], shown in Fig. 1. The first three rules are reflexivity, symmetry and transitivity; the fourth rule, called substitution, allows to instantiate an axiom with a substitution σ, provided one has proofs for every condition of the axiom; [7] finally, the last rule internalizes Leibniz rule, for replacing equals by equals in subterms. Notice that we can only prove identities and not quasi-identities. We define the relation of provability as an inductive type, parameterized in the theory E, and indexed by the conclusion of the proof. For conciseness, we only show the constructor for transitivity:

```
data _⊢_ {Σ X ar} (E : Theory Σ X ar) : ∀ {s} → Eq Σ X s → Set where
  ptrans : ∀ {s} {t₀ t₁ t₂} →
             E ⊢ (t₀ , t₁) → E ⊢ (t₁ , t₂) → E ⊢ (t₀ , t₂)
```

Let E be a theory over a signature Σ. It is straightforward to define a setoid over |T| Σ (| X |) by letting $t_1 \approx t_2$ if $E \vdash t_1 \approx t_2$; this equivalence relation (thanks to the first three rules) is a congruence (because of the last rule) over the term algebra. We can also use the facility provided by the standard library to write proofs with several transitive steps more nicely, as can be seen in the next example.

The proofs of soundness and completeness are proved as in the mono-sorted case. For soundness one proceeds by induction on the derivations; completeness is a consequence that the quotient of the term algebra by provable equality is a model.

Theorem 3.1 (Soundness and Completeness) *$E \vdash t \approx t'$ iff $E \models_\Sigma t \approx t'$.*

Let us remark that completeness does not imply that there is a decidability algorithm for every theory; i.e. this result gives no decision procedure at all.

Let E and E' be two theories over the signature Σ. We say that E is *stronger* than E' if every axiom $e \in E'$ can be deduced from E, written $E \vdash \mathrm{T}\, E'$. Obviously if E is stronger than E', then any equation that can be deduced from E' can also be deduced from E and any model of E is also a model of E'.

3.3 A theory for Boolean Algebras

As an example we show a fragment of the formalization of a Boolean Theory discussed in [21]. [8] The signature is mono-sorted, so we use the unit type as its only sort.

[7] In our formalization this rule is slightly less general because we assume all the equations are quantified over the same set of variables.

[8] The full code is available at https://cs.famaf.unc.edu.ar/~mpagano/universal-algebra/html/Examples.EqBool.html.

```
data bool-ops : List ⊤ × ⊤ → Set where
  f t : bool-ops ([] ↦ tt)
  neg : bool-ops ([ tt ] ↦ tt)
  and or : bool-ops (([ tt , tt ]) ↦ tt)
bool-sig : Signature
bool-sig = record {sorts = ⊤; ops = bool-ops}
```

We let X tt = ℕ be the set of variables, and let Form stand for terms over bool-sig ⦇ Vars ⦈ with the following smart-constructors:

```
true false : Form
true = term (inj₁ t) ⟨⟩
false = term (inj₁ f) ⟨⟩
p q : Form
p = term (inj₂ 0) ⟨⟩
q = term (inj₂ 1) ⟨⟩
_∧_ : Form → Form → Form
φ ∧ ψ = term and ⟨ φ , ψ ⟩
¬ : Form → Form
¬ φ = term neg ⟨ φ ⟩
```

We show only two of the twelve axioms of the theory E-Bool:

```
commAnd leastDef : Equation bool-sig Vars tt
commAnd = ⋀ (p ∧ q) ≈ (q ∧ p) if ([] , ⟨⟩)
leastDef = ⋀ (p ∧ (¬ p)) ≈ false if ([] , ⟨⟩)
E-bool : Theory bool-sig Vars [ tt , tt , ... ]
E-bool = ⟨ commAnd , leastDef , ... ⟩
```

The following example shows an equational proof using the facility for equational reasoning provided by the standard library of Agda. In the justification steps we use the substitution rule (called psubst) and the pattern-synonyms commAndAx,leastDefAx as short-hands for commAnd ∈ E-bool and leastDef ∈ E-bool, respectively.

```
p₁ : E-bool ⊢ (⋀ ¬ p ∧ p ≈ false)
p₁ = begin
  ¬ p ∧ p
  ≈⟨ psubst commAndAx σ₁ ∼⟨⟩ ⟩
  p ∧ ¬ p
  ≈⟨ psubst leastDefAx idSubst ∼⟨⟩ ⟩
  false
  ■
```

The relevant actions of the substitution σ_1 are $\sigma_1\ \mathsf{p} = \neg\ \mathsf{p}$ and $\sigma_1\ \mathsf{q} = \mathsf{p}$.

$$\frac{}{[s_1,\ldots,s_n] \Vdash^{\Sigma} \sharp i : s_i}\ (\text{PRJ}) \qquad \frac{f : [s_1,...,s_n] \Rightarrow_{\Sigma} s \;\; ar \Vdash^{\Sigma} t_1 : s_1 \;\cdots\; ar \Vdash^{\Sigma} t_n : s_n}{ar \Vdash^{\Sigma} f\,(t_1,...,t_n) : s}\ (\text{OP})$$

Fig. 2. Type system for formal terms

4 Morphisms between signatures

The propositional calculus of Dijkstra and Scholten [8] is an alternative boolean theory whose only non-constant operations are equivalence and disjunction.

```
data bool-ops' : List ⊤ × ⊤ → Set where
  f' t' : bool-ops' ([] ↦ tt)
  equiv' or' : bool-ops' ([ tt , tt ] ↦ tt)
bool-sig' : Signature
bool-sig' = record {sorts = ⊤ , ops = bool-ops'}
```

It is clear that one can translate recursively any term over bool-sig to a term in bool-sig' preserving its semantics. An alternative and more general way is to specify how to translate each operation in bool-sig using operations in bool-sig'. In this way, any bool-sig'-algebra can be seen as a bool-sig-algebra: a bool-sig-operation f is interpreted as the semantics of the translation of f. In particular, the translation of formulas is recovered as the initial homomorphism between |T| bool-sig and the transformation of |T| bool-sig'. In this section we formalize the concepts of *derived signature morphism* and *reduct algebra* as introduced, for example, by Sanella et al. [23].

4.1 Derived signature morphism

Although the disjunction from bool-sig can be directly mapped to its namesake in bool-sig', there is no unary operation in bool-sig' to translate the negation. In fact, we should be able to translate an operation as a combination of operations in bool-sig' and also refer to the arguments of the original operation.

We introduce the notion of *formal terms* which are formal composition of projections and operations. We introduce a type system, shown in Fig. 2, ensuring the well-formedness of these terms: the contexts are arities, i.e. lists of sorts, and identifiers are pointers (like de Bruijn indices). It can be formalized as an inductive family parameterized by arities and indexed by sorts.

```
data _⊩_ (ar' : Arity Σ) : (sorts Σ) → Set where
  #_ : (n : Fin (length ar')) → ar' ⊩ (ar' !! n)
  _|$|_ : ∀ {ar s} → ops Σ (ar ⇒ s) → HVec (ar' ⊩_) ar → ar' ⊩ s
```

A formal term specifies how to interpret an operation from the source signature in the target signature. The arity ar' specifies the sort of each argument of the original operation. For example, since the operation neg is unary, we can use one identifier when defining its translation. Notice that bool-sig and bool-sig' share the sorts; in general, one also considers a mapping between sorts.

A *derived signature morphism* consists of a mapping between sorts and a mapping from operations to formal terms:

```
record _↪_ (Σs Σt : Signature) : Set where
  field
    ↪s : sorts Σs → sorts Σt
    ↪o : ∀ {ar s} → ops Σs (ar , s) → (map ↪s ar) ⊩ (↪s s)
```

We show the action of the morphism on the operations neg and and

```
ops↪ : ∀ {ar s} → (f : bool-ops (ar ↦ s)) → map id ar ⊩ s
ops↪ neg = equiv' |$| ⟨ p , f' ⟩
ops↪ and = equiv' |$| ⟨ equiv' |$| ⟨ p , q ⟩ , or' |$| ⟨ p , q ⟩ ⟩
```

where p = # zero and q = # (suc zero).

4.2 Transformation of Algebras

A signature morphism $m\colon \Sigma_s \hookrightarrow \Sigma_t$ induces a functor from Σ_t-algebras to Σ_s-algebras. Given a Σ_t-algebra $\mathcal{A}$, we denote with $\langle\mathcal{A}\rangle$ the corresponding Σ_s-algebra, which is known as the *reduct algebra with respect to the morphism* m. Let us sketch the construction of the functor on algebras: the interpretation of a Σ_s-sort s is given by $\langle\mathcal{A}\rangle_s = \mathcal{A}_{(m\,s)}$ and for interpreting an operation f in the reduct algebra $\langle\mathcal{A}\rangle$ we use the interpretation of the formal term mf, which is recursively defined by

```
⟦_⟧t : ∀ {ar s} → ar ⊩ s → ∥ A ⟦ ar ⟧s* ∥ → ∥ A ⟦ s ⟧s ∥
⟦ # n ⟧t as = as !!v n
⟦ f |$| ts ⟧t as = A ⟦ f ⟧o ⟨$⟩ ⟦ ts ⟧t* as
```

Identifiers denote projections and the application of the operation f to formal terms ts is interpreted as the interpretation of f applied to the denotation of each term in ts, the function ⟦_⟧t* extends ⟦_⟧t to vectors.

We can formalize the reduct algebra in a direct way, however the interpretation of operations is a little more complicated, since we need to convince Agda that any vector vs : HVec (A ⟦_⟧s ∘ ↪s) is has also the type HVec A (map ↪s is), which is accomplished by reindex-ing the vector (we omit the proof of cong):

```
module ReductAlg (m : Σs ↪ Σt) (A : Algebra Σt) where
  ⟨_⟩s : → (s : sorts Σs) → Setoid
  ⟨ s ⟩s = A ⟦ ↪s m s ⟧s

  ⟨_⟩o : ∀ {ar s} → ops Σs (ar ⇒ s) → (⟨_⟩s) ∗ ar ⟶≈ ⟨ s ⟩s
  ⟨ f ⟩o = record { _⟨$⟩_ = ⟦ ↪o m f ⟧t ∘ reindex (↪s m); cong = ... }

  _⟨_⟩ : Algebra Σs
  _⟨_⟩ = record { _⟦_⟧s = ⟨_⟩s , _⟦_⟧o = ⟨_⟩o }
```

The action of the functor on homomorphisms is also straightforward, we do not it show for lack of space.

4.3 Translation of theories

From a signature morphism $m : \Sigma_s \hookrightarrow \Sigma_t$ one gets the translation of ground Σ_s terms as the initial homomorphism from $|\mathsf{T}|\ \Sigma_s$ to $\langle\ |\mathsf{T}|\ \Sigma_t\ \rangle$. With an appropriate extension to variables, this translation applied to a theory E_s over Σ_s yields the theory $\widetilde{E_s}$ over Σ_t. Moreover if $\mathcal{A}_t \models \widetilde{E_s}$, one would think that the reduct $\langle \mathcal{A}_t \rangle$ is a model of the original theory, i.e. $\langle \mathcal{A}_t \rangle \models E_s$. Even better, if E_t is a stronger theory than the translated theory $\widetilde{E_s}$ and if $\mathcal{A}_t$ is a model for E_t, we would like that the reduct algebra models E_s. In Agda such a result would be realized as a function $\models\!\hookrightarrow$ with the following type (where $\hookrightarrow\!^*\ \mathsf{E}_s$ is the translation of E_s):

$$\models\!\hookrightarrow\ :\ \forall\ \mathsf{A}_t\ \mathsf{E}_t\ \mathsf{E}_s \to \mathsf{A}_t \models_m \mathsf{E}_t \to (\mathsf{E}_t \vdash \mathsf{T} \hookrightarrow\!^*\ \mathsf{E}_s) \to \langle\ \mathsf{A}_t\ \rangle \models_m \mathsf{E}_s$$

With the morphism $m : \Sigma_s \hookrightarrow \Sigma_t$, one can define the translation of open terms from $|\mathsf{T}|\ \Sigma_s\ (\!|\ \mathsf{X}_s\ |\!)$ to $|\mathsf{T}|\ \Sigma_t\ (\!|\ \mathsf{X}_t\ |\!)$ using initiality if we also have a renaming function $\hookrightarrow_v\ :\ \{\mathsf{s}\ :\ \mathsf{sorts}\ \Sigma_s\} \to \mathsf{X}_s\ \mathsf{s} \to \mathsf{X}_t\ (\mathsf{m} \hookrightarrow_s \mathsf{s})$. In general, however, we cannot prove the *satisfaction property*: if a Σ_t-algebra models the translation of an equation, then its reduct models the original equation. The technical issue is the impossibility of defining a Σ_t-environment from a Σ_s-environment. There is a well-known solution which consists on restricting the set of variable of the target signature by letting $X_t = \bigcup_{s \in \Sigma_s, t = m \hookrightarrow s} X_s$. Under this restriction, we can prove the satisfaction property and furthermore define the function $\models\!\hookrightarrow$. Such a restriction over the set of variables seems to us as an impediment, which can be alleviated if the original variables of E_t are included in the calculated set of variables.

5 Conclusions

As far as we know, heterogeneous universal algebra has not attracted a great interest in the academic community of type theory. In this paper, we have developed in Agda a library with the main concepts of heterogeneous universal algebra, up to the proof of the three isomorphisms theorems and the freeness of the term algebra over a set of variables. In order to define the term algebra we have introduced heterogeneous vectors, which later turned out to be very useful in other parts of the library, for example as the set of axioms of finite theories and as premises of deduction rules. We further introduced a formal system for conditional equational logic and proved its soundness and completeness with respect to Goguen and Meseguer semantics (we refer the reader to [27] for a deeper explanation of this result recasting it on a categorical setting). Finally, we defined a novel representation for (derived) signature morphisms and its associated contra-variant functor on algebras. We also showed that, under some restrictions, this functor also preserves models.

Related Work. Let us contrast our work with other formalizations covering some aspects of universal algebra. As far as we know, since Capretta's [6] first mechanization of universal algebra and its further extension to equational logic in his thesis, the closest new works are Kahl's [19]'s formalization of allegories and the development of the algebraic hierarchy lead by Spitters [24]. Capretta considered only finitary signatures and his work does not encompass signature morphisms. Spitters and his co-workers developed some very preliminary definitions of universal algebra,

because their goal is to use the notion of variety to define the algebraic hierarchy up to the construction of the reals; in particular they use Coq's typeclasses to have a cleaner representation of algebraic structures.

Future Work. We think that this development opened the path to several further work, in particular: (i) a natural step is to formalize institutions;(ii) consider algebras of binding structures as proposed by [9];(iii) introduce multi-sorted rewriting;(iv) formalize more of the mathematical theory behind universal algebra, for example Birkhoff's (quasi)-variety characterization; and(v) explore the idea of using completeness and soundness for automating the proof of identities in algebraic structures.

References

[1] Barthe, G., V. Capretta and O. Pons, *Setoids in type theory*, J. Funct. Program. **13** (2003), pp. 261–293.

[2] Birkhoff, G., *On the structure of abstract algebras*, Mathematical Proceedings of the Cambridge Philosophical Society **31** (1935), p. 433–454.

[3] Birkhoff, G. and J. D. Lipson, *Heterogeneous algebras*, J. of Combinatorial Theory **8** (1970), pp. 115–133.

[4] Bove, A., P. Dybjer and U. Norell, *A brief overview of agda - A functional language with dependent types*, in: *TPHOLs*, Lecture Notes in Computer Science **5674** (2009), pp. 73–78.

[5] Burstall, R. M., *Proving Properties of Programs by Structural Induction*, The Computer Journal **12** (1969), pp. 41–48.

[6] Capretta, V., *Universal algebra in type theory*, in: *International Conference on Theorem Proving in Higher Order Logics*, Springer, 1999, pp. 131–148.

[7] Danielsson, N. A. and The Agda Team, *The agda standard library, version 0.12*, https://github.com/agda/agda-stdlib (2015).

[8] Dijkstra, E. W. and C. S. Scholten, "Predicate Calculus and Program Semantics," Springer New York, New York, NY, 1990.

[9] Fiore, M. and O. Mahmoud, *Second-order algebraic theories*, in: *International Symposium on Mathematical Foundations of Computer Science*, Springer, 2010, pp. 368–380.

[10] Goguen, J. A., *Memories of ADJ*, Bulletin of the EATCS **39** (1989), pp. 96–102.

[11] Goguen, J. A. and R. M. Burstall, *Institutions: Abstract model theory for specification and programming*, J. ACM **39** (1992), pp. 95–146.

[12] Goguen, J. A. and K. Lin, *Specifying, programming and verifying with equational logic.*, in: *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two* (2005), pp. 1–38.

[13] Goguen, J. A. and J. Meseguer, *Completeness of many-sorted equational logic*, SIGPLAN Notices **17** (1982), pp. 9–17.

[14] Goguen, J. A. and J. Meseguer, *Remarks on remarks on many-sorted equational logic*, SIGPLAN Notices **22** (1987), pp. 41–48.

[15] Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright, *Abstract data types as initial algebras and the correctness of data representations*, in: *Conference on Computer Graphics, Pattern Recognition, & Data Structure, UCLA* (1975), pp. 89–93.

[16] Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright, *Initial algebra semantics and continuous algebras*, J. ACM **24** (1977), pp. 68–95.

[17] Gonthier, G., A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi and L. Théry, *A machine-checked proof of the odd order theorem*, in: *ITP*, Lecture Notes in Computer Science **7998** (2013), pp. 163–179.

[18] Huet, G. and D. C. Oppen, *Equations and rewrite rules: a survey*, Technical Report STAN//CS-TR-80-785, Stanford University, Department of Computer Science (1980).

[19] Kahl, W., *Dependently-typed formalisation of relation-algebraic abstractions*, in: *RAMICS*, Lecture Notes in Computer Science **6663** (2011), pp. 230–247.

[20] Meinke, K. and J. V. Tucker, *Universal algebra*, in: S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 1)*, Oxford University Press, Inc., New York, NY, USA, 1992 pp. 189–368.

[21] Rocha, C. and J. Meseguer, *Theorem proving modulo based on boolean equational procedures*, in: *RelMiCS*, Lecture Notes in Computer Science **4988** (2008), pp. 337–351.

[22] Salvesen, A. and J. M. Smith, *The strength of the subset type in Martin-Löf's type theory*, in: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988* (1988), pp. 384–391.

[23] Sannella, D. and A. Tarlecki, "Foundations of algebraic specification and formal software development," Springer Science & Business Media, 2012.

[24] Spitters, B. and E. van der Weegen, *Type classes for mathematics in type theory*, Mathematical Structures in Computer Science **21** (2011), pp. 795–825.

[25] Tarlecki, A., *Some nuances of many-sorted universal algebra: A review*, Bulletin of the EATCS **104** (2011), pp. 89–111.

[26] Thatcher, J. W., E. G. Wagner and J. B. Wright, *More on advice on structuring compilers and proving them correct*, Theoretical Computer Science **15** (1981), pp. 223–249.

[27] Vidal, J. C. and J. S. Tur, *On the completeness theorem of many-sorted equational logic and the equivalence between Hall Algebras and Bénabou theories*, Reports on Mathematical Logic **40** (2006), pp. 127–158.

Formal Meta-level Analysis Framework for Quantum Programming Languages

Mohamed Yousri Mahmoud[1] Amy P. Felty[2]

School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada

Abstract

The design and development of quantum programming languages (QPLs) is an important and active area of quantum computing. This paper addresses the problem of developing a standard methodology for verifying a QPL against major quantum computing concepts. We propose a framework dedicated to the meta-level analysis of QPLs, in particular, functional quantum languages. To this aim, we choose the Hybrid system as the tool in which to build our framework. Hybrid is a logical framework that supports higher-order abstract syntax, on top of which we develop an intuitionistic linear specification logic used for reasoning about QPLs. We provide a formal proof of some important meta-theoretic properties of this logic, and in addition, showcase a number of examples that can be tackled under the proposed framework.

Keywords: Quantum Lambda Calculus, Linear Logic, Hybrid, Coq

1 Introduction

Quantum computing has the potential to radically change the way computing is done. The existence of large scale quantum machines would increase computational power exponentially and provide unbreakable security systems [2]. Quantum programming languages (QPLs) are an integral element required for achieving successful quantum machines, as they introduce quantum concepts at a high-level, allowing better understanding of quantum aspects and increasing access to research in quantum domains. QPLs were initiated by Knill [6] who provided a number of conventions to express quantum algorithms (i.e., quantum pseudo-code). The development of QPLs has evolved for both imperative languages, e.g., [10] and [12], and functional languages, e.g., [15] and [11]. QPLs are based on the QRAM computation model: the quantum state or data is stored in a RAM and the program is a sequence of primitive quantum operations (or controls) that update such a global quantum state [6]. There is a trade-off between offering a high-level quantum language and capturing the full quantum aspects, e.g., no-cloning and superposition

[1] Email: myousri@uottawa.ca
[2] Email: afelty@uottawa.ca

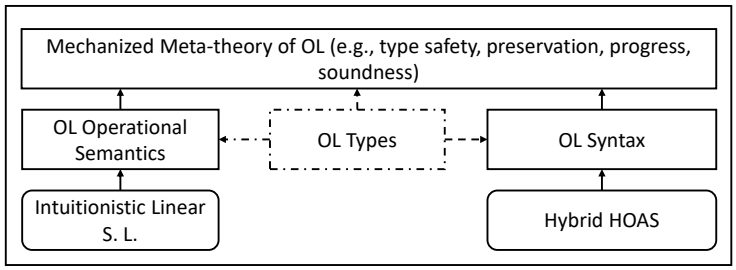


Fig. 1. Formal Meta-level Analysis Framework QPLs

properties. Accordingly, it is very crucial, when developing a QPL, to ensure that the proposed language, the computational model, and the operational semantics (as well as the type system if any) is practical, in terms of how the language respects the quantum properties and how it deals with *quantum non-determinism* (i.e., a quantum storage unit can hold both the zero and one values at the same time) and the *measurement process* (i.e., determining the exact value of a storage unit based on some probabilities). In this paper, we introduce a formal framework that aims to standardize meta-level analysis of quantum languages. Such a framework can help a language designer to focus on enhancing the language specifications itself, leaving many of the details of correctness checking to the proposed framework. As a result, it becomes easier to make changes to the language as it evolves and study the effect of these changes, updating only the parts of proofs that are affected at each step.

Our concern in this paper is with functional quantum programming languages which typically map to quantum lambda calculi [13]. The major difference that a quantum calculus provides with respect to ordinary lambda calculus is that it addresses resource limitations: a quantum variable (i.e., quantum bit) is not duplicable (i.e., no cloning) and should be consumed only once and cannot be erased. This makes linear logic a good candidate for modeling the operational semantics for both typed and untyped quantum lambda calculi. To avoid involving the user in low-level details of language formalization, e.g., variable binding and substitution, we opt to use the Hybrid system [5], a two-level *logical framework* that supports higher-order abstract syntax (HOAS), implemented in both the Coq and Isabelle/HOL proof assistants. Figure 1 illustrates the design of our proposed framework for meta-level analysis of QPLs. OL stands for *object language*, which is the programming language subject to analysis. OL syntax contains the encoding of all possible expressions of a language (which often includes expressions that are not correct with respect to a well-formedness or other kind of judgment such as typing or evaluation). The use of Hybrid involves defining a *specification logic* (SL). An SL is developed independently from any OL, but is customizable through a parameter for atomic predicates used to express OL judgments, e.g., typing and evaluation rules. An SL is generally the formalization of a sequent calculus. We choose to implement intuitionistic linear logic which is well-suited to modeling the QRAM computation model that allows both intuitionistic resources (i.e., controls) and linear resources (i.e., quantum data).

The middle block defines the syntax and basic properties of the types supported by an OL. Of course, this block is not included in the case of untyped quantum lambda calculi. QPL designers often choose to define untyped calculi to capture

the maximum amount of quantum features (i.e., to build a quantum Turing complete language), which could be sacrificed by adding certain type systems. Nevertheless, the proposed framework supports both Turing complete and incomplete QPLs. The formalized syntax and semantics are then used to reason about the OL, e.g., proving subject reduction (type soundness) or Turing completeness.

The proposed framework will be implemented in both Coq and Isabelle/HOL. This paper presents completed work on the Coq version [8], which builds on earlier (unpublished) work [7] where we formalize the Proto-Quipper language [11] along with some of its meta-theory. Here, we generalize the ideas from that case study to develop a general framework, focusing on two main ideas: generalizing the SL and providing a more complete and general set of meta-level properties that can be reused by many OLs, and illustrating its use on two different QPLs, namely Proto-Quipper and Q [15]. Although the work on Q is in its early stages, it illustrates the general nature of the framework. The rest of the paper discusses each box of Fig. 1 in more detail.

2 Encoding OL Syntax in Hybrid

The first file in the Coq library implementing Hybrid (bottom right of Fig. 1) introduces a special type `expr`, defined as an inductive type, that encodes a de Bruijn representation of λ-terms. The inductive definition has one parameter `con` which is filled in when defining the constants used to represent an OL. The two constructors for `expr` that we will see in this paper are `CON` of type `con -> expr` and `APP` of type `expr -> expr -> expr`, which encode constants and application, respectively, of de Bruijn terms. The library then includes a series of definitions used to define the operator `lambda` of type `(expr -> expr) -> expr`, which provides the capability to express OL syntax using HOAS. Expanding its definition fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user when reasoning about meta-theory. One other predicate from the Hybrid library that will appear later is `abstr`, which is applied to arguments of `lambda` and rules out functions of type (`expr` $\rightarrow$ `expr`) that do not adequately encode object-level syntax. (See [5] for its definition and a discussion of *adequacy* of HOAS encodings.)

We now give two examples filling in the middle right box in Fig. 1.

Example 1. The following is a segment of the context-free grammar of Proto-Quipper [11], a typed QPL:

$$a \ ::= \ x \mid q \mid \langle a_1, a_2 \rangle \mid \lambda x.a \mid \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 \mid \texttt{let}\ \langle x, y \rangle = a_1\ \texttt{in}\ a_2$$

where x is a term variable, q is a quantum variable, and $\langle a_1, a_2 \rangle$ is *tensor product* of two Proto-Quipper expressions. The *let-statement* has the constraint that the variables x and y should be available at the same time. Typically, x and y evaluate to quantum bits which are linear resources. For this object language, the type `con` is instantiated with the type `Econ`, implemented as:

```
Inductive Econ: Set :=
 Qvar: nat -> Econ| qPROD: Econ | qABS: Econ  | qIF: Econ | qLET: Econ.
```

We do not represent term variables explicitly since they appear as bound variables in the HOAS representation of OL terms. Using these constants, we then define the OL operators. For example:

```
Definition Prod: qexp -> qexp -> qexp :=
 fun e1:qexp => fun e2:qexp => (APP (APP (CON qPROD) e1) e2).
Definition Let:  (qexp -> qexp-> qexp) -> qexp -> qexp :=
 fun f:qexp->qexp->qexp => fun e1:qexp =>
 (APP (CON qLET) (APP (lambda
  (fun x => (lambda (fun y => f x y)))) e1)).
```

The type `qexp` is the type `expr` discussed above with the parameter `con` instantiated with `Econ` for this OL. Note the use of `lambda` in defining an HOAS encoding of the *let statement*. With this definition in place, the de Bruijn representation is hidden, and does not appear in further proof development.

Example 2. The following is an example of a segment of the context-free grammar of Q [15], an untyped QPL:

$$a \ ::= \ x \mid r \mid \langle a_1, a_2, ..., a_n \rangle \mid \lambda!x.a \mid \lambda x.a$$

where r refers to quantum variables, $\langle a_1, a_2, ..., a_n \rangle$ is a *tensor product* of n Q expressions (or a linear pattern). A major difference between Q and Proto-Quipper is that Q allows two types of lambda abstraction: over linear variables where the bound variable appears in the function body only once, and over duplicable variables where bound variables may appear zero or more times. Proto-Quipper distinguishes these two kinds of abstraction using its type system (which we do not present here, see [8]). Accordingly, the definition of `con` for Q will include two distinct constants for the two abstractions, namely `LABS` and `IABS`. It also has constants `qPROD` and `Qvar` as before. Linear and intuitionistic abstraction are defined using an HOAS representation as follows:

```
Definition LAbs: (qexp-> qexp) -> qexp := fun f:qexp->qexp =>
  APP (CON LABS) (lambda (fun x => f x)).
Definition IAbs:  (qexp-> qexp) -> qexp:= fun f:qexp->qexp =>
  APP (CON IABS) (lambda (fun x => f x)).
```

3 Encoding the SL and OL Inference Rules

The sequents of the intuitionistic linear logic we adopt as an SL (bottom left of Fig. 1) have the form $\Gamma; \Delta \vdash_\Pi G$, where G is a formula, Γ is an intuitionistic context of formulas, Δ is a linear context, and Γ and Δ contain only atomic formulas. The restriction to atomic formulas is sufficient for the examples we have considered so far, but it should be straightforward to extend the SL to allow more general formulas in the contexts, as was done for an intuitionistic SL in [1]. Π is a set of formulas expressing the inference rules of an OL, which we omit when presenting the sequent rules because it is fixed for each OL and does not change within a proof. They can be considered as a fixed subset of Γ. We include the connectives $\otimes$ for multiplicative conjunction, & for additive conjunction, $\multimap$ for linear implication, $\Rightarrow$

for intuitionistic implication, and $\top$ for the universal resource consumer. Below are some sample rules of this logic (where . represents an empty context):

$$\frac{}{\Gamma; A \vdash A}\ \text{l\_init} \qquad \frac{}{\Gamma, A; . \vdash A}\ \text{i\_init} \qquad \frac{}{\Gamma; \Delta \vdash \top}\ \top\text{-R}$$

$$\frac{\Gamma; \Delta_1 \vdash B \qquad \Gamma; \Delta_2 \vdash C}{\Gamma; \Delta_1, \Delta_2 \vdash B \otimes C}\ \otimes\text{-R} \qquad \frac{\Gamma; \Delta \vdash B \qquad \Gamma; \Delta \vdash C}{\Gamma; \Delta \vdash B \& C}\ \&\text{-R}$$

There are two initial rules, the linear rule (l_init) strictly prohibits the existence of any hypothesis inside Δ except A, and does not care about the contents of Γ. The intuitionistic rule (i_init) strictly requires an empty linear context, whereas A should be in the intuitionistic context. We can use $\otimes$ if its operands can be proven linearly at the same time, i.e., they do not share linear resources. On the other hand, additive conjunction is used when the operands are sharing the linear resources. Because they both consume all resources, only one of them can be made available at a time. Missing from this set are rules for each of the implication connectives ($\Rightarrow$-R and $\multimap$-R), the standard introduction rule for universal quantification ($\forall$-R), and a form of a backchain rule which encapsulates the L rules, and is standard for two-level systems (see [5]).

There are a number of formalizations of linear logic, e.g., [3] in Abella and [9] in Coq. The main purpose of these formalizations is handling the meta-analysis of different fragments of linear logic, not using them as intermediate logics in which to study object languages such as QPLs. Our objective is broader since it includes both. Our formalization is inspired by the work in [5], which implements ordered intuitionistic linear logic as a specification logic in the Isabelle/HOL version of Hybrid, where it is used to study a continuation-machine presentation of the operational semantics of a functional language that is much simpler than the QPLs considered here.

The SL sequent rules are formalized as an inductive predicate `seq: list atm -> list atm -> oo -> Prop`, where the type `atm` is the parameter for atomic predicates, implemented for each OL. Formulas of the sequent calculus are encoded as an inductive type `oo` using an HOAS representation where `Conj` represents $\otimes$, `And` represents &, and `atom` coerces atomic formulas (type `atm`) to formulas (type `oo`). The first list of atoms refers to the intuitionistic context, and whereas the second list contains linear atoms. We also define an inductive predicate `prog: atm -> list oo -> list oo -> Prop` that contains the encoding of inference rules of an OL (such as operational semantics). This predicate is used by the clause of the `seq` definition that encodes the backchaining rule. The formula `(prog A IL L)` reads as follows: an atom `A` (the conclusion of an inference rule of the OL) is provable if we can prove each member of the list of intuitionistic subgoals `IL` and of the linear subgoals `L` (together representing the premises of an OL rule). The definition of `seq` does not include the structural rules because they are admissible in our encoding of the SL. The reader is referred to [8] for full versions of omitted definitions.

The implementation of the SL is significantly generalized from that in [7], allowing a more complete validation in the form of more general metatheoretic properties that can be applied to any OL. For example, we have proved the admissible structural properties and cut-elimination rules for both intuitionistic and linear contexts.

We show only the statements of the cut-elimination rules. The first is:

```
Lemma seq_cut_aux: forall (a:atm) (b:oo) (IL L:list atm),
  seq IL L b -> seq (remove eq_dec a IL) [] (atom a) ->
  seq (remove eq_dec a IL) L b.
```

The theorem states that if we remove all instances of the hypothesis `a` from the list of intuitionistic hypotheses `IL`, and `a` is found to be provable under the new list of hypotheses, then eliminating `a` does not affect the provability of `b`. Note that the `remove` operator requires an equality operator on atoms as an argument. On the other hand, the linear version of the cut elimination rule allows the removal of only one instance of the linear resource `a`:

```
Lemma seq_cut_linear: forall (a:atm) (b:oo) (IL L L':list atm),
  seq IL L b -> seq IL L' (atom a) ->
   seq IL (L' ++ remove_one eq_dec a L) b).
```

The following two examples illustrate the middle left box in Fig. 1.

Example 3. Continuing Example 1, the following is an example of a Proto-Quipper typing rule:

$$\frac{\Gamma; \Delta_1 \vdash a : A \qquad \Gamma; \Delta_2 \vdash b : B}{\Gamma; \Delta_1, \Delta_2 \vdash \langle a, b \rangle : (A \otimes B)} \otimes_i$$

For the Proto-Quipper expressions a and b, whose types are A and B, respectively, and linearly provable and do not share linear resources, their product $\langle a, b \rangle$ is of type $(A \otimes B)$ ($\otimes$ overloaded here). This rule is encoded as part of the `prog` clauses dedicated to the Proto-Quipper language as follows:

```
| ttensorl: forall (a b:qexp) (A B: qtp),
  prog (typeof (Prod a b) (tensor A B)) []
  [Conj (atom (typeof a A)) (atom (typeof b B))]
```

Here, the type `qtp` encodes Proto-Quipper types (whose definition was omitted from Example 1), `typeof` is an `atm` constructor that associates an expression with its type, `Conj` encodes $\otimes$ as multiplicative conjunction of the SL, `tensor` encodes $\otimes$ as an OL type, and `Prod` encodes OL products inhabiting that type. This small example justifies the use of linear logic.

Example 4. We choose to illustrate the Q language by considering well-formedness of terms of the form $\lambda x.a$ and $\lambda !x.a$ from Example 2. Let Γ and Δ be contexts of intuitionistic and linear term variables, respectively. The term $\lambda x.a$ (respectively $\lambda !x.a$) is well-formed in $\Gamma; \Delta$ if a is well-formed in $\Gamma, x; \Delta$ (respectively $\Gamma; \Delta, x$). The `prog` clauses for these lambda expressions are as follows:

```
| lambda1: forall (M:qexp -> qexp), abstr M ->
  prog (is_qexp (LAbs M)) [] [(All (fun x : qexp => lImp (is_qexp x)
  (atom_ (is_qexp (M x)))))]
| lambda2: forall (M:qexp -> qexp), abstr M ->
  prog (is_qexp (IAbs M)) [] [(All (fun x : qexp => Imp (is_qexp x)
  (atom_ (is_qexp (M x)))))]
```

Here, `is_qexp` is one of the constructors of `atm` and `All`, `lImp`, and `Imp` are con-

structors of `oo` encoding $\forall$, $\multimap$ and $\Rightarrow$ of the SL. Note the difference between the two rules: linear implication is used to define the linear lambda abstraction (which emphasis that the function body contains one copy of `x`), whereas intuitionistic implication is used to define intuitionistic lambda abstraction that allows multiple copies of `x` or even zero occurrences.

Now, we conclude our formalization by showing an example meta-theorem, namely subject reduction. The following theorem will prove the soundness of the Proto-Quipper language based on the definitions provided earlier:

```
Theorem subject_reduction:forall IL LL C C' a a',
  seq IL [] (atom_ (reduct C a C' a')) ->
  seq IL LL (atom_ (typeof a A)) ->
  seq IL LL (atom_ (typeof a' A)).
```

The above theorem states that if an expression `a`produced by a quantum circuite `C` reduces to an expression `a'` produced by a quantum circuite `C'`, then the reduced expression maintains the same type as the original expression. For an untyped programming language, the above theorem will be slightly different, where `typeof` atom will be replaced by `is_qexp`.

4 Conclusion

We have proposed a meta-level analysis framework for functional QPLs implemented using the Hybrid system with a linear specification logic. The framework provides a standard way to tackle common components and concepts of QPLs, e.g., operational semantics and type safety. Formalization examples of the *Proto-Quipper* and *Q* languages have been presented to show the practical potential of the proposed system. We plan a more complete formalization of these languages in addition to others, e.g., [4] and [14].

References

[1] Battell, C. and A. Felty, *The logic of hereditary harrop formulas as a specification logic for hybrid*, in: *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice* (2016), pp. 3:1–3:10.

[2] Brassard, G., C. Crepeau, R. Jozsa and D. Langlois, *A quantum bit commitment scheme provably unbreakable by both parties*, in: *IEEE Annual Symposium on Foundations of Computer Science*, 1993, pp. 362–371.

[3] Chaudhuri, K., L. Lima and G. Reis, *Formalized meta-theory of sequent calculi for substructural logics*, in: *Workshop on Logical and Semantic Frameworks with Applications*, ENTCS **332** (2017), pp. 57 – 73.

[4] Díaz-Caro, A. and G. Dowek, *Simply typed lambda-calculus modulo type isomorphisms*, CoRR **arXiv/1501.06125** (2014).

[5] Felty, A. P. and A. Momigliano, *Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax*, Journal of Automated Reasoning **48** (2012), pp. 43–105.

[6] Knill, E., *Conventions for quantum pseudocode*, Technical Report LAUR-96-2724, Los Alamos National Laboratory (1996), https://www.osti.gov/scitech/servlets/purl/366453, accessed 2017-06-19.

[7] Mahmoud, M. Y. and A. P. Felty, *Formalization of metatheory of the Quipper quantum programming language in a linear logic* (2017), http://www.site.uottawa.ca/~afelty/dist/HybridProtoQuipper17.pdf.

[8] Mahmoud, M. Y. and A. P. Felty, *Quantum programming language Coq scripts* (2017), https://bitbucket.org/snippets/myousri/Gj7qX.

[9] Olivier Laurent, *YALLA : an LL library for Coq*, https://perso.ens-lyon.fr/olivier.laurent/yalla/ (2017).

[10] Ömer, B., "A Procedural Formalism for Quantum Computing," Master's thesis, Technical University of Vienna (1998).

[11] Ross, N. J., "Algebraic and Logical Methods in Quantum Computation," Ph.D. thesis, Dalhousie University (2015), CoRR arXiv:1510.02198 [quant-ph].

[12] Sanders, J. W. and P. Zuliani, *Quantum programming*, in: *Mathematics of Program Construction*, LNCS (2000), pp. 80–99.

[13] Selinger, P. and B. Valiron, *A lambda calculus for quantum computation with classial control*, Mathematical Structures in Computer Science **16** (2006), pp. 527–552.

[14] Vizzotto, J. K., B. Calegaro and E. K. Piveta, *A double effect λ-calculus for quantum computation*, in: *Brazilian Symposium on Programming Languages*, LNCS (2013), pp. 61–74.

[15] Zorzi, M., *On quantum lambda calculi: a foundational perspective*, Mathematical Structures in Computer Science **26** (2016), pp. 1107–1195.

Mechanizing Linear Logic in Coq

Bruno Xavier and Carlos Olarte

Universidade Federal do Rio Grande do Norte (UFRN), Natal, Brazil

Giselle Reis[1]

Carnegie Mellon University, Doha, Qatar

Vivek Nigam

Centro de Informática, Universidade Federal da Paraíba João Pessoa, Brazil, and Fortiss, Munich, Germany

Abstract

Linear logic has been used as a foundation (and inspiration) for the development of programming languages, logical frameworks and models for concurrency. Linear logic's cut-elimination and the completeness of focusing are two of its fundamental properties that have been exploited in such applications. Cut-elimination guarantees that linear logic is consistent and has the so-called sub-formula property. Focusing is a discipline for proof search that was introduced to reduce the search space, but has proved to have more value, as it allows one to specify the shapes of proofs available. This paper formalizes linear logic in Coq and mechanizes the proof of cut-elimination and the completeness of focusing. Moreover, the implemented logic is used to encode an object logic, such as in a linear logical framework, and prove adequacy.

Keywords: linear logic, focusing, Coq, Abella

1 Introduction

Linear Logic was proposed by Girard [12] more than 30 years ago, but it still inspires computer scientists and proof theorists alike, being used as a foundation for programming languages, models of concurrency [18,8,19] and logical frameworks [16]. Such developments rely on two fundamental properties of linear logic: cut-elimination [11] and the completeness of focusing [1].

Cut-elimination states that any proof with instances of the cut-rule:

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}$$

[1] Giselle Reis was funded by grant NPRP 097-988-1-178 from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

can be transformed into a proof of the same formula without any instance of the cut-rule. The two main consequences of this theorem are: (1) the system's consistency, *i.e.*, it is not possible to prove both $\vdash A$ and $\vdash A^{\perp}$; and (2) all proofs satisfy the sub-formula property, *i.e.*, a proof of a formula A, contains only sub-formulas of A. *Focusing* is a proof search discipline proposed by Andreoli [1] which constraints proofs by enforcing that rules sharing some structural property, like invertibility, are grouped together. The completeness of focusing states that if a formula has a proof, then it has a focused proof.

The combination of cut-elimination and focusing allows for the construction of powerful linear logical frameworks. By relying on these two properties, proof search is considerably improved. Moreover, these properties can be used to engineer the types of proofs, thus allowing the specification/encoding of a number of different proof systems (e.g., sequent calculus, natural deduction and tableaux systems) for different logics [16,17].

This paper presents a formalization of first-order classical linear logic (LL) in Coq, including proofs for cut-elimination and completeness of focusing. Up to the best of our knowledge, this is the first formalization the meta-theory of first-order linear logic in Coq (see Section 6). Our main contributions are listed below:

(1) Quantifiers: Most of the formalizations of cut-elimination in the literature deal with propositional systems (see Section 6). The first-order quantifiers of LL are fundamental for the adequacy results in Section 5. We use the technique of Parametric HOAS [4] (*i.e.*, dependent types in Coq) to encode the LL quantifiers in the meta-logic. This alleviates the burden of specifying substitution and freshness conditions. However, it comes at a price, as substitution lemmas and structural preservation under substitutions need to be assumed as axioms (see details in Section 2).

(2) Focusing completeness: While cut-elimination theorems for a number of proof systems have been formalized, including propositional linear logic [3], this is, as far as we know, the first formalization of first-order LL's cut-elimination and LL's focused proof system completeness. The proof formalized is exactly the one presented by Andreoli [1]. It involves a number of non-trivial proof transformations.

(3) Encoding proof systems: By relying on linear logic's cut-elimination and focusing property, it is possible to encode a number of proof systems [17,16]. Focusing is used to achieve the highest level of adequacy (*i.e.*, from rules to partial derivations). This is done by engineering derivations available in the LL framework to match derivations of the encoded proof system. We build a tactic that automatically handles the whole negative phase of the proof, making proofs shorter and simpler for the specifier/programmer. The mechanized proofs are similar to the paper proofs and quite direct. We demonstrate this by encoding the system LJ for intuitionistic propositional logic into LL.

(4) Treatment of contexts: A main challenge in LL arises from the fact that contexts are treated as multisets of formulas and not as sets (due to the lack of weakening and contraction) . We show that the exchange rule is admissible in LL, i.e., if $\vdash \Gamma$ is provable in LL and Γ' is a permutation of Γ then $\vdash \Gamma'$ is provable in LL. We use the library `Morphisms` of Coq to easily substitute, in any proof, equivalent multisets. For doing that, we extended the standard library for multisets in Coq with additional theorems. Moreover, we developed tactics that handle (automatically) most of the proofs of multisets in our formalization.

(5) Induction measures: Although more relaxed measures for the height of derivations

can be used as induction measures (*e.g.*, the axiom rule can have height n for any n), we decided to follow carefully the induction measures used in the proof of cut-elimination and the completeness of focusing in, *e.g.*, [25] and [1]. Hence, our proofs reflect exactly the procedures described in the literature.

Our formalization is available at `https://github.com/meta-logic/coq-ll` and the organization of the paper follows closely the structure of the files. Section 2 deals with the syntax of LL and the snippets of code are from `SyntaxLL.v`. Section 3 deals with different sequent calculi for LL and a focused system for it (`SequentCalculi.v`). Section 4 presents several results about LL: structural properties (`StructuralRules.v`), cut-elimination (`Cut_Elim.v`) and completeness of focusing (`Completeness.v`). Section 5 shows the application of our formalization to prove correct the encoding of LJ into LL (`LJLL.v`). Section 6 discusses related and future work. We present here some of the most important definitions and key cases in the proofs of the theorems. It is important to note that, for the sake of presentation, we omit some cases (e.g., in inductive definitions) and we also change marginally the notation to improve readability. Also, in theorems' statements, all the variables are implicitly universally quantified. The reader may always consult the complete definitions and proofs in the source files.

2 Linear Logic Syntax

Linear logic (LL) [12] is a resource conscious logic, in the sense that formulas are consumed when used during proofs, unless they are marked with the exponential ? (whose dual is !), in which case, they behave *classically*. LL connectives include the additive conjunction & and disjunction $\oplus$ and their multiplicative versions $\otimes$ and $\parr$, together with their units and the first-order quantifiers:

$$\begin{array}{rcl} A, B, \ldots & ::= & \underbrace{a}_{\text{LITERAL}} \mid \underbrace{A \otimes B \mid 1 \mid A \parr B \mid \bot}_{\text{MULTIPLICATIVE}} \mid \underbrace{A \oplus B \mid 0 \mid A \& B \mid \top}_{\text{ADDITIVE}} \mid \underbrace{\forall x.A \mid \exists x.A}_{\text{QUANTIFIED}} \mid \underbrace{?A \mid !A}_{\text{EXPONENTIAL}} \\ & \mid & \underbrace{a^{\perp}}_{\text{LITERAL}} \end{array} \tag{1}$$

The main issue when formalizing first-order logics in proof assistants is how to encode quantifiers. At first glance, one might consider the following naive signature for the constructors `fx` and `ex` of universally and existentially quantified formulas, respectively:

```
| fx : (var → formula) → formula        | ex : (var → formula) → formula
```

In order to define substitutions of variables for terms on such formulas, it is necessary to define a `term` type as a union of, *e.g.*, `var`s and `function`s; and also to implement substitution from scratch. This means dealing with variable capture and equality of terms.

It is possible to avoid this unnecessary bureaucracy if substitution is handled by the meta-level β-reduction. This means that a quantified formula $Qx.F$ ($Q \in \{\forall, \exists\}$) is represented as $Q(\lambda x.F)$, where λ is a meta-level binder. In this case, we have:

```
| fx : (term → formula) → formula        | ex : (term → formula) → formula
```

This approach is called *higher-order abstract syntax* (HOAS) or λ-tree syntax [21,15]. In a functional framework, the type `(term → formula)` ranges over all functions of this type. This is not desirable as it allows functions, called *exotic terms* [6], to pattern match on the

term and return a structurally (or logically) different formula for each case. Note that, in a relational framework, this is not a problem, since all definitions must have the type of propositions on the meta-level (different from the type of formulas of the object logic).

A solution for this problem is either to require that `term` is an uninhabited type or to quantifying over all types as below:

```
| fx : forall T, (T → formula) → formula        | ex : forall T, (T → formula) → formula
```

However, in both specifications, it is impossible to write a function that computes the size of a formula (a good description of these problems can be found at http://adam.chlipala.net/cpdt/html/Hoas.html). A solution proposed in [4] consists of parametrizing the type `T` for quantified variables not in quantifiers' constructors, but outside the whole specification. This approach is called *parametric* HOAS. Using this technique, linear logic's syntax is formalized as follows.

```
Section Sec_lExp.
  Variable T: Type.              (* Parameter for building variables *)
  Inductive term :=              (* Terms *)
   | var (t: T)  | cte (e: A)    (* variables and constants *)
   | fc1 (n: nat) (t: term)  | fc2 (n: nat) (t1 t2: term). (* family of functions of 1/2 argument *)
  Inductive aprop :=                (* Atomic propositions *)
   | a0: nat → aprop                (* 0-ary predicates *)
   | a1: nat → term → aprop | a2: nat → term → term → aprop. (* family of 1/2-ary predicates *)
  Inductive lexp := (* Formulas *)
   | atom (a : aprop) ) | perp (a: aprop)   (* positive/negated atoms *)
   | top | bot | zero | one                 (* units *)
   | tensor (F G: lexp) | par (F G: lexp)  (* multiplicative *)
   | plus (F G: lexp) | witH (F G: lexp)   (* additive *)
   | bang (F: lexp) | quest (F: lexp)      (* exponentials *)
   | ex (f: T → lexp) | fx (f: T → lexp).  (* quantifiers *)
End Sec_lExp.
```

The type `A` of constant terms is a global parameter. The signature of first-order terms includes a family of functions `fc1` and `fc2` of one and two arguments respectively. In each case, the first parameter (`nat`) is the identifier, *i.e.*, the name of the function. Atomic propositions can be 0-ary (`a0`), unary (`a1`) or binary (`a2`) predicates. As in the case of functions, a natural number acts as the identifier. The rest of the code should be self-explanatory. We note that more general constructors for functions and predicate using a list (of arbitrary length) of parameters could have been defined. However, the current signature is general enough for our encodings and it greatly simplifies the notation.

All types defined in `Section Sec_lExp` are parametrized by the type `T`. Therefore, the type of `top`, for example, is not `lexp`, but `forall T: Type, lexp T`. Hence, for any type `T`, the expression `top T` is of type `lexp T` (*e.g.*, `top nat: lexp nat`). Clearly, the definition of linear logic formulas must be independent of `T`. In particular, it should not allow pattern matching on terms of type `T`. Therefore, the type `Lexp` of linear logic formulas is defined over all types `T` (*i.e.*, it is a dependent type). The same holds for atoms and terms:

```
Definition Term := forall T: Type, term T.    (* type for terms *)
Definition AProp := forall T: Type, aprop T.  (* type for atomic propositions *)
Definition Lexp := forall T: Type, lexp T.    (* Type for formulas *)
```

Note that `top nat` is *not* of type `Lexp` and connectives must be functions on `T`, e.g.,

```
Definition Top: Lexp := fun T ⇒ top. (* formula ⊤*)
Definition Atom (P: AProp): Lexp := fun T ⇒ atom (P T). (* building atomic propositions *)
Definition Tensor (F G: Lexp): Lexp := fun T ⇒ tensor (F T) (G T).  (* formula F ⊗ G *)
```

Since T is never destructed, all its occurrences in the above code can be replaced by "_" (meaning "*irrelevant*"). This means that an arbitrary type T is passed as a parameter, and it does not interfere with the structure of formulas.

Some of the forthcoming proofs proceed by structural induction on a LL formula F: Lexp. However, since Lexp is a (polymorphic) function, not an inductive type, Coq's usual destruct, induction or inversion tactics do not work. Following [4], the solution is to define inductively the notion of *closed formulas* as follows:

```
Inductive ClosedT: Term → Prop :=
  | cl_cte: forall C, ClosedT (Cte C)
  | cl_fc1: forall n t1, ClosedT t1 → ClosedT (FC1 n t1)
  | cl_fc2: forall n t1 t2, ClosedT t1 → ClosedT t2 → ClosedT (FC2 n t1 t2).

Inductive ClosedA : AProp → Prop :=
  | cl_a0: forall n, ClosedA (A0 n)
  | cl_a1: forall n t, ClosedT t → ClosedA (fun _ ⇒ a1 n (t _)).
  | cl_a2: forall n t t', ClosedT t → ClosedT t' → ClosedA (fun _ ⇒ a2 n (t _) (t' _)).

Inductive Closed : Lexp → Prop :=
  | cl_atom: forall A, ClosedA A → Closed (Atom A )
  | cl_perp: forall A, ClosedA A → Closed (Perp A )
  | cl_one: Closed One
  | cl_tensor: forall F G, Closed F → Closed G → Closed (Tensor F G)
  | cl_fx: forall FX, Closed (Fx FX)
  [...]
```

Such definitions rule out the occurrences of the *open* term var x in the type Term and, consequently, in atomic propositions and formulas. Now we need the axioms that only *closed* structures can be built.

```
Axiom ax_closedT: forall X: Term, ClosedT X.
Axiom ax_closedA: forall A: AProp, ClosedA A.
Axiom ax_closed : forall F: Lexp, Closed F.
```

The statements above cannot be proved in Coq, mainly because it is not possible to induct on function types (such as Term). This would require, *e.g.*, a meta-model of the Calculus of Inductive Constructions [2] itself inside Coq. Let us give some intuition of why those axioms are consistent with the theory of Coq. This will also clarify the closeness condition we impose on Lexp. If 1 is the identifier for the predicate P and c is a constant of type A, the atomic proposition $P(c)$ can be defined as `Definition Pc: Lexp := fun T: Type ⇒ atom (a1 1 (cte c)).` However, the same exercise does not work for $P(x)$ when x is a free variable. If we were to write `Definition Px: Lexp := fun T: Type ⇒ atom (a1 1 (var ??)).`, then "??" must be an inhabitant of T. Since we do not know anything about T, we cannot name an element of it and Px will never type check. Hence, both terms and formulas are necessarily closed (without free variable).

Another consequence of the functional representation of terms is that we require the axiom of Functional Extensionality of the standard library of Coq to check whether two terms/formulas are the same:

```
Axiom functional_extensionality_dep : forall {A} {B : A → Type},
  forall (f g :  forall x : A,  B x),  (forall x, f x = g x) → f = g.
```

Roughly, given two function f and g, we conclude $f = g$ whenever $f(x) = g(x)$ for all x.

Substitutions. According to the definition of quantifiers, substitutions should be performed on formulas of type T → lexp, where T is a parameter. Following the same idea as before (and as in [4]), we define a type Subs for such formulas as a (closed) dependent type:

`Definition Subs := forall T: Type, T → lexp T.` By quantifying over all `T`s, we prevent functions that destruct the term and change the structure of the formula. Now we need to define a wrapper for substitutions. This substitution function will take as parameters `S: Subs` and `X: Term`, and return a `Lexp`. The first step is to apply Coq's β-reduction to `S`, the type of `X` (instantiating the `forall`) and `X` (first argument). The result of this reduction is of type `lexp (term T)`, which is different from `Lexp`, defined as `forall T: lexp T`. A `lexp T` is constructed by the function `flatten: lexp (term T) → lexp T`, which is defined in a section parametrized by `T`.

```
Definition Subst (S: Subs) (X: Term) : Lexp := fun T: Type ⇒ flatten (S (term T) (X T)).
```

As an example, the steps below apply $\lambda x.\top \otimes Q(f(x), c)$, to the constant d. The resulting inhabitant of `Lexp` is commented out.

```
Definition S: Subs := fun (T: Type) (x: T) ⇒ tensor top (atom (a2 1 (fc1 1 (var x)) (cte c))).
Definition t1: Term := fun T ⇒ (cte d).
Eval compute in Subst S t1. (* fun T:Type ⇒ tensor one (atom (a2 4 (fc1 1 (cte d)) (cte c))) *)
```

Formula complexity. Even if it is now possible to reason on the structure of the formula via the `Closed` definition, some of our proofs proceed by induction on the weight of a formula. Our definition follows the standard one (*i.e.*, $W(\top) = 0, W(F \otimes G) = 1 + W(F) + W(G)$, etc). It should be the case that $W(F[t/x]) = W(F[t'/x])$ for any two terms t and t'. This is true since substitutions cannot perform "case analysis" on the term t to return a different formula. This simple fact requires extra work in Coq. First, we define when two formulas are equivalent modulo renaming of bound variables (some cases are omitted):

```
Inductive xVariantT: term T → term T'→ Prop :=
  | xvt_var: forall x y, xVariantT (var x) (var y)
  | xvt_cte: forall c, xVariantT (cte c) (cte c)
  [...]
Inductive xVariantA: aprop T → aprop T'→ Prop :=
  | xva_eq: forall n, xVariantA (a0 n) (a0 n)
  | xva_a1: forall n t t', xVariantT t t' → xVariantA (a1 n t) (a1 n t')
  [...]
Inductive EqualUptoAtoms: lexp T → lexp T' → Prop :=
  | eq_atom: forall A A', xVariantA A A' → EqualUptoAtoms (atom A) (atom A')
  | eq_ex: forall FX FX', (forall t t', EqualUptoAtoms (FX t) (FX' t')) → EqualUptoAtoms (ex (FX )) (ex (FX'))
  [...]
```

For similar arguments as the ones given above for *closeness*, we need to add as axioms that inhabitants of `Subs` and `Lexp` cannot make choices based on its arguments:

```
Axiom ax_subs_uptoAtoms: forall (T T': Type) (t: T) (t': T') (FX: Subs), EqualUptoAtoms (FX T t) (FX T' t').
Axiom ax_lexp_uptoAtoms: forall (T T': Type) (F: Lexp), EqualUptoAtoms (F T) (F T').
```

The needed results for the weight function (defined as `Exp_weight`) can thus be proved:

```
Theorem subs_weight: forall (FX: Subs) x y, Exp_weight(Subst FX x) = Exp_weight(Subst FX y).
```

We also formalized propositional linear logic, including all the theorems in this section as well as those in Sections 3 and 4 (see https://github.com/meta-logic/coq-ll). In the propositional case, the type of formulas is a standard inductive type and there is no need for parametric (polymorphic) definitions. Hence, none of the axioms above were needed. In the first-order case, removing those axioms will amount to, e.g., formalize in a finer level the binders and substitutions. This is definitely not an easy task (see e.g., [9,10]) and we would not get for free all the benefits inherited from the PHOAS approach (e.g., substitutions are, by definition, capture avoiding).

$$\frac{}{\vdash a^\perp, a}\ I \qquad \frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{}{\vdash 1}\ 1 \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \bot \qquad \frac{\vdash \Gamma, A[x/e]}{\vdash \Gamma, \forall x.F}\ \forall \qquad \frac{\vdash \Gamma, A[x/t]}{\vdash \Gamma, \exists x.F}\ \forall_d$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\ \& \qquad \frac{}{\vdash \Gamma, \top}\ \top \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \oplus_2 \qquad \frac{\vdash ?A_1, \ldots, ?A_n, A}{\vdash ?A_1, \ldots, ?A_n, !\,A}\ !$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ ? \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ W \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ C$$

Fig. 1. Linear logic introduction rules: e is fresh, *i.e.*, does not appear in Γ.

3 Sequent Calculi

The proof system for one-sided (classical) first-order linear logic is depicted in Figure 1. A sequent has the form $\vdash \Gamma$ where Γ is a multiset of formulas (*i.e.*, exchange is implicit). While this system is the one normally used in the literature, LL's focused proof system is equipped with some more structure. As shown by Andreoli [1], it is possible to incorporate contraction (C) and weakening (W) into the introduction rules. The key observation is that formulas of the form $?F$ can be contracted and weakened. This means that such formulas can be treated as in classical logic, while the remaining formulas are treated linearly. This is reflected into the syntax in the so called *dyadic sequents* which have two contexts:

$$\frac{\vdash \Theta, F : \Gamma}{\vdash \Theta : \Gamma, ?F}\ ? \qquad \frac{\vdash \Theta, F : \Gamma, F}{\vdash \Theta, F : \Gamma}\ copy \qquad \frac{}{\vdash \Theta : a^\perp, a}\ I \qquad \frac{\vdash \Theta : \Gamma_1, A \quad \vdash \Theta : \Gamma_2, B}{\vdash \Theta : \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{\vdash \Theta : \Gamma, A \quad \vdash \Theta : \Gamma, B}{\vdash \Theta : \Gamma, A \,\&\, B}\ \&$$

Here Θ is a set of formulas and Γ a multiset of formulas. The sequent $\vdash \Theta : \Gamma$ is interpreted as the linear logic sequent $\vdash ?\Theta, \Gamma$ where $?\Theta = \{?A \mid A \in \Theta\}$. It is then possible to define a proof system for LL without explicit weakening (implicit in rule I above) and contraction (implicit in e.g., *copy* and $\otimes$ above). Notice that only the linear context Γ is split among the premises in the $\otimes$ rule. The complete proof system can be found in [1].

Rules of the dyadic system are specified as inductively defined predicates. The following code is an excerpt of the definition of the dyadic system (called `sig2` in our files):

```
Inductive sig2: list Lexp → list Lexp → Prop :=
 | sig2_init : forall B L A, L =mul= (A ⁺) :: [A ⁻] → ⊢ B ; L
 | sig2_bang: forall B F L , L =mul= [! F] → ⊢ B ; [F] → ⊢ B ; L
 | sig2_ex :  forall B L FX M t, L =mul= E{FX} ::  M → ⊢ B ; (Subst FX t) ::  M → ⊢  B ; L
 | sig2_fx :  forall B L FX M,   L =mul= (F{FX}) ::  M → (forall x, ⊢ B ; [Subst FX x] ++  M) → ⊢ B ; L
 [...]
```

Given an atomic proposition `A:Aprop`, `A⁺` and `A⁻` stand, respectively, for `Atom(A)` (A) and `Perp(A)` ($A^\perp$). Given a substitution `FX:Subs`, the LL quantifiers are represented as `E{FX}` and `F{FX}`. The rule for the universal quantifier relies on Coq's (dependent type constructor) `forall` that takes care of generating a fresh variable. Finally, ++ stands for concatenation of lists.

Given two multisets of formulas `M` and `N`, `M=mul=N` denotes that `M` is multiset equivalent to `N`. Coq's library `Coq.Sets.Multiset` defines multisets as bags (of type `A → nat`), thus specifying the number of occurrences for each element of a given type A. Reasoning about such bags is hard and automation becomes trickier. Using lists as representation of multisets seems to be a better (and cleaner) choice. In fact, the library CoLoR (`http://color.inria.fr/`) follows that direction. CoLoR is quite general and formalizes, among several other theorems, properties about data structures such as relations, finite sets, vectors, etc. CoLoR transforms the lists M and N to `Coq.Sets.Multiset` and uses the multiplicity definition of Coq in order to define multiset equivalence. Our `Multisets` module does not use this transforma-

tion. Instead, it uses directly `Coq.Lists` that features mechanisms to count the number of occurrences of a given element. We also added some useful theorems for our formalization and implemented automatic techniques (e.g., `solve_permutation`) that discharge most of the proofs about multisets we needed in our developments.

Inductive Measures. In our proofs, we usually require measures for the height of the derivation as well as for the number (and complexity) of the cuts used. For this reason, we also specified other variants of the sequent rules where such measures are explicit. For instance, the proof of cut-elimination was performed on the following system:

```
Inductive sig3: nat → nat → list lexp → list lexp → Prop :=
  | sig3_init : forall (B L: list lexp) A, L =mul= (A ⁺) :: [A ⁻] → 0 ⊢ 0 ; B ; L
  | sig3_CUT : forall (B L: list lexp) n c w h, sig3_cut_general w h n c B L → S n ⊢ S c ; B ; L
  [...]
with sig3_cut_general : nat → nat → nat → nat → list lexp → list lexp → Prop :=
 | sig3_cut : forall ... L =mul= (M ++ N) → m ⊢ c1;B ; F :: M → n ⊢ c2 ; B ; F :: N → sig3_cut_general ...
 | sig3_ccut : forall ... L =mul= (M ++ N) → m ⊢ c1;B ; (! F) :: M → n ⊢ c2 ; F°:: B ; N → sig3_cut_general ...
```

Note that there are two cut-rules: *Cut* (`sig3_cut`) and *Cut*! (`sig3_ccut`). The second rule is needed in the proof of cut-elimination as shown in Section 4.1. We later show that the system with only the *Cut* rule (`sig2`) is equivalent to `sig3`. Sequents in `sig3` take the form $n \vdash c; B; L$ where n is the height of the derivation, c the number of times the cut-rule was used and B and L the classical and linear contexts respectively. Definition `sig3_cut_general` makes also explicit the following measures : w (the complexity of the cut formula) and $h = m + n$ (the cut-height, including the height of the two premises of the cut-rule). Such measures will be useful in Section 4.1.

3.1 Focused System

Focusing was first proposed by Andreoli [1] as a discipline on proofs in order to reduce non-determinism. Proofs are organized in two alternating phases: the negative phase contains only invertible rules, and the positive phase contains only non-invertible rules. The connectives $⅋, \bot, \&, \top, ?, \forall$ have invertible introduction rules and are thus classified as *negative*. The remaining connectives $\otimes, 1, \oplus, !, \exists$ are *positive*. Formulas inherit their polarity according to their main connective, *e.g.*, $A \otimes B$ is positive and $A ⅋ B$ is negative.

In LL's focused proof system LLF (also called triadic system), there are two types of sequents where Θ is a set of formulas, Γ a multiset of formulas, and L a list of formulas:

- $\vdash \Theta : \Gamma \Uparrow L$ belongs to the negative phase. During this phase, all negative formulas in L are introduced and all positive formulas and atoms are moved to Γ.
- $\vdash \Theta : \Gamma \Downarrow A$ belongs to the positive phase. During this phase, all positive connectives at the root of A are introduced.

Let us present some rules (the complete system is depicted in the Appendix):

$$\frac{\vdash \Theta : \Gamma \Uparrow A, L \quad \vdash \Theta : \Gamma \Uparrow B, L}{\vdash \Theta : \Gamma \Uparrow A \,\&\, B, L} \qquad \frac{\vdash \Theta : \Gamma_1 \Downarrow A \quad \vdash \Theta : \Gamma_2 \Downarrow B}{\vdash \Theta : \Gamma_1, \Gamma_2 \Downarrow A \otimes B} \qquad \frac{\vdash \Theta : \Gamma \Downarrow P}{\vdash \Theta : \Gamma, P \Uparrow} D_1 \qquad \frac{\vdash \Theta, P : \Gamma \Downarrow P}{\vdash \Theta, P : \Gamma \Uparrow} D_2 \qquad \frac{\vdash \Theta : \Gamma \Uparrow N}{\vdash \Theta : \Gamma \Downarrow N} R$$

Notice that focusing ($\Downarrow$) persists on the premises of the $\otimes$ rule. The negative phase ends when the list of formulas L is empty. Then, the decision rules D_1 (linear) and D_2 (classical) are used to start a new positive phase. Finally, the release rule switches to a negative phase

when the current focused formula is negative. This restriction on proofs has two main applications: it considerably reduces proof search space and it allows specifiers to engineer proofs as we illustrate in Section 5.

The rules of the focused system (`TriSystem`) were formalized as the previous ones. We used `⊢B ; M ; UP L` to denote the negative phase and `⊢B ; M ;DW F` for a positive phase focused on F. Similar to the (unfocus) sequent systems `sig2`, we also defined for the triadic system a version with explicit height of derivation (`TriSystemh`) that we later show to be equivalent.

An interesting feature of `TriSysem` is that we can define automatic tactics to handle the negative phase. For instance the formula $p^- \parr q^- \parr \bot \parr (?p^+) \parr (?q^+)$ is proved as follows:

```
Example sequent: ⊢ [] ; [] ; UP( [ (p⁻ & q⁻) ⅋ ⊥ ⅋ ?p⁺ ⅋ ?q⁺ ] ).
Proof with unfold p;unfold q;InvTac.
  NegPhase. (* Negative phase *)
  eapply tri_dec2 with (F:= p⁺) ... (* apply the decision rule on the classical context *)
  eapply tri_dec2 with (F:= q⁺) ...
Qed.
```

We first decompose all the negative connectives and store the atoms (`NegPhase`). Then, we have to prove two sequents (due to the & rule). For proving those sequents, we only need to decide to focus on the formulas p^+ and q^+ respectively. The "..." notation in Coq applies the tactic `InvTac` that solves all the needed intermediary results (e.g., checking the polarities of the atoms and applying the initial rules when needed).

3.2 Structural Properties

Using strong induction on the height of the derivation, we show several structural properties for the above systems. For instance, we proved that equivalent multisets prove the same formulas (preserving the height of the derivation):

```
Theorem sig2h_exchange: B1 =mul= B2 → L1 =mul= L2 → n ⊢ B1; L1 → n ⊢ B2; L2.
```

Moreover, using the library `Morphisms`, we are able to easily substitute equivalent multisets during proofs (using the tactic `rewrite`). Moreover, we proved height preserved weakening and contraction for the classical context:

```
Theorem weakening_sig2h: n ⊢ B; L → n ⊢ B ++ D; L.
Theorem contraction_sig2h: n ⊢ F :: F :: B; L → n ⊢ F :: B; L.
```

Similar properties were proved for the triadic system. The only interesting case is exchange for the focused context (needed in the proof of completeness):

```
Theorem EquivUpArrow : n ⊢ B ; M ; UP L → L =mul= L' → exists m, m ⊢ B ; M ; UP L'.
```

In this case, the height of the derivation is not preserved since the last context is a list and not a multiset. The proof of such theorem required some lemmata showing the invertibility of the negative connectives. In particular, in a sequent `⊢B ; M ; UP L ++ [a] ++ L'`, if a is a negative connective, then, it can be applied any time during the proof. Those results correspond to the following theorems (all the variables are universally quantified):

```
Theorem EquivAuxTop : ⊢ B ; M ; UP (L ++ [⊤] ++ L').
Theorem EquivAuxBot : ⊢ B ; M ; UP (L ++ L') → ⊢ B ; M ; UP (L ++ [⊥] ++ L').
Theorem EquivAuxWith : ⊢ B;M ;UP (L++[F]++ L') → ⊢ B ;M ;UP (L++[G] ++ L') → ⊢ B; M ;UP (L ++ [F&G] ++ L').
Theorem EquivAuxPar : n ⊢ B ; M ; UP (L ++ [F ; G] ++ L') → ⊢ B ; M ; UP (L ++ [F ⅋ G] ++ L').
Theorem EquivAuxSync : ~ Asynchronous F → ⊢ B ; M ++ [F] ; UP (L ++ L') → ⊢ B ; M ; UP (L ++ [F] ++ L').
Theorem EquivAuxForAll : (forall x, ⊢ B ; M; UP (L++[Subst FX x]++ L')) → ⊢ B ; M; UP (L ++ [F{FX}] ++ L').
```

```
Theorem EquivAuxQuest : n ⊢ B ++ [F] ; M ; UP (L ++ L') →  ⊢ B ; M ;  UP (L ++ [? F] ++ L').
```

The proofs of these lemmas proceed by induction on the sum of the complexity of the formulas in L (i.e., summing up the complexities of the formulas in L). The `Asynchronous F` predicate asserts that F is a negative formula.

4 Meta-Theory

This section presents the main results formalized in our system: cut-elimination (for the dyadic system) and completeness of the focused system. Hence, as a corollary, we show the equivalence of all these systems (dyadic, triadic, with/without cut rules and with/without measures) and prove the consistency of LL. We show the key cases and relevant strategies to complete the proofs of the main theorems. For that, we shall use the following notation. We start listing, using "H" ids, the relevant hypotheses of the theorem and the current Goal (using "G" ids). Then, we write the relevant steps (using Coq's comments) to generate new hypotheses or transform the current goal. For instance,

```
HI : forall m <=n, m ⊢ B ; M ; UP L → exists x, x ⊢ B ; M ++ L (* inductive hypothesis *)
H1: n ⊢ B ; M ; UP L
G:  ⊢ B ; M ++ L (* current goal *)
(* apply HI in H1 to conclude H2 *)        H2 : exists x, x ⊢ B ; M ++ L
(* using inversion in H2 we show H2' *)    H2' : x ⊢ B ; M ++ L
(* conclude G by using adequacy (with/without measures) in H2' *)
```

In proofs involving the triadic system (Section 4.2), the focusing discipline determines easily the next step/goal in the proof. In those cases, we do not use comments to explain the proof but we use directly Coq's tactics. Roughly, those tactics correspond to one application of a logical rule of the triadic system (`TriSystem`). Finally, some of the proofs in Section 4.1 correspond to standard sequent transformations that we show in Appendix B.

4.1 Cut Elimination

The proof is structured as follows. The main theorem consists in showing that a proof with one cut can be replaced with a proof with zero cuts. Recall that the measure of the number of cuts is explicit (as c) in the system `sig3`:

```
Lemma cut_elimination_base: n ⊢ 1 ; B ; L → exists m, m ⊢ 0 ;  B ;  L.
```

This lemma represents the case where we are eliminating the upper-most cut in a derivation tree. The proof, proceeds by double induction on the complexity of the cut-formula (w) and the cut-height (h), i.e., the sum of the premises' heights of the cut-rule. The proof of this lemma requires several additional lemmas/cases:

(i) The base case corresponds to $w = h = 0$. The cut-formula is one of the units or an atomic proposition. Moreover, since $h = 0$, both premises are either the initial rule or $\top$. We grouped those cases in the following theorem.

```
Theorem cut_aux: L =mul= M1 ++ M2 → 0 ⊢ 0 ;  B ;  F ::  M1 → 0 ⊢ 0 ;  B ;  F° ::  M2 → exists m, m ⊢ 0 ;  B ;  L.
```

(ii) In the cases where the formula is not principal, one has to permute the cut and reduce the inductive measure h. For instance, the case where rule $\oplus$ is used in one of the cut-premises is proved as follows (see item (i) in Appendix B):

```
H :  n1 ⊢ 0; B; a :: F :: T (* 0 cuts, height n1. "a" is the cut-formula *)
Hn1 : S n1 ⊢ 0; B; a :: F ⊕ G :: T
Hn2 : n2 ⊢ 0; B; a° :: M2  (* 0 cuts, height n2 *)
HI : forall h<= n1 + n2 → inductive hypothesis on the cut−height
G: exists m : nat, m ⊢ 0; B; F ⊕ G :: T ++ M2
(* apply cut on H and Hn2 to conclude Hc*)          Hc: S(max n1 n2) ⊢ 1 ; B; F ::  T ++ M2
(* use HI to produce a cut-free proof Hc' *)        Hc': x ⊢ 0 ; B; F ::  T ++ M2
(* using exists with t:= S x the new goal is G' *)  G' : S x  ⊢ 0; B; F ⊕ G :: T++ M2
(*  conclude G' from Hc' and the rule ⊕ *)
```

(iii) When the cut formula is principal in both premises, we perform (possible several) cuts with simpler formulas. For instance, the case of ⊗ is (see item (ii) in App. B):

```
H1 : S (max n m) ⊢ 0; B; F ⊗ G :: (M1 ++ M2) (* cut formula is F * G *)
H2 : S n0 ⊢ 0; B; F° ⅋ G° :: D
H3 : m ⊢ 0; B; F :: M1
H4 : n ⊢ 0; B; G :: M2
H5 : n0 ⊢ 0; B; G° :: F° :: D
HI : forall w<= w(F ⊗ G) → inductive hypothesis on the weight
G: exists m0 : nat, m0 |− 0; B; M1 ++ M2 ++ D
(* apply cut on H4 and H5 to conclude H6 *)         H6 : S(max n n0) ⊢ 1 ; B; F° :: (M2 ++ D)
(* use HI on H6 to produce a cut-free proof H6' *)  H6' : x ⊢ 0 ; B; F° :: (M2 ++ D)
(* apply cut on H3 and H6 to conclude H7 *)         H7 : S(max m x) ⊢ 1 ; M1 + + M2 + + D
(* use HI on H7 to produce a cut-free proof H7' *)  H7' : y ⊢ 0 ; M1 + + M2 + + D
(* conclude G from H7' *)
```

The case when ! is the cut-formula and principal requires and additional rule *Cut* !

$$\frac{\vdash \Theta : \Delta, !F \quad \vdash \Theta, F^{\perp} : \Gamma}{\vdash \Theta : \Delta, \Gamma}$$

that we show to be admissible (Theorem `sig2_iff_sig3` below). This rule is encoded in the `sig3` system (constructor `sig3_ccut`). We then transform an application of *Cut* into an application of *Cut* !, reducing the cut-height (see item (iii) in Appendix B):

```
H1 : S n |− 0; B; [! F]
H2 : S n0 |− 0; B; ? F° :: L
H3 : n |− 0; B; [F]
H4 : n0 |− 0; F° :: B; L
HI : forall h <= S (n + n0) → inductive hypothesis on cut−height
G:  exists m0 : nat, m0 ⊢ 0; B; L
(* apply ccut on H1 and H4 to conclude H5*)    H5 : S(max (S n) n0) ⊢ 1; B ; L
(* use HI to obtain the cut-free proof H5' *)  H5' : x ⊢ 0; B ; L
(* conclude G from H5' *)
```

(iv) The cases for eliminating an application of *Cut*! are similar.

Using all the lemmas above, the proof of cut-elimination considers all the cases (including the symmetric ones) generated by Coq. The final step is to show that a proof with an arbitrary number of cuts can be transformed into a proof without cuts. This can be easily done by induction on the number of cuts and using the previous results:

```
Theorem cut_elimination : forall B L n c, n |− c ; B ; L → exists m, m |− 0 ; B ; L.
```

As a corollary, we can show the consistency of LL:

```
Theorem consistency : ~ sig3 n c [] []   ∧   ~ sig3 n c [] [0]   ∧  ~ sig3 n c [] [⊥].
```

Now we can prove that the *Cut*! rule is admissible and hence, the systems with (`sig3`) and without (`sig2`) this rule are equivalent:

```
Theorem sig2_iff_sig3: sig2 B; L ↔ sig3 B; L.
```

The most interesting (inductive) case in the proof considers the transformation of an appli-

cation of *Cut*! into an application of *Cut* (see item (iv) in Appendix B).

4.2 Completeness of Focusing

The following theorem shows that focused proofs can be mimicked by the dyadic system:

```
Theorem Soundness : LexpPos M → n |−F− B ; M ; A  → |−− B ;  M  ++ (Arrow2LL A).
```

where the predicate `LexpPos` states that all the formulas in the list/multiset M are all positive and the function `Arrow2LL` simply transforms "⇑ L" into L and "⇓ F" into the list $[F]$. The proof is easy by induction on the height of the derivation n: we just need to apply exactly the same rule used in the focused proof.

The proof of the inverse theorem, i.e., completeness, is of course more involved. First, we proved the invertibility theorems in Section 3.1 (for the negative connectives). Then we proved that applications of positive rules can be switched:

```
Theorem InvCopy : |−F− B ++ [F]  ;M ;  UP (F ::  L) → LexpPos M → |−F− B ++ [F] ; M  ;  UP L .
Theorem InvEx : |−F− B  ;M ;  UP (Subst FX t ::  L) → LexpPos M → |−F− B ; M ++ [E{ FX}]  ;  UP L .
Theorem InvPlus : |−F− B  ;M ;  UP (F ::  L) → LexpPos M → |−F− B ; M ++ [F ⊕ G]  ;  UP L .
Theorem InvTensor : LexpPos (M ++ M') → |−F− B  ;M ;  UP (F ::  L) → |−F− B; M';  UP (G ::  L') →
                    |−F− B ;  M ++ M' ++ [F ⊗ G]  ;  UP (L ++ L') .
```

In [1] Andreoli detailed the proof of the case for ⊗ (`InvTensor`). Let us explain the case for ⊕. First we define the following predicates:

```
Definition RUp (n:nat) := forall B L M F G,  LexpPos M →
       n |−F− B ; M;  UP (L ++ [F])  → |−F− B ; M ++ [F ⊕ G]; UP L.
Definition RDown (n:nat) := forall B M H F G,  LexpPos M → PosOrNegAtom F →
       n |−F− B ;  M ++ [F]  ;  DW H → |−F− B ; M ++ [F ⊕ G]  ;  DW H.
Definition RInd (n:nat) := RUp n ∧ RDown (n −1).
```

The predicate `RUp` determines how ⊕ permutes with the negative connectives. We proceed by induction on n. In the inductive cases, we consider two cases, namely, when L is empty or not. Let us explain the first case. We consider the last rule applied. The cases of the negative connectives are easy, e.g., the case ⊥ is as follows:

```
Hyp1:   S n |−F− B;  M1;  UP [⊥]
G1: |−F− B;  M1  ++ [⊥ ⊕ G];  UP []    eapply tri_dec1 with (F:= ⊥ ⊕ G) ...
G2: |−F− B;  M1 ++ [];  DW (⊥ ⊕ G)     eapply tri_plus1 ...
G3: |−F− B;  M1 ++ [];  DW ⊥           eapply tri_rel ...
G4: |−F− B;  M1 (2++·) []; UP⊥         (* conclude by using AdequacyTri1 and Hyp1 *)
```

The last step (`AdequacyTri1`) uses the adequacy result relating the system with measures (sequent in `Hyp1`) and the current goal.

The interesting case is the store case, i.e., when the the formula F is an atom or a positive formula. Then we have the following situation:

```
HDown :  RDown (n −1)
Hyp1 :   n |−F− B;  M1 ++ [F];  UP [ ]
```

Since the sequent in `Hyp1` is provable, due to focusing, we know that it is provable by using a decision rule. We then need to consider three cases, namely, focusing on either `F`, on a formula in `M1` or on a formula in `B`. Let us consider the second case. We have the following situation (after some substitutions):

```
HDown :  RDown (n')
Hyp1' :    n' |−F− B;  M1' ++ [F];  DW F'
HML :   M1 =mul= F' ::  M1'
```

```
G1: |−F− B; (F' :: M1') ++ [F ⊕ G]; UP []        eapply tri_dec1 with (F:=F0) ...
G2: |−F− B; M1' ++ [F ⊕ G]; DW F'
```

The proof of G2 ends by applying the inductive hypotheses HDown in Hyp1'.

The second predicate (RDown) allows us to permute two positive phases in the proof. We proceed by induction on n and consider all the cases for H in DW H. Let us show the case when $H = \exists x.H'$. The main hypotheses and goal are:

```
HDown : RDown n'
H : n' |−F− B; M ++ [F]; DW (Subst FX t)
G1 : |−F− B; M ++ [F ⊕ G]; DW (E{FX})     apply tri_ex with (t:=t) ...
G2: |−F− B; M ++ [F ⊕ G]; DW (Subst FX t)
```

The proof ends by applying the inductive hypothesis HDown in H. The other cases are similar (see the case of ⊗ in the Appendix B.1).

5 Applications

In [16], LL was used as the logical framework for specifying a number of logical and computational systems. The idea is to use two meta-level predicates $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ for identifying objects that appear on the left or on the right side of the sequents in the object logic. Hence, for instance, object-level sequents of the form $B_1, \ldots, B_n \vdash C_1, \ldots, C_m$ (where $n, m \geq 0$) are specified as the multiset $\lfloor B_1\rfloor, \ldots, \lfloor B_n\rfloor, \lceil C_1\rceil, \ldots, \lceil C_m\rceil$. Here, as an application of our developments, we specify an encoding of intuitionistic propositional logic (LJ) into LL and prove the adequacy of the encoding. The machinery we develop is general enough to mechanize the proof of other adequacy theorems for logical systems [16] and also for concurrent computational systems [18,19].

We first specify the syntax of LJ in the usual way:

```
Inductive LForm :Set :=
  | bot (* false *) | atom : nat → LForm (* atomic propositions *)
  | conj : LForm → LForm → LForm (* conjunction *)
  | disj : LForm → LForm → LForm (* disjunction *)
  | impl : LForm → LForm → LForm. (* intuitionistic implication *)
```

and the logical rules:

```
Inductive sq : list LForm → nat → LForm → Prop :=
  | init : forall (L L' :list LForm) a, L =mul= atom a :: L' → L ; 0 |−P− atom a
  | cR : forall L F G n m , L ; n |−P− F → L ; m |−P− G → L ; S (max n m) |−P− conj F G
  | cL : forall L G G' F L' n, L =mul= (conj G G') :: L' → G :: G' :: L' ; n |−P− F → L ; S n |−P− F
  [...]
```

We name the two meta-level predicates $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ as 1 and 3 respectively, and we also name the (meta-level) functional symbols that represent each of the connectives in LJ:

```
Definition rg := 1. (* UP PREDICATE *)  Definition lf := 3. (* DOWN predicate *)
Definition bt := 0. (* bottom *)         Definition pr := 1. (* atoms / propositions *)
Definition cj := 2. (* conjunction *)    Definition dj := 3. (* disjunction *)
Definition im := 4. (* implication *)
```

We note that even (resp. odd) predicates are assumed to be positive (resp. negative). Hence, rg and lf are negative atoms and it is not possible to focus on them.

Now we are able to encode LJ's logical rules as bipoles [16]. Roughly, a bipole is a positive formula in which no positive connective can be in the scope of a negative one.

Focusing on such a formula will produce a single positive and a single negative phase. This two-phase decomposition enables the adequate capturing of the application of an object-level inference rule by the meta-level logic. For instance, LJ initial rule is encoded as the following LL formula:

```
Definition INIT :Lexp := Ex ( fun _ x ⇒ tensor
        ( tensor  ( perp ( a1 rg ( fc1 pr ( var x ))))  ( perp ( a1 lf ( fc1 pr ( var x )))))  top).
```

If we decide to focus on INIT, then, the rule $\exists$ requires to chose a value for x and the focus continues on the formula $(\lceil (A1(pr(x)))^{\perp} \rceil \otimes \lfloor (A1(pr(x)))^{\perp} \rfloor) \otimes \top$. Hence, the two atoms $\lceil (A1(pr(x))) \rceil$ and $\lfloor (A1(pr(x))) \rfloor$ must be already in the context (since focusing cannot be lost). The proof finishes by loosing the focusing on $\top$ and then, in the negative phase, by using the rule $\top$. In other words, in one change of polarity, we check that a given atom a is on the right and on the left, thus finishing the proof.

The other rules follow a similar pattern. For instance, $\wedge_R$ is specified as:

```
Definition CRIGHT :Lexp := Ex ( fun _ x ⇒ ex( fun y ⇒
                    tensor (perp (a1 rg ( fc2 cj ( var x)  ( var y))))
                      (  witH (atom (a1 rg ( var x)))  ( atom (a1 rg ( var y)))  ) )).
```

Again, in a positive phase, $\exists$ and $\otimes$ are applied and the atom $\lceil A1(cj(x,y)) \rceil$ is consumed. Focusing is lost on & and we obtain two LL sequents. In each of them, the atoms $\lceil A1(x) \rceil$ and $\lceil A1(y) \rceil$ are stored. What we observe is exactly the behavior of $\wedge_R$.

Thanks to the automatic tactics developed, the proof of soundness and completeness are relatively simple. Let us explain the main arguments. In the case of soundness, we have:

```
Definition encodeSequent (L: list PL.LForm) (F: PL.LForm) :=
                        |−F− Theory ; ( encodeFR F) ::  encodeList L ; UP [].
Theorem Soundness: |−P− n ; F → ( encodeSequent L F ).
```

where the function encodeFR (resp. encodeList) simply generate the needed $\lceil \cdot \rceil$ atoms (resp. list of $\lfloor \cdot \rfloor$ atoms). Moreover, Theory is a list with the encoding of all the LJ rules.

We proceed by induction on n. Let us consider the case of conjunction right. It suffices to build the focused proof, starting by focusing on the rule CRIGHT:

```
HI: (* Inductive hypothesis  x ≤ max(n,m)  *)
HF: L |−P− n;  F
HG: L |−P− m;  G
G1: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; UP []                    eapply tri_dec2 with (F:= CRIGHT) ...
G2: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; DW CRIGHT                eapply tri_ex with (t:= encodeTerm F).
G3: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; DW E{ ...}    ...        eapply tri_ex with (t:= encodeTerm G).
G4: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; DW  AtR ⊗ (AtF & AtG)    eapply tri_tensor ...
G5: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; DW  AtF & AtG            apply tri_rel ...
G6: |−F− Theory ;  ⌈cj(F,G)⌉ :: encodeList L; UP  AtF & AtG            apply tri_with; apply tri_store ...
(* Branch F *) apply HI in HF ...
(* Branch G *) apply HI in HG ...
```

Recall that the "..." solves the easy cases (see Section 3.1). On the other side, completeness is stated as follows:

```
Theorem Completeness : ( encodeSequent L F ) → exists n, L |−P− n ; F.
```

We proceed by induction on the height of the (LL) derivation. The interesting part of this proof is that, thanks to focusing, we only need to use the inversion tactic on the hypotheses until the goal is proved. For instance, in the inductive case ($n > 0$) the most relevant hypothesis and goal are:

```
H: S n |-F- Theory; encodeFR F :: encodeList L; UP []
G: exists n : nat, L |-P- n; F
```

Since we have the proof in H, focusing tells us that that such proof must proceed by deciding either to focus on a formula in the classical context (`Theory`) or in the linear context (`encodeFR F :: encodeList L`). However, atoms in the linear context are negative and we cannot focus on them. Therefore, the only alternative is to focus on one of the formulas in `Theory`. This makes the proof rather simple. For instance, consider the case `INIT`:

```
H' : n |-F- Theory; encodeFR F :: encodeList L; DW INIT
```

We continue applying inversion on `H'` to "generate" the consequences of having the proof `H'`. The main consequence, in this case, is

```
H'' : exists L' a, L =mul= (F :: L') ∧ F = PL.atom a
```

This means that F is necessarily an atom and such atom is in the list of (PL) formulas L. Hence, the proof concludes easily by applying the initial rule of LJ.

6 Related and Future Work

Intuitionistic propositional linear logic was implemented in Coq [22] and Isabelle [14], but the main goal of these works was to provide proof search, and thus no meta-theorems were proved. Cut-elimination and invertibility lemmas were proved for a formalization of several linear logic calculi in Abella [3]. Even though the paper presents only the propositional part, the proofs for first-order and focused fragments were later completed. The first-order implementation requires the use of the two-level logic approach, particular to this tool.

A generic method for formalizing sequent calculi in Isabelle/HOL is proposed in [5]. The meta-theorems are parametrized by the set of rules and for cut-elimination, weakening must be admissible. The authors have applied the method to provability logics.

Completeness of focusing was proved for intuitionistic propositional logic in Twelf and Agda [24]. The proof follows the technique developed in [20] where sequents are annotated with terms and the problem is reduced to type-checking. Focusing follows as a corollary of two other properties proved about the calculus. In their approach, context management is handled in the meta-level, so much of the bureaucracy in handling multisets is avoided. The method in [24] could be used to prove completeness of focusing of LL as well. A possible candidate implementation is the one from [23, Chapter 6] on a modified version of Twelf. Unfortunately this development was never carried out. Another approach for proving completeness of focusing, formalized in Coq, uses an algebraic implementation of the calculus [13]. To use this solution, the calculus must have "dual" rules for all connectives (harmony) and admissibility of contraction and weakening.

In the short term, we plan to use the machinery in Section 5 in order to mechanize the proofs of adequacy of other formalisms into Linear Logic, e.g., Hybrid Linear Logic [7], and Concurrent Constraint Process Calculi (CCP) [19]. In order to prove the adequacy of CCP calculi featuring modalities, we have to specify also the so-called subexponentials [19] (roughly, exponentials decorated with indexes). Finally, a more interesting outcome will be formalizing the theorems in [16]. This should allow us to use the meta-theory of linear logic to prove meta-theorems (e.g., cut-elimination) of other logics encoded into LL.

References

[1] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 1992.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, EATCS Series, 2004.

[3] K. Chaudhuri, L. Lima, and G. Reis. Formalized Meta-Theory of Sequent Calculi for Substructural Logics. In *Workshop on Logical and Semantic Frameworks, with Applications (LSFA-11)*, 2016.

[4] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP*, 2008.

[5] J. E. Dawson and R. Goré. Generic Methods for Formalising Sequent Calculi Applied to Provability Logic. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17*, 2010.

[6] J. Despeyroux, A. P. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in Coq. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA*, 1995.

[7] J. Despeyroux, C. Olarte, and E. Pimentel. Hybrid and Subexponential Linear Logics. *To appear in Electronic Notes in Theoretical Computer Science*, 2017.

[8] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic (CSL'12)*, 2012.

[9] A. P. Felty and A. Momigliano. Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *Journal of Automated Reasoning*, 2012.

[10] A. P. Felty, A. Momigliano, and B. Pientka. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey. *Journal of Automated Reasoning*, 2015.

[11] G. Gentzen. Investigations into Logical Deductions. In *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[12] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 1987.

[13] S. Graham-Lengrand. *Polarities & Focussing: a journey from Realisability to Automated Reasoning*. Habilitation thesis, Université Paris-Sud, 2014.

[14] S. Kalvala and V. D. Paiva. Mechanizing linear logic in Isabelle. In *In 10th International Congress of Logic, Philosophy and Methodology of Science*, 1995.

[15] D. Miller and C. Palamidessi. Foundational Aspects of Syntax. *ACM Computing Surveys*, 1999.

[16] D. Miller and E. Pimentel. A formal framework for specifying sequent calculus proof systems. *Theoretical Computer Science*, 2013.

[17] V. Nigam and D. Miller. A Framework for Proof Systems. *Journal of Automated Reasoning*, 2010.

[18] V. Nigam, C. Olarte, and E. Pimentel. A General Proof System for Modalities in Concurrent Constraint Programming. In *Concurrency Theory (CONCUR)*, 2013.

[19] C. Olarte, E. Pimentel, and V. Nigam. Subexponential concurrent constraint programming. *Theoretical Computer Science*, 2015.

[20] F. Pfenning. Structural Cut Elimination. *Information and Computation*, 2000.

[21] F. Pfenning and C. Elliott. Higher-order Abstract Syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[22] J. Power and C. Webster. Working with linear logic in coq. In *12th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16, 1999.

[23] J. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

[24] R. J. Simmons. Structural focalization. *CoRR*, abs/1109.6273, 2011.

[25] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2000.

Introduction Rules

$$\frac{\vdash \Theta : \Gamma \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, \bot}\ [\bot] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F, G}{\vdash \Theta : \Gamma \Uparrow L, F \parr G}\ [\parr] \qquad \frac{\vdash \Theta, F : \Gamma \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, ?F}\ [?]$$

$$\frac{}{\vdash \Theta : \Gamma \Uparrow L, \top}\ [\top] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F \quad \vdash \Theta : \Gamma \Uparrow L, G}{\vdash \Theta : \Gamma \Uparrow L, F \,\&\, G}\ [\&] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F[c/x]}{\vdash \Theta : \Gamma \Uparrow L, \forall x\, F}\ [\forall]$$

$$\frac{}{\vdash \Theta :\Downarrow 1}\ [1] \qquad \frac{\vdash \Theta : \Gamma \Downarrow F \quad \vdash \Theta : \Gamma' \Downarrow G}{\vdash \Theta : \Gamma, \Gamma' \Downarrow F \otimes G}\ [\otimes] \qquad \frac{\vdash \Theta :\Uparrow F}{\vdash \Theta :\Downarrow\, !F}\ [!]$$

$$\frac{\vdash \Theta : \Gamma \Downarrow F}{\vdash \Theta : \Gamma \Downarrow F \oplus G}\ [\oplus_l] \qquad \frac{\vdash \Theta : \Gamma \Downarrow G}{\vdash \Theta : \Gamma \Downarrow F \oplus G}\ [\oplus_r] \qquad \frac{\vdash \Theta : \Gamma \Downarrow F[t/x]}{\vdash \Theta : \Gamma \Downarrow \exists x\, F}\ [\exists]$$

Identity, Reaction, and Decide rules

$$\frac{}{\vdash \Theta : A_p^{\perp} \Downarrow A_p}\ [I_1] \qquad \frac{}{\vdash \Theta, A_p^{\perp} :\Downarrow A_p}\ [I_2] \qquad \frac{\vdash \Theta : \Gamma, S \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, S}\ [R\Uparrow]$$

$$\frac{\vdash \Theta : \Gamma \Downarrow P}{\vdash \Theta : \Gamma, P \Uparrow}\ [D_1] \qquad \frac{\vdash \Theta, P : \Gamma \Downarrow P}{\vdash \Theta, P : \Gamma \Uparrow}\ [D_2] \qquad \frac{\vdash \Theta : \Gamma \Uparrow N}{\vdash \Theta : \Gamma \Downarrow N}\ [R\Downarrow]$$

Fig. A.1. The focused proof system, LLF, for linear logic [1]. Here, L is a list of formulas, Θ is a multiset of formulas, Γ is a multiset of literals and positive formulas, A_p is a positive literal, N is a negative formula, P is not a negative literal, and S is a positive formula or a negated atom.

A Linear Logic Focused Proof System

The complete set of rules for the triadic (focused) system is in Figure A.1.

B Proof Transformations (Cut-Elimination Proof)

(i) Case $\oplus$ when the cut-formula is not principal:

$$\frac{\dfrac{\overset{\pi_1}{\vdash [\Psi]; \Gamma, G_1, F}}{\vdash [\Psi]; \Gamma, G_1 \oplus G2, F}\ \oplus_1 \qquad \overset{\pi_2}{\vdash \Delta, F^{\perp}}}{\vdash [\Psi]; \Gamma, G_1 \oplus G2, \Delta}\ cut \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\pi_1}{\vdash [\Psi]; \Gamma, G_1, F} \qquad \overset{\pi_2}{\vdash [\Psi]; \Delta, F^{\perp}}}{\vdash [\Psi]; \Gamma, G_1, \Delta}\ cut}{\vdash [\Psi]; \Gamma, G_1 \oplus G2, \Delta}\ \oplus_1$$

(ii) The cut-formula is principal (case $\otimes$):

$$\frac{\dfrac{\overset{\pi_1}{\vdash [\Psi]; \Gamma_1, F} \qquad \overset{\pi_2}{\vdash [\Psi]; \Gamma_2, G}}{\vdash [\Psi]; \Gamma_1, \Gamma_2, F \otimes G}\ \otimes \qquad \dfrac{\overset{\pi_3}{\vdash [\Psi]; \Delta, F^{\perp}, G^{\perp}}}{\vdash [\Psi]; \Delta, F^{\perp} \parr G^{\perp}}\ \parr}{\vdash [\Psi]; \Gamma_1, \Gamma_2, \Delta}\ cut \quad \rightsquigarrow$$

$$\frac{\overset{\pi}{\vdash [\Psi]; \Gamma_1, F} \qquad \dfrac{\overset{\pi_2}{\vdash [\Psi]; \Gamma_2, G} \qquad \overset{\pi_3}{\vdash [\Psi]; \Delta, F^{\perp}, G^{\perp}}}{\vdash [\Psi]; \Gamma_2, F^{\perp}, \Delta}\ cut}{\vdash [\Psi]; \Gamma_1, \Gamma_2, \Delta}\ cut$$

(iii) The cut-formula is principal (case !):

$$\dfrac{\pi_1 \quad \dfrac{\pi_2}{\vdash [\Psi, F^{\perp}]; \Delta}}{\vdash [\Psi]; !F \quad \vdash [\Psi]; \Delta, ?F^{\perp}} \; ? \; cut \quad \rightsquigarrow \quad \dfrac{\pi_1 \quad \dfrac{\pi_2}{copy}}{\vdash [\Psi]; !F \quad \vdash [\Psi, F^{\perp}]; \Delta} \; cut!$$

(iv) Equivalence between the system with $CUT!$ and the system with only the standard cut-rule:

$$\dfrac{\pi_1 \quad \pi_2}{\vdash [\Psi]; !F \quad \vdash [\Psi, F^{\perp}]; \Delta} \; cut! \quad \rightsquigarrow \quad \dfrac{\dfrac{\pi_1}{\vdash [\Psi]; F} \quad \dfrac{\pi_2}{\vdash [\Psi, F^{\perp}]; \Delta}}{\vdash [\Psi]; !F \quad \vdash [\Psi]; \Delta, ?F^{\perp}} \; ? \; cut$$

B.1 Proof of the case $H = H_1 \otimes H_2$

The case $H = H_1 \otimes H_2$ leads to the following:

```
H  : M' ++ N' =mul= [F] ++ M
HF' : n1 |-F- B; M'; DW H1
HG' : n2 |-F- B; N'; DW H2
G:  |-F- B; M ++ [F ⊕ G]; DW (H1 ⊗ H2)
```

We have to consider two cases: when F is in M' and when F is in N'. Both cases are proved in a similar way. Consider the case $F \in M$:

```
HDown: RDown (S n1)
HMN: M' =mul= [F] ++ M''
HF' : n1 |-F- B; M'; DW H1
HG' : n2 |-F- B; N'; DW H2
------------------------------------------------
G': |-F- B; (M' ++ N') ++ [F ⊕ G]; DW (H1 ⊗ H2)  eapply tri_tensor ...
(* Case H1 *) |-F- B; M'' ++ [F ⊕ G]; DW H1   (* proved by using HDown in HG'*)
(* Case H2 *) |-F- B; N'; DW H2               (* proved from HG' *)
```

Hierarchical hybrid logic

Alexandre Madeira

HASLab INESC TEC, U. Minho & CIDMA, U. Aveiro, Portugal

Renato Neves

HASLab INESC TEC, U. Minho

Manuel A. Martins

CIDMA, U. Aveiro, Portugal

Luís S. Barbosa

HASLab INESC TEC, U. Minho

Abstract

We introduce $\mathcal{HHL}$, a hierarchical variant of hybrid logic. We study first order correspondence results and prove a Hennessy-Milner like theorem relating (hierarchical) bisimulation and modal equivalence for $\mathcal{HHL}$. Combining hierarchical transition structures with the ability to refer to specific states at different levels, this logic seems suitable to express and verify properties of hierarchical transition systems, a pervasive semantic structure in Computer Science.

Keywords: Hybrid logic, Hierarchical systems.

1 Introduction

From D. Harel's *statecharts* [Har87] to the mobile *ambients* [CG98] proposed by A. Gordon and L. Cardelli, models of hierarchical systems are pervasive in Computer Science. In practice, hierarchical, multi-level transitions often coexist with local ones. The ability to represent both and reason uniformly about them is essential to such models, for example in specific applications such as coordination protocols in the context of distributed systems [BAPG03], or to handle software which operates in different modes of execution and is able to commute between them. The global transition structure defines how such systems evolve from a mode (or configuration) to another [MFMB11].

This paper introduces a hierarchical variant of hybrid logic [Bla00,Bra10] that adds to the modal description of hierarchical transition structures the ability to

refer to specific states at any level of description. As discussed by the authors in [MFMB11], hybrid logic, which allows one to refer to specific states in a system, became the specification *lingua franca* for reconfigurable systems. The hierarchical variant proposed here sets the ground for a uniform framework to express and verify properties of any kind of hierarchical transition system.

The paper is organised as follows: after a section on preliminaries, Section 3 introduces hierarchical hybrid logic, $\mathcal{HHL}$. The relevant first-order correspondences are discussed in Section 4. Section 5 discusses bisimulation for this sort of systems and proves a Hennessy-Milner like theorem relating, under the usual conditions of image-finiteness, bisimulation and modal equivalence for $\mathcal{HHL}$. Finally, Section 6 concludes and briefly discusses future work.

2 Hybrid logic

The qualifier *hybrid* [Bla00,Bra10] applies to extensions of modal languages with symbols, called *nominals*, that explicitly refer to individual states in the underlying Kripke frames. A *hybrid signature* is a pair $(\mathrm{Prop}, \mathrm{Nom})$, where Prop and Nom are disjoint sets of symbols of *propositional variables* and *nominals*, respectively. The set of *hybrid formulas over* $(\mathrm{Prop}, \mathrm{Nom})$ extends the corresponding modal language with formulas i, which only hold at the state named by i, and $@_i\rho$, which asserts that formula ρ holds in the state named by i, for $i \in \mathrm{Nom}$. Formally, the set of formulas, denoted by $\mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$, is defined by the grammar

$$\rho ::= p \mid i \mid @_i\rho \mid \Diamond\rho \mid \neg\rho \mid \rho \wedge \rho,$$

for $i \in \mathrm{Nom}$ and $p \in \mathrm{Prop}$.

Note that the remaining Boolean connectives and the box modality are introduced as abbreviations. The set $\mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$ of *basic formulas* is defined by

$$\begin{aligned}\mathrm{Prop} \cup \mathrm{Nom} \cup \{\Diamond\rho : \rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})\} \\ \cup \{@_i\rho : \rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom}), i \in \mathrm{Nom}\}\end{aligned}$$

Models of $\mathcal{HL}$ for a signature $(\mathrm{Prop}, \mathrm{Nom})$ are Kripke structures with named states, i.e., structures $M = (W, R, V)$ where W is a set of *states*, $R \subseteq W \times W$ is the *accessibility relation*, and $V : \mathrm{Prop} \cup \mathrm{Nom} \to \mathcal{P}(W)$ is a function that interprets propositions and nominals, such that for any $i \in \mathrm{Nom}$, $V(i)$ is a singleton. The set of all models over a signature $(\mathrm{Prop}, \mathrm{Nom})$ is denoted by $\mathrm{Mod}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$.

The *satisfaction* relation between a model $M = (W, R, V)$ in $\mathrm{Mod}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$ and a formula $\rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$ at state $w \in W$, is recursively defined as follows:

- $M, w \vDash^{\mathcal{HL}} \rho$ iff $w \in V(\rho)$, $\rho \in \mathrm{Nom} \cup \mathrm{Prop}$;
- $M, w \vDash^{\mathcal{HL}} @_i\varphi$ iff $M, V(i) \vDash^{\mathcal{HL}} \varphi$;
- $M, w \vDash^{\mathcal{HL}} \Diamond\varphi$ iff there is a $v \in W$ such that $(w, v) \in R$ and $M, v \vDash^{\mathcal{HL}} \varphi$;
- $M, w \vDash^{\mathcal{HL}} \neg\varphi$ iff it is false that $M, w \vDash^{\mathcal{HL}} \varphi$ (in symbols, $M, w \nvDash^{\mathcal{HL}} \varphi$);

- $M, w \models^{\mathcal{HL}} \varphi \wedge \varphi'$ iff $M, w \models^{\mathcal{HL}} \varphi$ and $M, w \models^{\mathcal{HL}} \varphi'$.

As usual, we write $M \models^{\mathcal{HL}} \rho$ when, for any $w \in W$, $M, w \models^{\mathcal{HL}} \rho$, and $\models^{\mathcal{HL}} \rho$ when $M \models^{\mathcal{HL}} \rho$ for all $M \in \mathrm{Mod}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$.

Applications often justify the introduction of a distinguished state in the underlying Kripke structure, regarded as the initial point of evaluation. As discussed in the sequel, such is the case of hierarchical transition systems representing software configurations: each configuration 'starts' at a specific entry point, or initial state. Models for such *pointed* versions of $\mathcal{HL}$ are pairs $((W, R, V), s)$ where $s \in W$. Accordingly, the satisfaction relation is defined by

$$((W, R, V), s) \models \rho \text{ iff } (W, R, V), s \models^{\mathcal{HL}} \rho$$

3 Hierarchical hybrid logic

A signature in *hierarchical hybrid logic*, $\mathcal{HHL}$-signature in short, is a tuple $(\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ where Prop, Nom, PROP and NOM are four disjoint sets of *propositions* and *nominals* corresponding to the two levels of assertion, called the 'lower' and the 'upper' level, respectively.

The set of formulas for a signature $(\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ is organised in a two-levels hierarchy.

Definition 3.1 ($\mathcal{HHL}$-formulas) *Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. The set $\mathrm{Fm}^{\mathcal{HHL}}(\Delta)$ of $\mathcal{HHL}$-formulas is the smallest set such that:*

- $\mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom}) \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $\mathrm{PROP} \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $\mathrm{NOM} \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $@_{\mathtt{i}}\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$, *for any* $\mathtt{i} \in \mathrm{NOM}$ *and* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $\circledast\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$, *for any* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $\neg\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$, *for any* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$;
- $\rho \wedge \rho' \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$, *for any* $\rho, \rho' \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$.

As usual, Boolean connectives and the box modality are defined by abbreviation. Note also that $\mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom}) \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$.

Definition 3.2 ($\mathcal{HHL}$-models) *Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$ signature. A Kripke* Δ-model *is a tuple*

$$M = (W, R, V, (M_w)_{w \in W})$$

where

- *W is a non empty set of the so called* upper, *or* super, *states;*
- *$R \subseteq W \times W$ is a binary relation called the* upper accessibility relation*;*
- *$V : \mathrm{PROP} \cup \mathrm{NOM} \to \mathcal{P}(W)$ is a function where, for any $\mathtt{i} \in \mathrm{NOM}$, $V(\mathtt{i})$ is a singleton. When it is implicitly clear, the element $w \in V(\mathtt{i})$ will be identified as the set $V(\mathtt{i})$ itself.*

- *For any* $w \in W$, M_w *is a* $\mathcal{HL}$*-pointed model* $M_w = \big(H_w, s_w\big)$, *where* $H_w = (W_w, R_w, V_w) \in \mathrm{Mod}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$ *and* $s_w \in W_w$.

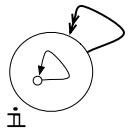


Fig. 1. An almost trivial $\mathcal{HHL}$ model.

Definition 3.3 ($\mathcal{HHL}$-Satisfaction) *Let* $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature and* $M = (W, R, V, (M_w)_{w \in W})$ *be a* Δ*-model. The* satisfaction relation *between formulas, models and points is recursively defined as follows:*

(i) $M, w \vDash \rho$ *iff* $H_w, s_w \vDash^{\mathcal{HL}} \rho$, *for* $\rho \in \mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$;

(ii) $M, w \vDash \mathtt{p}$ *iff* $w \in V(\mathtt{p})$, *for* $\mathtt{p} \in \mathrm{PROP}$;

(iii) $M, w \vDash \mathtt{i}$ *iff* $V(\mathtt{i}) = \{w\}$, *for* $\mathtt{i} \in \mathrm{NOM}$;

(iv) $M, w \vDash @_{\mathtt{i}}\rho$ *iff* $M, V(\mathtt{i}) \vDash \rho$;

(v) $M, w \vDash \Diamond\!\!\!\!-\rho$ *iff there is a* $w' \in W$ *such that* $(w, w') \in R$ *and* $M, w' \vDash \rho$;

(vi) $M, w \vDash \neg\rho$ *iff it is not the case that* $M, w \vDash \rho$;

(vii) $M, w \vDash \rho \wedge \rho'$ *iff* $M, w \vDash \rho$ *and* $M, w \vDash \rho'$

As in the standard case we write $M \vDash \rho$ when, for any $w \in W$, $M, w \vDash \rho$, and $\vDash \rho$ when $M \vDash \rho$ for all $M \in \mathrm{Mod}^{\mathcal{HHL}}(\Delta)$. These definitions extend to sets of formulas as expected. Finally, for $\Gamma \cup \{\rho\} \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$, ρ is said to be a global consequence of Γ, $\Gamma \vDash \rho$, if for any model $M \in \mathrm{Mod}^{\mathcal{HHL}}(\Delta)$, $M \vDash \Gamma$ implies $M \vDash \rho$.

4 First-order correspondences

As usual in the introduction of a modal language, this section discusses how formulas in hierarchical hybrid logic can be transformed into first-order ones. This is done through the introduction of two possible correspondences: the first one follows the well known recipe used in the standard translation of modal logic; the second entails a different, less common perspective taking explicitly into account definability in each possible world. Beyond the theoretical interest of these correspondences, they pave the way to the effective use of a number of proof assistants.

4.1 The standard translation

Definition 4.1 Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. We define the two-sorted first-order signature $\Delta^* = (S, F, P)$ as follows:

- the set of sorts $S = \{W, U\}$, where W is the sort of super-states and U the sorts of sub-states.
- the set of operation symbols $F = \{i : W \to U \mid i \in \mathrm{Nom}\} \cup \{\mathtt{i} : \to W \mid \mathtt{i} \in \mathrm{NOM}\} \cup \{Init : W \to U\}$;

- the set of predicate symbols $P = \{R : W \times W,\ r : W \times U \times U,\ Sub : W \times U\} \cup \{p : W \times U \mid p \in \mathrm{Prop}\} \cup \{\mathtt{p} : W \mid \mathtt{p} \in \mathrm{PROP}\}$.

The purpose of operation symbol Sub is to explicitly relate (sub)states to super-states, defining the inhabitants of each possible super-state. Although this construction is not required for defining the standard translation, it plays a role in the alternative translation introduced in the next section.

Definition 4.2 Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. Given a model $M = (W, R, (M_w)_{w \in W}, V) \in \mathrm{Mod}(\Delta)$, we define the model M^* as follows: sorts are realized by the carrier sets $M^*_W = W$ and $M^*_U = \bigcup_{w \in W} W_w$. The definition for functions and predicates, respectively, is given by

$$
\begin{array}{l}
M^*_i(w) = V_w(i) \text{ for } i \in \mathrm{Nom} \\
M^*_{\mathtt{i}} = V(\mathtt{i}) \text{ for } \mathtt{i} \in \mathrm{NOM} \\
M^*_{Init}(w) = s_w \\
M^*_R(w, w') \text{ iff } (w, w') \in R \\
M^*_r(w, u, v) \text{ iff } (u, v) \in R_w \\
M^*_{Sub}(w, u) \text{ iff } u \in W_w \\
M^*_p(w, u) \text{ iff } u \in V_w(p), p \in \mathrm{Prop} \\
M^*_{\mathtt{p}}(w) \text{ iff } w \in V(\mathtt{p}), \mathtt{p} \in \mathrm{PROP}
\end{array}
$$

Finally, we obtain the translation of formulas as follows:

Definition 4.3 [Standard translation] The standard translation ST consists of the map

$$\mathrm{ST} : \mathrm{Fm}^{\mathcal{HHL}}(\Delta) \longrightarrow \mathrm{Fm}^{FOL}(\Delta^*)$$

recursively defined as follows:

$$
\begin{array}{rcll}
\mathrm{ST}_{X,u}(p) & = & p(X, u) & p \in \mathrm{Prop} \\
\mathrm{ST}_{X,u}(i) & = & u = i(X) & i \in \mathrm{Nom} \\
\mathrm{ST}_{X,u}(@_i \rho) & = & \mathrm{ST}_{X,i(X)}(\rho) & i \in \mathrm{Nom}, \rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom}) \\
\mathrm{ST}_{X,u}(\Diamond \rho) & = & (\exists v : U)(r(X, u, v) \wedge \mathrm{ST}_{X,v}(\rho)) & \rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom}) \\
\mathrm{ST}_{X,u}(\mathtt{p}) & = & \mathtt{p}(X) & \mathtt{p} \in \mathrm{PROP} \\
\mathrm{ST}_{X,u}(\mathtt{i}) & = & X = \mathtt{i} & \mathtt{i} \in \mathrm{NOM} \\
\mathrm{ST}_{X,u}(@_{\mathtt{i}} \rho) & = & \mathrm{ST}_{X,u}(\rho)[X \mapsto \mathtt{i}, u \mapsto Init(\mathtt{i})] & \mathtt{i} \in \mathrm{NOM} \\
\mathrm{ST}_{X,u}(\circledast \rho) & = & (\exists Y : W)(R(X, Y) \wedge \mathrm{ST}_{Y,Init(Y)}(\rho)) & \\
\mathrm{ST}_{X,u}(\neg \rho) & = & \neg \mathrm{ST}_{X,u}(\rho) & \\
\mathrm{ST}_{X,u}(\rho \wedge \rho') & = & \mathrm{ST}_{X,u}(\rho) \wedge \mathrm{ST}_{X,u}(\rho') &
\end{array}
$$

Notation $\mathrm{ST}_{\mathtt{i}, Init(\mathtt{i})}(\rho)$ is used for $\mathrm{ST}_{X,u}(\rho)[X \mapsto \mathtt{i}, u \mapsto Init(\mathtt{i})]$, when clear from context.

Lemma 4.4 *Let* $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature and* $M = (W, R, (M_w)_{w \in W}, V)$ *a* Δ*-model and* $\rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$*. Then, for any* $w \in W$*,* $z \in H_w$*,*

$$H_w, z \vDash^{\mathcal{HL}} \rho \;\; \textit{iff} \;\; M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\rho)[X \mapsto w, u \mapsto z]$$

where $M_w = (H_w, s_w)$*.*

Proof. By induction on the structure of formulas.

for $\rho = p$**,** $p \in \mathrm{Prop}$

$$\begin{array}{ll} & H_w, z \vDash^{\mathcal{HL}} p \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash^{\mathcal{HL}} \} \\ & z \in V_w(p) \\ \Leftrightarrow & \quad \{ \text{ defn. of } M^* \text{ and } z \in W_w \} \\ & M_p^*(w, z) \\ \Leftrightarrow & \quad \{ \text{ defn of } \vDash^{FOL} \} \\ & M^* \vDash^{FOL} p(X, u)[X \mapsto w, u \mapsto z] \\ \Leftrightarrow & \quad \{ \text{ defn of ST} \} \\ & M^* \vDash^{FOL} \mathrm{ST}_{X,u}(p)[X \mapsto w, u \mapsto z] \end{array}$$

for $\rho = i$**,** $i \in \mathrm{Nom}$

$$\begin{array}{ll} & H_w, z \vDash^{\mathcal{HL}} i \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash^{\mathcal{HL}} \} \\ & z = V_w(i) \\ \Leftrightarrow & \quad \{ \text{ defn. of } M^* \} \\ & M_i^*(w) = z \\ \Leftrightarrow & \quad \{ \text{ defn of } \vDash^{FOL} \} \\ & M^* \vDash^{FOL} i(X) = u[X \mapsto w, u \mapsto z] \\ \Leftrightarrow & \quad \{ \text{ defn of ST} \} \\ & M^* \vDash^{FOL} \mathrm{ST}_{X,u}(i)[X \mapsto w, u \mapsto z] \end{array}$$

for $\rho = @_i\varphi$**,** $i \in \mathrm{Nom}$

$$\begin{array}{ll} & H_w, z \vDash^{\mathcal{HL}} @_i\varphi \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash^{\mathcal{HL}} \} \\ & H_w, V_w(i) \vDash^{\mathcal{HL}} \varphi \\ \Leftrightarrow & \quad \{ \text{ I.H.} \} \\ & M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\varphi)[X \mapsto w, u \mapsto V_w(i)] \\ \Leftrightarrow & \quad \{ \text{ since } V_w(i) = M_i^*(w) + \text{I.H} \} \\ & M^* \vDash^{FOL} \mathrm{ST}_{X,i(X)}(\varphi)[X \mapsto w, u \mapsto z] \\ \Leftrightarrow & \quad \{ \text{ defn of ST} \} \\ & M^* \vDash^{FOL} \mathrm{ST}_{X,u}(@_i\varphi)[X \mapsto w, u \mapsto z] \end{array}$$

for $\rho = \Diamond\varphi$

$$
\begin{array}{ll}
& H_w, z \vDash^{\mathcal{HL}} \Diamond\varphi \\
\Leftrightarrow & \quad \{ \text{ defn. of } \vDash^{\mathcal{HL}} \} \\
& H_w, v \vDash^{\mathcal{HL}} \varphi, \text{for some } v \in W_w \text{ such that } (z, v) \in R_w \\
\Leftrightarrow & \quad \{ \text{ I.H + defn of } M^* \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\varphi)[X \mapsto w, u \mapsto v] \\
& \text{for some } v \text{ such that } M_r^*(w, z, v) \\
\Leftrightarrow & \quad \{ \text{ defn of } \vDash^{FOL} \} \\
& M^* \vDash^{FOL} (\exists v)\, r(X, u, v) \wedge \mathrm{ST}_{X,v}(\varphi)[X \mapsto w, u \mapsto z] \\
\Leftrightarrow & \quad \{ \text{ defn of ST} \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\Diamond\varphi)[X \mapsto w, u \mapsto z]
\end{array}
$$

The cases dealing with conjunction and negation follow directly from the induction hypothesis. □

Theorem 4.5 *Let* $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature,* $M = (W, R, (M_w)_{w \in W}, V)$ *a* Δ*-model and* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$*. Then, for* $w \in W$*,*

$$M, w \vDash \rho \text{ iff } M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\rho)[X \mapsto w, u \mapsto M^*_{Init}(w)]$$

Proof. The proof proceeds by induction on the structure of formulas. Thus,

for $\rho \in \mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$

$$
\begin{array}{ll}
& M, w \vDash \rho \\
\Leftrightarrow & \quad \{ \text{ defn. of } \vDash \} \\
& H_w, s_w \vDash^{\mathcal{HL}} \rho \\
\Leftrightarrow & \quad \{ \text{ since } M^*_{Init}(w) = s_w \text{ and Lemma 4.4} \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\rho)[X \mapsto w, u \mapsto M^*_{Init}(w)]
\end{array}
$$

for $\rho = @_{\mathtt{i}}\varphi$**,** $\mathtt{i} \in \mathrm{NOM}$

$$
\begin{array}{ll}
& M, w \vDash @_{\mathtt{i}}\varphi \\
\Leftrightarrow & \quad \{ \text{ defn. of } \vDash \} \\
& M, V(\mathtt{i}) \vDash \varphi \\
\Leftrightarrow & \quad \{ \text{ I.H. } \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{\mathtt{i},u}(\varphi)[X \mapsto w, u \mapsto M^*_{Init}(V(\mathtt{i}))] \\
\Leftrightarrow & \quad \{ \text{ since } V(\mathtt{i}) = M^*_{\mathtt{i}} \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{\mathtt{i},Init(\mathtt{i})}(\varphi) \\
\Leftrightarrow & \quad \{ \text{ defn. of ST} \} \\
& M^* \vDash^{FOL} \mathrm{ST}_{X,u}(@_{\mathtt{i}}\varphi)[X \mapsto w, u \mapsto M^*_{Init}(w)]
\end{array}
$$

for $\rho = \circledcirc\varphi$

$$M, w \vDash \circledcirc\varphi$$

$\Leftrightarrow$ $\{$ defn. of $\vDash\}$

$M, w' \vDash \varphi$, for some $w' \in W$ such that $R(w, w')$

$\Leftrightarrow$ $\{$ I.H. + defn. of $M^*\}$

$M^* \vDash^{FOL} \mathrm{ST}_{Y,v}(\varphi)[X \mapsto w', u \mapsto M^*_{Init}(w')]$, for some $w' \in W$ such that $M^*_R(w, w')$

$\Leftrightarrow$ $\{$ defn. of $\vDash^{FOL}\}$

$M^* \vDash^{FOL} (\exists Y : W)R(X, Y) \wedge \mathrm{ST}_{Y,Init(Y)}(\varphi)[X \mapsto w]$

$\Leftrightarrow$ $\{$ defn. of ST$\}$

$M^* \vDash^{FOL} \mathrm{ST}_{X,u}(\Diamond\varphi)[X \mapsto w, u \mapsto M^*_{Init}(w)]$

Again, the cases dealing with conjunction and negation follow directly from the induction hypothesis. □

4.2 A different perspective

As mentioned above, an alternative translation, not standard in modal logic, will be considered now.

Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. Then $\mathrm{Mod}^{FOL}_D(\Delta^*)$ denotes the class of all models of $M \in \mathrm{Mod}(\Delta^*)$ such that for each $w \in W$, $M_{Init}(w)$ and $M_i(w)$, for any nominal i, belong to the universe associated to w, that is $M_{Sub}(w, M_{Init}(w))$ and $M_{Sub}(w, M_i(w))$ for any nominal i. If Nom is finite, we denote the formula

$$(\forall X : W)\Big(Sub(X, Init(X)) \wedge \bigwedge_{i \in \mathrm{Nom}} Sub(X, i(X))\Big)$$

by $D(\Delta)$. And we have, $\mathrm{Mod}^{FOL}_D(\Delta^*) = \{M \in \mathrm{Mod}(\Delta^*) \mid M \vDash^{FOL} D(\Delta)\}$.

Definition 4.6 Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. The operator

$$\_^\circ : \mathrm{Mod}^{FOL}_D(\Delta^*) \to \mathrm{Mod}^{\mathcal{HHL}}(\Delta)$$

is defined as follows: given a model $M \in \mathrm{Mod}^{FOL}_D(\Delta^*)$, we construct the model $M^\circ = (W^\circ, R^\circ, (M^\circ_w)_{w \in W^\circ}, V^\circ)$ as follows:

- $W^\circ = M_W$
- $R^\circ = M_R$
- for any $\mathtt{p} \in \mathrm{PROP}$, $V^\circ(\mathtt{p}) = M_{\mathtt{p}}$
- for any $\mathtt{i} \in \mathrm{NOM}$, $V^\circ(\mathtt{i}) = \{M_{\mathtt{i}}\}$

and for any $w \in W^\circ$, $M^\circ_w = \big(H^\circ_w, s^\circ_w\big)$ where $H^\circ_w = (W^\circ_w, R^\circ_w, V^\circ_w)$ is such that

- $W^\circ_w = \{a \mid M_{Sub}(w, a)\}$
- $R^\circ_w = \{(a, b) \mid M_r(w, a, b)$ and $M_{Sub}(w, a)$ and $M_{Sub}(w, b)\}$
- for any $p \in \mathrm{Prop}$, $V^\circ_w(p) = \{a \mid M_p(w, a)$ and $M_{Sub}(w, a)\}$
- for any $i \in \mathrm{Nom}$, $V^\circ_w(i) = M_i(w)$
- $s^\circ_w = M_{Init}(w)$

Observe the role of $D(\Delta)$ in asserting the definability of the local valuations V_w with respect to its functionality over Nom, as well as with respect to definability of the (local) initial states. In order to obtain a translation that is compatible with the operator $^\circ$ the standard translation defined above has to be constrained, leading to the following definition:

Definition 4.7 [(constrained) standard translation]

$$\begin{aligned}
\mathrm{ST}^\circ_{X,u}(p) &= Sub(X,u) \wedge p(X,u),\ p \in \mathrm{Prop} \\
\mathrm{ST}^\circ_{X,u}(i) &= Sub(X,u) \wedge u = i(X),\ i \in \mathrm{Nom} \\
\mathrm{ST}^\circ_{X,u}(\Diamond\rho) &= (\exists v : U)(Sub(X,v) \wedge r(X,u,v) \wedge \mathrm{ST}^\circ_{X,v}(\rho)),\ \rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop},\mathrm{Nom})
\end{aligned}$$

and it is defined as in ST for the remaining cases.

Lemma 4.8 *Let* $\Delta = (\mathrm{Prop},\mathrm{Nom},\mathrm{PROP},\mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature,* $M \in \mathrm{Mod}^{FOL}_D(\Delta^*)$ *and* $\rho \in \mathrm{Fm}^{\mathcal{HL}}(\mathrm{Prop},\mathrm{Nom})$*. Then, for any* $w \in W^\circ$ *and* $z \in H^\circ_w$*,*

$$H^\circ_w, z \models^{\mathcal{HL}} \rho \text{ iff } M \models^{FOL} \mathrm{ST}^\circ_{X,u}(\rho)[X \mapsto w, u \mapsto z]$$

Proof. The proof is by induction on the structure of formulas. Thus,

for $\rho = p$**,** $p \in \mathrm{Prop}$

$$\begin{aligned}
& M \models^{FOL} \mathrm{ST}^\circ_{X,u}(p)[X \mapsto w, u \mapsto z] \\
\Leftrightarrow & \quad \{ \text{ defn. of } \mathrm{ST}^\circ \} \\
& M \models^{FOL} (Sub(X,u) \wedge p(X,u))[X \mapsto w, u \mapsto z] \\
\Leftrightarrow & \quad \{ \text{ defn. of } \models \} \\
& M_{Sub}(w,z) \text{ and } M_p(w,z) \\
\Leftrightarrow & \quad \{ \text{ defn. of } M^\circ \} \\
& z \in V^\circ_w(p) \\
\Leftrightarrow & \quad \{ \text{ defn. of } \models^{\mathcal{HL}} \} \\
& H^\circ_w, z \models^{\mathcal{HL}} p
\end{aligned}$$

for $\rho = i$**,** $i \in \mathrm{Nom}$

$$\begin{aligned}
& M \models^{FOL} \mathrm{ST}^\circ_{X,u}(i)[X \mapsto w, u \mapsto z] \\
\Leftrightarrow & \quad \{ \text{ defn. of } \mathrm{ST}^\circ \} \\
& M \models^{FOL} (Sub(X,u) \wedge u = i(x))[X \mapsto w, u \mapsto z] \\
\Leftrightarrow & \quad \{ \text{ defn. of } \models \} \\
& M_{Sub}(w,z) \text{ and } z = M_i(w)) \\
\Leftrightarrow & \quad \{ \text{ defn. of } M^\circ \} \\
& z = V^\circ_w(i) \\
\Leftrightarrow & \quad \{ \text{ defn. of } \models^{\mathcal{HL}} \} \\
& H^\circ_w, z \models^{\mathcal{HL}} p
\end{aligned}$$

for $\rho = @_i\varphi$**,** $i \in \mathrm{Nom}$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(@_i\varphi)[X \mapsto w, u \mapsto z]$

$\Leftrightarrow$ $\{$ defn. of $\mathrm{ST}^{\circ}\}$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,i(X)}(\varphi)[X \mapsto w, u \mapsto z]$

$\Leftrightarrow$ $\{$ substitution$\}$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(\varphi)[X \mapsto w, u \mapsto M_i(w)]$

$\Leftrightarrow$ $\{$ I.H.$\}$

$H^{\circ}_w, M_i(w) \vDash^{\mathcal{HL}} \varphi$

$\Leftrightarrow$ $\{$ defn. of $\vDash^{\mathcal{HL}}\}$

$H^{\circ}_w, z \vDash^{\mathcal{HL}} @_i\varphi$

for $\rho = \Diamond\varphi$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(\Diamond\varphi)[X \mapsto w, u \mapsto z]$

$\Leftrightarrow$ $\{$ defn. of $\mathrm{ST}^{\circ}\}$

$M \vDash^{FOL} (\exists v : U)(Sub(X, v) \wedge r(X, u, v) \wedge \mathrm{ST}^{\circ}_{X,v}(\varphi))[X \mapsto w, u \mapsto z]$

$\Leftrightarrow$ $\{$ defn. of $\vDash\}$

there is $a \in M_U$ such that $M_{Sub}(w, a)$ and $M_r(w, z, a)$ and

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,v}(\varphi)[X \mapsto w, v \mapsto a]$

$\Leftrightarrow$ $\{$ I.H.$\}$

$H^{\circ}_w, a \vDash^{\mathcal{HL}} \varphi$, for some $a \in M_U$ such that $M_{Sub}(w, a)$ and $M_r(w, z, a)$

$\Leftrightarrow$ $\{$ by def. of $\vDash^{\mathcal{HL}}\}$

$H^{\circ}_w, z \vDash^{\mathcal{HL}} \Diamond\varphi$

□

Theorem 4.9 *Let* $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature,* $M \in \mathrm{Mod}^{FOL}_D(\Delta^*)$ *and* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$*. Then, for every* $w \in W$

$$M^{\circ}, w \vDash \rho \text{ iff } M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(\rho)[X \mapsto w, u \mapsto M_{Init}(w)]$$

Proof.

for $\rho \in \mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(\rho)[X \mapsto w, u \mapsto M_{Init}(w)]$

$\Leftrightarrow$ $\{$ Lemma 4.8 $\}$

$H^{\circ}_w, s^{\circ}_w \vDash^{\mathcal{HL}} \rho$

$\Leftrightarrow$ $\{$ defn. of $\vDash\}$

$M^{\circ}, w \vDash \rho$

for $\rho = @_{\mathtt{i}}\varphi$**,** $\mathtt{i} \in \mathrm{NOM}$

$M \vDash^{FOL} \mathrm{ST}^{\circ}_{X,u}(@_{\mathtt{i}}\varphi)[X \mapsto w, u \mapsto M_{Init}(w)]$

$\Leftrightarrow$ $\{$ defn. of $\mathrm{ST}^{\circ}\}$

$$
\begin{aligned}
& M \vDash^{FOL} \mathrm{ST}_{X,u}(\varphi)[X \mapsto \dot{\imath}, u \mapsto Init(\dot{\imath})] \\
\Leftrightarrow \quad & \{ \text{ I.H.} \} \\
& M^\circ, \dot{\imath} \vDash \varphi \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M^\circ, w \vDash @_{\dot{\imath}}\varphi
\end{aligned}
$$

for $\rho = \Diamond\varphi$

$$
\begin{aligned}
& M \vDash^{FOL} \mathrm{ST}^\circ_{X,u}(\Diamond\varphi)[X \mapsto w, u \mapsto M_{Init}(w)] \\
\Leftrightarrow \quad & \{ \text{ defn. of } \mathrm{ST}^\circ \} \\
& M \vDash^{FOL} (\exists Y : W) R(X,Y) \wedge \mathrm{ST}^\circ_{Y,Init(Y)}(\varphi))[X \mapsto w, u \mapsto M_{Init}(w)] \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& \text{there is } w' \in M_W \text{ st } M_R(w,w') \text{ and } M \vDash^{FOL} \mathrm{ST}^\circ_{Y,Init(Y)}(\varphi))[Y \mapsto w', u \mapsto M_{Init}(w)] \\
\Leftrightarrow \quad & \{ \text{ substitution } \} \\
& \text{there is } w' \in M_W \text{ st } M_R(w,w') \text{ and } M \vDash^{FOL} \mathrm{ST}^\circ_{Y,u}(\varphi))[Y \mapsto w', u \mapsto M_{Init}(w')] \\
\Leftrightarrow \quad & \{ \text{ I.H.} \} \\
& \text{there is } w' \in M_W \text{ st } M_R(w,w') \text{ and } M^\circ, w' \vDash \varphi \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash^{\mathcal{HL}} \} \\
& M^\circ, w \vDash \Diamond\varphi
\end{aligned}
$$

The cases of conjunction and negation are dealt similarly, easily achieved by direct application of induction hypothesis. □

The operator $\circ$ is not in general injective. However it is surjective. In fact, given an $M \in \mathrm{Mod}^{\mathcal{HHL}}(\Delta)$ it is not difficult to see that $M = (M^*)^\circ$, and $M^* \in \mathrm{Mod}^{FOL}_D(\Delta^*)$.

Corollary 4.10 *Let* $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ *be a* $\mathcal{HHL}$*-signature with finite sets* Nom *and* NOM. *Then, for any* $\Gamma \cup \{\rho\} \subseteq \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$*, we have*

$$\Gamma \vDash \rho \;\; \textit{iff} \;\; \Gamma^* \cup D(\Delta) \vDash^{FOL} (\forall X : W)\, \mathrm{ST}^\circ_{X,Init(X)}(\rho)$$

where $\Gamma^* = \{(\forall X : W), \mathrm{ST}^\circ_{X,Init(X)}(\rho) \,|\, \rho \in \Gamma\}$

Proof. Suppose that $\Gamma \vDash \rho$. Let M be a Δ^* first order model of $\Gamma^* \cup D(\Delta)$. Then, by Theorem 4.9, $M^\circ \vDash^{\mathcal{HL}} \Gamma$. Hence, $M^\circ \vDash^{\mathcal{HL}} \rho$. That is, for all $w \in W$ $M^\circ, w \vDash^{\mathcal{HL}} \rho$. Again, by Theorem 4.9 (in the opposite direction), $M \vDash^{FOL} (\forall X : W)\mathrm{ST}^\circ_{X,Init(X)}(\rho)$.

Conversely, suppose $\Gamma^* \cup D(\Delta) \vDash^{FOL} (\forall X : W)\mathrm{ST}^\circ_{X,Init(X)}(\rho)$. Let N be a Δ-model such that $N \vDash \Gamma$. Since $\circ$ is surjective there is an $M \in \mathrm{Mod}^{FOL}_D(\Delta^*)$ such that $N = M^\circ$. Since $N \vDash \Gamma$, by Theorem 4.9, M is a model of $\Gamma^* \cup D(\Delta)$. Therefore $M \vDash^{FOL} (\forall X : W)\, \mathrm{ST}^\circ_{X,Init(X)}(\rho)$. Again, by Theorem 4.9, $M^\circ = N \vDash \rho$. □

5 Hennessy-Milner Theorem for $\mathcal{HHL}$

Bisimulation is a main tool for the study of transition systems which, on their turn, are pervasive structures in computational phenomena. It is also a good example of the fruitful interaction between modal logic and Computer science. This section characterises a notion of hierarchical bisimulation for models of $\mathcal{HHL}$ and proves a corresponding Hennessy-Milner result relating hybrid equivalence between two models with the existence of a bisimulation relating them.

Definition 5.1 Let $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ be a $\mathcal{HHL}$-signature. An *hierarchical bisimulation* between two Δ-models $M = (W, R, (M_w)_{w \in W}, V)$ and $M' = (W', R', (M'_w)_{w \in W'}, V')$ consists of a relation $\mathbb{B} \subseteq W \times W'$ such that,

(NOM) for any $\mathtt{i} \in \mathrm{NOM}$, $V(\mathtt{i})\, \mathbb{B} V'(\mathtt{i})$

- for any $w \in W, w' \in W'$, $w \mathbb{B} w'$ implies:
  (ATOMS) for any $\mathtt{a} \in \mathrm{PROP} \cup \mathrm{NOM}$, $w \in V(\mathtt{a})$ iff $w' \in V'(\mathtt{a})$;
  (ZIG) for any $v \in W$ such that $(w, v) \in R$ there is a $v' \in W'$ such that $v \mathbb{B} v'$ and $(w', v') \in R'$;
  (ZAG) for any $v' \in W'$ such that $(w', v') \in R'$ there is a $v \in W$ such that $v \mathbb{B} v'$ and $(w, v) \in R$.

(LOCAL) M_w and $M'_{w'}$ are bisimilar, *i.e.*, there is a relation $\mathrm{B}^{w}_{w'} : H_w \times H_{w'}$ such that
  (init) $s_w \, \mathrm{B}^{w}_{w'} s'_{w'}$;
  (nom) for any $i \in \mathrm{Nom}$, $V_w(i) \, \mathrm{B}^{w}_{w'} V'_{w'}(i)$;
  - for any $u \in H_w, u' \in H'_w$ such that $u \, \mathrm{B}^{w}_{w'} u'$,
    (atoms) for any $p \in \mathrm{Prop} \cup \mathrm{Nom}$, $u \in V_w(p)$ iff $u' \in V'_{w'}(p)$;
    (zig) for any $v \in H_w$ such that $(u, v) \in R_w$ there is a $v' \in H'_{w'}$ such that $v \, \mathrm{B}^{w}_{w'} v'$ and $(u', v') \in R'_{w'}$;
    (zag) for any $v' \in H'_{w'}$ such that $(u', v') \in R'_{w'}$ there is a $v \in H_w$ such that $v \, \mathrm{B}^{w}_{w'} v'$ and $(u, v) \in R_w$.

An example in depicted in Fig. 2. The reader may easily notice the existence of local bisimulations relating the transition systems inside each of the two states of the system in the left with the one in the right, plus a global bisimulation relating precisely those (global) states.

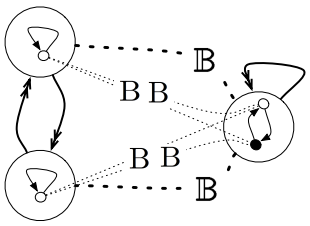


Fig. 2. A $\mathcal{HHL}$-bisimulation.

Lemma 5.2 *Let M and M' be two $\mathcal{HHL}$-models over the same signature. The set of hierarchical bisimulations between M and M' is closed under unions.*

Proof. Let $\mathbb{B}^1, \mathbb{B}^2 \subseteq |W| \times |W'|$ be two bisimulations between models M and M'. Their union $\mathbb{B} = \mathbb{B}^1 \cup \mathbb{B}^2$ is also an hierarchical bisimulation because

- Clearly, all points named by nominals are related by $\mathbb{B}$ as they are related both by $\mathbb{B}^1$ and $\mathbb{B}^2$. Moreover, for any pair (w, w') such that $w \mathbb{B} w'$ either $w \mathbb{B}^1 w'$ or $w \mathbb{B}^2 w'$. As both $\mathbb{B}^1$ and $\mathbb{B}^2$ are hierarchical bisimulations, clauses of (i.) in Definition. 5.1 hold for $\mathbb{B}$.
- A similar argument applies to both (ZIG) and (ZAG) conditions. For clause (iv) let $(w, v) \in R$ and $w \mathbb{B} w'$. Then, either $w \mathbb{B}^1 w'$ or $w \mathbb{B}^2 w'$. Since, $\mathbb{B}^1$ and $\mathbb{B}^2$ are bisimulations, we have that there is a $v' \in W'$ such that $v \mathbb{B}^1 v'$ or $v \mathbb{B}^2 v'$. Hence $v \mathbb{B} v'$. The condition (ZAG) condition is proved similarly.

□

Similarly, one may prove that hierarchical bisimulations are closed for composition as well. Bisimulation invariance is, on the other hand, a main, expected result.

Theorem 5.3 (Bisimulation invariance) *Let M and M' be two $\mathcal{HHL}$-models over the same signature $\Delta = (\mathrm{Prop}, \mathrm{Nom}, \mathrm{PROP}, \mathrm{NOM})$ and $\mathbb{B}$ a bisimulation between them. Then, for any $w \mathbb{B} w'$ and for any $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$,*

$$M, w \vDash \rho \text{ iff } M', w' \vDash \rho$$

Proof. The proof is by induction on the structure of the sentences.

$\rho \in \mathrm{BFm}^{\mathcal{HL}}(\mathrm{Prop}, \mathrm{Nom})$

$$\begin{array}{ll} & M, w \vDash \rho \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash \} \\ & (H_w, s_w) \vDash^{\mathcal{HL}} \rho \\ \Leftrightarrow & \quad \{ \star \} \\ & (H'_{w'}, s'_{w'}) \vDash^{\mathcal{HL}} \rho \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash \} \\ & M', w' \vDash \rho \end{array}$$

Step $\star$ comes from the (init) clause in Definition 5.1, $s_w \, \mathrm{B}^{\,w}_{w'} s_{w'}$, and the standard *bisimulation invariance* of (propositional)-hybrid logic (e.g. [tC05]). However, this proof can be achieved in a complete analogy with the (top-level) cases proved above. For instance, in order to proof the invariance of $\rho = i$, for $i \in \mathrm{Nom}$, we take the bisimilar initial states s_w and $s'_{w'}$ (by (init)) and we reproduce exactly the same steps of the $\rho = \mathtt{i}$ proof, but considering the condition (nom) in the place of (NOM). The other cases is obtained exactly in the same way.

$\rho = \mathtt{i}$, $\mathtt{i} \in \mathrm{NOM}$

$$\begin{array}{ll} & M, w \vDash \mathtt{i} \\ \Leftrightarrow & \quad \{ \text{ defn. of } \vDash \} \\ & V(\mathtt{i}) = w \\ \Leftrightarrow & \quad \{ \text{ ATOMS of Defn. 5.1} \} \end{array}$$

$$
\begin{aligned}
& V'(\mathtt{i}) = w' \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M', w' \vDash \mathtt{i}
\end{aligned}
$$

$\rho = \mathtt{p}$, $\mathtt{p} \in \mathrm{PROP}$

$$
\begin{aligned}
& M, w \vDash \mathtt{p} \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& w \in V(\mathtt{p}) \\
\Leftrightarrow \quad & \{ \text{ ATOMS of Defn. 5.1} \} \\
& w' \in V'(\mathtt{p}) \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M', w' \vDash \mathtt{p}
\end{aligned}
$$

$\rho = @_{\mathtt{i}}\varphi$

$$
\begin{aligned}
& M, w \vDash @_{\mathtt{i}}\varphi \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M, V(\mathtt{i}) \vDash \varphi \\
\Leftrightarrow \quad & \{ \text{ I.H. + NOM of Defn. 5.1} \} \\
& M', V'(\mathtt{i}) \vDash \varphi \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M', w' \vDash @_{\mathtt{i}}\varphi
\end{aligned}
$$

$\rho = \circledast\varphi$

$$
\begin{aligned}
& M, w \vDash \circledast\varphi \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M, v \vDash \varphi \text{ for some } v \in W \text{ such that } (w, v) \in R \\
\Leftrightarrow \quad & \{ \text{ I.H. + ZIG for } \Rightarrow \text{ + ZAG for } \Leftarrow \} \\
& M', v' \vDash \varphi \text{ for some } v' \in W' \text{ such that } (w', v') \in R' \\
\Leftrightarrow \quad & \{ \text{ defn. of } \vDash \} \\
& M', w' \vDash \circledast\varphi
\end{aligned}
$$

□

A $\mathcal{HHL}$-model M is *image-finite* if for each state $w \in W$, the set $\{v : (w, v) \in R\}$ and the sets $\{v : (u, v) \in R_w, w \in W\}$, $u \in W_w$, are finite. Note that no condition is imposed on the cardinality of W.

Theorem 5.4 *Let Δ be a $\mathcal{HHL}$-signature and M and M' two* image-finite Δ-*models, respectively. Then, for every $w \in W$ and $w' \in W'$, the following conditions are equivalent:*

(i) $M, w \vDash \rho$ *iff* $M', w' \vDash \rho$, *for any formula* $\rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta)$

(ii) There is a bisimulation $\mathbb{B}$ *between* M *and* M' *such that* $w \mathbb{B} w'$.

Proof. We have just to prove that (i) implies (ii). Let us show that

$$\mathbb{Z} = \left\{(w, w') \in W \times W' : \text{ for any } \rho \in \mathrm{Fm}^{\mathcal{HHL}}(\Delta), M, w \vDash \rho \text{ iff } M', w' \vDash \rho\right\}$$

is a bisimulation. The conditions (ATOM) and (NOM) follow directly from the invariance of the sentences $\rho \in \mathrm{NOM} \cup \mathrm{PROP}$. Since the image-finitness of $\mathcal{HHL}$-models entails the image-finitness of its local $M_w, w \in W$, we have that the condition (ATOMS) corresponds to the standard Hennessy-Milner result of the propositional hybrid logic (e.g. [tC05]).

For the (ZIG) condition, assume that $w \mathbb{Z} w'$ and let $u \in W$ such that $(w, u) \in R$. To obtain a contradiction, suppose that there is no $u' \in W'$ with $(w', u') \in R'$ and $u \mathbb{Z} u'$. As in the standard case the image-finite condition makes $S' = \{u' : (w', u') \in R'\}$ finite. Moreover, S' cannot be empty since in such a case $M, w \vDash \neg \Diamond\!\!\!\!\circ (@_{\mathtt{i}}\mathtt{i})$, which is incompatible with the fact that $M, w \vDash \Diamond\!\!\!\!\circ(@_{\mathtt{i}}\mathtt{i})$ (since $(w, u) \in R$). By assumption, for every $z \in S'$ there is a formula ψ_z such that $M, u \vDash \psi_z$ and it is false that $M', z \vDash \psi_z$ Consider now the conjunction

$$\psi = \bigwedge_{z \in S'} \psi_z$$

of all of these formulas. Hence we have that $M, w \vDash \Diamond\!\!\!\!\circ \psi$ and $M', w' \nvDash \Diamond\!\!\!\!\circ \psi$, which contradicts $w \mathbb{Z} w'$. □

6 Discussion and future work

In this paper we introduced $\mathcal{HHL}$ – a hierarchical variant of hybrid logic. We presented first order correspondence results and proved a Hennessy-Milner like theorem relating (hierarchical) bisimulation and modal equivalence for $\mathcal{HHL}$.

On the more practical side, it is clear that $\mathcal{HHL}$ is appropriate to reason about hierarchical transition systems, as they appear in, e.g. reconfigurable programs. The logic, however, is unable to express arbitrary multi-level transitions, thus enforcing a particular specification discipline. Actually, there are some variants (e.g. [MMB15]) whose motivation stems directly from Computer Science applications which may require more complex features. For example, statecharts, already mentioned in the Introduction, comprise different forms of inter-level transitions, including multiple-source and multiple-target ones as well as simultaneous firing of non-conflicting transitions and their prioritisation, which cannot be captured in $\mathcal{HHL}$.

The process of constructing $\mathcal{HHL}$ on top of standard propositional hybrid logic can be made generic through *hybridisation*, a procedure introduced in [MMDB11] that consists of taking an arbitrary logic and to systematically develop on top of it the syntax and semantic features of hybrid logic. To be completely general, this is framed in the context of the institution theory of Goguen and Burstall [GB92], each logic (base and hybridised) being treated abstractly as an *institution*. Actually, $\mathcal{HHL}$ can be obtained through hybridisation of propositional hybrid logic. The latter, however, can be replaced by other logics resulting from the same process being applied to whatever logics are found interesting to specify configurations (states) at the lower level of the hierarchy — e.g., equational, first-order, fuzzy, etc. The

application of this idea on the rigorous development of reconfigurable systems was discussed in [Mad13,MMBH15,MNBM16].

Concerning strictly logical properties, we would like to discuss decidability and completeness properties of $\mathcal{HHL}$. For this reason, we intend to explore, among other things, the finite model property for this logic, as well complete proof calculi, resorting to our previous work [NMMB16].

Acknowledgements

This work is supported by ERDF European Regional Development Fund, through the COMPETE Programme, and by National Funds through FCT - Portuguese Foundation for Science and Technology - within projects `POCI-01-0145-FEDER-016692` and `UID/MAT/04106/2013`, as well by project "SmartEGOV: Harnessing EGOV for Smart Governance (Foundations, Methods, Tools) / NORTE-01-0145-FEDER-000037", supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement. A. Madeira and R. Neves are further supported by the FCT individual grants `SFRHBPD103004/2014` and `SFRH/BD/52234/2013` respectively .

References

[BAPG03] Richard Banach, Farhad Arbab, George A. Papadopoulos, and John R. W. Glauert. A multiply hierarchical automaton semantics for the iwim coordination model. *J. UCS*, 9(1):2–33, 2003.

[Bla00] Patrick Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8(3):339–365, 2000.

[Bra10] Torben Braüner. *Hybrid Logic and its Proof-Theory*. Applied Logic Series. Springer, 2010.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Mad13] Alexandre Madeira. *Foundations and techniques for software reconfigurability*. PhD thesis, Universidades do Minho, Aveiro and Porto (Joint MAP-i Doctoral Programme), July 2013.

[MFMB11] Alexandre Madeira, José M. Faria, Manuel A. Martins, and Luís Soares Barbosa. Hybrid specification of reactive systems: An institutional approach. In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and Formal Methods (SEFM 2011, Montevideo, Uruguay, November 14-18, 2011)*, volume 7041 of *Lecture Notes in Computer Science*, pages 269–285. Springer, 2011.

[MMB15] Alexandre Madeira, Manuel A. Martins, and Luís Soares Barbosa. A logic for n-dimensional hierarchical refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, volume 209 of *EPTCS*, pages 40–56, 2015.

[MMBH15] Alexandre Madeira, Manuel A. Martins, Luís Soares Barbosa, and Rolf Hennicker. Refinement in hybridised institutions. *Formal Asp. Comput.*, 27(2):375–395, 2015.

[MMDB11] Manuel A. Martins, Alexandre Madeira, Răzvan Diaconescu, and Luís Soares Barbosa. Hybridization of institutions. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science (CALCO 2011, Winchester, UK, August 30 - September 2, 2011)*, volume 6859 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2011.

[MNBM16] Alexandre Madeira, Renato Neves, Luís Soares Barbosa, and Manuel A. Martins. A method for rigorous design of reconfigurable systems. *Sci. Comput. Program.*, 132:50–76, 2016.

[NMMB16] Renato Neves, Alexandre Madeira, Manuel A. Martins, and Luís Soares Barbosa. Proof theory for hybrid(ised) logics. *Sci. Comput. Program.*, 126:73–93, 2016.

[tC05] Balder David ten Cate. *Model Theory for Extended Modal Languages*. PhD thesis, Institute for Logic, Language and Computation Universiteit van Amsterdam, 2005.

Inversion, Fixed Points, and the Art of Dual Wielding

Robin Kaarsgaard[1,2]

DIKU, Department of Computer Science, University of Copenhagen

Abstract

In category theory, the symbol † ("dagger") is used to denote (at least) two very different operations on morphisms: Taking their *adjoint* (in the context of dagger categories) and finding their *least fixed point* (in the context of domain theory and categories enriched in domains). In the present paper, we wield both of these daggers at once and consider dagger categories enriched in domains. Exploiting the view of dagger categories as enriched in involutive monoidal categories, we develop a notion of a monotone dagger structure as a dagger structure that is well behaved with respect to the enrichment, and show that such a structure leads to pleasant inversion properties of the fixed points that arise as a result of this enrichment. Notably, such a structure guarantees the existence of *fixed point adjoints*, which we show are intimately related to the conjugates arising from the canonical involutive monoidal structure in the enrichment. Finally, we relate the results to applications in the design and semantics of reversible programming languages.

Keywords: reversible computing, dagger categories, domain theory, enriched category theory

1 Introduction

Dagger categories are categories that are canonically self-dual, assigning to each morphism an *adjoint* morphism in a contravariantly functorial way. In recent years, dagger categories have been used to capture central aspects of both reversible [28,29,31] and quantum [2,35,13] computing. Likewise, domain theory and categories enriched in domains (see, *e.g.*, [3,15,16,4,7,38]) have been successful since their inception in modelling both recursive functions and data types in programming.

In the present paper, we develop the art of dual wielding the two daggers that arise from respectively dagger category theory and domain theory (where the very same †-symbol is occasionally used to denote fixed points, *cf.* [15,16]). Concretely, we ask how these structures must interact in order to guarantee that fixed points are well-behaved with respect to the dagger, in the sense that each functional has a *fixed point adjoint* [31]. Previously, the author and others showed that certain

[1] Email: robin@di.ku.dk

[2] The author would like to thank Martti Karvonen, Mathys Rennela, and Robert Glück for their useful comments, corrections, and suggestions; and to acknowledge the support given by *COST Action IC1405 Reversible computation: Extending horizons of computing.*

domain enriched dagger categories, join inverse categories, had such well-behaved fixed points [31]. Here, we identify a sufficient condition for fixed points to be well-behaved in the presence of a dagger, allowing us not only to generalize previous results, but also to show new ones about parametrized fixed points.

A slogan of domain theory could be that *well-behaved functions are continuous* – and as a corollary, that *well-behaved functors are locally continuous*. When augmented with a dagger, the proper addendum to this slogan turns out to be that *well-behaved inversion is monotone*, captured in the definition of a monotone dagger structure.

Given a domain enriched category $\mathscr{C}$ with a monotone dagger structure, we develop an induced involutive monoidal category of domains enriching $\mathscr{C}$, which we think of as the category of *continuous functionals* on $\mathscr{C}$. This canonically constructed involutive structure at the level of functionals proves fruitful in unifying seemingly disparate concepts from the literature under the banner of *conjugation of functionals*. Notably, we show that the conjugate functionals arising from this involutive structure coincide with *fixed point adjoints* [5,31], and that they occur naturally both in proving the ambidexterity of dagger adjunctions [23] and in natural transformations that preserve the dagger (which include dagger traces [36]).

While these results could be applied to model a reversible functional programming language with general recursion and parametrized functions (such as an extended version of Theseus [29]), they are general enough to account for even certain probabilistic and nondeterministic models of computation.

Overview: A brief introduction to the relevant background material on dagger categories, (**DCPO**-)enriched categories, iteration categories, and involutive monoidal categories is given in Section 2. In Section 3 the concept of a *monotone dagger structure* on a **DCPO**-category is introduced, and it is demonstrated that such a structure leads to the existence of fixed point adjoints for (ordinary and externally parametrized) fixed points, given by their conjugates. We also explore natural transformations in this setting, and develop a notion of *self-conjugate* natural transformations, of which †-trace operators are examples. Finally, we discuss potential applications and avenues for future research in Section 4, and end with a few concluding remarks in Section 5.

2 Background

Though familiarity with basic category theory, including monoidal categories, is assumed, we recall here some basic concepts relating to dagger categories, (**DCPO**)-enriched categories, iteration categories, and involutive monoidal categories [26,8]. The material is only covered here briefly, but can be found in much more detail in the numerous texts on dagger category theory (see, *e.g.*, [35,2,21]), enriched category theory (for which [33] is the standard text), and domain theory and iteration categories (see, *e.g.*, [3,16]).

2.1 Dagger categories

A dagger category (or †-category) is a category equipped with a suitable method for flipping the direction of morphisms, by assigning to each morphism an *adjoint* in a

manner consistent with composition. They are formally defined as follows.

Definition 2.1 A dagger category is a category $\mathscr{C}$ equipped with an functor $(-)^\dagger : \mathscr{C}^{\mathrm{op}} \to \mathscr{C}$ satisfying that $\mathrm{id}_X^\dagger = \mathrm{id}_X$ and $f^{\dagger\dagger} = f$ for all identities $X \xrightarrow{\mathrm{id}_X} X$ and morphisms $X \xrightarrow{f} Y$.

Dagger categories, dagger functors (*i.e.*, functors F satisfying $F(f^\dagger) = F(f)^\dagger$), and natural transformations form a 2-category, **DagCat**.

A given category may have several different daggers which need not agree. An example of this is the groupoid of finite-dimensional Hilbert spaces and linear isomorphisms, which has (at least!) two daggers: One maps linear isomorphisms to their linear inverse, the other maps linear isomorphisms to their hermitian conjugate. The two only agree on the unitaries, *i.e.*, the linear isomorphisms which additionally preserve the inner product. For this reason, one would in principle need to specify *which* dagger one is talking about on a given category, though this is often left implicit (as will also be done here).

Let us recall the definition of the some interesting properties of morphisms in a dagger category: By theft of terminology from linear algebra, say that a morphism $X \xrightarrow{f} Y$ in a dagger category is *hermitian* or *self-adjoint* if $f = f^\dagger$, and *unitary* if it is an isomorphism and $f^{-1} = f^\dagger$. Whereas objects are usually considered equivalent if they are isomorphic, the "way of the dagger" [23] dictates that all structure in sight must cooperate with the dagger; as such, objects ought to be considered equivalent in dagger categories only if they are isomorphic via a unitary map.

We end with a few examples of dagger categories. As discussed above, **FHilb** is an example (*the* motivating one, even [35]) of dagger categories, with the dagger given by hermitian conjugation. The category **PInj** of sets and partial injective functions is a dagger category (indeed, it is an *inverse category* [32,12]) with $f^\dagger$ given by the partial inverse of f. Similarly, the category **Rel** of sets and relations has a dagger given by $R^\dagger = R^\circ$, *i.e.*, the relational converse of R. Noting that a dagger subcategory is given by the existence of a faithful dagger functor, it can be shown that **PInj** is a dagger subcategory of **Rel** with the given dagger structures.

2.2 **DCPO**-*categories and other enriched categories*

Enriched categories (see, *e.g.*, [33]) capture the idea that homsets on certain categories can (indeed, ought to) be understood as something other than sets – or in other words, as objects of a another category than **Set**. A category $\mathscr{C}$ is *enriched* in a monoidal category $\mathscr{V}$ if all homsets $\mathscr{C}(X,Y)$ of $\mathscr{C}$ are objects of $\mathscr{V}$, and for all objects X, Y, Z of $\mathscr{C}$, $\mathscr{V}$ has families of morphisms $\mathscr{C}(Y,Z) \otimes \mathscr{C}(X,Y) \to \mathscr{C}(X,Z)$ and $I \to \mathscr{C}(X,X)$ corresponding to composition and identities in $\mathscr{C}$, subject to commutativity of diagrams corresponding to the usual requirements of associativity of composition, and of left and right identity. As is common, we will often use the shorthand "$\mathscr{C}$ is a $\mathscr{V}$-category" to mean that $\mathscr{C}$ is enriched in the category $\mathscr{V}$.

We focus here on categories enriched in the category of *domains* (see, *e.g.*, [3]), *i.e.*, the category **DCPO** of pointed directed complete partial orders and continuous maps. A partially ordered $(X, \sqsubseteq)$ is said to be directed complete if every directed set (*i.e.*, a *non-empty* $A \subseteq X$ satisfying that any pair of elements of A has a

supremum in A) has a supremum in X. A function f between directed complete partial orders is monotone if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y, and continuous if $f(\sup A) = \sup_{a \in A}\{f(a)\}$ for each directed set A (note that continuity implies monotony). A directed complete partial order is *pointed* if it has a least element $\bot$ (or, in other words, if also the empty set has a supremum), and a function f between such is called *strict* if $f(\bot) = \bot$ (*i.e.*, if also the supremum of the empty set is preserved [3]). Pointed directed complete partial orders and continuous maps form a category, **DCPO**.

As such, a category enriched in **DCPO** is a category $\mathscr{C}$ in which homsets $\mathscr{C}(X, Y)$ are directed complete partial orders, and composition is continuous. Additionally, we will require that composition is strict (meaning that $\bot \circ f = \bot$ and $g \circ \bot = \bot$ for all suitable morphisms f and g), so that the category is actually enriched in the category **DCPO**! of directed complete partial orders and strict continuous functions, though we will not otherwise require functions to be strict.

Enrichment in **DCPO** provides a method for constructing morphisms in the enriched category as least fixed points of continuous functions between homsets: This is commonly used to model recursion. Given a continuous function $\mathscr{C}(X, Y) \xrightarrow{\varphi} \mathscr{C}(X, Y)$, by Kleene's fixed point theorem there exists a least fixed point $X \xrightarrow{\operatorname{fix}\varphi} Y$ given by $\sup_{n \in \omega}\{\varphi^n(\bot)\}$, where φ^n is the n-fold composition of φ with itself.

2.3 Parametrized fixed points and iteration categories

Related to the fixed point operator is the *parametrized fixed point operator*, an operator pfix assigning morphisms of the form $X \times Y \xrightarrow{\psi} X$ to a morphism $Y \xrightarrow{\operatorname{pfix}\psi} X$ satisfying equations such as the *parametrized fixed point identity*

$$\operatorname{pfix}\psi = \psi \circ \langle \operatorname{pfix}\psi, \mathrm{id}_Y \rangle$$

and others (see, *e.g.*, [25,15]). Parametrized fixed points are used to solve domain equations of the form $x = \psi(x, p)$ for some given parameter $p \in Y$. Indeed, if for a continuous function $X \times Y \xrightarrow{\psi} X$ we define $\psi^0(x, p) = x$ and $\psi^{n+1}(x, p) = \psi(\psi^n(x, p), p)$, we can construct its parametrized fixed point in **DCPO** in a way reminiscent of the usual fixed point by

$$(\operatorname{pfix}\psi)(p) = \sup_{n \in \omega}\{\psi^n(\bot_X, p)\} \ .$$

In fact, a parametrized fixed point operator may be derived from an ordinary fixed point operator by $(\operatorname{pfix}\psi)(p) = \operatorname{fix}\psi(-, p)$. Similarly, we may derive an ordinary fixed point operator from a parametrized one by considering a morphism $X \xrightarrow{\varphi} X$ to be parametrized by the terminal object 1, so that the fixed point of $X \xrightarrow{\varphi} X$ is given by the parametrized fixed point of $X \times 1 \xrightarrow{\pi_1} X \xrightarrow{\varphi} X$.

The parametrized fixed point operation is sometimes also called a *dagger operation* [15], and denoted by $f^\dagger$ rather than $\operatorname{pfix} f$. Though this is indeed the other

[3] This is *not* the case in general, as continuous functions are only required to preserve least upper bounds of directed sets, which, by definition, does not include the empty set.

dagger that we are wielding, we will use the phrase "parametrized fixed point" and notation "pfix" to avoid unnecessary confusion.

An *iteration category* [16] is a cartesian category with a parametrized fixed point operator that behaves in a canonical way. The definition of an iteration category came out of the observation that the parametrized fixed point operator in a host of concrete categories (notably **DCPO**) satisfy the same identities. This lead to an elegant semantic characterization of iteration categories, due to [16].

Definition 2.2 An *iteration category* is a cartesian category with a parametrized fixed point operator satisfying all identities (of the parametrized fixed point operator) that hold in **DCPO**.

Note that the original definition defined iteration categories in relation to the category $\mathbf{CPO}_m$ of ω-complete partial orders and monotone functions, rather than to **DCPO**. However, the motivating theorem [16, Theorem 1] shows that the parametrized fixed point operator in $\mathbf{CPO}_m$ satisfies the same identities as the one found in **CPO** (*i.e.*, with continuous rather than monotone functions). Since the parametrized fixed point operator of **DCPO** is constructed *precisely* as it is in **CPO** (noting that ω-chains are directed sets), this definition is equivalent to the original.

2.4 Involutive monoidal categories

An involutive category [26] is a category in which every object X can be assigned a *conjugate* object $\overline{X}$ in a functorial way such that $\overline{\overline{X}} \cong X$. A novel idea by Egger [14] is to consider dagger categories as categories enriched in an *involutive monoidal category*. We will return to this idea in Section 3.1, and recall the relevant definitions in the meantime (due to [26], compare also with *bar categories* [8]).

Definition 2.3 A category $\mathscr{V}$ is *involutive* if it is equipped with a functor $\mathscr{V} \xrightarrow{\overline{(-)}} \mathscr{V}$ (the *involution*) and a natural isomorphism $\mathrm{id} \overset{\iota}{\Rightarrow} \overline{\overline{(-)}}$ satisfying $\iota_{\overline{X}} = \overline{\iota_X}$.

Borrowing terminology from linear algebra, we call $\overline{X}$ (respectively $\overline{f}$) the *conjugate* of an object X (respectively a morphism f), and say that an object X is *self-conjugate* if $X \cong \overline{X}$. Note that since conjugation is covariant, any category $\mathscr{C}$ can be made involutive by assigning $\overline{X} = X$, $\overline{f} = f$, and letting $\mathrm{id} \overset{\iota}{\Rightarrow} \overline{\overline{(-)}}$ be the identity in each component; as such, an involution is a structure rather than a property. Non-trivial examples of involutive categories include the category of complex vector spaces $\mathbf{Vect}_{\mathbb{C}}$, with the involution given by the usual conjugation of complex vector spaces; and the category **Poset** of partially ordered sets and monotone functions, with the involution given by order reversal.

When a category is both involutive and (symmetric) monoidal, we say that it is an *involutive (symmetric) monoidal category* when these two structures play well together, as in the following definition [26].

Definition 2.4 An *involutive (symmetric) monoidal category* is a (symmetric) monoidal category $\mathscr{V}$ which is also involutive, such that the involution is a monoidal functor, and $\mathrm{id} \Rightarrow \overline{\overline{(-)}}$ is a monoidal natural isomorphism.

This specifically gives us a natural family of isomorphisms $\overline{X \otimes Y} \cong \overline{X} \otimes \overline{Y}$, and when the monoidal product is symmetric, this extends to a natural isomorphism $\overline{X \otimes Y} \cong \overline{Y} \otimes \overline{X}$. This fact will turn out to be useful later on when we consider dagger categories as enriched in certain involutive symmetric monoidal categories.

3 Domain enriched dagger categories

Given a dagger category that also happens to be enriched in domains, we ask how these two structures ought to interact with one another. Since domain theory dictates that the well-behaved functions are precisely the continuous ones, a natural first answer would be to that the dagger should be locally continuous; however, it turns out that we can make do with less.

Definition 3.1 Say that a dagger structure on **DCPO**-category is *monotone* if the dagger is locally monotone, *i.e.*, if $f \sqsubseteq g$ implies $f^\dagger \sqsubseteq g^\dagger$ for all f and g.

In the following, we will use the terms "**DCPO**-category with a monotone dagger structure" and "**DCPO**-†-category" interchangably. That this is sufficient to get what we want – in particular to obtain local continuity of the dagger – is shown in the following lemma.

Lemma 3.2 *In any* **DCPO**-†*-category, the dagger is an order isomorphism on morphisms; in particular it is continuous and strict.*

Proof. For $\mathscr{C}$ a dagger category, $\mathscr{C} \cong \mathscr{C}^{\mathrm{op}}$ so $\mathscr{C}(X,Y) \cong \mathscr{C}^{\mathrm{op}}(X,Y) = \mathscr{C}(Y,X)$ for all objects X, Y; that this isomorphism of hom-objects is an order isomorphism follows directly by local monotony. □

Let us consider a few examples of **DCPO**-†-categories.

Example 3.3 The category **Rel** of sets and relations is a dagger category, with the dagger given by $R^\dagger = R^\circ$, the relational converse of R (*i.e.*, defined by $(y,x) \in R^\circ$ iff $(x,y) \in R$) for each such relation. It is also enriched in **DCPO** by the usual subset ordering: Since a relation $\mathcal{X} \to \mathcal{Y}$ is nothing more than a subset of $\mathcal{X} \times \mathcal{Y}$, equipped with the subset order $- \subseteq -$ we have that $\sup(\Delta) = \bigcup_{R \in \Delta} R$ for any directed set $\Delta \subseteq \mathbf{Rel}(\mathcal{X}, \mathcal{Y})$. It is also pointed, with the least element of each homset given by the empty relation.

To see that this is a monotone dagger structure, let $\mathcal{X} \xrightarrow{R,S} \mathcal{Y}$ be relations and suppose that $R \subseteq S$. Let $(y,x) \in R^\circ$.Since $(y,x) \in R^\circ$ we have $(x,y) \in R$ by definition of the relational converse, and by the assumption that $R \subseteq S$ we also have $(x,y) \in S$. But then $(y,x) \in S^\circ$ by definition of the relational converse, so $R^\dagger = R^\circ \subseteq S^\circ = S^\dagger$ follows by extensionality.

Example 3.4 We noted earlier that the category **PInj** of sets and partial injective functions is a dagger subcategory of **Rel**, with $f^\dagger$ given by the partial inverse (a special case of the relational converse) of a partial injection f. Further, it is also a **DCPO**-subcategory of **Rel**; in **PInj**, this becomes the relation that for $X \xrightarrow{f,g} Y$, $f \sqsubseteq g$ iff for all $x \in X$, if f is defined at x and $f(x) = y$, then g is also defined at x and $g(x) = y$. Like **Rel**, it is pointed with the nowhere defined partial function as

the least element of each homset. That $\sup(\Delta)$ for some directed $\Delta \subseteq \mathbf{PInj}(X, Y)$ is a partial injection follows straightforwardly, and that this dagger structure is monotone follows by an argument analogous to the one for **Rel**.

Example 3.5 More generally, any *join inverse category* (see [17]), of which **PInj** is one, is a **DCPO**-†-category. Inverse categories are canonically dagger categories enriched in partial orders. That this extends to **DCPO**-enrichment in the presence of joins is shown in [5,31]; that the canonical dagger is monotonous with respect to the partial order is an elementary result (see, *e.g.*, [5, Lemma 2]).

Example 3.6 The category $\mathbf{DStoch}_{\leq 1}$ of finite sets and *doubly substochastic maps* is an example of a probabilistic **DCPO**-†-category. A doubly substochastic map $X \xrightarrow{f} Y$, where $|X| = |Y| = n$, is given by an $n \times n$ matrix $A = [a_{ij}]$ with non-negative real entries such that $\sum_{i=1}^{n} a_{ij} \leq 1$ and $\sum_{j=1}^{n} a_{ij} \leq 1$. Composition is given by the usual multiplication of matrices.

This is a dagger category with the dagger given by matrix transposition. It is also enriched in **DCPO** by ordering doubly substochastic maps entry-wise (*i.e.*, $A \leq B$ if $a_{ij} \leq b_{ij}$ for all i, j), with the everywhere-zero matrix as the least element in each homset, and with suprema of directed sets given by computing suprema entry-wise. That this dagger structure is monotone follows by the fact that if $A \leq B$, so $a_{ij} \leq b_{ij}$ for all i, j, then also $a_{ji} \leq b_{ji}$ for all j, i, which is precisely to say that $A^\dagger = A^T \leq B^T = B^\dagger$.

As such, in terms of computational content, these are examples of deterministic, nondeterministic, and probabilistic **DCPO**-†-categories. We will also discuss the related category $\mathbf{CP}^*(\mathbf{FHilb})$, used to model quantum phenomena, in Section 4.

3.1 The category of continuous functionals

We illustrate here the idea of dagger categories as categories enriched in an involutive monoidal category by an example that will be used throughout the remainder of this article: Enrichment in a suitable subcategory of **DCPO**. It is worth stressing, however, that the construction is *not* limited to dagger categories enriched in **DCPO**; any dagger category will do. As we will see later, however, this canonical involution turns out to be very useful when **DCPO**-†-categories are considered.

Let $\mathscr{C}$ be a **DCPO**-†-category. We define an induced (full monoidal) subcategory of **DCPO**, call it $\mathbf{DcpoOp}(\mathscr{C})$, which enriches $\mathscr{C}$ (by its definition) as follows:

Definition 3.7 For a **DCPO**-†-category $\mathscr{C}$, define $\mathbf{DcpoOp}(\mathscr{C})$ to have as objects all objects Θ, Λ of **DCPO** of the form $\mathscr{C}(X, Y)$, $\mathscr{C}^{\mathrm{op}}(X, Y)$ (for all objects X, Y of $\mathscr{C}$), 1, and $\Theta \times \Lambda$ (with 1 initial object of **DCPO**, and $- \times -$ the cartesian product), and as morphisms all continuous functions between these.

In other words, $\mathbf{DcpoOp}(\mathscr{C})$ is the (full) cartesian subcategory of **DCPO** generated by objects used in the enrichment of $\mathscr{C}$, with all continuous maps between these. That the dagger on $\mathscr{C}$ induces an involution on $\mathbf{DcpoOp}(\mathscr{C})$ is shown in the following theorem.

Theorem 3.8 $\mathbf{DcpoOp}(\mathscr{C})$ *is an involutive symmetric monoidal category.*

Proof. On objects, define an involution $\overline{(-)}$ with respect to the cartesian (specifically symmetric monoidal) product of **DCPO** as follows, for all objects Θ, Λ, Σ of $\mathbf{DcpoOp}(\mathscr{C})$: $\overline{\mathscr{C}(X,Y)} = \mathscr{C}^{\mathrm{op}}(X,Y)$, $\overline{\mathscr{C}^{\mathrm{op}}(X,Y)} = \mathscr{C}(X,Y)$, $\overline{1} = 1$, and $\overline{\Theta \times \Lambda} = \overline{\Theta} \times \overline{\Lambda}$. To see that this is well-defined, recall that $\mathscr{C} \cong \mathscr{C}^{\mathrm{op}}$ for any dagger category $\mathscr{C}$, so in particular there is an isomorphism witnessing $\mathscr{C}(X,Y) \cong \mathscr{C}^{\mathrm{op}}(X,Y)$ given by the mapping $f \mapsto f^\dagger$. But then $\mathscr{C}^{\mathrm{op}}(X,Y) = \{f^\dagger \mid f \in \mathscr{C}(X,Y)\}$, so if $\mathscr{C}(X,Y) = \mathscr{C}(X',Y')$ then $\overline{\mathscr{C}(X,Y)} = \mathscr{C}^{\mathrm{op}}(X,Y) = \{f^\dagger \mid f \in \mathscr{C}(X,Y)\} = \{f^\dagger \mid f \in \mathscr{C}(X',Y')\} = \mathscr{C}^{\mathrm{op}}(X',Y') = \overline{\mathscr{C}(X',Y')}$. That $\overline{\mathscr{C}^{\mathrm{op}}(X,Y)} = \mathscr{C}(X,Y)$ is well-defined follows by analogous argument.

On morphisms, we define a family ξ of isomorphisms by $\xi_I = \mathrm{id}_I$, $\xi_{\mathscr{C}(X,Y)} = (-)^\dagger$, $\xi_{\mathscr{C}^{\mathrm{op}}(X,Y)} = (-)^\dagger$, and $\xi_{\Theta\times\Lambda} = \xi_\Theta \times \xi_\Lambda$, and then define

$$\overline{\Theta \xrightarrow{\varphi} \Lambda} = \overline{\Theta} \xrightarrow{\xi_\Theta^{-1}} \Theta \xrightarrow{\varphi} \Lambda \xrightarrow{\xi_\Lambda} \overline{\Lambda}.$$

This is functorial as $\overline{\mathrm{id}_\Theta} = \xi_\Theta \circ \mathrm{id}_\Theta \circ \xi_\Theta^{-1} = \xi_\Theta \circ \xi_\Theta^{-1} = \mathrm{id}_{\overline{\Theta}}$, and for $\Theta \xrightarrow{\varphi} \Lambda \xrightarrow{\psi} \Sigma$,

$$\overline{\psi \circ \varphi} = \xi_\Sigma \circ \psi \circ \varphi \circ \xi_\Theta^{-1} = \xi_\Sigma \circ \psi \circ \xi_\Lambda^{-1} \circ \xi_\Lambda \circ \varphi \circ \xi_\Theta^{-1} = \overline{\psi} \circ \overline{\varphi}.$$

Finally, since the involution is straightforwardly a monoidal functor, and since the natural transformation $\mathrm{id} \Rightarrow \overline{\overline{(-)}}$ can be chosen to be the identity since all objects of $\mathbf{DcpoOp}(\mathscr{C})$ satisfy $\overline{\overline{\Theta}} = \Theta$ by definition, this is an involutive symmetric monoidal category. □

The resulting category $\mathbf{DcpoOp}(\mathscr{C})$ can very naturally be thought of as the induced *category of (continuous) functionals* (or second-order functions) of $\mathscr{C}$.

Notice that this is a special case of a more general construction on dagger categories: For a dagger category $\mathscr{C}$ enriched in some category $\mathscr{V}$ (which could simply be **Set** in the unenriched case), one can construct the category $\mathscr{V}\mathbf{Op}(\mathscr{C})$, given on objects by the image of the hom-functor $\mathscr{C}(-,-)$ closed under monoidal products, and on morphisms by all morphisms of $\mathscr{V}$ between objects of this form. Defining the involution as above, $\mathscr{V}\mathbf{Op}(\mathscr{C})$ can be shown to be involutive monoidal.

Example 3.9 One may question how natural (in a non-technical sense) the choice of involution on $\mathbf{DcpoOp}(\mathscr{C})$ is. One instance where it turns out to be useful is in the context of dagger adjunctions (see [23] for details), that is, adjunctions between dagger categories where both functors are dagger functors.

Dagger adjunctions have no specified left and right adjoint, as all such adjunctions can be shown to be ambidextrous in the following way: Given $F \dashv G$ between endofunctors on $\mathscr{C}$, there is a natural isomorphism $\mathscr{C}(FX,Y) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(X,GY)$. Since $\mathscr{C}$ is a dagger category, we can define a natural isomorphism $\mathscr{C}(X,FY) \xrightarrow{\beta_{X,Y}} \mathscr{C}(GX,Y)$ by $f \mapsto \alpha_{Y,X}(f^\dagger)^\dagger$, *i.e.*, by the composition

$$\mathscr{C}(X,FY) \xrightarrow{\xi} \mathscr{C}(FY,X) \xrightarrow{\alpha_{Y,X}} \mathscr{C}(Y,GX) \xrightarrow{\xi} \mathscr{C}(GX,Y)$$

which then witnesses $G \dashv F$ (as it is a composition of natural isomorphisms). But then $\beta_{X,Y}$ is defined precisely to be $\overline{\alpha_{Y,X}}$ when F and G are endofunctors.

3.2 Daggers and fixed points

In this section we consider the morphisms of $\mathbf{DcpoOp}(\mathscr{C})$ in some detail, for a **DCPO**-†-category $\mathscr{C}$. Since least fixed points of morphisms are such a prominent and useful feature of **DCPO**-enriched categories, we ask how these behave with respect to the dagger. To answer this question, we transplant the notion of a *fixed point adjoint* from [5,31] to **DCPO**-†-categories, where an answer to this question in relation to the more specific *join inverse categories* was given:

Definition 3.10 A functional $\mathscr{C}(Y,X) \xrightarrow{\varphi_\ddagger} \mathscr{C}(Y,X)$ is *fixed point adjoint* to a functional $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ iff $(\operatorname{fix}\varphi)^\dagger = \operatorname{fix}\varphi_\ddagger$.

Note that this is symmetric: If $\varphi_\ddagger$ is fixed point adjoint to φ then $\operatorname{fix}(\varphi_\ddagger)^\dagger = (\operatorname{fix}\varphi)^{\dagger\dagger} = \operatorname{fix}\varphi$, so φ is also fixed point adjoint to $\varphi_\ddagger$. As shown in the following theorem, it turns out that the conjugate $\overline{\varphi}$ of a functional φ is precisely fixed point adjoint to it. This is a generalization of a theorem from [31], where a more ad-hoc formulation was shown for join inverse categories, which constitute a non-trivial subclass of **DCPO**-†-categories.

Theorem 3.11 *Every functional is fixed point adjoint to its conjugate.*

Proof. The proof applies the exact same construction as in [31], since being a **DCPO**-†-category suffices, and the constructed fixed point adjoint turns out to be the exact same. Let $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ be a functional. Since $\overline{\varphi} = \xi_{\mathscr{C}(X,Y)} \circ \varphi \circ \xi^{-1}_{\mathscr{C}(X,Y)}$,

$$\bar{\varphi}^n = \left(\xi_{\mathscr{C}(X,Y)} \circ \varphi \circ \xi^{-1}_{\mathscr{C}(X,Y)}\right)^n = \xi_{\mathscr{C}(X,Y)} \circ \varphi^n \circ \xi^{-1}_{\mathscr{C}(X,Y)}$$

and so

$$\begin{aligned}\operatorname{fix}\overline{\varphi} &= \sup\{\bar{\varphi}^n(\bot_{Y,X})\}_{n\in\omega} = \sup\{\varphi^n(\bot^\dagger_{Y,X})^\dagger\} = \sup\{\varphi^n(\bot_{X,Y})^\dagger\} \\ &= \sup\{\varphi^n(\bot_{X,Y})\}^\dagger = (\operatorname{fix}\varphi)^\dagger\end{aligned}$$

as desired. □

This theorem is somewhat surprising, as the conjugate came out of the involutive monoidal structure on $\mathbf{DcpoOp}(\mathscr{C})$, which is not specifically related to the presence of fixed points. As previously noted, had $\mathscr{C}$ been enriched in another category $\mathscr{V}$, we would still be able to construct a category $\mathscr{V}\mathbf{Op}(\mathscr{C})$ of $\mathscr{V}$-functionals with the *exact same* involutive structure.

As regards recursion, this theorem underlines the slogan that *reversibility is a local phenomenon*: To construct the inverse to a recursively defined morphism $\operatorname{fix}\varphi$, it suffices to invert the local morphism φ at each step (which is essentially what is done by the conjugate $\overline{\varphi}$) in order to construct the global inverse $(\operatorname{fix}\varphi)^\dagger$.

Parametrized functionals and their external fixed points are also interesting to consider in this setting, as some examples of **DCPO**-†-categories (*e.g.*, **PInj**) fail to have an internal hom. For example, in a dagger category with objects $L(X)$ corresponding to "lists of X" (usually constructed as the fixed point of a suitable functor), one could very reasonably construe the usual map-function not

as a higher-order function, but as a family of morphisms $LX \xrightarrow{\mathrm{map}\langle f\rangle} LY$ indexed by $X \xrightarrow{f} Y$ – or, in other words, as a functional $\mathscr{C}(X,Y) \xrightarrow{\mathrm{map}} \mathscr{C}(LX,LY)$. Indeed, this is how certain higher-order behaviours are mimicked in the reversible functional programming language Theseus (see also Section 4).

To achieve such parametrized fixed points of functionals, we naturally need a parametrized fixed point operator on $\mathbf{DcpoOp}(\mathscr{C})$ satisfying the appropriate equations – or, in other words, we need $\mathbf{DcpoOp}(\mathscr{C})$ to be an *iteration category*. That $\mathbf{DcpoOp}(\mathscr{C})$ is such an iteration category follows immediately by its definition (*i.e.*, since $\mathbf{DcpoOp}(\mathscr{C})$ is a full subcategory of $\mathbf{DCPO}$, we can define a parametrized fixed point operator in $\mathbf{DcpoOp}(\mathscr{C})$ to be precisely the one in $\mathbf{DCPO}$), noting that parametrized fixed points preserve continuity.

Lemma 3.12 $\mathbf{DcpoOp}(\mathscr{C})$ *is an iteration category.*

For functionals of the form $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$, we can make a similar definition of a *parametrized fixed point adjoint*:

Definition 3.13 A functional $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi_\ddagger} \mathscr{C}(X,Y)$ is *parametrized fixed point adjoint* to a functional $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ iff $(\mathrm{pfix}\,\psi)(p)^\dagger = (\mathrm{pfix}\,\psi_\ddagger)(p^\dagger)$.

We can now show a similar theorem for parametrized fixed points of functionals and their conjugates:

Theorem 3.14 *Every functional is parametrized fixed point adjoint to its conjugate.*

Proof. Let $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ be a functional. We start by showing that $\bar{\psi}^n(f,p) = \psi^n(f^\dagger,p^\dagger)^\dagger$ for all $Y \xrightarrow{f} X$, $Q \xrightarrow{p} P$, and $n \in \mathbb{N}$, by induction on n. For $n = 0$ we have

$$\bar{\psi}^0(f,p) = f = f^{\dagger\dagger} = (f^\dagger)^\dagger = \psi^0(f^\dagger,p^\dagger)^\dagger.$$

Assuming now the induction hypothesis for some n, we have

$$\begin{aligned}\bar{\psi}^{n+1}(f,p) &= \bar{\psi}(\bar{\psi}^n(f,p),p) = \bar{\psi}(\psi^n(f^\dagger,p^\dagger)^\dagger,p) = \psi(\psi^n(f^\dagger,p^\dagger)^{\dagger\dagger},p^\dagger)^\dagger \\ &= \psi(\psi^n(f^\dagger,p^\dagger),p^\dagger)^\dagger = \psi^{n+1}(f^\dagger,p^\dagger)^\dagger\end{aligned}$$

Using this fact, we now get

$$\begin{aligned}(\mathrm{pfix}\,\overline{\psi})(p^\dagger) &= \sup_{n\in\omega}\{\bar{\psi}^n(\bot_{Y,X},p^\dagger)\} = \sup_{n\in\omega}\{\psi^n(\bot_{Y,X}^\dagger,p^{\dagger\dagger})^\dagger\} \\ &= \sup_{n\in\omega}\{\psi^n(\bot_{X,Y},p)\}^\dagger = (\mathrm{pfix}\,\psi)(p)^\dagger\end{aligned}$$

which was what we wanted. □

Again, this theorem highlights the local nature of reversibility, here in the presence of additional parameters. We observe further the following highly useful property of parametrized fixed points in $\mathbf{DcpoOp}(\mathscr{C})$:

Lemma 3.15 *Parametrized fixed points in* $\mathbf{DcpoOp}(\mathscr{C})$ *preserve conjugation.*

Proof. Let $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ be continuous, and $P \xrightarrow{p} Q$. Then $\overline{\operatorname{pfix}\psi}(p) = (\xi \circ (\operatorname{pfix}\psi) \circ \xi^{-1})(p) = (\operatorname{pfix}\psi)(p^\dagger)^\dagger = (\operatorname{pfix}\overline{\psi})(p)^{\dagger\dagger} = (\operatorname{pfix}\overline{\psi})(p)$, so $\overline{\operatorname{pfix}\psi} = \operatorname{pfix}\overline{\psi}$. □

Note that a lemma of this form only makes sense for parametrized fixed points, as the usual fixed point of a functional $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ results in a morphism $X \xrightarrow{\operatorname{fix}\varphi} Y$ in $\mathscr{C}$, not a functional in $\mathbf{DcpoOp}(\mathscr{C})$.

3.3 Naturality and self-conjugacy

We now consider the behaviour of functionals and their parametrized fixed points when they are natural. For example, given a natural family of functionals $\mathscr{C}(FX,FY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(GX,GY)$ natural in X and Y (for dagger endofunctors F and G on $\mathscr{C}$), what does it mean for such a family to be well-behaved with respect to the dagger on $\mathscr{C}$? We would certainly want that such a family preserves the dagger, in the sense that $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$ in each component X,Y. It turns out that this, too, can be expressed in terms of conjugation of functionals.

Lemma 3.16 *Let* $\mathscr{C}(FX,FY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(GX,GY)$ *be a family of functionals natural in* X *and* Y*. Then* $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$ *for all* $X \xrightarrow{f} Y$ *iff* $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$*.*

Proof. Suppose $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$. Then $\alpha_{X,Y}(f) = \alpha_{X,Y}(f)^{\dagger\dagger} = \alpha_{Y,X}(f^\dagger)^\dagger = \overline{\alpha_{Y,X}}(f)$, so $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$. Conversely, assuming $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$ we then have for all $X \xrightarrow{f} Y$ that $\alpha_{X,Y}(f) = \alpha_{Y,X}(f^\dagger)^\dagger$, so $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)^{\dagger\dagger} = \alpha_{Y,X}(f^\dagger)$. □

If a natural transformation α satisfies $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$ in all components X,Y, we say that it is *self-conjugate*. An important example of a self-conjugate natural transformation is the *dagger trace operator*, as detailed in the following example.

Example 3.17 A trace operator [30] on a braided monoidal category $\mathscr{D}$ is family of functionals

$$\mathscr{D}(X \otimes U, Y \otimes U) \xrightarrow{\operatorname{Tr}^U_{X,Y}} \mathscr{D}(X,Y)$$

subject to equations such as naturality in X and Y, dinaturality in U, and others. Traces have been used to model features from traces in tensorial vector spaces [20] to tail recursion in programming languages [1,9,19], and occur naturally in compact closed (or, more generally, tortile monoidal) categories [30] and unique decomposition categories [18,24].

A *dagger trace operator* on a dagger category (see, *e.g.*, [36]) is precisely a trace operator on a dagger monoidal category (*i.e.*, a monoidal category where the monoidal functor is a dagger functor) that satisfies $\operatorname{Tr}^U_{X,Y}(f)^\dagger = \operatorname{Tr}^U_{Y,X}(f^\dagger)$ in all components X,Y. Such traces have been used to model reversible tail recursion in reversible programming languages [28,29,31], and also occur in the *dagger compact closed categories* (see, *e.g.*, [37]) used to model quantum computation. In light of Lemma 3.16, dagger traces are important examples of self-conjugate natural transformations on dagger categories.

Given the connections between (di)naturality and parametric polymorphism [39,6], one would wish that parametrized fixed points preserve naturality. Luckily, this does turn out to be the case, as shown in the proof of the following theorem.

Theorem 3.18 *If $\mathscr{C}(FX,FY) \times \mathscr{C}(GX,GY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(FX,FY)$ is natural in X and Y, so is its parametrized fixed point.*

Proof. Suppose that α is natural in X and Y, *i.e.*, the following diagram commutes for all X, Y.

$$\begin{array}{ccc} \mathscr{C}(FX,FY) \times \mathscr{C}(GX,GY) & \xrightarrow{\alpha_{X,Y}} & \mathscr{C}(FX,FY) \\ {\scriptstyle Ff \times Gf \circ - \circ Fg \times Gg}\downarrow & & \downarrow{\scriptstyle Ff \circ - \circ Fg} \\ \mathscr{C}(FX',FY') \times \mathscr{C}(GX',GY') & \xrightarrow[\alpha_{X',Y'}]{} & \mathscr{C}(FX',FY') \end{array}$$

Under this assumption, we start by showing naturality of α^n for all $n \in \mathbb{N}$, *i.e.*, for all $GX \xrightarrow{p} GY$

$$\alpha^n_{X',Y'}(\bot_{X',Y'}, Gf \circ p \circ Gg) = Ff \circ \alpha^n_{X,Y}(\bot_{X,Y}, p) \circ Fg$$

by induction on n. For $n = 0$ we have

$$\begin{aligned} \alpha^0_{X',Y'}(\bot_{X,Y}, Gf \circ p \circ Gg) &= \bot_{X',Y'} \\ &= Ff \circ \bot_{X,Y} \circ Fg \\ &= Ff \circ \alpha^0_{X,Y}(\bot_{X,Y}, p) \circ Fg. \end{aligned}$$

where $Ff \circ \bot_{X,Y} \circ Fg = \bot_{X',Y'}$ by strictness of composition. Assuming the induction hypothesis now for some n, we have

$$\begin{aligned} \alpha^{n+1}_{X',Y'}(\bot_{X',Y'}, Gf \circ p \circ Gg) &= \alpha_{X',Y'}(\alpha^n_{X',Y'}(\bot_{X',Y'}, Gf \circ p \circ Gg), Gf \circ p \circ Gg) \\ &= \alpha_{X',Y'}(Ff \circ \alpha^n_{X,Y}(\bot_{X,Y}, p) \circ Fg, Gf \circ p \circ Gg) \\ &= Ff \circ \alpha_{X,Y}(\alpha^n_{X,Y}(\bot_{X,Y}, p), p) \circ Fg \\ &= Ff \circ \alpha^{n+1}_{X,Y}(\bot_{X,Y}, p) \circ Fg \end{aligned}$$

so α^n is, indeed, natural for any choice of $n \in \mathbb{N}$. But then

$$\begin{aligned} (\mathrm{pfix}\, \alpha_{X',Y'})(Gf \circ p \circ Gg) &= \sup_{n \in \omega} \left\{ \alpha^n_{X',Y'}(\bot_{X',Y'}, Gf \circ p \circ Gg) \right\} \\ &= \sup_{n \in \omega} \left\{ \alpha^n_{X',Y'}(Ff \circ \bot_{X,Y} \circ Fg, Gf \circ p \circ Gg) \right\} \\ &= \sup_{n \in \omega} \left\{ Ff \circ \alpha^n_{X,Y}(\bot_{X,Y}, p) \circ Fg \right\} \\ &= Ff \circ \sup_{n \in \omega} \left\{ \alpha^n_{X,Y}(\bot_{X,Y}, p) \right\} \circ Fg \\ &= Ff \circ (\mathrm{pfix}\, \alpha_{X,Y})(p) \circ Fg \end{aligned}$$

so $\mathrm{pfix}\, \alpha_{X,Y}$ is natural as well. □

This theorem can be read as stating that, just like reversibility, a recursive polymorphic map can be obtained from one that is only locally polymorphic. Com-

bining this result with Lemma 3.16 regarding self-conjugacy, we obtain the following corollary.

Corollary 3.19 *If* $\mathscr{C}(FX,FY)\times\mathscr{C}(GX,GY)\xrightarrow{\alpha_{X,Y}}\mathscr{C}(FX,FY)$ *is a self-conjugate natural transformation, so is* $\operatorname{pfix}\alpha_{X,Y}$.

Proof. If $\alpha_{X,Y}=\overline{\alpha_{Y,X}}$ for all X,Y then also $\operatorname{pfix}\alpha_{X,Y}=\operatorname{pfix}\overline{\alpha_{Y,X}}$, which is further natural in X and Y by Theorem 3.18. But then $\overline{\operatorname{pfix}\alpha_{X,Y}}=\operatorname{pfix}\overline{\alpha_{X,Y}}=\operatorname{pfix}\alpha_{Y,X}$, as parametrized fixed points preserve conjugation. □

4 Applications and future work

Reversible programming languages

Theseus [29] is a typed reversible functional programming language similar in syntax and spirit to Haskell. It has support for recursive data types, as well as reversible tail recursion using so-called *typed iteration labels* as syntactic sugar for a dagger trace operator. Theseus is based on the Π-family of reversible combinator calculi [28], which bases itself on dagger traced symmetric monoidal categories augmented with a certain class of algebraically ω-compact functors.

Theseus also supports *parametrized functions*, that is, families of reversible functions indexed by reversible functions of a given type, with the proviso that parameters must be passed to parametrized maps statically. For example, (if one extended Theseus with polymorphism) the reversible map function would have the signature $map :: (a \leftrightarrow b) \to ([a] \leftrightarrow [b])$, and so *map* is not in itself a reversible function, though $map\ \langle f\rangle$ is (for some suitable function f passed statically). This gives many of the benefits of higher-order programming, but without the headaches of higher-order reversible programming.

The presented results show very directly that we can extend Theseus with a fixed point operator for general recursion while maintaining desirable inversion properties, rather than making do with the simpler tail recursion. Additionally, the focus on the continuous functionals of $\mathscr{C}$ given by the category $\mathbf{DcpoOp}(\mathscr{C})$ also highlights the feature of parametrized functions in Theseus, and our results go further to show that even parametrized functions that use general recursion not only have desirable inversion properties, but also preserve naturality, the latter of which is useful for extending Theseus with parametric polymorphism.

Quantum programming languages

An interesting possibility as regards quantum programming languages is the category $\mathbf{CP}^*(\mathbf{FHilb})$ (see [13] for details on the $\mathbf{CP}^*$-construction), which is dagger compact closed and equivalent to the category of finite-dimensional C^*-algebras and completely positive maps [13]. Since finite-dimensional C^*-algebras are specifically von Neumann algebras, it follows (see [10,34]) that this category is enriched in the category of *bounded* directed complete partial orders; and since it inherits the dagger from **FHilb** (and is locally ordered by the pointwise extension of the Löwner order restricted to positive operators), the dagger structure is monotone, too. As such, the presented results ought to apply in this case as well – modulo concerns of boundedness – though this warrants more careful study.

Dagger traces in DCPO-†-categories

Given a suitable monoidal tensor (*e.g.*, one with the zero object as tensor unit) and a partial additive structure on morphisms, giving the category the structure of a *unique decomposition category* [18,24], a trace operator can be constructed by means of the so-called *trace formula*

$$\mathrm{Tr}^U_{X,Y}(f) = f_{11} + \sum_{n\in\omega} f_{21} \circ f_{22}^n \circ f_{12}$$

where $f_{mn} = \rho_m \circ f \circ \iota_n$, with $X_n \xrightarrow{\iota_n} \bigoplus_{i\in I} X_i$ and $\bigoplus_{i\in I} X_i \xrightarrow{\rho_m} X_m$ are families of canonical *quasi-injections* respectively *quasi-projections* of the monoidal tensor. In previous work [5,31], the author (among others) demonstrated that a certain class of **DCPO**-†-categories, namely join inverse categories, had a dagger trace under suitably mild assumptions. It is conjectured that this theorem may be generalized to other **DCPO**-†-categories that are not necessarily inverse categories, again provided that certain assumptions are satisfied.

Involutive iteration categories

As it turned out that the category $\mathbf{DcpoOp}(\mathscr{C})$ of continuous functionals on $\mathscr{C}$ was both involutive and an iteration category, an immediate question to ask is how the involution functor ought to interact with parametrized fixed points in the general case. A remarkable fact of iteration categories is that they are defined to be cartesian categories that satisfy all equations of parametrized fixed points that hold in the category $\mathbf{CPO}_m$ of ω-complete partial orders and *monotone* functions, yet also have a complete (though infinite) equational axiomatization [16].

We have provided an example of an interaction between parametrized fixed points and the involution functor here, namely that $\mathbf{DcpoOp}(\mathscr{C})$ satisfies $\overline{\mathrm{pfix}\,\psi} = \mathrm{pfix}\,\overline{\psi}$. It could be interesting to search for examples of involutive iteration categories in the wild (as candidates for a semantic definition), and to see if Ésik's axiomatization could be extended to accomodate for the involution functor in the semantic category.

Algebraic compactness of dagger functors

Another useful feature of categories enriched in domains, as shown independently by Adámek [4] and Barr [7], is the algebraic compactness of locally continuous functors, provided that certain (co)completeness requirements are met. Since the way of the dagger dictates that fixed points of (dagger) functors ought to be unique up to *unitaries* (rather than up to any old isomorphism), the entire machinery of **DCPO**-categories developed for this purpose needs readjustment in order to accomodate for this requirement of uniqueness up to unitary maps. This is the topic of another paper by the author.

5 Conclusion and related work

We have developed a notion of **DCPO**-categories with a monotone dagger structure (of which **PInj**, **Rel**, and $\mathbf{DStoch}_{\leq 1}$ are examples, and $\mathbf{CP}^*(\mathbf{FHilb})$ is closely related), and shown that these categories can be taken to be enriched in an induced

involutive monoidal category of continuous functionals. With this, we were able to account for (ordinary and parametrized) fixed point adjoints as arising from conjugation of the functional in the induced involutive monoidal category, to show that parametrized fixed points preserve conjugation and naturality, and that natural transformations that preserve the dagger are precisely those that are self-conjugate. We also described a number of potential applications in connection with reversible and quantum computing.

A great deal of work has been carried out in recent years on the domain theory of quantum computing, with noteworthy results in categories of von Neumann algebras (see, *e.g.*, [34,10,27,11]). Though the interaction between dagger structure and the domain structure on homsets was not the object of study, Heunen considers the similarities and differences of **FHilb** and **PInj**, also in relation to domain structure on homsets, in [22], though he also notes that **FHilb** fails to enrich in domains as composition is not even monotone (this is not to say that domain theory and quantum computing do not mix; only that **FHilb** is the wrong category to consider for this purpose). Finally, dagger traced symmetric monoidal categories, with the dagger trace serving as an operator for reversible tail recursion, have been studied in connection with reversible combinator calculi [28] and functional programming [29].

References

[1] Abramsky, S., *Retracing some paths in process algebra*, in: U. Montanari and V. Sassone, editors, *CONCUR '96*, Springer, 1996 pp. 1–17.

[2] Abramsky, S. and B. Coecke, *A categorical semantics of quantum protocol*, in: *Logic in Computer Science, 2004, Proceedings*, IEEE, 2004, pp. 415–425.

[3] Abramsky, S. and A. Jung, *Domain theory*, in: S. Abramsky, D. Gabbay and T. Maibaum, editors, *Handbook of Logic in Computer Science*, 3, Clarendon Press, 1994 pp. 1–168.

[4] Adámek, J., *Recursive data types in algebraically ω-complete categories*, Information and Computation **118** (1995), pp. 181–190.

[5] Axelsen, H. B. and R. Kaarsgaard, *Join inverse categories as models of reversible recursion*, in: B. Jacobs and C. Löding, editors, *FOSSACS 2016, Proceedings* (2016), pp. 73–90.

[6] Bainbridge, E. S., P. J. Freyd, A. Scedrov and P. J. Scott, *Functorial polymorphism*, Theoretical Computer Science **70** (1990), pp. 35–64.

[7] Barr, M., *Algebraically compact functors*, Journal of Pure and Applied Algebra **82** (1992), pp. 211–231.

[8] Beggs, E. J. and S. Majid, *Bar categories and star operations*, Algebras and Representation Theory **12** (2009), pp. 103–152.

[9] Benton, N. and M. Hyland, *Traced premonoidal categories*, Theoretical Informatics and Applications **37** (2003), pp. 273–299.

[10] Cho, K., "Semantics for a Quantum Programming Language by Operator Algebras," Master's thesis, University of Tokyo (2014).

[11] Cho, K., B. Jacobs, B. Westerbaan and A. Westerbaan, *An Introduction to Effectus Theory* (2015), `arXiv:1512.05813 [cs.LO]`.

[12] Cockett, J. R. B. and S. Lack, *Restriction categories I: Categories of partial maps*, Theoretical Computer Science **270** (2002), pp. 223–259.

[13] Coecke, B., C. Heunen and A. Kissinger, *Categories of quantum and classical channels*, Quantum Information Processing **15** (2016), pp. 5179–5209.

[14] Egger, J., *Involutive monoidal categories and enriched dagger categories* (2008), seminar talk, University of Oxford, available at `https://www.youtube.com/watch?v=75dTIkppk8Q` (fetched Mar. 31., 2017).

[15] Ésik, Z., *Fixed point theory*, in: M. Droste, W. Kuich and H. Vogler, editors, *Handbook of Weighted Automata*, Springer, 2009 pp. 29–65.

[16] Ésik, Z., *Equational properties of fixed point operations in cartesian categories: An overview*, in: G. Italiano, G. Pighizzini and D. Sannella, editors, *MFCS 2015, Proceedings, Part I*, Springer, 2015 pp. 18–37.

[17] Guo, X., "Products, Joins, Meets, and Ranges in Restriction Categories," Ph.D. thesis, University of Calgary (2012).

[18] Haghverdi, E., *Unique decomposition categories, Geometry of Interaction and combinatory logic*, Mathematical Structures in Computer Science **10** (2000), pp. 205–230.

[19] Hasegawa, M., *Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi*, in: P. de Groote and J. R. Hindley, editors, *TLCA '97*, Lecture Notes in Computer Science **1210**, Springer, 1997 pp. 196–213.

[20] Hasegawa, M., M. Hofmann and G. Plotkin, *Finite dimensional vector spaces are complete for traced symmetric monoidal categories*, in: A. Avron, N. Dershowitz and A. Rabinovich, editors, *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday* (2008), pp. 367–385.

[21] Heunen, C., "Categorical quantum models and logics," Ph.D. thesis, Radboud University Nijmegen (2009).

[22] Heunen, C., *On the functor ℓ^2*, in: *Computation, Logic, Games, and Quantum Foundations – The Many Facets of Samson Abramsky*, Springer, 2013 pp. 107–121.

[23] Heunen, C. and M. Karvonen, *Monads on dagger categories*, Theory and Applications of Categories **31** (2016), pp. 1016–1043.

[24] Hoshino, N., *A representation theorem for unique decomposition categories*, Electronic Notes in Theoretical Computer Science **286** (2012), pp. 213–227.

[25] Hyland, M., *Abstract and concrete models for recursion*, in: O. Grumberg, T. Nipkow and C. Pfaller, editors, *Proceedings of the NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness* (2008), pp. 175–198.

[26] Jacobs, B., *Involutive categories and monoids, with a GNS-correspondence*, Foundations of Physics **42** (2012), pp. 874–895.

[27] Jacobs, B., *New directions in categorical logic, for classical, probabilistic and quantum logic*, Logical Methods in Computer Science **11** (2015), pp. 1–76.

[28] James, R. P. and A. Sabry, *Information effects*, in: *POPL 2012, Proceedings* (2012), pp. 73–84.

[29] James, R. P. and A. Sabry, *Theseus: A high level language for reversible computing* (2014), work-in-progress report at RC 2014, available at https://www.cs.indiana.edu/~sabry/papers/theseus.pdf.

[30] Joyal, A., R. Street and D. Verity, *Traced monoidal categories*, Mathematical Proceedings of the Cambridge Philosophical Society **119** (1996), pp. 447–468.

[31] Kaarsgaard, R., H. B. Axelsen and R. Glück, *Join inverse categories and reversible recursion*, Journal of Logical and Algebraic Methods in Programming **87** (2017), pp. 33–50.

[32] Kastl, J., *Inverse categories*, in: H.-J. Hoehnke, editor, *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen **7**, Akademie-Verlag, 1979 pp. 51–60.

[33] Kelly, G. M., "Basic Concepts of Enriched Category Theory," London Mathematical Society Lecture Note Series **64**, Cambridge University Press, 1982.

[34] Rennela, M., *Towards a quantum domain theory: Order-enrichment and fixpoints in W*-algebras*, Electronic Notes in Theoretical Computer Science **308** (2014), pp. 289–307.

[35] Selinger, P., *Dagger compact closed categories and completely positive maps*, Electronic Notes in Theoretical Computer Science **170** (2007), pp. 139–163.

[36] Selinger, P., *A survey of graphical languages for monoidal categories*, in: B. Coecke, editor, *New Structures for Physics*, Springer, 2011 pp. 289–355.

[37] Selinger, P., *Finite dimensional Hilbert spaces are complete for dagger compact closed categories*, Logical Methods in Computer Science **8** (2012), pp. 1–12.

[38] Smyth, M. B. and G. D. Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM Journal on Computing **11** (1982), pp. 761–783.

[39] Wadler, P., *Theorems for free!*, in: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89 (1989), pp. 347–359.

A family of graded epistemic logics

Mario R. F. Benevides[2,3]

PESC/COPPE - Inst. de Matemática — Universidade Federal do Rio de Janeiro

Alexandre Madeira[1,4]

HASLab INESC TEC, U. Minho & CIDMA, U. Aveiro, Portugal

Manuel A. Martins[1,5]

CIDMA, Dep. Mathematics, U. Aveiro, Portugal

Abstract

Multi-Agent Epistemic Logic has been investigated in Computer Science [5] to represent and reason about agents or groups of agents knowledge and beliefs. Some extensions aimed to reasoning about knowledge and probabilities [4] and also with a fuzzy semantics have been proposed [6,13].
This paper introduces a parametric method to build graded epistemic logics inspired in the systematic method to build Multi-valued Dynamic Logics introduced in [11,12]. The parameter in both methods is the same: an action lattice [9]. This algebraic structure supports a generic space of agent knowledge operators, as choice, composition and closure (as a Kleene algebra), but also a proper truth space for possible non bivalent interpretation of the assertions (as a residuated lattice).

Keywords: Epistemic Logic, Action Lattice, Modal Logics

1 Introduction

The analysis and the applications of concepts such as agent's knowledge, everybody's knowledge and common knowledge became a stimulating research field, particularly in the last decades, when *epistemic logics* emerged. Although, the work of Hintikka [8] can be considered the founder of modern modal epistemic logic, most

[1] This work is supported by ERDF European Regional Development Fund, through the COMPETE Programme, and by National Funds through FCT - Portuguese Foundation for Science and Technology - within projects POCI-01-0145-FEDER-016692 and UID/MAT/04106/2013. A. Madeira is also supported by the FCT individual grant SFRH/BPD/103004/2014

[2] This work is supported by the Brazilian research agencies CNPq, FAPERJ and CAPES.

[3] Email: mario@cos.ufrj.br

[4] Email: amadeira@inesctec.pt

[5] Email: martins@ua.pt

of these logics are heavily influenced by the work of Halpern et al [5] on modal logics of knowledge in a multi-agent systems framework. Modal logics of knowledge describe how an agent reasons about his own knowledge and about the knowledge of other agents. We say that an agent knows a fact φ if φ is true in every state that the agent considers possible. "The intuition is that if an agent does not have complete knowledge about the world, he will consider a number of possible worlds. These are his candidates for the way the world actually is" [5].

Much of the agreement and cooperation in a group of agents is reached considering the interaction among the agents and the increasing group knowledge acquisition. A fact φ is mutual knowledge in a group of agents, if each agent knows φ. This group knowledge is also known as everybody's knowledge. Suppose, for instance, that each participant in a conference knows that the lecturer will arrive late. The fact that the lecturer will arrive late is mutual knowledge among the participants, but each participant may think that he is the only one who knows about that. However, suppose that one of the participants makes an announcement for the audience: "The lecturer told me that he will arrive late". From this moment onwards, each participant knows that each participant knows that the lecturer will arrive late, and each participant knows that each participant knows that each participant knows that the lecturer will arrive late, and so on. The participant's statement turned the fact that was mutually known into a common knowledge fact.

There are many situations where we have uncertainty in our knowledge and beliefs. It is not unusual to believe in some fact with some grade of possibility. For instance, *Anne believes that her father has a strong preference for Bob, which means that she believes that he will give a sweet to Bob rather than to Clara. In a scale from 0 to 5, her belief is 4.* This kind of belief is not true or false. In this work we deal with graded knowledge, but atomic propositions are true or false.

In [5] Multi-Agent Epistemic Logics has been investigated, to represent and reason about agents or groups of agents knowledge and beliefs. There are many proposals to extend these logics with uncertainty. Some extensions aimed to reasoning about knowledge and probabilities [4]. In general, this is accomplished extending the language with weighted formulas and adding probabilities to the semantics. There are other attempts that provide a fuzzy or many valued semantics [6,13]. This work goes in the later direction.

The work of Fitting [6] proposes a many valued modal logic where the truth values are taken from a lattice.It is presented two semantics, one where the atomic propositions are many valued and a second one where the accessibility relation also is many valued. Also, in [3], it is presented a many-valued modal logic over a finite residuated lattice. In [13] it is introduced an epistemic logic based on the work of Fitting. It differs from ours because they work with a particular lattice. Another related work that uses a complete, distributive lattices as semantics for epistemic and doxastic logics is presented in [7]. More recently, some interesting works have appeared to deal with many valued dynamic epistemic logic [16,10].

In [11,12] it is proposed a method to build Multi-valued Dynamic Logics. Inspired on this method, we introduce a method to build graded Multi-Agent Epistemic logics. Both methods are based on Action Lattices [9]. Using action lattices, we are able to support a generic space of agent knowledge operators, as choice,

composition and closure (as a Kleene algebra), but also a proper truth space for possible non bivalent interpretation of the assertions (as a residuated lattice). We use matricial algebra to be able to introduce knowledge representations as weighted graphs, which enables us to capture a wide class of weighted scenarios, from the classic bivalent perspective of knowledge, to other structured, discrete and continuous, domains. It should be notice that, in this work, we only deal with the epistemic notions of knowledge and their duals.

This paper is organized as follows. Section 2 presents all the background needed about multi-agent epistemic logic. Section 3, introduces our method for building graded Multi-Agent Epistemic logics. It also provides some concepts on Kleene algebras and action lattices. Section 4 illustrates the use of our method with two examples. Section 5 discusses some conditions where classical axioms of Multi-Agent Epistemic Logic are valid and points out some future works.

2 Multi-Agent Epistemic Logic

Multi-agent epistemic logic has been investigated in Computer Science [5] to represent and reason about agents or groups of agents knowledge and beliefs.

2.0.1 Language and Semantics

Definition 2.1 *The epistemic language consists of a set Φ of countably many proposition symbols, a finite set $\mathcal{A}$ of agents, the boolean connectives $\neg$ and $\wedge$, a modality K_a for each agent a. The formulas are defined as follows:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid C_G\varphi$$

where $p \in \Phi$, $a \in \mathcal{A}$ and $G \subseteq \mathcal{A}$.

The standard connectives can be presented as abbreviations, namely $\bot \equiv \neg\top$, $\varphi \vee \phi \equiv \neg(\neg\varphi \wedge \neg\phi)$, $\varphi \to \phi \equiv \neg(\varphi \wedge \neg\phi)$ and $E_G\varphi \equiv \bigwedge_{a\in G} K_a\varphi$.

The intuitive meaning of the modal formulas are:

- $K_a\varphi$ - agent a knows φ;
- $E_G\varphi$ - every agent $a \in G$ knows φ;
- $C_G\varphi$ - it is common knowledge among all members of group G that it is the case that φ.

We also introduce, by definition, the dual operators $B\varphi \equiv \neg K\neg\varphi$ and $M_G\varphi \equiv \neg E_G\neg\varphi$.

Definition 2.2 *A multi-agent epistemic* frame *is a tuple $\mathcal{F} = (W, R_a)$ where*

- *W is a non-empty set of states;*
- *R_a is a binary relation over W, for each agent $a \in \mathcal{A}$;*

We also define the following relations

- $R_G = \bigcup_{a\in G} R_a$
- $R_G^* = (R_G)^*$, *where $(R_G)^*$ is the reflexive, transitive closure of R_G.*

Definition 2.3 *A* multi-agent model *is a pair* $\mathcal{M} = (\mathcal{F}, \mathbf{V})$*, where* $\mathcal{F}$ *is a frame and* $\mathbf{V}$ *is a valuation function* $\mathbf{V} : \Phi \to 2^W$*.*

In most applications of multi-agent epistemic logic the relations R_a are equivalence relations. In this case, models are called *epistemic models* and, in these structures, if G is not the empty group of agents, R^*_G coincides with R^+_G, for R^+_G being the transitive closure of R_G.

Definition 2.4 *Given a multi-agent model* $\mathcal{M} = \langle S, R_a, V\rangle$*. The notion of satisfaction* $\mathcal{M}, s \models \varphi$ *is defined as follows*

- $\mathcal{M}, s \models p$ *iff* $s \in V(p)$
- $\mathcal{M}, s \models \neg\phi$ *iff* $\mathcal{M}, s \not\models \phi$
- $\mathcal{M}, s \models \phi \wedge \psi$ *iff* $\mathcal{M}, s \models \phi$ *and* $\mathcal{M}, s \models \psi$
- $\mathcal{M}, s \models K_a\phi$ *iff for all* $s' \in S : sR_as' \Rightarrow \mathcal{M}, s' \models \phi$
- $\mathcal{M}, s \models C_G\phi$ *iff for all* $s' \in S : sR^*_Gs' \Rightarrow \mathcal{M}, s' \models \phi$

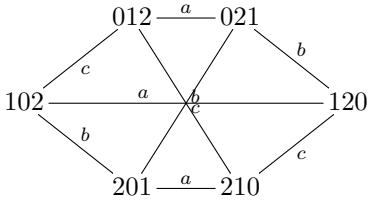
It is easy to see that $\mathcal{M}, s \models E_G\phi$ *iff for all* $s' \in S : sR_Gs' \Rightarrow \mathcal{M}, s' \models \phi$*.*

Example 1 (An adaptation from [17]) *Suppose a father has three envelopes, each containing:* **0**, **1** *and* **2** *dollars inside respectively. The father has three children:* **a***nne,* **b***ob and* **c***lara. Each child receives one envelope and do not know content of the envelopes of the other children.*

We use proposition symbols $0_x, 1_x, 2_x$ *for* $x \in \{a, b, c\}$ *meaning "child* x *has envelope* **0**, **1** *or* **2**. *We name each state by the envelope that each child has in that state, for instance* 012 *is the state where child* **a** *has* **0**, *child* **b** *has* **1** *and child* **c** *has* **2**. *A state name underlined means current state. The following epistemic model represents the epistemic state of each agent*[6] *.*

Hexa $= \langle S, R_a, R_b, R_c, V\rangle$*:*

- $S = \{012, 021, 102, 120, 201, 210\}$
- $R_a = \{(012, 012), (012, 021), (021, 021), \dots\}$, ...
- $V(0_a) = \{012, 021\}$, $V(1_a) = \{102, 120\}$, ...



It is not difficult to see that $012 \models B_b0_a$ *and* $012 \models B_aK_c2_c$ *hold, but* $021 \models E_{ac}2_b$ *does not hold.*

3 Parametric construction of Graded Epistemic Logics

We introduce, in this paper, a parametric method to build graded epistemic logics inspired in the systematic method to build multi-valued dynamic logics introduced in [11,12]. Both methods are based in the same parameter: an action lattice [9].

[6] We omit the reflexive loops in the picture

3.1 Kleene algebras, action lattices and graded knowledges representation

Action lattices support a generic space of agent knowledge operators, as choice, composition and closure (as a Kleene algebra), but also a proper truth space for possible non bivalent interpretation of the assertions (as a residuated lattice). Observe that the original motivations of Kozen to introduce Action Lattices were very different for these ones. Originally, the residues were introduced within *Action Algebra* [15] as a necessary technicality to obtain a finitely-based equational variety to reason about imperative programs. Then, Kozen adjusted this notion into the *Action Lattice* in [9] by introducing and axiomatizing a meet operation, in order to recover the closeness by matricial formation of the Kleene Algebras [2]. We overview, in the following, the action algebra with some relevant examples in the context of our purpose. A lot of other examples and properties can be found in [11]. The structure of Kleene algebra will be used to model the set of agent knowledge operators over a set of agents $\mathcal{A}$. In our setting, the valuations of propositions are crisp, i.e., true or false. This forces the integrability on the action lattices adopted.

$$a+(b+c)=(a+b)+c \quad (1) \qquad a;x\leq x\Rightarrow a^*;x\leq x \quad (11)$$
$$a+b=b+a \quad (2) \qquad x;a\leq x\Rightarrow x;a^*\leq x \quad (12)$$
$$a+a=a \quad (3) \qquad a;x\leq b\Leftrightarrow x\leq a\to b \quad (13)$$
$$a+0=0+a=a \quad (4) \qquad a\to b\leq a\to(b+c) \quad (14)$$
$$a;(b;c)=(a;b);c \quad (5) \qquad (x\to x)^*=x\to x \quad (15)$$
$$a;1=1;a=a \quad (6) \qquad a\cdot(b\cdot c)=(a\cdot b)\cdot c \quad (16)$$
$$a;(b+c)=(a;b)+(a;c) \quad (7) \qquad a\cdot b=b\cdot a \quad (17)$$
$$(a+b);c=(a;c)+(b;c) \quad (8) \qquad a\cdot a=a \quad (18)$$
$$a;0=0;a=0 \quad (9) \qquad a+(a\cdot b)=a \quad (19)$$
$$1+a+(a^*;a^*)\leq a^* \quad (10) \qquad a\cdot(a+b)=a \quad (20)$$

Fig. 1. Axiomatisation of action lattices (from [9])

Definition 3.1 (Kleene Algebra) *A* Kleene algebra *is an idempotent (and thus partially ordered) semiring endowed with a closure operator $*$, i.e. it consists of a tuple $(A,+,;,0,1,*)$ where A is a set, $+$ and $;$ are binary operations, $*$ is an unary operation and $0,1$ are constants satisfying the axioms (1)–(12) (the relation $\leq$ is the natural order induced by the operation $+$: $a\leq b$ iff $a+b=b$).*

Note that (4) implies that 0 is the minimum element in any Kleene algebra. Conway shown in [2] that we can endow the class of all matrices over a Kleene algebra with a Kleene structure. We recall this procedure here: given a Kleene algebra $\mathbf{A}=(A,+,;,0,1,*)$ we define a Kleene algebra $\mathbb{M}_n(\mathbf{A})=(M_n(\mathbf{A}),\mathbf{+},\mathbf{;},\mathbf{0},\mathbf{1},\mathbf{*})$ as follows:

(i) $M_n(\mathbf{A})$ is the space of $(n\times n)$-matrices over $\mathbf{A}$

(ii) for any $A,B\in M_n(\mathbf{A})$, define $M=A+B$ by $M_{ij}=A_{ij}+B_{ij}$, $i,j\leq n$.

(iii) for any $A,B\in M_n(\mathbf{A})$, define $M=A\ \mathbf{;}\ B$ by $M_{ij}=\sum_{k=1}^{n}(A_{ik};B_{kj})$ for any $i,j\leq n$.

(iv) $\mathbf{1}$ and $\mathbf{0}$ are the $(n \times n)$-matrices defined by $\mathbf{1}_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$ and $\mathbf{0}_{ij} = 0$, for any $i, j \leq n$.

(v) for any $M = [a] \in \mathbb{M}_1(\mathbf{A})$, $M^* = [a^*]$; for any $M = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \in M_n(\mathbf{A})$, $n > 1$, where A and D are square matrices, define

$$M^* = \left[\begin{array}{c|c} F^* & F^* ; B ; D^* \\ \hline D^*; C; F^* & D^* + (D^* ; C ; F^* ; B ; D^*) \end{array}\right]$$

where $F = A + B ; D^* ; C$. Note that this construction is recursively defined from the base case ($n = 2$) where the operations of the base action lattice $\mathbf{A}$ are used.

In the present work we take advantage of this matricial algebra to be able to operate knowledge representations as weighted graphs or, more precisely, weighted labelled transition systems. As we will see, this abstract structure capture a wide class of weighted scenarios, from the classic bivalent perspective of knowledge, to other structured, discrete and continuous, domains.

Moreover, as stated, we are interesting in the definition of Graded Epistemic logics with non necessarily boolean degrees of truth. In this view, in order to be able to interpret other logical connectives, we extend our Kleene Algebra of knowledge with some additional structure - namely, with a residue for the interpretation of the logical implication and an infimum to interpret the logical conjunction. This can be found in the following notion of Action Lattice introduced by D. Kozen in [9]. Note, however, that the seminal motivation for this definition was quite distinct of the stated one. In particular, it aimed to adjust the finitely-based equational variety "Action Algebra" of Pratt [15], to an algebra closed under the matricial constructions. Let us recall this notion:

Definition 3.2 *A* action lattice *is a tuple* $\mathbf{A} = (A, +, ;, 0, 1, *, \rightarrow, \cdot)$*, where* A *is a set,* 0 *and* 1 *are constants,* $*$ *is an unary operation in* A *and* $+, ;, \rightarrow$ *and* $\cdot$ *are binary operations in* A *satisfying the axioms enumerated in Figure 5, where the relation* $\leq$ *is induced by* $+$*:* $a \leq b$ *iff* $a + b = b$*. An* integral action lattice *consists of an action lattice satisfying* $a \leq 1$*.*

Beyond the bivalent $\{0, 1\}$-action lattice we consider the following two action lattice that will be used to illustrate our method in Section 4. More examples and properties of action lattices can be found in [11].

Definition 3.3 (Ł - the Łukasiewicz arithmetic lattice) *The* Łukasiewicz arithmetic lattice *is the structure* $\boldsymbol{Ł} = ([0, 1], \max, \odot, 0, 1, *, \rightarrow, \min)$*, where*

- $x \rightarrow y \ = \ \min(1, 1 - x + y)$,
- $x \odot y = \max(0, y + x - 1)$ *and*
- $x^* = 1$.

Definition 3.4 ($\mathbf{W}_k$ finite Wajsberg hoops) *We consider now an action lattice endowing the finite* Wajsberg hoops *[1] with a suitable star operation. Hence, for a fix natural $k > 0$ and a generator a, we define the structure $\mathbf{W}_k = (W_k, +, ;, 0, 1, *, \to, \cdot)$, where $W_k = \{a^0, a^1, \cdots, a^k\}$, $1 = a^0$ and $0 = a^k$, and for any $m, n \leq k$,*

- $a^m + a^n = a^{min\{m,n\}}$
- $a^m ; a^n = a^{min\{m+n,k\}}$
- $(a^m)^* = a^0$
- $a^m \to a^n = a^{max\{n-m,0\}}$
- $a^m \cdot a^n = a^{max\{m,n\}}$

3.2 A method to build Graded Epistemic Logic

In this section we introduce a method to build multi-agent epistemic logics parameterized by an action lattice. The "on-demand grading" of the logic is only reflected in its semantics; the syntax is the same as in the standard case. The proposition assignment is crisp and only the agent's relations are graded on the underlying action lattice. This non orthodox feature is naturally expressed on the definition of satisfaction.

Let us fix a complete action lattice $\mathbf{A} = (A, +, ;, 0, 1, *, \to, \cdot)$. We introduce, in the following, a method to generate an $\mathbf{A}$-graded epistemic logic $\mathcal{GE}(\mathbf{A})$:

- Signatures $(\mathrm{At}, \mathrm{Ag})$ where At is a set of atomic propositions and Ag is a finite set of agents.
- Sentences are the standard sentences of Multi-Agent Epistemic Logic:

$$\varphi ::= p \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid K_a\varphi \mid B_a\varphi \mid E_G\varphi \mid M_G\varphi \mid C_G\varphi$$

  where $p \in \mathrm{At}$, $a \in \mathrm{Ag}$, $G \subseteq \mathrm{Ag}$. Note that, here we are explicitly considering the or connective and the dual operators of the ones introduced in Definition 2.1 . Actually, here these operators are not definable because we do not have, in general, a negation.
- Models are structures (W, R, V) where W is a non empty set of states, with cardinality n; R is an Ag-family of $(n \times n)$-matrices of $\mathbb{M}(\mathbf{A})$ and $V : \mathrm{At} \times W \to \{0, 1\}$ is a valuation function. We use the notation $R_a(w, w')$ to denote the cell (w, w') of the matrix R_a.
- Satisfaction:
  $\cdot$ $(w \models \bot) = 0$
  $\cdot$ $(w \models p) = V(p, w)$, for any $p \in \mathrm{At}$
  $\cdot$ $(w \models \varphi \wedge \varphi') = (w \models \varphi) \cdot (w \models \varphi')$
  $\cdot$ $(w \models \rho \vee \rho') = (w \models \rho) + (w \models \rho')$
  $\cdot$ $(w \models \varphi \to \varphi') = (w \models \varphi) \to (w \models \varphi')$
  $\cdot$ $(w \models K_a\,\varphi) = \bigwedge_{w' \in W} \big(R_a(w, w') \to (w' \models \varphi)\big)$
  $\cdot$ $(w \models B_a\,\varphi) = \bigvee_{w' \in W} \big(R_a(w, w'); (w' \models \varphi)\big)$
  $\cdot$ $(w \models E_G\,\varphi) = \bigwedge_{w' \in W} \big(R_G(w, w') \to (w' \models \varphi)\big)$
  $\cdot$ $(w \models M_G\,\varphi) = \bigvee_{w' \in W} \big(R_G(w, w'); (w' \models \varphi)\big)$
  $\cdot$ $(w \models C_G\,\varphi) = \bigwedge_{w' \in W} \big(R^*_G(w, w') \to (w' \models \varphi)\big)$
  for $R_G = \sum_{a \in G} R_a$

4 Examples

We have already discussed an example of epistemic logic in the background section. Such example can be seen as an instantiation of our method over the $\{0, 1\}$ standard action lattice (see [11]). We present two more examples, namely one that deals with discrete degrees of knowledge and, on the same context, another one that admits knowledge ranging over a continuous scale.

Example 2 *Consider here the Graded Epistemic Logic generated by the Wajsberg hoop* $\mathbf{W}_5$ *over* $\{a^0, a^1, a^2, a^3, a^4, a^5\}$ *(Definition 3.4). Recall that the order in* $\mathbf{W}_5$ *is* $a^5 < a^4 < a^3 < a^2 < a^1 < a^0$*. In order to simplify the example, we denote* a^k *by* $5 - k$*, for* $k = 0, \ldots, 5$*. This logic is useful to reasoning about the following variant of Example 2.*
Suppose now that the children are jealous and they have the following beliefs:

(i) **a***nne believes that the father has a strong preference for* **b***ob, which means that she believes that he will give the envelop with higher value to bob than to* **c***lara. In a scale from 0 to 5, her belief is 4; Conversely, her belief that the envelop bob received has a smaller value is 1.*

(ii) **c***lara also believes that the father has a preference for* **b***ob . In a scale from 0 to 5, her belief is 3; and conversely, her belief that the envelop bob received has a smaller value is 1. But if she has the envelop* **2** *then she believes that the father has no preference between* **a***nne and* **b***ob; in that case her belief is 4.*

(iii) **b***ob does not believe that the father has any preference between* **a***nne and* **c***lara. So his belief is 3 indifferently about any situation.*

The following draws represent the beliefs of **a***nna,* **b***ob and* **c***lara. We draw it separately for clarity sake. Moreover, we omit the reflexive loops in the picture with value 5.*

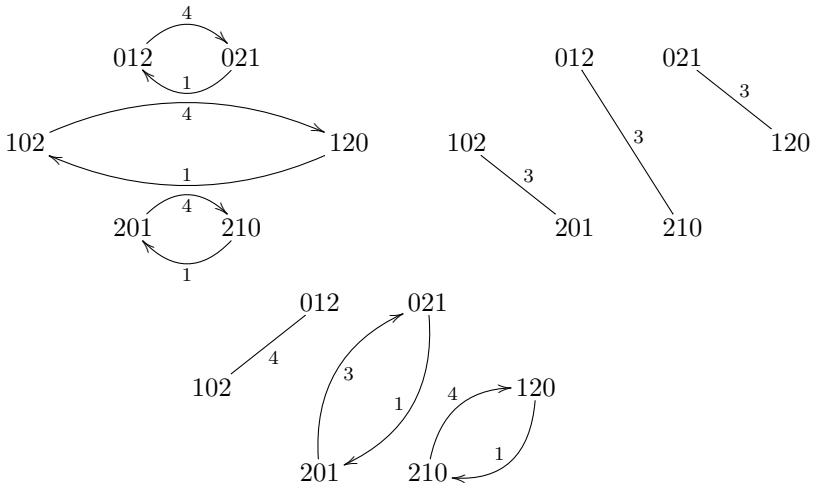


Fig. 2. anna's, **b**ob's and **c**lara's beliefs

We evaluate some formulas in this model. In order to simplify the calculations we use the fact that $a^5 \to x = a^0$*(i.e.,* $0 \to x = 5$*) and* $a^5; x = a^5$ *(i.e.,* $0; x = 0$*).*

$$\begin{aligned}
012 \models B_b 0_a = \quad & \bigvee \{R_b(012,012); 012 \models 0_a, R_b(012,210); 210 \models 0_a\} = \bigvee\{5;5,3;0\} = 5\\
012 \models B_a K_c 2_a = \quad & \bigvee \{R_a(012,012); 012 \models K_c 2_a, R_a(012,021); 021 \models K_c 2_a\}\\
= \quad & \bigvee \{5; \bigwedge \{R_c(012,012) \to 012 \models 2_a, R_c(012,102) \to 102 \models 2_a\},\\
& 4; \bigwedge \{R_c(021,021) \to 021 \models 2_a, R_c(021,201) \to 201 \models 2_a\}\}\\
= \quad & \bigvee \{5; \bigwedge \{5 \to 0, 4 \to 0\}, 4; \bigwedge \{5 \to 0, 1 \to 5\}\}\\
= \quad & \bigvee \{a^0; \bigwedge \{a^0 \to a^5, a^1 \to a^5\}, a^1; \bigwedge \{a^0 \to a^5, a^4 \to a^0\}\}\\
= \quad & \bigvee \{a^0; a^5, a^1; a^5\} = a^5 (= 0)
\end{aligned}$$

To calculate $M_{ac}2_b$ at 021 *we first calculate the matrix of $R_{ac} = R_a + R_c$.*

$R_{ac} =$

	012	021	102	120	201	210
012	5	4	4	0	0	0
021	1	5	0	0	1	0
102	4	0	5	4	0	0
120	0	0	1	5	0	1
201	0	0	0	4	5	4
210	0	0	0	4	1	5

Then we have,

$$\begin{aligned}
021 \models M_{ac}2_b = \quad & \bigvee \{R_{ac}(021,012); 012 \models 2_b, R_{ac}(021,021); 021 \models 2_b,\\
& R_{ac}(021,201); 201 \models 2_b\} = \bigvee\{1;5,5;5,1;0\} = 5
\end{aligned}$$

If we consider the group knowledge we have

$$\begin{aligned}
021 \models E_{ac}2_b = \quad & \bigwedge \{R_{ac}(021,012) \to 012 \models 2_b, R_{ac}(021,021) \to 021 \models 2_b,\\
& R_{ac}(021,201) \to 201 \models 2_b\}\\
= \quad & \bigwedge\{1 \to 5, 5 \to 5, 1 \to 0\} = \bigwedge\{5,5,1\} = 5
\end{aligned}$$

Example 3 *Consider now the Graded Epistemic Logic generated by the Łukasiewicz arithmetic lattice* $\mathbf{Ł} = ([0,1], \max, \odot, 0, 1, *, \to, \min)$ *(Definition 3.3). This logic is adequate to reasoning about knowledge expressed in the continuous scale* $[0,1]$. *Let us look to the following variant of Example 2.*
Suppose now that the children have the following beliefs:

(i) **a***nne believes that the father has a strong preference for* **b***ob, which means that she believes that he will give the envelop with higher value to bob than to* **c***lara. Her belief is* $\frac{4}{5}$*; moreover her belief that the value is less is* $\frac{1}{5}$

(ii) **c***ath also believes that the father has a preference for* **b***ob. Her belief is* $\frac{3}{5}$*. But if she has the envelop* **2** *then she believes that the father has no preference between* **a***nne and* **b***ob.In such case her belief is* 1.

(iii) **b***ob does not believe that the father has any preference between* **a***nne and* **c***lara. So, his beliefs are all* 1.

The draws in figure 3 represent the beliefs of **a***nna,* **b***ob and* **c***lara. We draw it separately for clarity sake.*

We will evaluate the same formulas as in previous example:

$$\begin{aligned}
012 \models B_b 0_a = \quad & \bigvee \{R_b(012,012) \odot 012 \models 0_a, R_b(012,210) \odot 210 \models 0_a\}\\
= \quad & \bigvee\{1 \odot 1, 1 \odot 0\} = 1
\end{aligned}$$

$$\begin{aligned}
012 \models B_a K_c 2_a = \quad & \bigvee \{R_a(012,012) \odot 012 \models K_c 2_a, R_a(012,021) \odot 021 \models K_c 2_a\}\\
= \quad & \bigvee \{1 \odot \bigwedge \{R_c(012,012) \to 012 \models 2_a, R_c(012,102) \to 102 \models 2_a\},\\
& \tfrac{4}{5} \odot \bigwedge \{R_c(021,021) \to 021 \models 2_a, R_c(021,201) \to 201 \models 2_a\}\}\\
= \quad & \bigvee \{1 \odot \bigwedge \{1 \to 0, 1 \to 0\}, \tfrac{4}{5} \odot \bigwedge \{1 \to 0, \tfrac{2}{5} \to 1\}\}\\
= \quad & \bigvee \{1 \odot \bigwedge \{0,0\}, \tfrac{4}{5} \odot \bigwedge \{0,1\}\}\\
= \quad & \bigvee \{1 \odot 0, \tfrac{4}{5} \odot 0\} = \bigvee \{0,0\} = 0
\end{aligned}$$

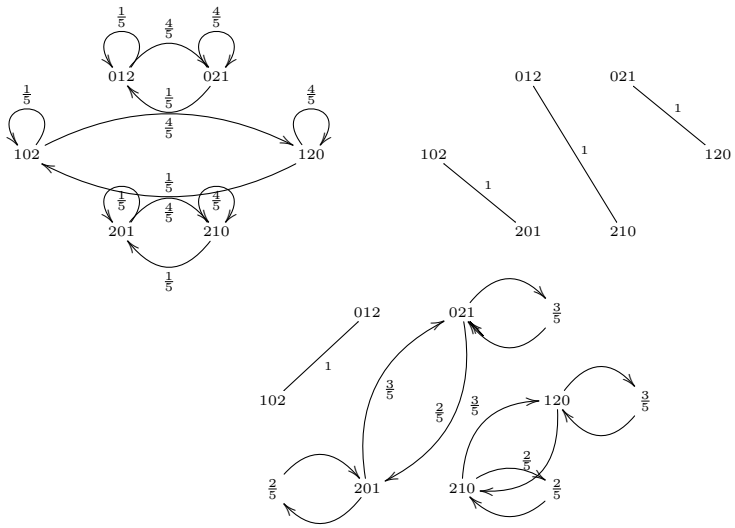


Fig. 3. **a**nna's, **b**ob's and clara's beliefs

To calculate $M_{ac}2_b$ *at* 021 *we first calculate the matrix of* $R_{ac} = R_a + R_c$.

$$R_{ac} = \begin{array}{c|cccccc} & 012 & 021 & 102 & 120 & 201 & 210 \\ \hline 012 & 1 & \frac{4}{5} & 1 & 0 & 0 & 0 \\ 021 & \frac{1}{5} & \frac{4}{5} & 0 & 0 & \frac{2}{5} & 0 \\ 102 & 1 & 0 & 1 & \frac{4}{5} & 0 & 0 \\ 120 & 0 & 0 & \frac{1}{5} & \frac{4}{5} & 0 & \frac{3}{5} \\ 201 & 0 & \frac{3}{5} & 0 & 0 & \frac{2}{5} & \frac{4}{5} \\ 210 & 0 & 0 & 0 & \frac{3}{5} & 1 & \frac{4}{5} \end{array}$$

Then we have,

$$\begin{aligned} 021 \models M_{ac}2_b = \quad & \bigvee \{R_{ac}(021,012) \odot 012 \models 2_b, R_{ac}(021,021) \odot 021 \models 2_b, \\ & R_{ac}(021,201) \odot 201 \models 2_b\} \\ = \quad & \bigvee\{\tfrac{1}{5} \odot 1, \tfrac{4}{5} \odot 1, \tfrac{2}{5} \odot 0\} = \bigvee\{\tfrac{1}{5}, \tfrac{4}{5}, 0\} = \tfrac{4}{5} \end{aligned}$$

If we consider the group knowledge we have

$$\begin{aligned} 021 \models E_{ac}2_b = \quad & \bigwedge \{R_{ac}(021,012) \to 012 \models 2_b, R_{ac}(021,021) \to 021 \models 2_b, \\ & R_{ac}(021,201) \to 201 \models 2_b\} \\ = \quad & \bigwedge\{\tfrac{1}{5} \to 1, \tfrac{4}{5} \to 1, \tfrac{2}{5} \to 0\} = \bigwedge\{1, 1, \tfrac{3}{5}\} = \tfrac{3}{5} \end{aligned}$$

5 How epistemic GE(A) logics are?

The study of each one of these instantiation of the logics generated in the previous section, as logics with 'its own rights', is very challenging. Obviously, there are aspects that have to be studied instantiation-by-instantiation. In this section, however we approach this in a more systematic perspective, trying to respond the question *How epistemic* $\mathcal{GE}(\mathbf{A})$ *logics are?* by studying the validity of the standard axioms of epistemic logic in Fig 5 on the generated logics.

We obtain some generic results for specific classes of generated logics, with respect to specific classes of action lattices and imposing constrains on the achieved models. The latter also happens in the standard epistemic logic, which the completeness is established for a restricted class of models, for instance, the epistemic ones (i.e., models whose accessible relations are equivalence relations) [17].

(i) *All instantiations of propositional tautologies*,
(ii) $K_a(\varphi \to \psi) \to (K_a\varphi \to K_a\psi)$,
(iii) $K_a\varphi \to \varphi$,
(iv) $K_a\varphi \to K_aK_a\varphi$ (+ *introspection*),
(v) $\neg K_a\varphi \to K_a\neg K_a\varphi$ (− *introspection*),
(vi) $C_G\varphi \leftrightarrow E_GC_G\varphi$
(vii) $C_G(\varphi \to E_G\varphi) \to (\varphi \to C_G\varphi)$

Fig. 4. Axiomatics of epistemic logic [5,17]

We follow the strategy adopted in [11,12] (in the context of generated graded dynamic logics). The integrability ($a \leq 1$) on action lattices provides a nice proof strategy to work at this generic level: as it is well known, in any integral action lattice, we have

$$(a \to b) = 1 \;\Leftrightarrow\; a \leq b \tag{21}$$

Theorem 5.1 *Let* $\mathbf{A}$ *be an integral* ;*-idempotent,* ;*-commutative action lattice. The property*

(ii) $K_a(\varphi \to \psi) \to (K_a\varphi \to K_a\psi)$

is valid in the logic $\mathcal{GE}(\mathbf{A})$.

Proof. This proof can be extracted from Lemma 9 of [11]. □

In a similar way, but by imposing commutativity on the operation ; we can extract the proof for the axiom (vii):

Theorem 5.2 *Let* $\mathbf{A}$ *be an integral action lattice such that* $; = \cdot$. *Then the property*

(vii) $C_G(\varphi \to E_G\varphi) \to (\varphi \to C_G\varphi)$

is valid in the logic $\mathcal{GE}(\mathbf{A})$.

Proof. This can be directly adapted from Lemma 10 of [11]. □

So, we have to study the remaining axioms, specifically the ones that distinguish epistemic logic from other modal logics - the axioms (iii), (iv), (v) and (vi). In this view, we have to impose further properties on the structure of the models. In particular, we have to generalize the reflexivity and transitivity conditions for our graded setting to guarantee the validity of (iii) and (iv). What the conditions needed for the cases (iii) and (iv) are still in study.

Definition 5.3 Let $\mathbf{A}$ be an action lattice and M be a model in $\mathcal{GE}(\mathbf{A})$. We say that M is *graded-reflexive* if for any $a \in \mathrm{Ag}$, $w \in W$,

$$R_a(w, w) = 1 \tag{22}$$

and that it is *graded-transitive*, whenever any $a \in \mathrm{Ag}$

$$\text{for any } w, w', w'' \in W, R_a(w, w'') \geq R_a(w, w') \;;\; R_a(w', w'') \tag{23}$$

Theorem 5.4 *Let* $\mathbf{A}$ *be an integral action lattice. Then, the axiom*

(iii) $K_a\varphi \to \varphi$,

is valid in graded-reflexive models.

Proof. Since **A** is integral, we have by (21) that it is sufficient to prove that, for any model M, and for any state $w \in W$, $(w \models K_a\varphi) \leq (w \models \varphi)$. In this view, we observe that:

$$
\begin{aligned}
& (w \models K_a\varphi) \\
= \quad & \{ \models \text{defn} \} \\
& \bigwedge_{w' \in W} \big(R_a(w,w') \to (w' \models \varphi)\big) \\
\leq \quad & \{ \text{infimum properties} \} \\
& \big(R_a(w,w) \to (w \models \varphi)\big) \\
= \quad & \{ (22) \} \\
& \big(1 \to (w \models \varphi)\big) \\
= \quad & \{ \text{in any action lattice } 1 \to a = a \text{ (cf. [11])} \} \\
& (w \models \varphi)
\end{aligned}
$$

□

Theorem 5.5 *Let* **A** *be an integral* ;*-commutative action lattice. Then, the axiom*

(iv) $K_a\varphi \to K_aK_a\varphi$ (+ *introspection*),

is valid in graded-transitive models.

Proof. Since **A** is integral, we have by (21) that it is sufficient to prove that, for any model M, and for any state $w \in W$, $(w \models K_a\varphi) \leq (w \models K_aK_a\varphi)$. In this view, we observe that:

$$
\begin{aligned}
& \text{for any } w', w'' \in W, R_a(w,w'') \geq R_a(w,w'); R_a(w',w'') \\
\Leftrightarrow \quad & \{ \text{;-commutative} \} \\
& \text{for any } w', w'' \in W,\ R_a(w,w'') \geq R_a(w',w''); R_a(w,w') \\
\Leftrightarrow \quad & \{ a \leq b \Rightarrow b \to c \leq a \to c \text{ (cf. [11])} \} \\
& \text{for any } w', w'' \in W,\ R_a(w,w'') \to (w'' \models \varphi) \leq \\
& (R_a(w',w''); R_a(w,w')) \to (w'' \models \varphi) \\
\Leftrightarrow \quad & \{ \text{infimum properties} \} \\
& \text{for any } w'' \in W, R_a(w,w'') \to (w'' \models \varphi) \leq \\
& \bigwedge_{w' \in W} \big((R_a(w',w''); R_a(w,w')) \to (w'' \models \varphi)\big) \\
\Leftrightarrow \quad & \{ \text{in any action lattice } a \to (b \to c) = (b;a) \to c \text{ (cf. [11])} \} \\
& \text{for any } w'', R_a(w,w'') \to (w'' \models \varphi) \leq
\end{aligned}
$$

$$\bigwedge_{w' \in W} \big(R_a(w,w') \to (R_a(w',w'') \to (w'' \models \varphi))\big)$$

$\Leftrightarrow$ $\{$ inf. monotocity$\}$

$$\bigwedge_{w'' \in W} R_a(w,w'') \to (w'' \models \varphi) \leq \bigwedge_{w',w'' \in W} \big(R_a(w,w') \to (R_a(w',w'') \to (w'' \models \varphi))\big)$$

$\Leftrightarrow$ $\{$ in any complete action lattice, $x \to (\bigwedge_{i\in I} y_i) = \bigwedge_{i\in I}(x \to y_i)$ (cf. [11])$\}$

$$\bigwedge_{w'' \in W} R_a(w,w'') \to (w'' \models \varphi) \leq \bigwedge_{w' \in W} \big(R_a(w,w') \to \bigwedge_{w'' \in W} (R_a(w',w'') \to (w'' \models \varphi))\big)$$

$\Leftrightarrow$ $\{ \models$ defn twice$\}$

$$(w \models K_a\varphi) \leq (w \models K_aK_a\varphi)$$

□

6 Conclusions and future work

This paper starts with a research program on the parametric generation of graded epistemic logics. The approach is based on the application of the method introduced in Section 3, and should be explored as an effective source of logics to reason on agent knowledge scenarios with distinct degrees of Knowledge/Belief. The generality of the method was illustrated with three graded epistemic logics (note that the standard multi-agent epistemic logic corresponds to the instantiation of the action lattice **2**), but a lot of other examples can be considered - from a $\{false, unknown, true\}$-three valued epistemic logic, achieved by instantiating the action lattice **3** to a more 'esoteric' graded epistemic logic to deal with knowledge/belief scenarios involving resource aware constraints (built on the Floyd Warshall algebra - see [11]). Beyond of their philosophical interest, the study of each one of these instantiations as a logic with 'its own rights' is very challenging. Indeed, as discussed in Section 5, it is possible to characterize specific classes of graded epistemic logics (parametric on specific subclasses of action lattices and by imposing further condition on the models) that preserves the essence of the bivalent epistemic logic.

There is, however, a lot of work to do in this line of research. To establish sufficient conditions for validating the negative introspection axiom (and of (vi)) is still work in progress for us. It seems that, beyond of a generalization of the Euclidean property on models, some new conditions should be imposed in the action lattices, particularly with respect to their negation (note that, in its generic form, there is no negation involution in general). The parametric generation of calculus and the study of complexity of generated epistemic logic w.r.t. to specific classes of action lattices are also in our agenda. Another interesting line of research is to investigate the concepts of simulation and bisimulation for our knowledge representations on the lines proposed in [18,14] for generic fuzzy labelled transition systems.

Finally, it would be interesting to investigate whether our approach allows for the representation of epistemic actions. Public announcements or private communications. More interesting is to look for epistemic actions that make sense only in this (or similar) setting. For example, one can think of situations in which the agent has a belief of some grade n, and then some new information 'downgrades' or 'upgrades' this belief (some form of belief revision, but now in a 'graded' fashion).

References

[1] W. J. Blok and I. M. A. Ferreirim. On the structure of hoops. *algebra universalis*, 43(2-3):233–257, 2000.

[2] J. H. Conway. *Regular Algebra and Finite Machines*. Printed in GB by William Clowes & Sons Ltd, 1971.

[3] L. Godo F. Bou, F. Esteva and R.O. Rodriguez. Many-valued modal logic over a nite residuated lattice. *Journal of Logic and Computation*, 21(5):739–790, 2011.

[4] R. Fagin and J. Halpern. Reasoning about knowledge and probability. *Journal of the ACM*, 41(2):340–367, 1994.

[5] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, USA, 1995.

[6] Melvin Fitting. Many-valued modal logics. *Fundam. Inform.*, 15(3-4):235–254, 1991.

[7] Michell Guzmán, Stefan Haar, Salim Perchy, Camilo Rueda, and Frank D. Valencia. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *J. Log. Algebr. Meth. Program.*, 86(1):107–133, 2017.

[8] J. Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, N.Y, 1962.

[9] Dexter Kozen. On action algebras. *Logic and Information Flow*, pages 78–88, 1994.

[10] Alexander Kurz and Alessandra Palmigiano. Epistemic updates on algebras. *Logical Methods in Computer Science*, 9(4), 2013.

[11] Alexandre Madeira, Renato Neves, and Manuel A. Martins. An exercise on the generation of many-valued dynamic logics. *J. Log. Algebr. Meth. Program.*, 85(5):1011–1037, 2016.

[12] Alexandre Madeira, Renato Neves, Manuel A. Martins, and Luís Soares Barbosa. A dynamic logic for every season. In Christiano Braga and Narciso Martí-Oliet, editors, *Formal Methods: Foundations and Applications - 17th Brazilian Symposium, SBMF 2014, Maceió, AL, Brazil, September 29-October 1, 2014. Proceedings*, volume 8941 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2014.

[13] Yoshihiro Maruyama. Reasoning about fuzzy belief and common belief: With emphasis on incomparable beliefs. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1008–1013, 2011.

[14] Haiyu Pan, Yongming Li, and Yongzhi Cao. Lattice-valued simulations for quantitative transition systems. *Int. J. Approx. Reasoning*, 56:28–42, 2015.

[15] Vaughan R. Pratt. Action logic and pure induction. In *JELIA*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120. Springer, 1990.

[16] Umberto Rivieccio. Bilattice public announcement logic. *Advances in Modal Logic*, 10:459–477, 2014.

[17] H. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*. Synthese Library Series, volume 337. Springer, The Netherland, 2008.

[18] Hengyang Wu and Yuxin Deng. Logical characterizations of simulation and bisimulation for fuzzy transition systems. *Fuzzy Sets and Systems*, 301:19–36, 2016.

An interpretation of CCS into Ludics

Stefano Del Vecchio[1,2] and Virgile Mogbil[1]

[1] *LIPN, CNRS – Université Paris 13, Villetaneuse, France*
[2] *Dipartimento di Matematica e Fisica, Università Roma Tre, Roma, Italia.*

Abstract

Starting from works aimed at extending the Curry-Howard correspondence to process calculi through linear logic, we give another Curry-Howard counterpart for Milner's Calculus of Communicating Systems (CCS) by taking Ludics as the target system. Indeed interaction, Ludics' dynamic, allows to fully represent both the non-determinism and non-confluence of the calculus.
We give an interpretation of CCS processes into carefully defined behaviours of Ludics using a new construction, called *directed behaviour*, that allows controlled interaction paths by using pruned designs. We characterize the execution of processes as interaction on behaviours, by implicitly representing the causal order and conflict relation of event structures. As a direct consequence, we are also able to interpret deadlocked processes, and identify deadlock-free ones.

Keywords: Calculus of Communicating Systems (CCS), Linear logic, Ludics interaction, non-determinism.

1 Introduction

Process algebras are an approach to concurrent theory, to model interactive systems, based on communication, often as message-passing, and reasoning on primitive operators like parallel composition; among the most known and used systems there are Milner's *Calculus of Communicating Systems* (CCS) [15], and the π-calculus ([16], [17]). In our work we focus on the former to give an interpretation in the proof theory setting of *Ludics* [13], which bring back together syntax and semantics following the paradigm of interactive computation, similarly to what is done in game semantics. Finding a proper Curry-Howard counterpart for such calculi could provide a logical foundation to concurrent computation, and some insight into its dynamic.

Usual models for concurrency can be used to give semantics to process algebra, e.g. event structures [24] for CCS [23]. Like Petri Nets [19], event structures are a *true concurrent* model but based on explicit causal order and conflict relations to reveal concurrency. Using closure on these notions, semantical properties easily characterize behaviours such as (no) conflicts, choice independence and confluence. Such closures are also present in our interpretation but internalized in *Ludics' behaviours* directly by bi-orthogonality. From another view point, our Ludics interpretation of CCS expresses all possible schedulings under non-deterministic choices. Somehow

this brings it closer to *interleaving* models for concurrency like traces and *labeled transition systems* [25] whose main ability is to characterize observational equivalence.

Motivations and related works. The motivations of this work come from studies aiming at extending the *Curry-Howard* correspondence outside the functional setting, to model concurrency. Many approaches achieved such a correspondence, though by imposing determinism to processes, i.e. limited to concurrent systems where only a deterministic behaviour can occur. However, reduction on processes is, in general, non-deterministic and non-confluent: this intrinsically limits a possible correspondence with cut elimination or normalization of proofs, whose nature is confluent and deterministic, effectively restraining a process to functional behaviour.

We start from works based on Linear logic [12] (which inspired many systems, being a logic particularly tied to interaction) that stress the difficulties with the mentioned approaches; in particular [4], [3], [5] and [8]. In [8] the authors show that differential nets are a suitable target system for a correspondence, though the translation proposed is not *modular*, in the sense that we cannot compose two differential nets to represent the (parallel) composition of two processes. But even proof nets, that have similar dynamics to process calculi, with a local and asynchronous cut elimination procedure, were unable to express the behaviour of concurrent systems without a shift of correspondence, based on the *scheduling* of executions [6]: from proof-nets as processes to proof-nets as *executions*, stressing the fact that the meaning of a proofs lies in its normal form, reached by cut elimination, while the meaning of a process is not one of its *multiple* irreducible forms, but how each form is reached: which channels communicate, and what is the form of the process at each step. The aim of our work is to try to carry out the same correspondence, but in the setting of *Ludics* [13] to overcome the limitations of cut elimination by using a logical system with **interaction** at its core.

Ludics has already been related to process calculi in works by C. Faggian and co.: *Ludics net* [7], *Ludics as a model for the finitary and linear π-calculus* [9], and an interpretation of linear strategies in *confusion-free event structures* [10] to extend Ludics with non-determinism. Indeed, from this connection the authors aim at gaining a way to represent replication (as in [2]) and non determinism inside Ludics – whose standard version is lacking both; however the correspondence between execution and interaction is not developed at all. As we understand now, our work is rather taking the opposite direction of [10], by effectively representing the *causal order* and *conflict relation* of event structures in Ludics using *behaviours*. This is achieved using a new construction, found in [11], called *directed behaviour*, that allows controlled interaction paths from carefully *pruned* designs. Finally we form a strong correspondence between execution in CCS and interaction in Ludics, without losing nor its non-determinism nor its non-confluence (i.e. multiple normal forms).

Outline. After background notions and notations for both Ludics (designs, behaviours and interaction) and MCCS (a simple fragment of CCS), we define in the third section our interpretation $[\![P]\!]$ of a process P in term of a Ludics' *behaviour* equipped with an assignment function. Section 4 summarizes the main results for MCCS and finishes with their extension to replication-free CCS. At first we focus on the correspondence between process execution and Ludics interaction: $[\![P]\!]$ char-

acterizes all executions on P. We then present the parallel composition as merging of interpretations, and an operational version of the interpretation corresponding to the process execution. We also give a characterization of deadlock-free processes, in the form of a sufficient condition to check on the interpretation for deadlock-freedom. In the last section we explain how to apply our technique to full CCS, that is an extension to replication using Terui's computational Ludics [21].

2 Background

Ludics is an abstraction of multiplicative-additive linear logic proofs in sequent calculus, under focused discipline (used for optimizing proof-search space by Andreoli [1]), thus cut-free and with a strict alternation between **positive** and **negative** rules. *Designs* the objects of Ludics, replace formulas with *addresses* $\xi, \zeta, \ldots$ (sequences of natural numbers); subformulas becomes *sub-addresses* $\xi.1, \xi.2.1, \ldots$. To give an intuition, proofs are rewritten in the following way:

Example 2.1 *A* focused *proof of the formula* $((A_1 \otimes A_2)\,\&\,B) \parr C$ *is:*

$$\dfrac{\dfrac{\vdash A_1, C, \Delta_1 \qquad \vdash A_2, \Delta_2}{\vdash A_1 \otimes A_2, C, \Delta} \qquad \vdash B, C, \Delta}{\vdash ((A_1 \otimes A_2)\,\&\,B) \parr C, \Delta}$$

where the connectives $\&$ *and* $\parr$ *are introduced at the same time. Going further, and considering only positive formulas, it becomes*

$$\dfrac{\dfrac{A_1^\perp \vdash C, \Delta_1 \qquad A_2^\perp \vdash \Delta_2}{\vdash A_1 \otimes A_2, C, \Delta} \qquad \vdash B, C, \Delta}{((A_1 \otimes A_2)^\perp \oplus B^\perp) \otimes C^\perp \vdash \Delta}$$

Finally in Ludics, formulas are forgotten, and we have

$$\dfrac{\dfrac{\xi.1.1 \vdash \xi.3, \Delta_1 \qquad \xi.1.2 \vdash \Delta_2}{\vdash \xi.1, \xi.3, \Delta} \qquad \vdash \xi.2, \xi.3, \Delta}{\xi \vdash \Delta}$$

where the Δ_i*s are sets of addresses.*

Formally, designs can be defined thus:

Definition 2.2 *A* design $\mathcal{D}$ *is a tree of* abstract sequents *built and labeled by rules called actions, such that each branch ends with a positive action. The rules are the following:*

- Daimon *(*Dai^+_Δ *:)* $\dfrac{}{\vdash \Delta}$ ✠
  where Δ *is a finite set of addresses. The Daimon is a* positive *action.*
- Positive action*:* $\dfrac{\ldots \quad \xi.i \vdash \Delta_i \quad \ldots}{\vdash \Delta, \xi}\ (+, \xi, I)$
  where $i \in I$*, with* $I \subset \mathbb{N}$*. I is the* ramification *of the rule and* ξ *is its* focus*.* Δ_i*s are* disjoint *and* included *in* Δ*.*
- Negative action*:* $\dfrac{\ldots \quad \vdash \xi.I, \Delta_I \quad \ldots}{\xi \vdash \Delta}\ (-, \xi, \mathcal{N})$
  where $\mathcal{N}$ *is the ramification of the rule, and* ξ *its focus.* $\mathcal{N} \subset \mathscr{P}_{fin}(\mathbb{N})$*,* $\xi.I = \xi.1, \ldots, \xi.n$*, with* $1, \ldots, n \in I$ *and* $I \in \mathcal{N}$*; the* Δ_I*s are* included *in* Δ*.*

The ramification *denotes the number of branches generated by the rule; i.e. the premises containing the sub-addresses $\xi.i$, with $i \in I$, or $\zeta.i_1, \ldots, \zeta.i_n$ with $i_1, \ldots, i_n \in I$ and $I \in \mathcal{N}$. Any rule with a sub-address $\xi.i$ as focus is* justified *by the rule introducing ξ.*

Positive and negative actions are strictly alternated *and Dai^+ can only be the* last *action of a design.*

Interaction is the equivalent in Ludics of cut elimination, and define its dynamic; the way a design interact is explicited by its syntax, closing the gap with semantics. We use the presentation and definitions found in [18], essentially the same as the seminal article [13]. The following example gives an idea of two orthogonal designs and interaction between them:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1,\{1\})^4}{\xi.1 \vdash}\,(-,\xi.1,\{\{1\}\})^3 \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1,\{1\})^8}{\xi.2 \vdash}\,(-,\xi.2,\{\{1\}\})^7}{\vdash \xi}\,(+,\xi,\{1,2\})^0 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}}{\xi.2.1 \vdash \xi.1.1.1}\,\maltese^{10}}{\vdash \xi.1.1.1, \xi.2}\,(-,\xi.2.1,\{\{1\}\})^9}{\xi.1.1 \vdash \xi.2}\,(+,\xi.2,\{1\})^6}{\vdash \xi.1, \xi.2}\,(-,\xi.1.1,\{\{1\}\})^5}{\xi \vdash}\,(+,\xi.1,\{1\})^2 \quad (-,\xi,\{\{1,2\}\})^1$$

Here the numbers denote the n-th interaction step. Interaction starts from the cut on the bases (i.e. conclusions $\vdash \xi$ and $\xi \vdash$), and checks the premises of the + rule: if the ramification $\{1,2\}$ finds a match in the corresponding negative rule, interaction continues, and ends *successfully* if it reaches a **daimon** $\maltese$: it fulfill the role of axioms, stopping proof-search. In case of successful interaction between two designs $\mathcal{D}$ and $\mathcal{C}$, we say that they are **orthogonals**, denoted $\mathcal{D} \perp \mathcal{C}$ or $\mathcal{C} \in \mathcal{D}^\perp$ (and vice-versa). We will interpret processes as *behaviours*; formally we have

Definition 2.3 *A **behaviour** is a set of designs $\mathscr{B} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ of same base* closed by bi-orthogonality*, i.e. such that $\mathscr{B}^{\perp\perp} = \mathscr{B}$.*

MCCS, a fragment of Milner's CCS. At first we restrict our setting to the *multiplicative* fragment of CCS, as presented in [4], in short called MCCS, to keep it as simple as possible. In section 4.5 the same tools will be used to represent non-deterministic choice and *name* hiding (aka *restriction*). Replication is addressed in section 5 and treated in a reformulated version of Ludics by K. Terui, called c-ludics [21]. MCCS terms are described by the following syntax:

$$P, Q := 1 \mid a^l.P \mid \overline{a}^m.P \mid (P \mid Q)$$

$a, b, c, \ldots$ denotes *channel names* occurrences, taken from a countable set $\mathcal{A}$ of names, and are labeled with *locations* $l, m, o, \ldots$, taken from a countable set Loc of locations (Loc_P denotes the locations of a process P). They are denoted a^l, b^m, c^o, etc. $a^l.P$ is the *positive* action prefix, $\overline{a}^m.P$ the *negative* one, and $P \mid Q$ the *parallel composition* of two processes. We use 1 to denote the *empty process*, instead of the traditional 0, since it is not only the neutral element of the non-deterministic choice (+), but also of the parallel composition ($P \mid 1 \equiv P$), which shows a *multiplicative* behaviour in our interpretation (as we will see, there is also a connection with the linear logic unit

1). A partial order on locations $<_P$ is induced by the *prefix order* of the occurrences of channel names they label: if $P = a^l.b^m \mid Q$, then $l <_P m$. A *synchronisation*, denoted $i, j, u, v, \ldots$, is a pair $(a^l, \overline{a}^m)$ of dual channel-name occurrences, which we can synchronize to perform *execution*, by the local rule

$$a^l.P \mid \overline{a}^m.Q \to_{(l,m)} P \mid Q.$$

The set of synchronisations of P is denoted $\mathcal{S}_P$. If two synchronisations u, v have a location in common, then $(u, v) \in \mathcal{X}_P$, the set of *xor* conditions of P, a conflict-like relation. With $xor(u) = \{v_1, \ldots, v_n \mid (u, v_i) \in \mathcal{X}_P, 1 \leq i \leq n\}$ we denote the set of synchronisations in conflict with u, i.e. the ones that cannot be in the same execution sequence with u. The partial order on locations $<_P$ is naturally extended to synchronisations and denoted $\lessdot_{\mathcal{S}_P}$: $u \lessdot_{\mathcal{S}_P} v$ if there is a location $l \in u$ and a location $m \in v$ such that $l <_P m$. Equality comes from the fact that a location of u could be smaller or greater than a location of v and vice-versa, forming a cycle $u \lessdot v \lessdot u$ in the order.

3 The interpretation

We present here the interpretation of the MCCS fragment. Firstly, the elements of the process relevant for execution must be represented: via an assignment of locations, cuts, and *xor* conditions to addresses, we can code them into designs. We interpret each element of Loc_P, $\mathcal{S}_P$ and $\mathcal{X}_P$ into *negative* designs, then put them together as premises (sub-designs) of a single positive design, the *base design*, denoted $\mathcal{D}_P$, that is a preliminary and naive interpretation of a process P.

Secondly, also the order and conflict relations between these elements need to be coded, and respected during interaction: a new operation on designs is used to restrict interaction paths accordingly to the said relations, and closure properties are obtained by bi-orthogonality, generating a behaviour. The partial orders $<_P$ and $\lessdot_{\mathcal{S}_P}$ and the conflict relation $\mathcal{X}_P$ are represented using particular *directed* (or *non commutative*) modifications of $\mathcal{D}_P$, called *restriction designs*, denoted $\mathcal{R}(P)$. These modifications exploit a technique found in [11], the *pruning* of a branch of a design, which affects interaction in the context of a behaviour. The role of $\mathcal{R}(P)$ is restricting the possible interactions on $\mathcal{D}_P$ by forcing them to respect the prefix order and conflict relation, once we put the designs together as the set of generators of a *behaviour*. Using the designs of the previous example, we try to give an intuition of the idea behind the pruning.

Example 3.1 $\mathcal{D}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1,\{1\})}{\xi.1 \vdash}\,(-,\xi.1,\{\{1\}\}) \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1,\{1\})}{\xi.2 \vdash}\,(-,\xi.2,\{\{1\}\})}{\vdash \xi}\,(+,\xi,\{1,2\})$$

$$\mathcal{C}:\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\ \maltese}{\xi.2.1 \vdash \xi.1.1.1}\ (-,\xi.2.1,\{\{1\}\})}{\vdash \xi.1.1.1, \xi.2}\ (+,\xi.2,\{1\})}{\xi.1.1 \vdash \xi.2}\ (-,\xi.1.1,\{\{1\}\})}{\vdash \xi.1, \xi.2}\ (+,\xi.1,\{1\})}{\xi \vdash}\ (-,\xi,\{\{1,2\}\})$$

$$\mathcal{E}:\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1.1, \xi.2.1.1}\ \maltese}{\xi.1.1 \vdash \xi.2.1.1}\ (-,\xi.1.1,\{\{1\}\})}{\vdash \xi.2.1.1, \xi.2}\ (+,\xi.1,\{1\})}{\xi.2.1 \vdash \xi.2}\ (-,\xi.2.1,\{\{1\}\})}{\vdash \xi.2, \xi.1}\ (+,\xi.2,\{1\})}{\xi \vdash}\ (-,\xi,\{\{1,2\}\})$$

Both $\mathcal{C}$ and $\mathcal{E}$ are orthogonal to $\mathcal{D}$. However

$$\mathcal{D}^*:\ \dfrac{\dfrac{}{\xi.1 \vdash}\ p \qquad \dfrac{\dfrac{}{\vdash \xi.2.1}\ \maltese}{\xi.2 \vdash}\ (-,\xi.2,\{\{1\}\})}{\vdash \xi}\ (+,\xi,\{1,2\})$$

where p denotes a pruning *on the branch starting with $\xi.1 \vdash$, is orthogonal only to $\mathcal{E}$: interaction cannot continue on $\xi.1$, since it is not introduced by a rule anymore, but can only pass through $\xi.2 \vdash$, stopping at the $\maltese$ above $\xi.2.1$. In conclusion, $\mathcal{E} \in \{\mathcal{D}, \mathcal{D}^*\}^\perp$, since it visits the $\xi.2$ branch first, while $\mathcal{C} \notin \{\mathcal{D}, \mathcal{D}^*\}^\perp$; in this way we have forced interaction to respect the order $\xi.2 < \xi.1$.*

Formally, let $[\ \ ]_P$ be an assignment function from $Loc_P \cup \mathcal{S}_P \cup \mathcal{X}_P$ to addresses: for $x \in Loc_P \cup \mathcal{S}_P$, we build the negative design $G[x]$ described by the device of Fig.1 where $[x]_P$ is the address assigned to the synchronisation or location in question; for instance $[u]_P = \xi.1$ and $[l]_P = \xi.2$, with $u \in \mathcal{S}_P$ and $l \in Loc_P$. Each $(u,v) \in \mathcal{X}_P$, to fork the interaction path, is interpreted by $w[u,v]$, in Fig.1, where we note the action as & since it is a binary negative rule, and $xor^u \,\&\, xor^v$ is an address assigned to the clause (u,v). Then, the base design is:

$$\mathcal{D}_P = \Big(\bigotimes_{x \in (\mathcal{S}_P \cup Loc_P),(u,v) \in \mathcal{X}_P} \{G[x], w[u,v]\}\Big)$$

where $\otimes$ stands for a sole **positive** rule $(+,\xi,I)$ with each element of the set as an element of the ramification I (we assume that the assigned addresses have all the same prefix ξ). Each negative design is thus a different premise, each containing a sub-address of the focus. A *restriction* $\mathcal{R}(i)$ for a synchronisation $i = (a^m, \overline{a}^o)$ such that, for instance, $l <_P o$, is an alteration of a copy of $\mathcal{D}_P$ obtained by forcing a $\maltese$ on $G[l]$ and a pruning on $G[i]$. Such restriction $\mathcal{R}(i)$ is of the form described in Fig.1, where the dots $\cdots$ stands for all the other sub-designs of $\mathcal{D}_P$, which remain unaffected. Any interaction with a design orthogonal to both $\mathcal{R}(i)$ and $\mathcal{D}_P$ will be forced to visit $[l] \vdash$ before $[i] \vdash$. For $(u,v) \in \mathcal{X}_P$, instead, we need two restriction designs described in Fig.2; thus tying each member of the xor pair (u or v) to a

$$G[x] = \dfrac{\dfrac{[x]_P.1.1 \vdash}{\vdash [x]_P.1}}{[x]_P \vdash} \qquad w[u,v] = \dfrac{\dfrac{xor^u.1 \vdash}{\vdash xor^u} \quad \dfrac{xor^v.1 \vdash}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash} \qquad \mathcal{R}(i) = \dfrac{.. \quad \dfrac{\dfrac{}{\vdash [l].1}\ \maltese}{[l] \vdash} \quad .. \quad \dfrac{}{[i] \vdash}\ \mathrm{p} \quad ..}{\vdash \xi}$$

Fig. 1. Device designs, and restriction design for $i \in \mathcal{S}_P$.

different branch of the negative rule, which effectively fork the interaction path. To finish we need to put the base design and all restriction designs together: they form *the generator* of a behaviour, obtained by bi-orthogonality. Therefore, let

$$\mathscr{B}_P = \mathbb{B}_P^{\perp\perp} = (\{\mathcal{D}_P\} \cup \mathcal{R}(x) \cup \mathcal{R}((u,v)))^{\perp\perp}$$

for all $x \in (\mathcal{S}_P \cup Loc_P)$, $(u,v) \in \mathcal{X}_P$ (note that $\mathcal{R}(x/(u,v))$ is actually a *set* of restrictions, usually more than one design). Then, the **interpretation of P** is $[\![P]\!] = (\mathscr{B}_P, [\ ]_P)$, the pair formed by the behaviour $\mathscr{B}_P$ and an assignment function $[\ ]_P$ from $\mathcal{S}_P$, $\mathcal{X}_P$ and Loc_P to addresses.

4 Main results

In this section we present results for MCCS, and will extend the setting to replication-free CCS only later, for which all the results remain valid. Solutions for the extension to replication are presented in the next section, giving a correspondence for full CCS.

The results consist of the expected correspondence between process execution and interaction in Ludics (Theorem 4.3), and between parallel composition and the merging of interpretations – that is, the operation $\otimes$ on behaviours, the ludical correspective of the linear logic tensor – when composing independent processes (Theorem 4.5); then, an operation mimicking execution on behaviours, giving us an intended weak subject reduction property (Theorem 4.6), and a characterization of deadlock-free processes, based on the notion of *visitable paths* of a behaviour (Theorem 4.8).

4.1 Correspondence between execution and interaction

In order to form a correspondence between the *dynamic* of a process P and its interpretation $[\![P]\!]$ we need to be able to extract from interaction the relevant information describing execution. The core notion is the one of *visited actions* inside an *interaction path*. The actions considered at each step by interaction are said *visited*, and an *interaction path* on a design is the sequence of visited actions. Our aim is to give a definition of *associated execution* which will make any orthogonal design describe an execution sequence, even if the empty one; at the same time the directed restrictions will ensure that *if* a design is orthogonal to $\mathscr{B}_P$ *then* its associated execution on the process P is *admissible*, i.e. it respects the partial order on synchronisations $<_{\mathcal{S}_P}$, and the *xor* conditions $\mathcal{X}_P$, thus being a possible execution sequence on P. The following notion is therefore well defined. Let $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}}$ be the sequence of actions of $\mathcal{D}$ visited during interaction with $\mathcal{C}$, we have

Definition 4.1 *Let $\mathcal{C} \in \mathscr{B}_P^{\perp}$. The execution on P* associated *to $\mathcal{C}$ is $\rightarrow_{i_1} \cdots \rightarrow_{i_n}$, the*

$$\dfrac{\dfrac{\dfrac{}{\vdash xor^u} \ \bindnasrepma\ \dfrac{xor^v.1 \vdash}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash} \quad \dfrac{}{[u] \vdash}\, p}{\vdash \xi} \qquad \dfrac{\dfrac{\dfrac{xor^u.1 \vdash}{\vdash xor^u} \ \bindnasrepma\ \dfrac{}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash} \quad \dfrac{}{[v] \vdash}\, p}{\vdash \xi}$$

Fig. 2. Two restriction designs for $(u,v) \in \mathcal{X}_P$.

execution sequence such that for all synchronisation $i \in (i_1, \ldots, i_n)$,

$$(-,[i],\{\{1\}\})(+,[i].1,\{1\}) = G[i] \in \mathcal{K}_{\mathcal{C}}^{\mathcal{D}_P},$$

ordered as they are visited by the interaction path.

4.1.1 An Example of associated execution

Example 4.2 *Let* $\mathcal{D}_P =$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}(+,\xi.1.1,\{1\})^4}{\xi.1 \vdash}(-,\xi.1,\{\{1\}\})^3 \quad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}(+,\xi.2.1,\{1\})^8}{\xi.2 \vdash}(-,\xi.2,\{\{1\}\})^7}{\vdash \xi}(+,\xi,I)^0$$

and $\mathcal{C} =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1, \Delta}\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1, \Delta}(-,\xi.2.1,\{\{1\}\})^9}{\vdash \xi.1.1.1, \xi.2, \Delta}(+,\xi.2,\{1\})^6}{\xi.1.1 \vdash \xi.2, \Delta}(-,\xi.1.1,\{\{1\}\})^5}{\vdash \xi.1, \xi.2, \Delta}(+,\xi.1,\{1\})^2}{\xi \vdash}(-,\xi,\{I\})^1$$

Assume $\xi.1 = [i]$ *and* $\xi.2 = [j]$. *Then* $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}_P} =$
$(+,\xi,I)(-,[i],\{\{1\}\})(+,[i].1,\{1\})(-,[j],\{\{1\}\})(+,[j].1,\{1\})$.

Therefore the execution associated to $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}_P}$ *is* $\rightarrow_{i,j}$. *Furthermore* i, j *are* minimal synchronisations *with respect to* $\leqslant_{\mathcal{S}_P}$, *since they are visited first in* $\mathcal{D}_P$ *(and therefore also have no xor conditions).*

By construction of the interpretation and definition of associated execution, we find the expected correspondence between *execution* on processes and *interaction* on behaviours:

Theorem 4.3 *Let* P *be a MCCS process,* $[\![P]\!]$ *characterizes all executions on* P.

With *characterizes* we mean that to each interaction between $\mathscr{B}_P$ and $\mathscr{B}_P^{\perp}$ is associated an execution, and each execution is associated to *at least* one interaction. It correspond to an *admissible* execution since the restriction designs force interaction to respect the partial order $<_P$ and conflict relation $\mathcal{X}_P$.

4.2 Parallel composition as merging of interpretations

To make the translation **modular** we need to represent the parallel composition on processes $P \mid Q$ by a composition of their respective behaviours $\mathscr{B}_P$ and $\mathscr{B}_Q$, via a logical operation on them. This operation, called *merging*, is based on $\otimes$, the Ludics equivalent of the linear logic tensor $\otimes$, which is the extension to behaviours of the more primitive composition $\odot$ on positive designs, also called *merging* (of designs). Informally, let $\mathcal{D}$ and $\mathcal{C}$ be designs of the same base $\vdash \xi$ and of respective first action $(+,\xi,I)$ and $(+,\xi,J)$, such that $I \cap J = \varnothing$. Then,

$$\mathcal{D} \odot \mathcal{C} = \{(+,\xi,I \cup J)\mathrm{c} \mid (+,\xi,I)\mathrm{c} \in \mathcal{D} \text{ or } (+,\xi,J)\mathrm{c} \in \mathcal{C}\}$$

where c denotes a branch (usually called a *chronicle*) of the design in question. If $I \cap J \neq \varnothing$ or either $\mathcal{D}$ or $\mathcal{C}$ are $\maltese$, then $\mathcal{D} \odot \mathcal{C} = \maltese$.

The full operation is defined thus:

Definition 4.4 *[18] Let $\mathscr{B}$ and $\mathscr{E}$ be* positive *behaviours, with disjoint ramifications of the first rule, and of same base. Then*

$$\mathscr{B} \bigotimes \mathscr{E} = \{\mathcal{D} \odot \mathcal{C} \mid \mathcal{D} \in \mathscr{B}, \mathcal{C} \in \mathscr{E}\}^{\perp\perp}$$

This operation, along with a composition of the assignments $[\ \ ]_P \cup [\ \ ]_Q$ and a few intermediate steps, let us compose the interpretations of two processes P and Q to achieve the interpretation of $P \mid Q$. In the trivial case where P and Q cannot communicate, it is a straight correspondence:

Lemma 4.1 *Given $\mathscr{B}_P$ and $\mathscr{B}_Q$ as behaviours with the same base, and disjoint ramifications of the first rule, if there is no communication between P and Q, then the corresponding interpretation is*

$$[\![P \mid Q]\!] = (\mathscr{B}_P \bigotimes \mathscr{B}_Q, [\ \ ]_P \cup [\ \ ]_Q).$$

The assumption poses no issues, since the addresses assigned by the function $[\ \ ]_P$ are completely arbitrary, and it only matters which element is assigned to which address. We may also assume a renaming of either $[\ \ ]_P$ or $[\ \ ]_Q$ to have them *disjoint*: this would let $\mathcal{D}_P$ and $\mathcal{D}_Q$ have the same base, but a *disjoint* ramification of the first action; i.e. no common sub-address, and thus no conflict in the assignments $[\ \ ]_P$ and $[\ \ ]_Q$. If there are possible communications between P and Q a few modifications are needed on the construction of the interpretation, to account for the new synchronisations generated in the parallel composition $P \mid Q$, and their *xor* conditions; but the logical core operation $\odot$ is kept intact on the base designs. The new objects that need to appear in the merging can be captured by the sets

$$new\mathcal{S}_{P|Q} = \mathcal{S}_{P|Q} \setminus (\mathcal{S}_P \cup \mathcal{S}_Q) \quad \text{and} \quad new\mathcal{X}_{P|Q} = \mathcal{X}_{P|Q} \setminus (\mathcal{X}_P \cup \mathcal{X}_Q).$$

We can deduce the *new* synchronisations, *xor* conditions and their restriction designs by checking $[\ \ ]_P$ and $[\ \ ]_Q$, which carry the information about channel names, with no need to know the structure of the processes. What we need to add is a design $\mathcal{N}_{P|Q}$ which accounts for the new element generated in the parallel composition. Given two processes P and Q, let $(+, \xi, I)$ and $(+, \xi, J)$ be the first action of, respectively, $\mathcal{D}_P$ and $\mathcal{D}_Q$; then $\mathcal{N}_{P|Q}$ is the following design, for $N \cap I = N \cap J = \varnothing$

$$\frac{G[k_1] \ \cdots \ G[k_n] \ \ w[x_1, y_1] \ \cdots \ w[x_n, y_n]}{\vdash \xi} \ (+, \xi, N)$$

where we have $\{k_1, \ldots, k_n\} = new\mathcal{S}_{P|Q}$, and $\{(x_1, y_1), \ldots, (x_n, y_n)\} = new\mathcal{X}_{P|Q}$.

With $[\ \ ]_{\mathcal{N}}$ we denote the assignment of $new\mathcal{S}_{P|Q}$ and $new\mathcal{X}_{P|Q}$ to addresses introduced by the ramification N. We then consider $\mathcal{D}_P \odot \mathcal{D}_Q \odot \mathcal{N}_{P|Q}$, and build the restriction designs on this extended base design in the same way, to make interaction respect $<_{P|Q}$, which is the union of the two partial orders $<_P \cup <_Q$ (no new location is generated), and $\mathcal{X}_{P|Q}$. The only new restrictions will be the ones about elements of $new\mathcal{S}_{P|Q}$ and $new\mathcal{X}_{P|Q}$. The result of the operation takes the base design $\mathcal{D}_P \odot$

$\mathcal{D}_Q \odot \mathcal{N}_{P|Q}$ together with the restrictions re-built on it, to generate a behaviour by bi-orthogonal closure; the result of this operation is denoted $(\mathbb{B}_P \oplus \mathbb{B}_Q)^{\perp\perp}$. Then, by taking the union of the assignments we get the *merging of interpretations*:

$$[\![P]\!] \oplus [\![Q]\!] = ((\mathbb{B}_P \oplus \mathbb{B}_Q)^{\perp\perp}, [\ \ ]_{P\odot Q} \cup [\ \ ]_{\mathcal{N}}).$$

The following result is a generalization of the previous lemma:

Theorem 4.5 *Let P, Q be MCCS processes. We have $[\![P]\!] \oplus [\![Q]\!] = [\![P \mid Q]\!]$.*

4.3 Subject reduction property

An explanation of the interpretation can be found by understanding process execution inside $[\![P]\!]$. This can be seen through a subject reduction property. We define the *reduction* on $[\![P]\!]$ for a given $u \in \mathcal{S}_P$ as an operation that matches execution $\rightarrow_u$ on P, denoted $[\![P]\!] \rightsquigarrow_u ([\![P]\!])_u$. Technically, we use an operation called *trimming*, that is a carefully defined *projection* on behaviours (originally defined in [13] and [18]), essentially a removal of a sub-ramification from the first action $(+, \xi, I)$ of $\mathcal{D}_P$ (operation that affects the whole behaviour, since it is built on $\mathcal{D}_P$). Informally, reduction is defined by erasing a sub-ramification, thus the entire sub-designs $G[x]$ or $w[x, y]$, corresponding to a designed synchronisation $u = (l, m)$, called the branches *associated* to u: $G[u]$, $G[l]$ and $G[m]$, $G[x]$ and $w[u, x]$ for $x \in xor(u)$, and $w[x, y]$ for $y \in xor(x)$. Let $K = \{i, \ldots, m\}$ be the corresponding sub-ramification, then the first action of $\mathcal{D}_P$ becomes $(+, \xi, I \setminus K)$.

As a consequence, all the $\mathcal{R}(P)$ will miss the same sub-designs, as well as the whole behaviour $\mathscr{B}_P$. There is a sort of inclusion of $([\![P]\!])_u$ in $[\![P]\!]$ from the point of view of the possible interactions and associated executions; that is, for each interaction path $\mathcal{K}_{\mathcal{C}'}$ on $([\![P]\!])_u$, there is an interaction path $\mathcal{K}_{\mathcal{C}}$ on $[\![P]\!]$ such that the execution associated to $\mathcal{K}_{\mathcal{C}'}$ is either the same, or a *sub-execution*, of the one associated to $\mathcal{K}_{\mathcal{C}}$. This second case holds if $\mathcal{K}_{\mathcal{C}'}$ visits $G[v]$ for a synchronisation v such that $u \lesssim_{\mathcal{S}_P} v$ – since $G[u]$ has been erased in $([\![P]\!])_u$, then execution on $G[v]$ is directly possible, while on $[\![P]\!]$, $G[u]$ must be visited first. The first case holds, in general, since $(\mathcal{D}_P)_u$ is strictly smaller than $\mathcal{D}_P$, thus any interaction on $(\mathscr{B}_P)_u$ is also possible on $\mathscr{B}_P$. Note that the restrictions corresponding to the elements *associated* to u automatically disappear in the behaviour $(\mathscr{B}_P)_u$: since the branches of these elements are erased, there are no more ✣ and prunings in the restrictions in question, making them exactly *equal* to $(\mathcal{D}_P)_u$.

The reduction on the interpretation is the operational side of our interpretation:

Theorem 4.6 *Let P be a MCCS process interpreted by $[\![P]\!]$. Given a synchronisation $u \in \mathcal{S}_P$ such that $P \rightarrow_u P'$, we have $[\![P']\!] = ([\![P]\!])_u$, i.e.*

$$\begin{array}{ccc} P & \rightarrow_u & P' \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![P]\!] & \rightsquigarrow_u & ([\![P]\!])_u \end{array}$$

The interpretation is thus not preserved during execution, but this must be the case if we want to fully represent the *dynamic* of a process. Indeed we preserve the *non-*

confluence of execution: $[\![P]\!]$ keeps all the possible executions to normal forms of P. As a consequence, for some maximal execution sequences, we have:

Corollary 4.7 *Let* $\mathbb{O}ne = \frac{}{\vdash \xi}^{(+,\xi,\varnothing)}$ *(the design generating the behaviour that corresponds to the linear logic multiplicative unit* 1*), and let* P *be a MCCS process:*

$$\textit{if } P \to^* 1 \textit{ then } [\![P]\!] \rightsquigarrow^* \{\mathbb{O}ne\}^{\perp\perp} \textit{ for the same synchronisation sequence.}$$

4.4 Deadlocks

By restricting a behaviour interpreting a process P to the set of its **visitable paths**, i.e. the branches of $\mathscr{B}_P$ which are actually visited by some interaction with $\mathscr{B}_P^{\perp}$, we obtain what is called the *incarnation* of a behaviour, which can give us some important information about the process. The result can be seen as a projection on $\mathscr{B}_P$ that erases the sub-ramification – and thus the sub-designs – *never* visited during interaction. We can restrict the definition to the *visitable part* of $\mathcal{D}_P$, denoted $|\mathcal{D}_P|$, i.e. the branches of $\mathcal{D}_P$ visited by some interaction, with the following consequences:

Theorem 4.8 (i) *If* $|\mathcal{D}_P| = \mathcal{D}_P$*, then* P *is* deadlock free.

(ii) *If* $|\mathcal{D}_P| \neq \mathcal{D}_P$*, then the* non visitable *part of* $\mathcal{D}_P$ *is never accessed by interaction, and its process counterpart represent the common part of all normal forms, where no synchronizations can occur (for any execution path); moreover,* P is not reducible to 1.

A process P is *deadlocked* if P is in *normal form*, $P \neq 1$, and there is also a *cycle* in the partial order $\preccurlyeq_{\mathcal{S}_P}$ on synchronisations; other definitions focus on the lack of communication on certain chosen occurrences of channel names (as for instance in [14]). In case (i), P is *deadlock free* since any part of P is potentially synchronizable (is visitable in $[\![P]\!]$). We don't know if $P \to^* 1$, but for each channel there is at least one execution path where it is synchronized with a dual, i.e. any part of P can be accessed by some execution. Thus, there are no *cycles* in the order $\preccurlyeq_{\mathcal{S}_P}$, and we can potentially communicate with any channel of P (i.e for all channels there is a synchronizing execution sequence).

In case (ii), for the correspondence between interaction and execution, if a part of $\mathcal{D}_P$ is never visited, then there are some channels of the process that cannot be synchronized (either there is no dual, a deadlock, or the execution is blocked for some other reason) and, since each occurrence of a channel name is represented in $[\![P]\!]$, we also know which these channels are. Notice that $|\mathcal{D}_P|$ is the only relevant design to check, since the restriction designs are modifications of $\mathcal{D}_P$ only needed to restrain the possible interactions, and interaction-order. Thus, if an interaction visits any branch of a restriction $\mathcal{R}(\cdots)$, then it visits the same branch on $\mathcal{D}_P$.

For the same correspondence, we can interpret deadlocked processes in behaviours with no issues: the deadlocked part will just be *non visitable* in $[\![P]\!]$. For example:

$$P = a^1.\overline{b}^2.Q \mid b^3.\overline{c}^4.R \mid c^5.\overline{a}^6.S$$

is deadlocked, with a cycle $u_1 = (a^1, \overline{a}^6) \preccurlyeq_{\mathcal{S}_P} u_2 = (\overline{b}^2, b^3) \preccurlyeq_{\mathcal{S}_P} u_3 = (\overline{c}^4, c^5) \preccurlyeq_{\mathcal{S}_P} u_1$.

On the side of $[\![P]\!]$, we have restrictions which will prevent us to interact with any u_i, since each $G[u_i]$ $(1 \leq i \leq 3)$ have as requisite a $G[l]$ for a location l of

another synchronisation in the cycle, which is not accessible as well for the same reason; therefore, there will be no orthogonal design interacting with any $G[u_i]$.

4.5 Extension to replication-free CCS: Non-deterministic Choice and Hiding

We present here the replication-free fragment of CCS, and the extension of the previous results. The non-deterministic choice + (also called *sum*) is a mutual exclusion between its two members; it waits for an *external* choice, i.e. a context in parallel composition, which selects one process to use by synchronizing with the channels of one of the two members, dropping the other for that execution path. Execution is therefore generalized in the following way:

$$P = a^l.P' + b^m.Q' \mid \overline{a}^n.P'' + \overline{b}^o.Q'' \rightarrow_{(l,n)} P' \mid P''$$

From the point of view of interaction the interpretation is extended with a *xor* condition in $\mathcal{X}_P$ on the synchronisations on a and b, i.e $u = (a^l, \overline{a}^n)$ and $v = (b^m, \overline{b}^o)$. Indeed, once we have performed execution on one, the other is excluded from the same execution path. On the other hand, both are possible until a choice is made, and choosing one of the two synchronisations u and v, by transitivity of the partial order $<_P$ on locations, necessarily exclude from any further execution all the internal synchronisations of P' (if v is chosen) or P'' (if u is chosen) – as in event structures, where the conflict relation is hereditary w.r.t. causal implication. This is effectively described by the *xor* relation already present in the MCCS interpretation, that can mimic the non-deterministic choice, by extending it to members of a sum +.

The hiding operator extends the term syntax with $\nu a(P)$, also known as *restriction* $P\backslash a$ in early texts. It declares that a channel name is bound and *private* inside its scope, then hidden, i.e. that cannot communicate with channels outside its scope. If $P = \nu a(a^l.R) \mid \overline{a}^m.Q$ then the pair $(a^l, \overline{a}^m)$ cannot synchronize, and hence the channels would not be able to communicate. The execution rule is then considered under the scope of ν operators, up to common structural equivalence, which let push the hiding inside a process, until it reaches its maximal/minimal scope:

$$\nu a(P \mid Q) \equiv \nu a(P) \mid Q, \text{ if } a \notin fv(Q),$$
$$\nu a(P) \mid \nu b(Q) \equiv \nu a(\nu b(P \mid Q)) \equiv \nu b(\nu a(P \mid Q)), \text{ if } a \notin fv(Q) \text{ and } b \notin fv(P).$$

where fv notes the un-bound channels of a process. For a simple interpretation of hiding, we already have all the needed material: we restrict our definition of synchronisations so that only some pairs of dual channels are considered. We can denote with a^l_ν such a bound name; then we exclude from $\mathcal{S}_P$ any pair $(a^l, \overline{a}^m)$ such that *only one* channel is tagged, i.e. either is a^l_ν or $\overline{a}^m_\nu$. If both are tagged then it is still a synchronisation. This implies that inside $\mathcal{D}_P$ and $\mathcal{R}(P)$ there is no trace at all of the hiding that can occur in the process P, we can only check its presence from the static assignment $[\ ]_P$, which only tells us singularly which channels occurrences are hidden. The result is that we forbid some interaction paths by *not interpreting*, instead of resorting to more restrictions.

5 Replication: a reformulation in computational Ludics

To handle replication one can follow the ideas of [22] to type event structures, by considering a restricted version of the π-calculus[1], a linear typed version of Sangiorgi's πI-calculus [20]. Such calculus has the same expressive power as the version with free name passing, and linearity only breaks for replicated outputs, but determinism is preserved by the uniqueness of inputs. This allow us to keep both $\preccurlyeq_{\mathcal{S}_P}$ and locations for synchronisations, even with replication. By this way the presented tools can be applied.

Another possibility to represent replication comes from an already existing version of Ludics dedicated for this. K. Terui formulated a complementary syntax for Ludics called *computational Ludics* [21] (aka c-ludics), closer to higher order π-calculus, to achieve practical advantages with the general goal of developing an *interactive* theory of computability and *complexity* based on Ludics. The feature that seems most interesting to us is the possibility to represent infinite designs by a *finite generator*, allowing *recursive definitions*. Design generators let us easily extend the interpretation to the full calculus with replication.

Terui's syntax is based on a *signature* $\mathscr{A} = (A, ar)$, where A is a set of *names*, and $ar : A \to \mathbb{N}$ is a function giving an arity to each name. A denumerable set of variables V is needed, denoted $x, y, z, \ldots$. A *positive action* is either $\maltese$, Ω (noting the divergence), or $\overline{a}$, with $a \in A$; a *negative* action is $x \in V$, or $a(x_1, \ldots, x_n)$, with $a \in A$ and $ar(a) = n$. $x_1, \ldots, x_n$ are distinct variables, and $\overrightarrow{x}_a$ denotes a vector of variables of the arity of a. Informally, a design $\mathcal{D}$ is co-inductively defined by

$$P ::= \maltese \mid \Omega \mid (N_0 \mid \overline{a}\langle N_1, \ldots, N_n\rangle),$$
$$N ::= x \mid \sum a(\overrightarrow{x}_a).P_a.$$

P denotes the *positive* actions, N the negative actions. A name denotes both the polarity and cardinality of the ramification of a rule, and, in the negative rule, the variables stand for each sub-address of the ramification. If N_0 is not a variable x in a positive design, then it becomes a *cut*.

A reformulation of the designs used in the translation is possible, since it holds the following:

Remark 5.1 ***Standard*** *c-designs*[2] *(i.e.* $\neq \Omega$*, linear, cut-free and identity-free) correspond to the original designs.*

Consequently, it is easy to show that using the pruning to build restriction designs can naturally be applied to c-designs. Assuming an assignment from $Loc_P, \mathcal{S}_P, \mathcal{X}_P$ to names, we have the following correspondence:

- $G[u] = [u](x_u).(x_u \mid \overline{[u.1]}\langle 0\rangle)$.
- $w[u, v] = [xor^u](x_u).x_u \mid \overline{[xor^u.1]}\langle 0\rangle + [xor^v](x_v).x_v \mid \overline{[xor^v.1]}\langle 0\rangle$.
- $\mathcal{D}_P = x_0 \mid \overline{a}\langle G[x], \ldots, w[x, y], \ldots\rangle$, with x varying on $\mathcal{S}_P \cup Loc_P$, and (x, y) on $\mathcal{X}_P$;

where $[\ ]$ denotes the assignment function. Interaction is called *reduction*, and

[1] Notice that [22] introduces such calculus to bypass the main difficulty to extend CCS semantics to the π-calculus: to switch from dynamic α-conversion that allows to represent the dynamic creation of a name, to a static one at typing time.

[2] See [21], remark 2.2.

is defined in λ-calculus style on positive c-designs with a cut, by:

$$(\sum a(\overrightarrow{x_a}.P_a) \mid \overline{a}\langle\overrightarrow{N}\rangle) \to P_a[\overrightarrow{N}/\overrightarrow{x_a}]$$

where $\overrightarrow{N}$ is of lenght $ar(a)$. The reduction relation select the $a(\overrightarrow{x_a})$ that matches with $\overline{a}\langle\overrightarrow{N}\rangle$, thus assuring us that they have the same arity. Then, in the corresponding P_a, each variable is substituted by a negative design inside the scope of $\overline{a}$; reduction can then continue on $P_a[\overrightarrow{N}/\overrightarrow{x_a}]$, until a normal form is reached (a variable x or $\maltese$), or it diverges (Ω). Using the reduction relation, we can rewrite example 3.1 using prunings and $\maltese$ easily:

- $\mathcal{D} = x_0 \mid \overline{a}_0\langle a_1(x_1).x_1 \mid \overline{a_{11}}\langle 0\rangle,\ a_2(x_2).x_2 \mid \overline{a_{21}}\langle 0\rangle\rangle$
- $\mathcal{C} = a_0(x_1, x_2).x_1 \mid \overline{a}_1\langle a_{11}(x_{11}).x_2 \mid \overline{a}_2\langle a_{21}(x_{21}).\maltese\rangle\rangle$.
- $\mathcal{E} = a_0(x_1, x_2).x_2 \mid \overline{a}_2\langle a_{21}(x_{21}).x_1 \mid \overline{a}_1\langle a_{11}(x_{11}).\maltese\rangle\rangle$
- $\mathcal{D}^* = x_0 \mid \overline{a}_0\langle 0^p, a_2(x_2).\maltese\rangle$.

where ^{p} denotes a pruning, and 0 the empty negative action: it denotes that the + action has a premise, but there is no further action in the branch. Both reductions $\mathcal{C} \mid \mathcal{D}$ and $\mathcal{E} \mid \mathcal{D}$ reach $\maltese$ after five reduction steps. Instead, only $\mathcal{E}$ is orthogonal to $\mathcal{D}^*$. To perform reduction, we must substitute x_0 in $\mathcal{D}^*$ with $\mathcal{E}$, obtaining

$$\mathcal{E} \mid \mathcal{D}^* = (a_0(x_1, x_2).x_2 \mid \overline{a}_2\langle a_{21}(x_{21}).x_1 \mid \overline{a}_1\langle a_{11}(x_{11}).\maltese\rangle\rangle) \mid \overline{a}_0\langle 0^p, a_2(x_2).\maltese\rangle$$

Reduction reach $\maltese$ in only 2 steps:

(i) $(a_2(x_2).\maltese) \mid (\overline{a}_2\langle a_{21}(x_{21}).0^p) \mid \overline{a}_1\langle a_{11}(x_{11}).\maltese\rangle\rangle$.

(ii) $\maltese$.

Instead, the reduction

$$\mathcal{C} \mid \mathcal{D}^* = (a_0(x_1, x_2).x_1 \mid \overline{a}_1\langle a_{11}(x_{11}).x_2 \mid \overline{a}_2\langle a_{21}(x_{21}).\maltese\rangle\rangle) | \overline{a}_0\langle 0^p, a_2(x_2).\maltese\rangle$$

at the second step diverges:

(i) $(0^p) \mid (\overline{a}_1\langle a_{11}(x_{11}).\maltese) \mid a_2(x_2).\maltese \mid \overline{a}_2\langle a_{21}(x_{21}).\rangle\rangle$.

(ii) Ω; since 0^p has no P_0 and variables to perform the substitution on.

Therefore, the reformulation in c-ludics does not affect restriction designs, or the behaviour $\mathscr{B}_P$, and the correspondence between execution and interaction still holds. About the merging of interpretations, the operation $\odot$ is simply an extension of the arity of a positive rule – or a substitution with a name of the needed arity – by putting together in the scope of this action all the negative designs that we have. Thus from $\mathcal{D} = x_0 \mid \overline{a_1}\langle N_1, \ldots, N_k\rangle$ and $\mathcal{C} = x_0 \mid \overline{a_2}\langle N_{k+1}, \ldots, N_{k+n}\rangle$ we obtain $\mathcal{D} \odot \mathcal{C} = x_0 \mid \overline{a_3}\langle N_1, \ldots, N_k, N_{k+1}, \ldots, N_{k+n}\rangle$.

Trimming, instead, require us to erase the sub-ramification associated to a certain synchronisation from $\mathcal{D}_P$, thus reducing the arity of the first action $\overline{a}$, and removing the corresponding negative designs (the dual operation of $\odot$). A substitution to a name of the right arity might be required, but the operation itself poses no issues. When reducing $\mathcal{D}_P$ to $\mathbb{O}ne$, the form we obtain is $\mathbb{O}ne = x_0 \mid \overline{a}\langle\rangle = x_0 \mid \overline{a}$, a positive

action with a 0-ary name.

The main issue with c-ludics is that some difficulties arises when defining the assignment function, since names require modifications every time we act on $\mathcal{D}_P$, and variables are not absolute values. Both cases complicate the read back from $[\![P]\!]$ to elements and relations of the process. This forces the assignment to be *deduced* from the structure of a c-design, and not be independent from it anymore, while also losing the $1-1$ correspondence with elements of the process.

6 Conclusion

The interpretation of CCS into Ludics tries to overcome the problems, and satisfy the goals, that motivated our work. The main properties which characterize it are:

- A logical characterization of the full dynamic of processes without imposing functional behaviour, resorting to multiple translations by partially determinizing execution via scheduling, or sacrificing the non-determinism or non-confluence itself (by imposing linearity and other constraints on the syntax).
- A partial *modularity* in the interpretation, which let us combine the interpreting structures, behaviours in our case, as we do with processes via parallel composition. We are able to represent the composition via a ludical operation on behaviours that, in the trivial case where there is no communication between two processes, exactly interpret the linear logic tensor $\otimes$. Otherwise, some more artificial and non-ludical steps are required, by working on the assignment functions, but the core operation $\odot$ is kept intact on the base designs.
- Insights on the dynamics of processes, as expliciting what parallel composition entails when two process communicate, what causes forks in an execution paths, and how the different reduced forms of a process are related through their interpretations.

As a consequence of these properties, we have that:

- subject reduction describe a particular inclusion between the interpretations of a process and of one of its reduced forms, with respect to their structures and possible interactions;
- along with execution, *deadlocks* are also characterized. Instead of being a property of the interpretation – as it is with typing systems, where if a process is typable, then it is deadlock free – the interpretation in Ludics is oblivious of their presence, as is interaction on behaviours. Still, we have a way to know if a process is deadlock-free, or if it can't be reduced to 1, via the *visitable paths* of a behaviour.

Finally, the reformulation in c-ludics [21] let us have access to non-linearity and recursive definition in the form of *finite designs generators*, and thus extend the interpretation to the full calculus by using the same techniques presented here.

Another interesting point requiring further investigation is the evident strong connection with *event structures*, which seem naturally represented by directed behaviours, generated via restriction designs of a base one. Indeed, event structures are indirectly already represented, passing through processes, since they are a model of CCS-like calculi [23].

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[2] Michele Basaldella and Claudia Faggian. Ludics with repetitions (exponentials, interactive types and completeness). *Logical Methods in Computer Science*, 7(2), 2011.

[3] Emmanuel Beffara. A concurrent model for linear logic. *Electr. Notes Theor. Comput. Sci.*, 155:147–168, 2006.

[4] Emmanuel Beffara. A logical view on scheduling in concurrency. In *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014.*, pages 78–92, 2014.

[5] Emmanuel Beffara and François Maurel. Concurrent nets: A study of prefixing in process calculi. *Theor. Comput. Sci.*, 356(3):356–373, 2006.

[6] Emmanuel Beffara and Virgile Mogbil. Proofs as executions. In *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, pages 280–294, 2012.

[7] Pierre-Louis Curien and Claudia Faggian. L-nets, strategies and proof-nets. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, pages 167–183, 2005.

[8] Thomas Ehrhard and Olivier Laurent. Interpreting a finitary π-calculus in differential interaction nets. *Inf. Comput.*, 208(6):606–633, 2010.

[9] Claudia Faggian and Mauro Piccolo. Ludics is a model for the finitary linear π-calculus. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, pages 148–162, 2007.

[10] Claudia Faggian and Mauro Piccolo. Partial orders, event structures and linear strategies. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 95–111, 2009.

[11] Christophe Fouqueré and Myriam Quatrini. Study of behaviours via visitable paths. *Submitted to journal*, 2016.

[12] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[13] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[14] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, pages 489–503, 2000.

[15] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[16] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[17] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[18] Myriam Quatrini. *La Ludique : une théorie de l'interaction, de la logique mathématique au langage naturel.* Habilitation à diriger des recherches, Université Aix-Marseille, May 2014.

[19] Grzegorz Rozenberg and P. S. Thiagarajan. Petri nets: Basic notions, structure, behaviour. In *Current Trends in Concurrency, Overviews and Tutorials*, pages 585–668. 1986.

[20] Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.

[21] Kazushige Terui. Computational ludics. *Theor. Comput. Sci.*, 412(20):2048–2071, 2011.

[22] Daniele Varacca and Nobuko Yoshida. Typed event structures and the linear π-calculus. *Theor. Comput. Sci.*, 411(19):1949–1973, 2010.

[23] Glynn Winskel. Event structure semantics for CCS and related languages. In *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, pages 561–576, 1982.

[24] Glynn Winskel. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, pages 325–392, 1986.

[25] Glynn Winskel and Mogens Nielsen. *Models for Concurrency*, volume 4. Oxford Clarendon Press, 1995.

7 Appendix

Here we provide details on the paper in the form of a more formal presentation of the basic definitions, a more in-depth discussion of the problems and solutions, and proofs of the results.

7.1 Restriction designs

What we need to build the restriction designs is to associate to each synchronisation the local requisites that makes it available for execution, and to each location the synchronisations that let us erase it in execution *and* the 1-step smaller location by which is blocked. The associated set of elements to each synchronisation or location is the following:

Definition 7.1 (freedom requisite) *Let P be a MCCS process. A* freedom requisite *for $i = (a^l, \overline{a}^h) \in \mathcal{S}_P$ is the set*

$$\mathscr{F}(i) = \{m, n \in Loc_P \mid l <_P m \vee h <_P n\}.$$

Note that

1) *If i is* minimal*, $\mathscr{F}(i) = \varnothing$.*
2) *If i has only* one minimal location*, then $\mathscr{F}(i) = \{m\}$, where p is the* very precedent *location of l.*
3) *If i has* no minimal location*, then $\mathscr{F}(i) = \{m, n\}$, without loss of generality $m <_P l$, $n <_P h$, and m, n are the respective* predecessors *of l, h.*

Let $l \in Loc_P$, $\mathscr{F}(l) = \{i \in \mathcal{S}_P \mid l \in i\}$ if $\exists i \in \mathcal{S}_P$ such that $l \in i$; otherwise $\mathscr{F}(l) = \perp$, i.e in $\mathcal{R}(l)$, $G[l]$ is going to be pruned, without a $\maltese$ also present in the design: in this way interaction on $G[l]$ is not possible.

Each restriction $\mathcal{R}(i), \mathcal{R}(l), \ldots$ is thus shaped to represent the *local* order expressed by the freedom requisites.

Definition 7.2 *Let $i \in \mathcal{S}_P$. The restrictions designs for i (also called* non commutative restrictions*), noted $\mathcal{R}(i)$, is a set of designs built from $\mathcal{D}_P$. If $\mathscr{F}(i) = \varnothing$, then no restriction is added (we omit the subscript $[\ ]_P$ when noting the associated addresses).*

If $\mathscr{F}(i) = \{l\}$, then $\mathcal{R}(i)$ is $\mathcal{D}_P$ thus modified:

$$\mathcal{R}(i) = \dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [l].1}\ \maltese}{[l] \vdash} \quad \ldots \quad \dfrac{}{[i] \vdash}\ p \quad \ldots}{\vdash \xi}$$

If $\mathscr{F}(i) = \{m, n\}$, then $\mathcal{R}(i)$ is the following pair of designs:

$$\dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [m].1}\ \maltese}{[m] \vdash} \quad \cdots \quad \dfrac{}{[i] \vdash}\ p}{\vdash \xi} \qquad \dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [n].1}\ \maltese}{[n] \vdash} \quad \cdots \quad \dfrac{}{[i] \vdash}\ p}{\vdash \xi}$$

Let $l, m \in Loc_P$, $m <_P l$. Then, if $\mathscr{F}(l) = \{i_1, \ldots, i_n\}$, $\mathcal{R}(l)$ is the following two designs:

$$\dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [m].1}\,\clubsuit}{[m] \vdash} \quad \cdots \quad \dfrac{}{[l] \vdash}\,p}{\vdash \xi}$$

along with

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [i_1].1}\,\clubsuit}{[i_1] \vdash} \quad \ldots \quad \dfrac{\dfrac{}{\vdash [i_n].1}\,\clubsuit}{[i_n] \vdash} \quad \ldots \quad \dfrac{}{[l] \vdash}\,p \quad \ldots}{\vdash \xi}$$

If $\mathscr{F}(l) = \varnothing$, the second design becomes: $\dfrac{\ldots \quad \dfrac{}{[l] \vdash}\,p \quad \ldots}{\vdash \xi}$

$\mathcal{R}(\mathcal{X}_P)$ is, for each $(u, v) \in \mathcal{X}_P$, the set containing the following two designs:

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash xor^u}\,\clubsuit \quad \dfrac{xor^v.1 \vdash}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash} \quad \ldots \quad \dfrac{}{[u] \vdash}\,p \quad \ldots}{\vdash \xi}$$

and equivalently for v:

$$\dfrac{\ldots \quad \dfrac{\dfrac{xor^u.1 \vdash}{\vdash xor^u} \quad \dfrac{}{\vdash xor^v}\,\clubsuit}{xor^u \,\&\, xor^v \vdash} \quad \ldots \quad \dfrac{}{[v] \vdash}\,p \quad \ldots}{\vdash \xi}$$

With $\mathcal{R}(P)$ we note the set $\mathcal{R}(\mathcal{X}_P) \cup \{\mathcal{R}(i), \mathcal{R}(l) \mid i \in \mathcal{S}_P, l \in Loc_P\}$. The $\cdots$ notes all the other chronicles of $\mathcal{D}_P$, which are left untouched.

7.1.1 A corollary with a constructive proof

Lemma 7.3 *Let P be a MCCS process. Then $P \to_i$ is an admissible execution if and only if $\exists \mathcal{C} \in \mathscr{B}_P^{\perp}$ such that the execution associated to $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}_P}$ starts with i.*

Proof.

($\Leftarrow$) If $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}_P}$ visits $G[i]$ before any other $G[x]$, then its associated execution starts with i, i.e. i is a minimal synchronisation, therefore $P \to_i$ is an admissible execution.

For $\Rightarrow$ we can note that if $P \to_i$, then i is a minimal synchronisation, thus $\mathscr{F}(i) = \varnothing$. Then we can easily *build* an orthogonal design interacting first (and only

with) $G[i]$. Let $[i]^{\perp}$ =

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash [i].1.1, \Delta}\,\maltese}{[i].1 \vdash \Delta}\,(-, i.1, \{\{1\}\})}{\vdash [i], xor^{i_1}.1, \ldots, xor^{i_n}.1, \Delta}\,(+, i, \{1\})}{xor^{i_n} \vdash [i], xor^{i_1}.1, \ldots, xor^{i_{n-1}}\Delta}\,(-, xor^{i_n}, \{\{1\}\})}{\vdots}}{xor^{i_1} \vdash xor^{i_2} \,\&\, xor^{j_2} \ldots, xor^{i_n} \,\&\, xor^{j_n}, [i], \Delta}\,(-, xor^{i_1}, \{\{1\}\})}{\vdash xor^{i_1} \,\&\, xor^{j_1}, \ldots, xor^{i_n} \,\&\, xor^{j_n}, [i], \Delta}\,(+, xor^{i_1} \,\&\, xor^{j_1}, \{xor^{i_1}\})}{\xi \vdash}\,(-, \xi, \{I\})$$

Then $[i]^{\perp} \perp \mathscr{B}_P$ and the execution associated to $\mathcal{K}_{\mathcal{C}}^{\mathcal{D}_{\mathcal{P}}}$ is $\rightarrow_i$.

Note that in Δ must appear the address of each premise of $\mathcal{D}_P$, since the ramifications of $\mathcal{D}_P$ and $[i]^{\perp}$ must match. The addresses $xor^{i_1} \,\&\, xor^{j_1}, \ldots, xor^{i_n} \,\&\, xor^{j_n}$ are the xor clauses for each $(i, x) \in \mathcal{X}_P$.

□

7.2 Merging

Example 7.4 *Let* $\mathcal{C} = \dfrac{\dfrac{\vdots}{\xi 1 \vdash} \quad \dfrac{\vdots}{\xi 3 \vdash}}{\vdash \xi}$; $\mathcal{D} = \dfrac{\dfrac{\vdots}{\zeta 1 \vdash} \quad \dfrac{\vdots}{\zeta 2 \vdash} \quad \dfrac{\vdots}{\zeta 3 \vdash}}{\vdash \zeta}$

Let Θ *be an unification of the bases of* $\mathcal{C}$ *and* $\mathcal{D}$ *such that* $\Theta\mathcal{C} = \mathcal{C}$*. We have*

$\Theta\mathcal{D} = \dfrac{\dfrac{\vdots}{\xi 1 \vdash} \quad \dfrac{\vdots}{\xi 2 \vdash} \quad \dfrac{\vdots}{\xi 3 \vdash}}{\vdash \xi}$

Then, to make $\mathcal{C}$ *and* $\Theta(\mathcal{D})$ compatible *for merging, we need to rewrite one of the ramifications, to make them disjoint. Let* $I = \{1, 2, 3\}$ *the ramification of the first action of* $\mathcal{D}$*, then the* renaming *(of the ramification) of* $\mathcal{D}$ *will be*

$\mathcal{D}[^{J=\{4,5,6\}}/_{I}] = \dfrac{\dfrac{\vdots}{\xi 4 \vdash} \quad \dfrac{\vdots}{\xi 5 \vdash} \quad \dfrac{\vdots}{\xi 6 \vdash}}{\vdash \xi}$

This let us easily perform the merging *of* $\mathcal{C}$ *and* $\Theta\mathcal{D}$*:*

$\mathcal{C} \odot (\Theta\mathcal{D}) = \dfrac{\dfrac{\vdots}{\xi 1 \vdash} \quad \dfrac{\vdots}{\xi 3 \vdash} \quad \dfrac{\vdots}{\xi 4 \vdash} \quad \dfrac{\vdots}{\xi 5 \vdash} \quad \dfrac{\vdots}{\xi 6 \vdash}}{\vdash \xi}\,(+, \xi, \{1, \ldots, 6\})$

The new objects that need to appear in the merging can be captured by the sets:

$new\mathcal{S}_{P \mid Q} = \mathcal{S}_{P \mid Q} \setminus \mathcal{S}_P \cup \mathcal{S}_Q$

$new\mathcal{X}_{P \mid Q} = \mathcal{X}_{P \mid Q} \setminus \mathcal{X}_P \cup \mathcal{X}_Q$.

We have that $new\mathcal{S}_{P \mid Q}$ is the set of cuts $i, j, \ldots$ that arise in the parallel composition: the pairs $(a^l, \overline{a}^m)$ such that, w.l.o.g., $a \in P$ and $\overline{a} \in Q$. $new\mathcal{X}_{P \mid Q}$ by definition is the set containing only the new xor clauses: pairs of cuts (u, v) with a

common location, such that u or v belong to $new\mathcal{S}_{P \mid Q}$ (thus the clause was not in $\mathcal{X}_P$ or $\mathcal{X}_Q$).

While the definitions holds, we need a way to recover these sets *without looking* at the structure of $P \mid Q$ (and thus to the internal structure of P and Q); otherwise we would be pretty much re-building the interpretation from zero. To this end, we can just look at the domain of the assignments $[\ \ ]_P$ and $[\ \ ]_Q$ (which includes occurrences of channel names of P and Q): it suffice to take each pair $(a^l, \overline{a}^m)$ such that (w.l.o.g.) $a^l \in [\ \ ]_P$ and $\overline{a}^m \in [\ \ ]_Q$ to recover $new\mathcal{S}_{P \mid Q}$ (we skip the trivial proof). Being locations in place of channel occurrences, we assume to always be able to switch from one to the others, and thus know which channel name the location in question is labeling. In this way we can recover the information we need, while remaining oblivious of the structure of the processes involved.

Then $new\mathcal{X}_{P \mid Q}$ are all the pair (u, v) where u and v share a location, and such that, w.l.o.g (it suffice to be one of the two) $u \in new\mathcal{S}_{P \mid Q}$, and if $v \notin new\mathcal{S}_{P \mid Q}$ then $v \in [\ \ ]_P$ or $[\ \ ]_Q$ (again, a trivial proof). Therefore, if we have $[\ \ ]_P$ and $[\ \ ]_Q$ we can always assume to have $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$, which can be build by a local assignment check (the structure of the processes remain unknown).

From the definition of $\odot$, $\mathcal{D}_P$, $\mathcal{D}_Q$ and $\mathcal{N}_{P \mid Q}$ we have the following

Lemma 7.5 *Let P, Q be MCCS processes. Then*

$$\mathcal{D}_P \bigodot \mathcal{D}_Q = \mathcal{D}_P \odot \mathcal{D}_Q \odot \mathcal{N}_{P \mid Q} = \mathcal{D}_{P \mid Q},$$

under the assumption that we have built both designs $\mathcal{D}_P \bigodot \mathcal{D}_Q$ and $\mathcal{D}_{P \mid Q}$ with the same ramification in the first rule.

Proof. It suffice to show that $\mathcal{N}_{P \mid Q}$ has the ramification corresponding to the elements of $P \mid Q$ missing from either P or Q, i.e. the new cuts and their *xor* conditions. Trivially, it holds by construction of $\mathcal{N}_{P \mid Q}$, and definition of $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$. The other elements of the joint ramifications of $\mathcal{D}_P \bigodot \mathcal{D}_Q$ are obviously present in $\mathcal{D}_{P \mid Q}$: the locations are the same ($Loc_{P \mid Q} = Loc_P \cup Loc_Q$) as well as the partial order, since no new channel occurrence is added, thus $<_{P \mid Q} = <_P \cup <_Q$. All cuts of $\mathcal{S}_P \cup \mathcal{S}_Q$ are obviously preserved, as are their *xor* conditions. Therefore, since the cardinality of the ramifications is the same, as the elements to which they correspond via the assignment, we can assume that to each element is assigned the same address in both $\mathcal{D}_P \bigodot \mathcal{D}_Q$ and $\mathcal{D}_{P \mid Q}$, and therefore that their ramifications are the same.

□

The most direct way to prove the equivalence of $[\![P]\!] \mid [\![Q]\!]$ and $[\![P \mid Q]\!]$, when there is communication between P and Q, is via the sets of generators. We already proved the base designs equivalence: what we are missing from the set of generators are the restriction designs, which are derived from the set of freedom requisites.

We know that $<_{P \mid Q} = <_P \cup <_Q$, therefore $\mathscr{F}(i)$ and $\mathscr{F}((u, v))$ are unchanged for synchronisations i in $\mathcal{S}_P \cup \mathcal{S}_Q$, and (u, v) in $\mathcal{X}_P \cup \mathcal{X}_Q$. Instead, since $\mathcal{S}_P \cup \mathcal{S}_Q \subseteq \mathcal{S}_{P \mid Q}$, $\mathscr{F}(l)$ for locations of P or Q might be extended with any $i \in new\mathcal{S}_{P \mid Q}$ such that $l \in i$. We can then build $\mathcal{R}(P)$ and $\mathcal{R}(Q)$ in the same way as before - putting ✣

and prunings accordingly to freedom requisites and xor conditions - on $\mathcal{D}_P \odot \mathcal{D}_Q$; we note the resulting sets of restriction designs as $\mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q}$ and $\mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q}$.

The restrictions for $new\mathcal{X}_{P \mid Q}$ are immediate from the previous definitions (we have all the information we need, having the new synchronisations); instead $\mathcal{R}(j)$ for $j \in new\mathcal{S}_{P \mid Q}$ require us to check the order on locations. If $j = (a^l, \overline{a}^m)$, we must recover the immediate smaller neighbors of l and m, i.e. the locations h, k such that $h < l$ and $k < m$. Since we coded the local partial order on locations in the restriction designs, it suffice us to check $\mathcal{R}(l)$ and $\mathcal{R}(m)$: if there is a design with a $\maltese$ as last rule of $G[h]$, for some location h, and a pruning over $[l] \vdash$ (and equivalently for m, for some location k), then we have that $h, k \in \mathscr{F}(j)$ - remember that the freedom requisite of a synchronisation are *at most* two locations; therefore we can build $\mathcal{R}(j)$ based on $\mathscr{F}(j)$ (we skip the trivial proof of the fact that $\mathscr{F}(j)$ is the same set if build on $(P \mid Q, <_{P \mid Q})$).

We note the set of these restriction designs with $\mathcal{R}(new\mathcal{X}_{P \mid Q})$ and $\mathcal{R}(new\mathcal{S}_{P \mid Q})$. $\mathscr{B}_P \mid \mathscr{B}_Q$ becomes thus:

$$(\{\mathcal{D}_P \bigodot \mathcal{D}_Q\} \cup \mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(new\mathcal{X}_{P \mid Q}) \cup \mathcal{R}(new\mathcal{S}_{P \mid Q}))^{\perp\perp}$$

together with the assignment $[\ \ ]_{P \odot Q} \cup [\ \ ]_{\mathcal{N}} = [\ \ ]_{P \bigodot Q}$.

Let us note the set of generators with $\mathbb{B}_P \Cap \mathbb{B}_Q$, then

Definition 7.6 (Merging of interpretations) *Let P, Q be MCCS processes. The* **merging of interpretations** *of P and Q, in symbols $[\![P]\!] \Cap [\![Q]\!]$, is the pair*

$$((\mathbb{B}_P \Cap \mathbb{B}_Q)^{\perp\perp}, [\ \ ]_{P \bigodot Q}).$$

We want to prove the following theorem:

Theorem 7.7 *Let P, Q be MCCS processes. Then $[\![P]\!] \Cap [\![Q]\!] = [\![P \mid Q]\!]$.*

Proof. It suffice to prove that $\mathbb{B}_P \Cap \mathbb{B}_Q \equiv \mathbb{B}_{P \mid Q}$; this holds via an identification of assignments $[\ \ ]_{P \bigodot Q} = [\ \ ]_{P \mid Q}$, which in turn identify the ramifications of the designs in the generators. We use the symbol for equivalence, since it holds modulo a (possible) renaming of addresses in the assignments, and a substitution of the ramification on the base designs to avoid conflicts (to assure that the ramifications are disjoint).

The theorem is almost straightforward by construction of $\mathbb{B}_P \Cap \mathbb{B}_Q$ and $\mathbb{B}_{P \mid Q}$. What we need to check is that we can indeed identify the assignments by a renaming, which, as said, implies a renaming of the ramification of the corresponding base design. This holds since, by the previous lemma, we may assume $\mathcal{D}_P \bigodot \mathcal{D}_Q = \mathcal{D}_{P \mid Q}$.

Indeed, we have that the domain of $[\ \ ]_{P \mid Q}$ is:

- $Loc_{P \mid Q} = Loc_P \cup Loc_Q$.
- $\mathcal{S}_{P \mid Q} = \mathcal{S}_P \cup \mathcal{S}_Q \cup new\mathcal{S}_{P \mid Q}$.
- $\mathcal{X}_{P \mid Q} = \mathcal{X}_P \cup \mathcal{X}_Q \cup new\mathcal{X}_{P \mid Q}$.

Which, by construction, is the domain of $[\ ]_{P \odot Q}$. The co-domain of both is the ramification of the *first rule* of $\mathcal{D}_P \odot \mathcal{D}_Q = \mathcal{D}_{P \mid Q}$. Thus we can assume $[\ ]_{P \mid Q} = [\ ]_{P \odot Q}$.

As we noted in the discussion above, the freedom conditions for elements of $\mathcal{S}_P \cup \mathcal{S}_Q$ and $\mathcal{X}_P \cup \mathcal{X}_Q$ are the same, since are defined on the same domain, on which the same order and *xor* relations holds; while the freedom conditions for the new elements arising in $P \mid Q$ can be deduced by $<_P \cup <_Q$, $Loc_P \cup Loc_Q$, and $new\mathcal{S}_{P \mid Q}$ (for the *xor* conditions), so are the same in both cases as well. The freedom requisites induce the same conditions on the non commutative restrictions - built on the same base design, therefore we can conclude:

$$\mathbb{B}_P \mathbin{⦶} \mathbb{B}_Q \equiv \mathbb{B}_{P \mid Q}$$

□

7.3 Subject reduction

We want to define a *projection* operation on the interpretation that matches execution on the underlying process. For the relevant elements in an execution, we can note that if $P \rightarrow_u P'$, then $\mathcal{S}_{P'} = \mathcal{S}_P \setminus (\{u\} \cup xor(u))$, and $Loc_{P'} = Loc_P \setminus \{l, m\}$, with $u = (a^l, \overline{a}^m)$ and $xor(u) = \{x \mid (u, x) \in \mathcal{X}_P\}$. This means the whole branches $G[u], G[l], G[m]$ and $G[x]$, for all such x, are not in $\mathcal{D}_{P'}$, as well as $w[u, x]$ and $w[x, y]$, the xor conditions for u and for the $x \in xor(u)$. Still, except for the channels in the synchronisation u, the rest of the process is preserved, and thus its possible executions. What we want to do is *deep prune* (or *trim* to avoid misinterpretation) $\mathcal{D}_P$ to obtain a strictly smaller base designs, along with *less* non commutative restrictions.

We can define the operation on $\mathcal{D}_P$, which will naturally apply to all $\mathcal{R}(P)$.

Definition 7.8 (Trimming) *Let P be a MCCS process,$(a^l, \overline{a}^m) = u \in \mathcal{S}_P$ and $(+, \xi, I)$ the first action of $\mathcal{D}_P$. The* trimming *of u on $\mathcal{D}_P$, noted $(\mathcal{D}_P)_u$, is a rewriting of $\mathcal{D}_P$ consisting of the removing of the branches* associated to a synchronisation *u, which are:*

- *$G[u]$;*
- *$G[l]$ and $G[m]$;*
- *$[xor^u \ \& \ xor^{x_1}], \ldots, [xor^u \ \& \ xor^{x_j}]$ and $G[x_1], \ldots, G[x_j]$ for all clause $(u, x_1), \ldots, (u, x_j) \in \mathcal{X}_P$;*
- *$[xor^x \ \& \ xor^y]$ for each $x \in xor(u)$ and $y \in xor(x)$;*

and thus a rewriting of $(+, \xi, I)$ accordingly to this removal, i.e. a removing of a sub-ramification $\{i_1, \ldots, i_n\} \subset I$, becoming $(+, \xi, J)$ with $J = I \setminus \{i_1, \ldots, i_n\}$; where $[u] = \xi i_1$, $[l] = \xi i_2$, $[m] = \xi i_3$, $G[x_1] = \xi i_4$, ..., $[xor^u \ \& \ xor^{x_j}] = \xi i_m$, ..., $[xor^x \ \& \ xor^y] = \xi i_n$.

Being built on $\mathcal{D}_P$, this induce a projection *on the behaviour $\mathscr{B}_P$.*

This operation on the base design induce the same restriction on the ramification on all $\mathcal{R}(P)$, and on the assignment $[\ ]_P$, since its co-domain (the *directory* of the type, which is the ramification of the first action of the base design) get restricted.

For the latter it is a matter of a simple inclusion, since $[\ ]_{P'} \subset [\ ]_P$: it lacks the elements $[u], [l], [m]$, the $[x]$, $[xor^u \,\&\, xor^x]$ and $[xor^x \,\&\, xor^y]$, for each $x \in xor(u)$ and $y \in xor(x)$ which are removed from the domain of $[\ ]_P$ (since all $x \in xor(u)$ disappear once u is done, even all their xor conditions must as well).

In the following, let x, y be variables on the set $\mathcal{S}_P$ (i.e. any synchronisation satisfying the conditions on x or y).

Definition 7.9 *Let* $(a^l, \overline{a}^m) = u \in \mathcal{S}_P$. *Then* $([\ ]_P)_u$ *is* $[\ ]_P \restriction_{(\mathcal{D}_P)_u}$, *i.e. is* $[\ ]_P$ *on the restricted domain* $(Loc_P \setminus \{l, m\}) \cup (\mathcal{S}_P \setminus (\{u\} \cup \{x \mid \in xor(u)\}) \cup (\mathcal{X}_P \setminus \{(u, x) \cup (x, y)\})$ *for each* $x \in xor(u)$ *and* $y \in xor(x)$.

It is clear that by this definition $([\ ]_P)_u$ matches the ramifications of $(\mathcal{D}_P)_u$, and has the same domain of $[\ ]_{P'}$, with $P \to_u P'$ - for the consideration on the sets in question done above. Thus, via a renaming of the assignments as for the *merging* of types, we can conclude

Lemma 7.10 *If* $P \to_u P'$, *then, let* $([\ ]_P)_u$ *be the resulting assignment on the trimmed base design* $(\mathcal{D}_P)_u$; *we have that* $([\ ]_P)_u = [\ ]_{P'}$.

Proof. We already noted that the domains of the two assignments are the same. The co-domain, which is the ramification of the first action of $\mathcal{D}_{P'}$ for $[\ ]_{P'}$ and the one of $(\mathcal{D}_P)_u$ for $([\ ]_P)_u$ have the same cardinality, being built on the same domain of the respective assignments. Thus we can unify the two ramifications, say I for $\mathcal{D}_{P'}$ and J for $(\mathcal{D}_P)_u$, by a renaming of the latter (we could as well rename I, but in this way we confine the operations on the trimmed base design), and obtain the equality

$$([\ ]_P)_u = [\ ]_{P'}.$$

□

Since the ramifications of $\mathcal{D}_{P'}$ and $(\mathcal{D}_P)_u$ are the same, as well as their structure (all the $G[l]$, $G[u]$ and $w[x, v]$ are the same since are built on the same domain), then we have as a corollary

Corollary 7.11 *If* $P \to_u P'$, *then* $\mathcal{D}_{P'} = (\mathcal{D}_P)_u$.

About the non commutative restrictions, it holds that the *trimming* (of a synchronisation u) on the base design will induce the *disappearance from the type* of the ones regarding the branches *associated* to u. This holds because the restrictions in question have not anymore their meaningful branches (the ones ending with ✣ or pruned), therefore are *equal* to the base design, and they will simply disappear in the union with $\{\mathcal{D}_P\}$ - since they are the very same element. We can identify these relevant restrictions in the following way:

Definition 7.12 *Let* $(a^l, \overline{a}^m) = u \in \mathcal{S}_P$. $\mathcal{R}([u])$ *is the set of restrictions designs of* $[\![P]\!]$ *such that* $G[u], G[l]$, $G[m]$, $[xor^u \,\&\, xor^x]$, *or* $[xor^x \,\&\, xor^y]$, *for all* $x \in xor(u)$, $y \in xor(x)$, *end with* ✣ *(for the xor, have* a branch *ending with* ✣*), and where* $G[u]$ *or* $G[x]$ *are* pruned.

The first condition select all restrictions where u or its locations are a freedom requisite, and the restrictions on all xor conditions regarding u and $x \in xor(u)$ - note that this include $\mathcal{R}(l)$ and $\mathcal{R}(m)$ since $G[u]$ is a requisite for its locations,

ending with ✣. About the second conditions, what happens is that all such cuts x disappear along u, because one of their locations, the one in common with u, is erased. However the other location is still in the process, thus the restriction where $G[x]$ ends with ✣ (the restriction on the other location), is still relevant to interaction.

Indeed the restriction for the other location of such x will just be missing the branch $G[x]$, but have ✣ on $G[j]$ for any other synchronisation j to which it belongs, or just end with a pruning if no such j exists, noting an absence of interaction. Therefore only those where $G[x]$ is pruned are going to disappear (its *xor* conditions and the restriction based on its freedom requisites - at most a single location if u was executable). We do not require u to be a minimal synchronisation since we want the definition to be as general as possible, letting us perform the trimming even for non principal cuts. Still, in most cases the restriction where $G[u]$ is pruned are not relevant, since our aim is to represent reduction on u on the interpretation, and thus we are assuming to be in the case where u is available for execution in the process, i.e. its channels are not blocked by any prefix, and therefore there are no restrictions where $G[u]$ is pruned.

So, the rewriting of the sets of restrictions $\mathcal{R}(P)$ on the trimmed base design, noted $\mathcal{R}(P)\upharpoonright_{(\mathcal{D}_P)_u}$ does not change the positions of ✣ and *pruning*, except in the case of the erased branches, where obviously they do not appear at all. We can note the behaviour on the trimmed base design with:

$$(\mathscr{B}_P)_u = (\{(\mathcal{D}_P)_u\} \cup \mathcal{R}(P)\upharpoonright_{(\mathcal{D}_P)_u})^{\perp\perp}$$

As a last step, we need the following lemma

Lemma 7.13 *All designs of* $\mathcal{R}([u])$ *disappear in the union* $\{(\mathcal{D}_P)_u\} \cup \mathcal{R}(P)\upharpoonright_{(\mathcal{D}_P)_u}$.

Proof.

By rewriting $\mathcal{R}(P)$ on $(\mathcal{D}_P)_u$ all the branches *associated* to u are erased. These are exactly the branches ending with ✣ or a pruning of designs of $\mathcal{R}([u])$ (by definition of $\mathcal{R}([u])$), which therefore are exactly equal to $(\mathcal{D}_P)_u$.

□

We have now all the tools to define *reduction* on $[\![P]\!]$.

Definition 7.14 (Type reduction) *Let P be a MCCS process of type $[\![P]\!]$, and $u \in \mathcal{S}_P$. The* reduction *of u on $[\![P]\!]$ is $([\![P]\!])_u = ((\mathscr{B}_P)_u, ([\ ]_P)_u)$. We note the operation with $[\![P]\!] \rightsquigarrow_u ([\![P]\!])_u$.*

The reason we call this operation *reduction* instead of rewriting, other than the fact that we are erasing parts of the base design, is clear if we look at the interaction paths. Indeed, once proved that $([\![P]\!])_u = [\![P']\!]$, for $P \to_u P'$, it is immediate to show that the reduced type contains a *subset* of the interaction paths of $[\![P]\!]$, by the correspondence execution-interaction (Theorem 4.3). On P' we can perform $\to_{v_i}$ with $v_i = v_j$ or v_0, such that:

(i) v_j does not have a common location with u, and is a principal synchronisation.

(ii) v_0 has as only requisite the synchronisation u (one or both its locations), i.e.

$P \to_{u,v_0}$ is an admissible execution.

Thus for all interaction on $[\![P']\!]$ we have that, modulo renaming of the assignment, there is an interaction on $[\![P]\!]$ such that either they have the same associated execution, or the interaction on $[\![P]\!]$ extend – with an initial segment – the one on $[\![P']\!]$ (in the case its lacking the interaction on u).

However we cannot talk about *equality* of the set of visited actions $\mathcal{K}_{\mathcal{C}}$, because the set of orthogonals $\mathscr{B}_P^{\perp}$ and $\mathscr{B}_{P'}^{\perp}$ *cannot* be equal, due to the mismatch of their *directories* (the ramifications of the first action of $\mathcal{D}_P$ and $\mathcal{D}_{P'}$). Indeed the first action of $\mathcal{D}_P$ have a smaller cardinality than the one of $\mathcal{D}_{P'}$, since P' lacks all the branches associated to u, and thus $[\ ]_{P'}$ have a strictly smaller domain than $[\ ]_P$; however we can still assume that the same addresses are assigned to the same elements, the ones the domains have in common. It remains to prove that $(\mathscr{B}_P)_u$ is actually the type of the process after execution on u.

Theorem 7.15 *Let P be a MCCS process interpreted by $[\![P]\!]$. Given a synchronisation $u \in \mathcal{S}_P$ such that $P \to_u P'$, we have $[\![P']\!] = ([\![P]\!])_u$, i.e.*

$$\begin{array}{ccc} P & \to_u & P' \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![P]\!] & \rightsquigarrow_u & ([\![P]\!])_u \end{array}$$

Proof.

By applying lemma 7.10 and 7.13, we have that $\mathcal{D}_{P'} = (\mathcal{D}_P)_u$, $[\ ]_{P'} = ([\ ]_P)_u$, and that $\mathcal{R}(P') = (\mathcal{R}(P))_u$, since the restriction are based on the same freedom requisites and base designs (the domain, order on locations, and xor conditions are the very same). Therefore we can conclude:

$$[\![P']\!] = ([\![P]\!])_u.$$

□

For the reasons explained above, about the inclusion of interaction paths, we can give the following corollary:

Corollary 7.16 *Let $P \to_u P'$. It holds that $\forall \mathcal{C}' \in \mathscr{B}_{P'}, \exists \mathcal{C} \in \mathscr{B}_P$ such that either:*

- *$\mathcal{K}_{\mathcal{C}'} \cong \mathcal{K}_{\mathcal{C}}$ where $\cong$ means that $(\mathcal{K}_{\mathcal{C}'} \setminus (+,\xi,I)) = (\mathcal{K}_{\mathcal{C}} \setminus (+,\xi,J))$ and $I \subset J$, with $(+,\xi,I)$ the first action of $\mathcal{D}_{P'}$ and $(+,\xi,J)$ the first action of $\mathcal{D}_P$.*

  This is the case where we perform interaction on $[v_i], [\overrightarrow{v}]$ with v_i as in case 1 above, independent from u; or
- *$\mathcal{K}_{\mathcal{C}'} \sqsubset \mathcal{K}_{\mathcal{C}}$ where $\sqsubset$ means that $(\mathcal{K}_{\mathcal{C}'} \setminus (+,\xi,I)) = (\mathcal{K}_{\mathcal{C}} \setminus ((+,\xi,J), u^{\perp\perp})$ and $I \subset J$; with $(+,\xi,I)$ the first action of $\mathcal{D}_{P'}$, $(+,\xi,J)$ the first action of $\mathcal{D}_P$ and $u^{\perp}$ the sequence of action describing a minimal execution on u, found in example **??**, i.e. the sequence of action which visits the xor conditions for u, $G[u]$, and $G[l]$, $G[m]$, with $u = (a^l, \overline{a}^m)$. This generalize the notion of final segment - it is not necessarily a final segment since all the interaction steps not corresponding to synchronisations or locations may be done in very different order.*

In this case we are performing interaction on $[v_0], [\overrightarrow{v}]$*, with* v_0 *as in case* 2 *above, dependent from* u*.*

Proof.

By applying theorem 4.3 on the correspondence execution-interaction, and for the reasons outlined above, we can conclude that the interaction paths - as sequence of visited actions - on $\mathscr{B}_{P'}$ are in one of the relations $\cong$ or $\sqsubset$ with the ones of $\mathscr{B}_P$, and thus have either the same associated executions or the ones of $\mathscr{B}_{P'}$ are final segments (in our broad sense) of those of $\mathscr{B}_P$. □

The consequence on execution sequences reducing the process to 1 is:

Corollary 7.17 *Let* $\mathbb{O}ne = \dfrac{}{\vdash \xi}\ (+, \xi, \varnothing)$ *(the design generating the behaviour that corresponds to the linear logic multiplicative unit* 1*), and let* P *be a MCCS process;*

$$\textit{if } P \to^* 1 \textit{ then } [\![P]\!] \rightsquigarrow^* \{\mathbb{O}ne\}^{\perp\perp} \textit{ for the same synchronisation sequence.}$$

Proof. We just need to apply the reduction step by step, and check what is left in $([\![P]\!])_*$. Following the reduction operation, if $P \to^* 1$, then *the whole ramification* is going to be erased from designs of $[\![P]\!]$: every location, thus every synchronisation and *xor* condition. What is left is just the *base* of $\mathcal{D}_P$ with an *empty* ramification:

$$\frac{}{\vdash \xi}\ (+, \xi, \varnothing)$$

which is the design called $\mathbb{O}ne$, whose generated behaviour correspond to the linear logic multiplicative unit 1. This means that the reduction operation on the type has the intended intuitive meaning, and further justify the choice to name the empty process 1, the multiplicative unit, instead of 0.

Moreover, $\mathbb{O}ne$ is the neutral element of $\odot$, the merging of designs.

□

Formalizing Constructive Projective Geometry in Agda

Guillermo Calderón[1]

University of the Republic
Montevideo, Uruguay

Abstract

We present a formalization of Projective Geometry in the proof assistant and programming language Agda. We formalize a recent development on constructive Projective Geometry which has been showed appropriate to cover most traditional topics in the area applying only constructively valid methods. The equivalence with other well-known constructive axiomatic systems for projective geometry is proved and formalized. The implementation covers a basic fragment of intuitionistic synthetic Projective Plane Geometry including the axioms, principle of duality, Fano and Desargues properties and harmonic conjugates.
We focuse in an illustrative example of implementation of a complex and large proof which appears partially and vaguely described in the literature; namely the *uniqueness of the harmonic conjugate*.
The most important details of our implementation are described and we show how to take advantage of several interesting properties of Agda such as *modules*, *dependent record types*, *implicit arguments and instances* which result very helpful to reduce the typical verbosity of formal proofs.

Keywords: proof assistants, formalizations of mathematics, projective geometry, type theory, Agda

1 Introduction

Projective Geometry is a well-established branch of mathematics which studies the incidence properties of points, lines and planes. Typical textbooks relative to this area (e.g. [4,23]) cover topics such as: axiomatic definition of points and lines, incidence relation, principle of duality, Desargues and Fano theorem, harmonic conjugates, projectivities, polarities, conics, etc..

Projective geometry constitutes a very elegant, self-contained mathematical system. It is constructed from two primitive concepts: points and lines together with a relation of incidence among them. In addition, a few axioms determine the behaviour of the entire system. For this reason, projective geometry becomes a very attractive discipline to be formalized in a computer system. and an interesting case study in order to investigate problems involved with computer formalization of mathematics.

Despite its simplicity, projective geometry is considered as a unifying framework for all other geometries. Every result in projective geometry can be applied to affine

[1] Email: calderon@fing.edu.uy

geometry which in turn reduces to Euclidean geometry. This feature is implicit in the so called Erlangen program of F. Klein [9] where projective geometry is defined as the study of the properties invariant under projectivities.

On the other hand, projective geometry has a number of important applications in different areas of computer science, such as: computer vision, cryptography, computer graphics [3,6,2]. A computer formalization of projective geometry will contribute to the construction of *certified computer algorithms* to solve known problems in these areas.

This paper describes a computer formalization of *constructive projective plane geometry*. In a first instance, we implement the system of axioms presented in Mandekern's paper [15] which is based on previous constructive developments of projective geometry such as [7] and [22]. The constructive method implies among other things that: the principle of excluded middle is not used at all; existential proofs are always achieved by effective construction of a *witness*; it is not assumed anything about decidability of basic relations.

Our formalization [2] is written in *Agda* [19], a proof assistant and functional programming language based on *Intuitionistic Type Theory* [17]. In particular, we are concerned with the construction of a programming tool which helps us to write formal proofs in projective geometry in a comfortable way but without using automated tactics.

Outline. The paper is organized as follows. In Section 2, we present the formalization of apartness relation and setoids. Section 3 describes the representation of the projective plane and its axioms in Agda. In Section 4, a method is presented to prove equalities with expressions involving the *meet* and *join* functions of a projective plane. In Section 5, we summarize some proofs developed in our formalization, in particular we comment about the implementation of the principle of duality. In Section 6, an overview about the implementation of Fano and Desargues properties is presented. In Section 7, we describe the formalization of harmonic conjugates and the proofs of existence and uniqueness. Finally, conclusions and related work are discussed in Section 8.

2 Apartness and Constructive Setoids

In the constructive formulation of projective geometry we are following, it is necessary to define the equality relation as derived from a primitive *apartness* relation (i.e inequality). Detailed explanation of this methodology can be found in [24] and [15]. In this section we describe our implementation of *apartness* and *setoids*.

An *apartness relation* on a set A is a binary relation $\sharp$ on A satisfying three properties: irreflexivity, symmetry [3] and cotransitivity. It is implemented in Agda as a *dependent record* which takes A and $\sharp$ as parameters and whose fields postulate the three required properties:

[2] All the code described in this paper is freely available from the git repository: https://github.com/GuillermoCalderon/ProjectiveGeometryInAgda

[3] In fact, *symmetry* it is not required because we can derive it from the other two properties. However, symmetry is usually included in the definition of *apartness*

```
record IsApartness {a b}{A : Set a}
  (_♯_ : A → A → Set b) : Set (a ⊔ b) where
  field
    irreflexive  : ∀ {x}     → ¬(x ♯ x)
    symmetry     : ∀ {x y}   → x ♯ y → y ♯ x
    cotransitivity : ∀ {x y} z → x ♯ y → z ♯ x ⊎ z ♯ y
```

In order to obtain a definition as general as possible, we consider that A belongs to the universe Set_a for a generic level a. Analogous assumptions are adopted for all objects in our formalization. We will not give details about levels in the rest of this paper. The reader unfamiliar with *universe polymorphism* [20] in Agda can just ignore them without loss of understanding. Note that some arguments are declared as implicit (the ones enclosed between curly brackets). Implicit arguments have a nice property: in most cases we can omit these arguments and Agda will try to infer them from the context. We use implicit arguments very often in our implementation. We will not explain why some arguments are declared as implicit and others are not. In general, it is decided by some sort of heuristics given by practice.

Equality. We define the equality relation $\approx$ as the negation of apartness: $x \approx y \equiv \neg(x \sharp y)$. This approach is slightly different from the one adopted in most axiomatic definitions of projective geometry ([7,15,24]) where the equality is a primitive concept and the relationship between equality and apartness is given by an additional axiom (*tightness*): *if* $\neg(x \sharp y)$ *then* $x = y$. The converse of tightness follows from irreflexivity. Thus, a logical equivalence is established among equality and the negation of apartness. By simplicity, we obtain this equivalence defining the equality relation as an alias of *not-apartness* (without affecting the system as a whole) As expected, $\approx$ is proved to be an *equivalence* relation.

Setoids. A setoid is a pair $\langle A, \sharp \rangle$ where A is a set and $\sharp$ an apartness relation on A [4]:

```
record Setoid♯ a b : Set (suc (a ⊔ b)) where
  field
    {Carrier}   : Set a
    _♯_         : Carrier → Carrier → Set b
    isApartness : IsApartness _♯_
  open IsApartness isApartness public
```

Relations on Setoid. In order to work with setoids, we need to lift some set based operations on setoids. This process consists in the definition of a record which contains a field representing the operation at the level of the carriers together with another field that asserts the compatibility of the operation with the apartness/equality relation. In the following code, $\langle \_ \rangle$ denotes an operator which returns the carrier of a setoid.

[4] We use the name `Setoid♯` standing for an apartness based setoid. The identifier `Setoid` is preserved to denote a classical setoid such as it is defined in Agda standard library.

```
record IsRel♯ {a₁ b₁ a₂ b₂ c}
   (S₁ : Setoid♯ a₁ b₁)(S₂ : Setoid♯ a₂ b₂)
   (R : ⟨ S₁ ⟩ → ⟨ S₂ ⟩ → Set c) :
   Set (a₁ ⊔ b₁ ⊔ a₂ ⊔ b₂ ⊔ c) where
field
   sound : ∀ {a b c d}
           → Setoid♯._≈_ S₁ a  b
           → Setoid♯._≈_ S₂ c  d
           → R a c → R b d

record Rel♯  {a₁ b₁ a₂ b₂} c
   (S₁ : Setoid♯ a₁ b₁)(S₂ : Setoid♯ a₂ b₂)
   : Set (suc (a₁ ⊔ b₁ ⊔ a₂ ⊔ b₂ ⊔ c)) where
field
   R     :  ⟨ S₁ ⟩ → ⟨ S₂ ⟩ → Set c
   isRel :  IsRel♯ S₁ S₂ R
open IsRel♯ isRel public
```

Equality and Rewriting. The field `sound` of binary relations on setoids, give places to a method to construct proofs by substitution of equals for equals preserving the relation (also known as *rewriting*). If $\_ \bowtie \_$ is a binary relation between setoids and we have a proof of $a_1 \bowtie b_1$ and proofs of $a_1 \approx a_2$ and $b_1 \approx b_2$ then we obtain a proof of $a_2 \bowtie b_2$.

We introduce a couple of operators which allow us to mimic a very common pattern used in informal mathematical notation. For instance, the expression $P = Q \in A = B$ stands for a proof that $P \in B$, provided we have proofs of $P = Q$, $A = B$ and $Q \in A$.

We implement two operators for rewriting which apply substitution over the left ($\langle\_\rangle\Leftarrow\_$) and right ($\_\Rightarrow\langle\_\rangle$) arguments of the relation as follows

```
⟨_⟩⇐_                :  ∀ {a₁ a₂ b} → a₂ ≈ a₁ → R a₁ b → R a₂ b
⟨ a₂≈a₁ ⟩⇐ Ra₁b     =  sound (sym a₂≈a₁) refl Ra₁b
_⇒⟨_⟩                :  ∀ {a₁ a₂ b} → R b a₁ → a₁ ≈ a₂ → R b a₂
Rba₁ ⇒⟨ a₁≈a₂ ⟩      =  sound refl a₁≈a₂ Rba₁
```

We need to work with several setoids at the same time. In order to overload the appartnes operators for different setoids, we use an Agda feature called *instance arguments* [5]. The idea is rather simple: we can open the module `Setoid♯` with a special directive: `open Setoid♯ {{...}}`. This directive allows us to access all the operations of the module `Setoid♯` and we can omit the particular instance argument which will be inferred by Agda from the context by a special *instance resolution algorithm* [5].

[5] Instance arguments of Agda can be seen as an equivalent of Haskell type class.

3 Representing the Projective Plane

In this section, we describe the implementation in Agda of the system of axioms of projective geometry according to [15].

Point, Lines and Basic Relations. The basic objects of projective plane geometry are *points* and *lines* which are setoids; i.e. they are equipped with an apartness relation. In addition, we have the incidence relation ($\in$) and the outside relation ($\notin$). Both relations are assumed to be compatible with apartness.

We start defining a record `IsProjectivePlane` which constitutes the main structure of our formalization of projective geometry.

```
record IsProjectivePlane {a b c d e}
  { Point♯ : Setoid♯ a b }
  { Line♯ : Setoid♯ c d  }
  (Incidence  Outside : Rel♯ e Point♯ Line♯)
  : Set (a ⊔ b ⊔ c ⊔ d ⊔ e) where
```

The names *Incidence* and *Outside* will be soon forgotten since we will use the raw relations $\in$ and $\notin$ at the level of the carriers. The first step is to give some convenient definitions that allows we easily access the components of the structure.

```
Point = ⟨ Point♯ ⟩
Line  = ⟨ Line♯ ⟩
_∈_ : Point → Line → Set e
_∈_ = Rel♯.R Incidence
_∉_ : Point → Line → Set e
_∉_ = Rel♯.R Outside
```

Outside Relation. In [15] and [7] the *outside* relation is defined positively (i.e. without negation) from the incidence relation and point apartness: $P \notin l \equiv (\forall Q)(Q \in l) \to P \sharp Q$. We take a different approach in our formalization: we do not give an explicit definition of the relation outside. Instead, we add an axiom (C0) asserting that $P \notin l \Rightarrow (\forall Q)(Q \in l) \to P \sharp Q$. The converse of this implication is not postulated but it can be proved using the others axioms [6].

3.1 Axioms for Projective Plane

In this section we begin the description of the axioms of a projective plane which are represented as fields of the record `IsProjectivePlane`.

Axiom C0 was explained in the previous section:

```
C0 : ∀ {P l} → P ∉ l → (∀ {Q} → Q ∈ l → P ♯ Q)
```

A first group of axioms establish the existence of a line and an external point (axiom C1 of Mandelkern). These axioms exclude the possibility of trivial cases of projective plane

[6] However, the converse direction of the equivalence it is never used in our implementation.

$$
\begin{array}{lcl}
\mathsf{P}_0 & : & \mathsf{Point} \\
\mathsf{l}_0 & : & \mathsf{Line} \\
\mathsf{P}_0{\notin}\mathsf{l}_0 & : & \mathsf{P}_0 \notin \mathsf{l}_0
\end{array}
$$

The next group of axioms (C2 and C3) express conditions of existence and uniqueness for the line passing by two distinct points and the dual (a point on two distinct lines).

Here, the relation of apartness becomes essential to assure the existence of a line passing by two distinct points.

$$
\begin{array}{lcl}
\mathsf{join} & : & \forall\,\{P\ Q : \mathsf{Point}\} \to\ P \sharp Q \to \mathsf{Line} \\
\mathsf{join}_1 & : & \forall\,\{P\ Q\}\ \{P\sharp Q : P \sharp Q\} \to P \in \mathsf{join}\ P\sharp Q \\
\mathsf{join}_r & : & \forall\,\{P\ Q\}\ \{P\sharp Q : P \sharp Q\} \to Q \in \mathsf{join}\ P\sharp Q \\
\mathsf{unq\text{-}join} & : & \forall\,\{P\ Q\}\ (P\sharp Q : P \sharp Q) \\
 & \to & \forall\,\{l\} \to P \in l \to Q \in l \to l \approx \mathsf{join}\ P\sharp Q
\end{array}
$$

A function join is given, which receives two points and a proof of the apartness of them, and returns a line (existence). The functions join_1 and join_r allow to prove that the join of two points effectively pass through them. Finally, the function unq-join asserts that all lines passing thorough two points are equal to the join of these points.

Note the difference with the traditional notation PQ to represent the line passing by the points P and Q. In our implementation this line is denoted by the expression $(\mathsf{join}\ P\sharp Q)$, where $P\sharp Q$ is a proof that P and Q are distinct.

We define analogously the functions meet, meet_1, meet_r and unq-meet which represent the dual part of the family of *join* operations.

A few axioms are formulated in order to rule out too simple models of projective plane (C4). Then, it is assumed the existence of three distinct points on any line:

$$
\begin{array}{lcl}
\mathsf{point}_1\ \mathsf{point}_2\ \mathsf{point}_3 & : & \mathsf{Line} \to \mathsf{Point} \\
\in\text{-}\mathsf{point}_1 & : & \forall\ l \to \mathsf{point}_1\ l \in l \\
\in\text{-}\mathsf{point}_2 & : & \forall\ l \to \mathsf{point}_2\ l \in l \\
\in\text{-}\mathsf{point}_3 & : & \forall\ l \to \mathsf{point}_3\ l \in l \\
\mathsf{point\text{-}i\sharp j} & : & \forall\ l\ \to\ \mathsf{point}_1\ l \sharp \mathsf{point}_2\ l \\
 & & \qquad \times\ \ \mathsf{point}_1\ l \sharp \mathsf{point}_3\ l \\
 & & \qquad \times\ \ \mathsf{point}_2\ l \sharp \mathsf{point}_3\ l
\end{array}
$$

A lot of propositions can be inferred from the axioms above if we were working using classical reasoning. Several of these conclusions are not constructively valid. Therefore, it is necessary to add some other axioms in order to obtain a useful system and at the same time, preserving its constructive foundation. We keep Mandelkern's denomination for these axioms.

Axiom C5 permits to infer two lines are distinct if a point belongs to one of the lines and is outside to the other. In turn, C6 establish a strong relation between $\notin$ and $\in$. The last axiom C7 provides another view about the uniqueness of the intersection of two lines, but involving $\notin$ instead of $\in$. It has the particularity that its conclusion is a disjunction

$$\mathsf{C5} \;:\; \forall\,\{l\ m\ P\} \to P \in l \to P \notin m \to l \sharp m$$
$$\mathsf{C6} \;:\; \forall\,\{P\ l\} \to \neg\,(P \notin l) \to P \in l$$
$$\mathsf{C7} \;:\; \forall\,\{l\ m\ P\} \to (l\sharp m : l \sharp m) \to P \sharp \mathsf{meet}\ l\sharp m \to P \notin l \uplus P \notin m$$

The converse one of C6 is a valid proposition and it is easily proved. However, the contrapositive of C6 (i.e : $\neg(P \in l) \to P \notin l$) is not constructively valid.

3.2 Comparing with other Axiom Systems

Since the seminal work of Heyting [7], several different systems were given to define projective geometry in an acceptable constructive way. In particular, we investigate the relationship between our formalization (mostly based on [15] which in turn is similar to Heyting's formulation) with the ones given by von Plato [24] and van Dalen [22].

Von Plato system defines a core for all the geometries which is called *apartness geometry*. In this approach all the usual negative relations are considered as primitive and positive. Then, the relations of apartness (for points and lines) and the outside relations are considered as given and the relations of equality and incidence are defined as the respective negations of these primitive relations. The collection of axioms are very simple and symmetric. Several of them, have disjunctive conclusions.

We give a formal proof that every projective plane as we define in Section 3 constitutes an *apartness projective geometry* in the sense of von Plato. The converse requires to add to von Plato system some axioms postulating non-degeneracy conditions (like axioms C1 and C4 of Mandelkern). With this addition, we formalize the correspondence of a projective geometry à la von Plato with a Mandelkern's projective plane.

On the other hand, van Dalen describes an axiom system for projective geometry where the only primitive relation is outside ($\notin$). All the other binary relations of the system are defined in function of the referred outside relation. We implement the proof of equivalence of the van Dalen's system with ours.

We consider these equivalences very important and useful, because allow us to work freely with different collections of axioms. Our implementation of each system of axioms is represented as a dependent record in Agda. Thus, we can make available any of the systems by importing the respective module.

4 Equational Reasoning with *join* and *meet*

The functions join and meet permits the construction of complex expressions representing points. Often, we find different expressions (compositions of join and meet) denoting the same object (point or line). The cause of this fact relies on the uniqueness properties expressed by the functions unq-join and unq-meet. Below, we present some algebraic rules expressing equalities between expressions constructed with join and meet. We use a simplified notation: AB_q denotes an expression of the form (join A B q) where q is a proof of $A \sharp B$. The sub index q is omitted when it is irrelevant.

We have dual rules for meet. We implement simple functions which allows to prove these algebraic equations in a rather direct manner. An interesting feature of this implementation is that all the arguments of these function are implicit. Thus,

$$\frac{}{AB_p \approx AB_q}\ {\approx}\text{join} \qquad \frac{}{AB \approx BA}\ \text{join-comm}$$

$$\frac{A_1 \approx A_2 \quad B_1 \approx B_2}{A_1B_1 \approx A_2B_2}\ \text{join-cong} \qquad \frac{A_1 \approx A_2 \quad B_1 \approx B_2}{A_1B_1 \approx B_2A_2}\ \text{join-flip-cong}$$

Fig. 1. Rules for equational reasoning

we can combine the functions in a very direct way, guided by the simple intuition of the rules.

These rules of equational reasoning together with the rewriting operators for setoids constitutes one of the most important tool to write proofs in our implementation.

5 Propositions and Duality

Once we have defined the previous modules with the definitions of projective planes and the libraries for setoids and equational reasoning, we can construct the proof for several propositions such as the ones appearing in Section 2 of [15] and most traditional books. Playing with the system, one discover soon that most proofs follow the pattern of establishing a relation among certain geometric objects (any of the basic relations, apartness, equality, incidence and outside). Thus, the axioms required are invoked, and usually we need to apply rewriting in order to obtain the relation involving the correct objects. Overloaded operators for rewriting and the combinators for equality described in Section 4 are extensively used in the construction of proofs in our system. In addition, these operators can work together with the operators of the library EqReasoning of Agda ([18], [1]) providing a nice syntax to write some proofs of equality.

We provide also a formal proof of the *principle of duality* considered one of the most typical results of projective geometry. The classic formulation of this principle express that: *For every valid proposition, it is also valid the dual proposition which is obtained from the first one swapping the words "line" by "point", "pass" by "lie" and so on.* This informal statement seems rather a *meta-theorem* than a geometric proposition. However, it can be formalized in a very succinct way like a function among projective planes.

```
duality : ∀  {a b c d e} {Point♯ : Setoid♯ a b} {Line♯ : Setoid♯ c d}
             {∈ ∉ : Rel♯ e Point♯ Line♯}
          → IsProjectivePlane ∈ ∉
          → IsProjectivePlane (flip♯ ∈) (flip♯ ∉)
```

The function flip♯ interchange the order of the arguments of a binary relation on setoids. The proof of the principle of duality indicates how to construct a dual projective plane from a given projective plane.

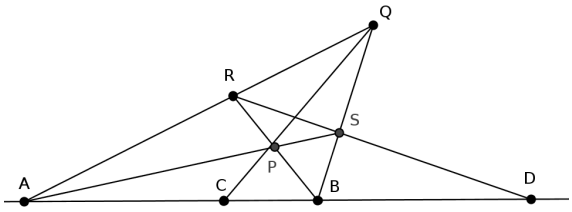


Fig. 2. A harmonic configuration for C with respect to A and B

6 Fano and Desargues

In planar projective geometry, the classic theorems of Fano and Desargues can not be proved. Therefore, we can assume them as new axioms. More precisely, we define extensions of the previously defined projective plane by adding fields representing these axioms.

Fano axiom is about *quadrangles*, and Desargues has to do with *triangles*. Quadrangles and triangles are instances of a kind of figure known as *complete n-point*. We define these concepts in our implementation providing modules with the necessary tools to work comfortably with them.

We define a structure FanoProjectivePlane as an extension of the structure ProjectivePlane by the addition of a new field that represents the Fano axiom. In turn, another structure Desarguesian is obtained as the extension of FanoProjectivePlane with a field corresponding to Desargues axiom. The principle of duality which was proved for projective plane, is generalized for the new defined structures by giving the proofs for the dual versions of Fano an Desargues axioms. Although conceptually simple, these proofs are rather extensive and verbose (about a thousand of lines of Agda code). A number of interesting mathematical machinery is implemented in these modules in order to work appropriately with the symmetry and circularity of the kind of figures involved. We do not provide a detailed explanation to this part of our implementation which can be consulted in the repository by the interested reader.

7 Harmonic Conjugates

Harmonic conjugates can be defined in Euclidean geometry in terms of metric concepts (the cross ratio). von Staudt [25] was the first one to propose a synthetic definition based in quadrangles. We follow the definition given in [15] which better reach the required constructivity since the definition is uniform over all the points of the base line.

7.1 Definition of Harmonic Conjugate

Assume two distinct points A and B. Let C be any point lying on the line AB. A *harmonic configuration* of C with respect to A and B is determined by a line l passing by C and different to AB, and a point R outside l and AB.

From the line l and the point R we can construct a point D which is called

the harmonic conjugate of C with respect to A, B. The method to obtain the conjugate of C proceeds as follows: Let Q be the intersection of l and AR. Let P be the intersection of l and BR. Let S be the intersection of AP and BQ Then, the *harmonic conjugate* of C with respect to A and B (relative to the configuration (l, R)) is the point $D = RS \cap AB$.

In order to have a valid definition, we must show that all the steps are well defined, i.e. all joins and meets are applied over different points and lines. In our formalization, this is forced by the Agda type checker since the functions join and meet require as argument a proof of apartness of the objects involved. For instance, to refer to the line AR we need a proof of $A \sharp R$ and to consider the intersection of l and AR we need a proof of $l \sharp AR$ and so on.

Harmonic Configuration. Given A and B distinct points and C a point belonging to AB, a harmonic configuration for C is the collection of lines and points constructed (following the method described above) from a particular choice of l and R in order to obtain the point D, (the harmonic conjugate of C with respect to A and B).

We represent a harmonic configuration in Agda, as a record:

```
record HarmonicConf
  {A B} {A♯B : A ♯ B} {C}
  (C∈AB : C ∈ join A♯B) : Set (a ⊔ b ⊔ c ⊔ d ⊔ e) where
field
  l       : Line
  C∈l     : C ∈ l
  l♯AB    : l ♯ join A♯B
  R       : Point
  R∉l     : R ∉ l
  R∉AB    : R ∉ join A♯B
```

The code above shows the fields of the record `HarmonicConf`, namely the line l and the point R together with the conditions they have to fulfill. In the same module `HarmonicConf` we define the points and lines required to construct the harmonic conjugate as well as the necessary proofs of apartness for each *join* and *meet* invoked. In particular, the points of the quadrangle (see below) are defined this way:

```
P Q S D : Point
P = meet RB♯l
Q = meet RA♯l
S = meet AP♯BQ
D = meet RS♯AB
```

Representing the Harmonic Conjugate. According to the previous development, the harmonic conjugate is defined as a function which returns a point from a given configuration:

```
HConjugate      :  ∀ {A B C : Point}{A♯B : A ♯ B}{C∈AB : C ∈ join A♯B}
                   → HarmonicConf C∈AB → Point
HConjugate HC =  HarmonicConf.D HC
```

Note the only explicit argument is the structure (record) corresponding to the harmonic configuration. Inside this record we give the construction of the point D which is the field selected for the configuration. All the other arguments are implicit, and they will be inferred from the argument `HC`.

Quadrangles and Harmonic Conjugate. It is usual to define the harmonic conjugate by using a quadrangle. In such definition, a quadrangle $PQRS$ is constructed with two diagonal points matching A and B and one of the sides determining the other diagonal passing by C. The point D is determined by the intersection of the remaining side with AB. This construction, is only possible when C does not coincide with A nor B. For this reason, we follow the definition given in [15] where we do not need to determine whether C is equal to A or B in order to define the harmonic conjugate.

7.2 Existence of the Harmonic Conjugate

The existence of the conjugate harmonic is in general admitted without any proof in the standard literature. In accordance with our constructive approach we have to provide a method to construct at least one harmonic configuration from any given points A, B and C.

```
∃HConf : ∀  {A B C : Point}{A♯B : A ♯ B}
         →  (C∈AB : C ∈ join A♯B)
         →  HarmonicConf C∈AB
```

By axiom C4 dual, we can obtain two distinct lines passing by C. By cotransitivity, at least one of these lines is distinct from AB. Then, we have the required line l passing by C and distinct from AB. The existence of the point R, outside both AB and l, is harder to prove. It follows also by an appropriate composition of axiom C4 and cotransitivity.

7.3 Uniqueness of Harmonic Conjugate

From the definition above, the conjugate of a given point with respect to A and B, will depend on the line l and the point R chosen (i.e. the configuration considered). One of the main results in projective geometry is that the harmonic conjugate does not depend on the configuration chosen. In other words, if we consider two configuration HC and HC′ the conjugates D and D' obtained using respectively HC and HC' are equal.

```
uniqueness : (A B C : Point)
  → (A♯B    : A ♯ B)
  → (C∈AB   : C ∈ join A♯B)
  → (HC HC' : HarmonicConf C∈AB)
  → HConjugate HC ≈ HConjugate HC'
```

The construction of this proof is the main goal of this paper. We are going to explain the general scheme used in the definition of this formal proof and we illustrate some parts with more details.

7.3.1 Constructive Proofs by Cases.
It is very common, in classic mathematics, to construct a proof by cases which derive from certain variant of the principle of the *Law of Excluded Middle* (LEM). If we prove a proposition ϕ from the assumption ψ and we also prove ϕ from assumption $\neg\psi$, then we have a proof ϕ (under no assumptions). This method it is not valid in general if we are working constructively and do not accept LEM as a general valid principle. However, in the particular case where we want to prove a contradiction ($\bot$) we can proceeds as follows: From the assumption ϕ, to prove a contradiction, and from the assumption $\neg\phi$, to prove a contradiction. Then we have a proof of $\bot$ (without assumptions). The first contradiction gives a proof of $\neg\phi$ and cancels the assumption ϕ. The second contradiction takes this proof of $\neg\phi$ and gives a proof of the absurdity ($\bot$).

This method can be generalized for an arbitrary number k of premises which are canceled by contradiction in sequence. Let $\phi_1, \ldots, \phi_k$ be a collection of assumptions. We have to provide $(k+1)$ proofs of $\bot$ in sequence. In each step we cancel a premise by negation. Finally, we obtain a proof of $\bot$ independent of the assumptions $\phi_1, \ldots, \phi_k$.

In the next section, we explain how to apply this pattern of reasoning to the construction of a proof for uniqueness of the harmonic conjugate.

7.3.2 The Proof of Uniqueness
In this section, we explain how the proof of uniqueness was constructed and formalized in Agda. This is a rather extended proof which considered a lot of different cases in order to cover all the possibilities [7].

The proof takes as input the points A, B and C as defined above and two harmonic configurations HC and HC'. Let D and D' be the harmonic conjugates with respect to HC and HC' respectively. We want to construct a proof of $D \approx D'$ which is defined as the negation of $D \sharp D'$. In other words, a proof of $D \approx D'$ is obtained by assuming $D \sharp D'$ and deriving a contradiction from this assumption.

In addition to the global assumption of $D \sharp D'$ we incorporate several additional assumptions which allows us to prove a contradiction by cases as explained in the previous section. These assumptions has to do with how distinct the configurations HC and HC' are.

For instance, we will describe the proof for the case where the two configurarion are completely disjoint.

We need eleven assumptions in order to configure this case:

$C \sharp A$, $C \sharp B$, $l \sharp l'$, $R \sharp R'$, $R' \notin RB$, $R' \notin RA$, $SP \sharp S'P'$, $SQ \sharp S'Q'$, $O \notin RS$, $O \notin R'S'$, $S \sharp S'$.

Note that all the premises are expressed as apartness ($\_\sharp\_$) or outside ($\_ \notin \_$)

[7] The code complete of the proof constitutes a number of files (about 15) which includes about 5000 lines of code.

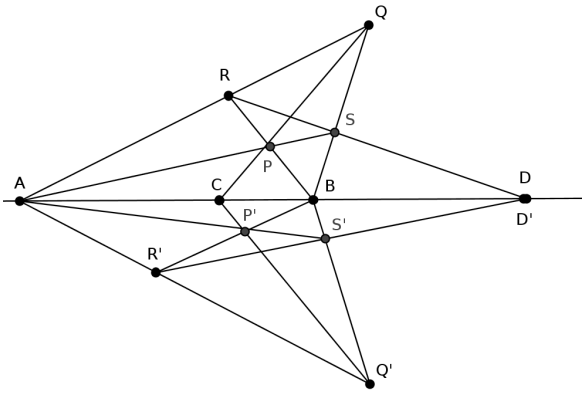


Fig. 3. Two harmonic configurations. The case of completely disjoint quadrangles

relation. This is important, since from the negation of $A \sharp B$ we can infer $A \approx B$ (by definition) and the negation $A \notin r$ entails that $A \in r$ (by axiom C6).

The first and second assumptions ensure the existence of the quadrangles for both configurations. The condition $l \sharp l'$ permits to deduce that $P \sharp P'$ and $Q \sharp Q'$. From $R' \notin RB$, and $R' \notin RA$ we can prove that the triangles PQR and $P'Q'R'$ are perspective from the axis AB. By Desargues converse, PQR and $P'Q'R'$ are perspective from a center. Let be $O = PP' \cap QQ'$ the center of perspectivity.

The triangles PQS and PQS are also perspective from the axis AB (using the assumptions $SP \sharp S'P'$, and $SQ \sharp S'Q'$). By Desargues converse, these triangles are perspective by a center. It is immediate to see that this center is equal to O. We can conclude that the triangles PRS and $P'R'S'$ are perspective from the center O (the assumptions $O \notin RS$, $O \notin R'S'$ and $S \sharp S'$ are required) Applying now Desargues, these triangles are also perspective from an axis. We can easily determine that this axis is the line AB and hence $D \approx D' = RS \cap R'S'$. From the general assumption $D \sharp D'$, we obtain the contradiction for this case.

Just this case is usually presented in most books about geometry projective as a complete proof of uniqueness of harmonic conjugate. Moreover, some of the required premises (like those involving the point O) are omitted.

Veblen and Young [23] present a proof like the above described. They argue that for the case where a vertex or side of HC coincides with a vertex or side of HC′, we can consider a third configuration HC″ whose vertexes and sides are distinct from those of HC and HC′. Hence, we can apply the previous proof to the configurations HC and HC″ and then to the configurations HC′ and HC″. By transitivity, it is deduced the equality of D and D'. If we want to formalize this argument, we have to give a method to construct this third configuration. Moreover, we should provide a method for each of the several cases where one of the previous hypothesis fails. This task seems to be rather complex in the context of a constructive approach [8] .

In turn, Coxeter's book [4] presents only the proof for the case of disjoint configurations and the proof for the other cases are left to the reader.

[8] Note the required method to obtain a third configuration becomes trivial if we consider a plane embedded in a projective space.

With respect to constructive approaches, Heyting [7] gives a quadrangle based definition of harmonic conjugates in the context of projective geometry of the space, providing a construction which only applies to points different from the base points (A and B). At the same time, the proof given by Mandelkern [15], is not rigorous enough in the construction of the sequence of contradiction and it has some inaccuracies [16].

In addition to the proof based upon the eleven assumptions, in our implementation, we have considered a lot of more cases negating the assumptions in sequence one at a time. For each case, we have to obtain a contradiction. The contradiction for some cases is rather evident, while for others it is required a considerable effort. Mostly, we obtain the required result by a number of applications of Desargues just as we do for the case of disjoint configurations. For a number of cases it is possible to take advantage of the symmetry of the configurations and to reuse the proof already given for an analogous situation.

8 Conclusion and Related Work

We described a formalization of a basic fragment of constructive projective plane geometry which covers: representation of basic objects and relations, definition of projective plane with its axioms, basic propositions of incidence, principle of duality, definition of complete n-point, representation of Fano and Desargues axioms and proof of duality, definition of the harmonic conjugate and proofs of existence and uniqueness. The implementation has been restricted to use only valid methods from constructive mathematics. We implement the axiom system described in [15] with some minor variants. In addition, the equivalence with other well-known constructive axiom systems was proved.

The proof of uniqueness of the harmonic conjugate turns out to be a quite complex proof when carried out just by using constructively valid methods. In fact, we could not find a complete and rigorous presentation of this proof in the literature. Therefore, a formal and automated verified construction of such a proof constitutes a major contribution o this paper.

We based our development on the programming language features of Agda rather than on the proof-assistant ones. In particular, we have not constructed proofs by tactics. We have taken advantage of a number of important features of the language. Modules and dependent record types have played an important role allowing us to implement setoids as an *abstract data type* and to view lines and points as instances of this ADT. In addition, the mechanism of *implicit instances* has provided the ability for overloading the operators of this ADT. We almost did not need any additional data type: only natural numbers and finitary sets are used in order to provide the definition of complete n-point. With respect to the logic involved in the implementation: we have used the standard connectives of first order intuitionistic logic, namely implication, negation, disjunction, conjunction and quantifiers.

Induction and recursion have been rarely used along our implementation. This is rather unusual for a formalization in type theory, but it is reasonable in a case where the main objects are ADTs instead of inductively defined data types. Accordingly, pattern matching is scarcely used (only to process disjunctions and finitary sets).

One of the goals of our work has been the implementation of a framework where one can easily construct proofs of projective geometry without using automatic tactics. By "easily construct" we means to write formal proofs in a similar way as it is done in mathematics textbooks. In other words, we want to reduce the complexity of a formal proof compared with a paper proof. We have obtained very elegant and simple proofs combining the operators provided by our implementation (e.g. rewriting, join-meet rules, eq-reasoning, circularity of configurations). However, some proofs (namely: Desargues an Fano duals and unicity of harmonic conjugates) are excessively verbose compared with their textbook versions. Some additional work is required in order to reduce this complexity. We conjecture that an appropriate redesign of some of the basic modules of our implementation would help to reach this goal. Moreover, some kind of automatization would be useful to avoid boilerplate code (e.g. proofs of equality for lines and points).

Related Work. To the best of our knowledge, there are no other formalizations of geometry (of any kind) implemented in Agda.

There exists an early formalization written in Alf (an ancestor of Agda) [13] by von Plato [24]. This is a very simple formalization of von Plato axiom system. A more advanced implementation is [8] which formalizes von Plato constructive geometry in Coq. These works only represent a small fragment of projective geometry without cover advanced concepts such as Desargues theorem and harmonic conjugates. However, they use the same representation for line/point existence and uniqueness: with functions *join* and *meet* defined as in our implementation. The definition of apartness relation is also very similar.

If we restrict ourselves to projective geometry formalizations in type theory, we should mention the work by Magaud, Narboux and Schreck [11,12]. They describe Coq implementations of the basic concepts of synthetic projective geometry using a tactic-based method to construct the proofs. They cover some complex theorems like Desargues and the principle of duality. There are some points in common with our formalization, namely the representation of the projective plane as a record and the proof of duality carried out by providing a function among records. The main and capital difference is that their formalization does not follow a pure constructive approach since the incidence and equality relations are postulated to be decidable.

At last, we note that we are not aware of any formalization of projective geometry covering the area of harmonic conjugates.

Future Work. This work is part of a more ambitious project consisting in a complete formalization of constructive projective geometry. Future topics to cover will be: projectivities and conics. In addition, there are a number of interesting related problems to be investigated such as: the relationship of projective geometry with affine geometry, i.e. the extension problem ([21], [14]); implementation of the coordinatization of the projective plane; relation with algebraic models of projective geometry like *Grassman-Cayley* algebra, geometric algebra, etc. [10].

Acknowledgments. I would like to thank Alberto Pardo and the anonymous referees for their useful comments. I also thank M. Mandelkern for helpful comments about his work on constructive projective geometry.

References

[1] Agda developers, *Agda standard library*, http://wiki.portal.chalmers.se/agda/pmwiki.php/Libraries/StandardLibrary, [Date Accesed: 2017-04-05].

[2] Beutelspacher, A. and U. Rosenbaum, "Projective geometry - from foundations to applications." Cambridge University Press, 1998, I-X, 1-258 pp.

[3] Birchfield, S., *An Introduction to Projective Geometry (for computer vision)*, http://robotics.stanford.edu/~birch/projective/ (1998), Date accessed: 2017.

[4] Coxeter, H. S. M., "Projective geometry," Springer Science & Business Media, New York, 2003.

[5] Devriese, D. and F. Piessens, *On the bright side of type classes: instance arguments in Agda*, in: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, 2011, pp. 143–155.

[6] Herman, I., "The use of projective geometry in computer graphics," Springer-Verlag Berlin ; New York, 1992, viii, 146 p. : pp.

[7] Heyting, A., *Zur intuitionistischen axiomatik der projektiven geometrie*, Mathematische Annalen **98** (1928), pp. 491–538.

[8] Kahn, G., *Constructive geometry according to Jan von Plato*, Coq contribution. Coq **5** (1995), p. 10.

[9] Klein, F., "Das Erlanger Programm," 253, Akademische Verlagsgesellschaft Geest & Portig, 1974.

[10] Li, H., "Invariant algebras and geometric reasoning," World Scientific, Singapore, 2008.

[11] Magaud, N., J. Narboux and P. Schreck, *Formalizing projective plane geometry in Coq*, in: *Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai, China, September 22-24, 2008. Revised Papers*, 2008, pp. 141–162.

[12] Magaud, N., J. Narboux and P. Schreck, *A case study in formalizing projective geometry in Coq: Desargues theorem*, Comput. Geom. **45** (2012), pp. 406–424.

[13] Magnusson, L. and B. Nordström, *The ALF proof editor and its proof engine*, in: *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, 1993, pp. 213–237.

[14] Mandelkern, M., *Constructive projective extension of an incidence plane*, Transactions of the American Mathematical Society **366** (2014), pp. 691–706.

[15] Mandelkern, M., *A constructive real projective plane*, Journal of Geometry (2015), pp. 1–42.

[16] Mandelkern, M., *About the proof of uniqueness of harmonic conjugates*, Personal Communication (by email) (2017).

[17] Nordström, B., K. Petersson and J. Smith, "Programming in Martin-Löf's type theory: an introduction," International series of monographs on computer science, Clarendon Press, 1990.

[18] Norell, U., "Towards a practical programming language based on dependent type theory," Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007).

[19] Norell, U., *Dependently typed programming in Agda*, in: *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, 2009, pp. 1–2.

[20] The Agda Team, *Universe polymorphism. The Agda wiki*, http://wiki.portal.chalmers.se/agda/pmwiki.php/ReferenceManual/UniversePolymorphism, [Date Accesed: 2017-07-03].

[21] van Dalen, D., *Extension problems in intuitionistic plane projective geometry, I and II*, *Indagationes Mathematicae* , **66**, Elsevier, 1963, pp. 349–368.

[22] van Dalen, D., *'Outside' as a primitive notion in constructive projective geometry*, Geometriae Dedicata **60** (1996), pp. 107–111.

[23] Veblen, O. and J. W. Young, "Projective geometry," **2**, Ginn, 1918.

[24] von Plato, J., *The axioms of constructive geometry*, Annals of pure and applied logic **76** (1995), pp. 169–200.

[25] von Staudt, K. G. C., "Geometrie der Lage," Nürnberg, 1847.

Formalizing Abstract Computability: Turing Categories in Coq

Polina Vinogradova[a,1] Amy P. Felty[a,b,2] Philip Scott[b,a,3]

[a] *School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada*

[b] *Department of Mathematics and Statistics, University of Ottawa, Ottawa, Canada*

Abstract

The concept of computable functions (as developed by Gödel, Church, Turing, and Kleene in the 1930's) has been extensively studied, leading to the modern subject of recursive function theory. However recent work by category theorists has led to a more conceptual and abstract foundation of computability theory– Turing categories. A Turing category models the notion of partial map as well as recursive computation, using methods of categorical algebra to formalize these concepts. The goal of this work is to provide a formal framework for analyzing this categorical model of computation. We use the Coq Proof Assistant, which implements the Calculus of (co)Inductive Constructions (CIC), and we build on an existing Coq library for general category theory. We focus on both formalizing Turing categories and on building a general framework in the form of a well-structured Coq library that can be further extended. We begin by formalizing definitions, propositions, and proofs pertaining to Turing categories, and then instantiate the more general Turing category formalism with a CIC description of the category which explicitly models the language of partial recursive functions.

Keywords: Category theory, Turing categories, Computability, Formalization, Calculus of Inductive Constructions (CIC), Coq proof assistant

1 Introduction

Traditional computation theory (Gödel, Turing, Kleene) originally aimed at capturing the informal notion of computable functions over the natural numbers and computable theories of real numbers (e.g. [17,10]). Even before Turing, Church's introduction of lambda calculus [2] attempted to capture certain properties of computation in a broader sense, through manipulation of strings of symbols representing function application and abstraction. It was subsequently shown that the numerical (partial) functions computable by these various abstract formalisms for computation coincided, thus suggesting the so-called Church-Turing thesis. Later independent formalisms for defining numerical computation (e.g. Register Machines, Markov's

[1] Email: polina.vino@gmail.com

[2] Email: afelty@uottawa.ca

[3] Email: phil@site.uottawa.ca

algorithms, etc.) were shown to again lead to the same class of computable numerical partial functions, lending yet more credance to the Church-Turing thesis. Nevertheless, practical as well as theoretical computer science requires more than just numerical computation: one has increasingly abstract theories of computation over various data types, higher-order computation, computation based on various programming language paradigms, newer paradigms of computation (parallel, probabilistic, quantum) etc. Category theory appears to be both general and expressive enough to be the tool of choice for modeling computation in these newer senses.

Many concepts of traditional recursion and computation theory have begun to be effectively analyzed categorically. This includes early fundamental work of Elgot on flowchart semantics, and its connections with denotational semantics (cf. M. Arbib and E. Manes [13]), to analysis of Church's theories of lambda calculi (both typed and untyped) in cartesian closed categories and associated higher-order categorical logics ([11]). In a series of works of increasing categorical generality, beginning with Longo and Moggi [12], Di Paola and Heller [15] and culminating in recent work of Cockett and Hofstra [6], we see the beginnings of a new and direct categorical development of the foundations of recursion theory

This paper is based on the thesis of the first author [19]. The model we study is the category-theoretic formalism of *Turing Categories*, introduced by Cockett and Hofstra [6]. Turing categories are a very general computational model, built from a categorical analysis of partial maps in categories. The partial maps of a Turing category arise as the computable maps of a partial combinatory algebra (PCA). Moreover, recent work establishes criteria for determining when various complexity classes of total maps can be made into a Turing category [3]. Thus the notion of Turing category provides a robust, abstract framework for discussing computation over a wide range of settings.

Our study of the Turing category computation model takes the form of building a type-theoretic formal language description (formalization) of the relevant concepts. The concepts we have selected to formalize lay the groundwork for (formally) proving abstract interpretations of standard theorems in recursion theory. The key motivation behind this approach is the level of organization, consistency, and guaranteed correctness it provides in working with proofs and definitions for which informal formulations may omit important and interesting details.

Turing category theory can be viewed as an (up until now) informally-presented mathematical framework that can be used to describe formal computation. As computation on a physical computer is a precise procedure, it seems natural to verify that a formal description of it exactly fits the selected categorical model. This is the motivating idea and the main objective of this work. There is not a huge amount of work done in this direction of research; specifically, in formalizing a category as an instance of an abstract computational model. Furthermore, we choose to work in the Coq Proof Assistant, with the Calculus of (co)Inductive Constructions (CIC) as its underlying formal language. Thus, we are using intuitionistic logic to build the proofs and definitions in this formalization. This further differentiates this development from traditional recursion theory, and adds interesting constructivist information to our proofs. For example, to verify if $f : A \to B$ is a function, we must confirm that for each proof that $x \in A$, we can prove $f(x) \in B$.

There have been previous attempts to formalize certain aspects of computation, both as categorically abstracted concepts and as direct formalizations of partial or total computation. Our project, in fact, builds on an existing constructive formalization of partial recursive functions in Coq [21], and makes use of the S_n^m theorem proved within the resulting language. There are other formalizations of traditional computation, such as primitive recursion in [14], a formalization of computable functions done directly using lambda abstractions (rather than a specific proof assistant, although the project was motivated by considerations of NUPRL) [8], and formalizations done in different proof assistants such as HOL [20] (this formalization is done using non-constructive logic). But we stress that our formalization (in Coq) is based on the novel structure of Turing categories, and an associated theory of partial maps in categories.

As far as formalization of categorical abstractions of computational structures goes, a formalization (using Coq) of cartesian closed categories, which have been previously used to model total computation, is found in [16]. Furthermore, a formalization of a categorical partiality structure which represents the same notion that we use as the partiality structure in Turing categories has also previously been done using the Agda proof assistant [1].

We start from a library for general category theory developed by Timany and Jacobs [16], designed to take advantage of advanced features in Coq 8.5 such as type classes and universe polymorphism. This library successfully develops many of the basic concepts, and thus we chose to adopt the style of definitions and formalization strategy used in this library. With this library as a starting point, we specify the mathematical definitions found in the framework of the Turing Category computation model, as well as abstract versions of other types of structures naturally occurring in the traditional computation model. We then formally prove (the abstract versions of) a number of results from traditional recursion theory.

In addition to formalizing the categorical concepts, we formalize several examples of categories. These examples provide validation of our formalization approach and formalized results. They also provide a mechanism to formally study these specific example categories. Our main example is the formalization of traditional computation on the natural numbers and the categorical interpretation of all the structure found therein, illustrating that these indeed conform to the Turing category model formalism. We base our formalization of traditional recursion theory on a formalization due to Zammit [21].

Our Coq scripts, compilation instructions, and a link to the library we build on is available at: https://github.com/polinavino/Turing-Category-Formalization.

2 Our Formalization

We divide our explanation of the formalization according to the Coq files we have built. The section corresponding to each file discusses definitions and their formal encoding as well as the challenges of reconciling the differences between them.

Restriction. In order to model partial computation using category theory, we need to first axiomatize partiality in a category. In a Turing category, this is done in terms of a restriction combinator [7]. A restriction combinator takes a map

$f : A \to B$ to an idempotent $\overline{f} : A \to A$ in a way that satisfies certain rules, and in a sense axiomatizes the notion of "domain" of f. In addition to a partiality structure, a category wherein we are able to axiomatize computation requires a version of cartesian product structure which interacts well with the restriction structure. The partial versions of products and terminal objects are called restriction (or partial) products and restriction-terminal objects, respectively. A category which admits these structures is called a *cartesian restriction category.*

Following the style of the Coq category theory library we have selected, we use type classes to formalize categorical notions. Type classes are a versatile and convenient way to encapsulate terms and propositions about them into a single term representing its informal counterpart, with a number of features particularly useful for reasoning about category theory. We have also formalized and proved a number of results about cartesian restriction structure, including:

(i) A restriction terminal object in a cartesian restriction category is a (true) terminal object in its total subcategory, and a corresponding result about restriction products [6];

(ii) Restriction products in a cartesian restriction category are (true) products in its total subcategory [6];

(iii) A cartesian category with $\overline{f} = 1$ for all maps f is a cartesian restriction category [6];

(iv) In an embedding-retraction pair (m, r), m is a total map [7];

(v) A ranges and retractions lemma from [18].

PCA and CompA. The underlying computational structure in a Turing category is a non-associative algebra called a partial combinatory algebra (abbreviated PCA), which consists of a pair $(A, \bullet)$ of an object $A \in \mathsf{C}$ and a map $A \times A \to A$ in a cartesian restriction category C, as well a combinatory completeness condition that $(A, \bullet)$ must satisfy in this category. Given an object A in an arbitrary restriction category C, the full subcategory of all objects of the form A^n is denoted Comp(A). We have formalized the definition of a PCA, as well as Comp(A) and Split(Comp(A)) (the Karoubi envelope of Comp(A)) as cartesian restriction categories. Furthermore, in the process, we have discovered additional conditions required to show intuitionistically that Split(Comp(A)) is a cartesian restriction category.

Turing. A Turing category is a category that contains a special kind of structure that models computation. A category T is Turing if it contains an object $A \in \mathsf{T}$, called a Turing object, as well as a morphism $\bullet : A \times A \to A$, called a Turing morphism, such that each map in T factors via $\bullet$ in a specific way, similar to the factorization of maps within a cartesian closed category (CCC) [11]. We have formalized Turing structure along with a number of results about it, including:

(i) Every object in a Turing category is a retract of a Turing object [6];

(ii) An object B in a Turing category with Turing object A is Turing if and only if it is a retract of A [6];

(iii) A CCC with trivial restriction structure and an object A of which every object

is a retract is a Turing category;

(iv) The halting domain is m-complete [6];

(v) An equivalent characterization of Turing categories in terms of the Turing morphism and object embeddings [6].

In addition, we have formalized the relationship between a Turing category T with a Turing object A and the related categories Comp(A) and Split(Comp(A)). These categories embed in the following order: Comp(A) $\subseteq \mathsf{T} \subseteq$ Split(Comp(A)).

Range. Range structure in a category can be expressed in terms of another type of combinator which (whenever it exists in a category) is in a sense dual to the restriction combinator [5]. The range combinator takes a map $f : A \to B$ to a map $\hat{f} : B \to B$, and, as with the restriction combinator, satisfies a number of axioms. It is related to the notion of open maps in a category, in that whenever all maps in a given category are open, it is a range category. We have chosen to formalize this particular abstraction because in the process of formalizing the motivating examples, it became apparent that representing partiality using a total formal language presented one of the biggest challenges as well as one of the greatest curiosities. We have formalized a number of results regarding the interactions between range structure and embedding-retraction pairs, as well as a criterion for a Turing category to admit cartesian range structure [18].

Par_Cat. The category Par (of Sets and partial maps) is the motivating example for the categorical structure discussed above, including cartesian restriction and cartesian range structure, but not including Turing or PCA structure. Due to the total nature of computation in CIC, it is impossible to directly represent a partial map. For this reason, we must define the type of the set of all partial maps $A \to B$ quite differently from the type of all total maps $A \to B$ in the category Set (i.e., `A -> B` in Coq). A partial map from `A` to `B` is a pair consisting of a domain predicate `P : A -> Prop` together with a map of type `forall x:A, P x -> B`, which takes two arguments: an "element" `x` of the set `A`, and a proof of the proposition `P x`.

Having defined the type of partial maps in this way, we proceeded to define the formal version of Par, which we call `Par_Cat`, as an instance of the `Category` type class from the original Coq library (i.e. define the required objects, morphisms and proofs of associativity, etc). Following a similar format, we have also instantiated Par as a cartesian restriction category and a cartesian range category (i.e., defined all the required maps, objects, and completed the accompanying proofs).

CompN_Cat. The category Par contains a subcategory of maps that are partial recursive, i.e., computable by a map which can be expressed in terms of the partial recursive constructors (zero, successor, projection, recursion, substitution and minimalization [9]). We use this definition of formal computation as the basis for our formalization of the category of computable maps. We build our subcategory using an existing formalization of this presentation of computation as well as the proof of the S_n^m theorem completed using this definition [21].

This formalization gives the definition and the semantics of the language of partial recursive maps separately. We define the language constructors as an inductive type `prf` in Coq, while the semantics are given as an inductively-defined relation, whose header is:

```
Inductive converges_to :  prf -> list nat -> nat -> Prop
```

where `(convergest_to f_prf ln n)` is provable whenever, informally, the partial recursive function `f_prf` applied to the list of natural numbers `ln` outputs `n`. Note that this "output" is unique, so we are able to build a partial map in the `Par_Cat` sense, described above, which corresponds to a given `prf` term.

Now, we formally consider a partial map with m components as a map in the category of partial recursive maps whenever there is a proof that each of its components is Kleene-equal to a `prf` computation. This category inherits cartesian restriction structure defined in the larger category, `Par_Cat`; however, we have additionally defined a Turing object (i.e., the natural numbers) and proved that the necessary diagrams commute (as maps in the larger `Par_Cat` category), thus demonstrating that it is indeed a Turing category.

3 Discussion and Future Work

The framework we have built develops the tools to study abstract recursion formally. It has the advantage that we are not required to model partial functions using total functions or relations (or total functions built using relations) in order to study partial recursion using (strongly normalizing and intuitionistic) CIC.

As with most proofs in category theory, informal results about Turing, restriction and range categorical structure do not require reasoning using the law of excluded middle, or any other application specifically of classical logic reasoning. Thus, proofs of the results we chose to formalize hold up in a constructive setting.

While most of the formal results we have proved confirm what has already been shown informally, formalization also gives us the ability to find omissions in the informal definitions, proofs and propositions. For example, in the process of formalizing a result about ranges in Turing categories, we saw that we were not able to directly express the result in terms of range structure, and had to instead formulate and prove a very closely related result in terms of open maps.

In the formalization of partial maps, however, it is not always the case that we can build definitions (and therefore proofs) directly following the informal strategy. For example, one major difference between our formal presentation of partial maps and the usual informal one is that a partial map `f` from `A` to `B` takes two arguments, an element `x : A` and a proof of membership of `x` in the domain of `f` (as discussed earlier), instead of the usual one argument, and this is expressed using dependent types. Now, because of this, in order to prove equality between partial maps formally, we require a stronger version of the (dependent) functional extensionality axiom as well as the proof irrelevance axiom (to identify all proofs of the same proposition as equal).

The most noteworthy result we have formalized is the constructive version of the category of sets of the form $\mathbb{N}^n$ and computable maps between them, which is meant to categorically represent traditional computation. This has not previously been done either formally or informally. Through our work, we have gained an understanding (as well as formal constructions) of the additional results, concepts and machinery that are needed to build such a category.

Here, again, there are far-reaching repercussions of not being able to use the

law of excluded middle, such as constructing partial maps out of a language of partial recursive maps `prf` (discussed in the previous section), the semantics of which can most directly be expressed as a relation. Some of the key recursion theory results have already been demonstrated using this language directly (such as the S_n^m theorem) [21], and therefore hold up in the (cartesian restriction) Turing category we have built. However, the purpose of Turing categories is, in part, to be able to do as much recursion outside of extensional reasoning of set theory as possible. For this reason, it would be beneficial to extend the scope of this formal framework to include other categorical structures useful for presenting elements of recursion theory in an abstract categorical way.

There are a number of promising directions for further applying this framework. The most natural, perhaps, is the formalization of the Turing category-formulated abstraction of Rice's theorem. This will require the formalization of a number of general categorical concepts such joins and meets, as emphasized in Cockett's lectures [4]. Such general concepts are widely applicable to other results as well. Applying our framework in another direction, it would be an extremely interesting and innovative pursuit to use it to study computational complexity classes of total maps in Turing categories. Other potentially interesting options for building on this framework include formalizing monoidal Turing categories (with differential structure) and conducting a formal study more focused on the PCA's (which, recall, are computation-modeling structures at the core of every Turing category) as well as relationships between them.

References

[1] Chapman, J., T. Uustalu and N. Veltri, *Formalizing restriction categories* (2017).

[2] Church, A., "The Calculi of Lambda Conversion. (AM-6)," Annals of Mathematics Studies, Princeton University Press, 2016.

[3] Cockett, J., P. Hofstra and P. Hrubeš, *Total maps of turing categories*, Electronic Notes in Theoretical Computer Science **308** (2014), pp. 129 – 146, proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX).

[4] Cockett, R., *Turing categories and computability* (2010), slides from 15th Estonian Winter School in Computer Science (EWSCS).
URL http://cs.ioc.ee/ewscs/2010/cockett/estonia-slides-4.pdf

[5] Cockett, R., X. Guo and P. Hofstra, *Range categories II: Towards regularity*, Theory and Applications of Categories **26** (2012), pp. 453–500.

[6] Cockett, R. and P. Hofstra, *Introduction to Turing categories*, Annals of Pure and Applied Logic **156(2-3)** (2008), pp. 183–209.

[7] Cockett, R. and S. Lack, *Restriction categories I*, Theoretical Computer Science **270** (2002), pp. 223–259.

[8] Constable, R. L. and S. F. Smith, *Computational foundations of basic recursive function theory*, Theoretical Computer Science **121** (1993), pp. 89 – 112.
URL //www.sciencedirect.com/science/article/pii/0304397593900858

[9] Cutland, N., "Computability: An introduction to recursive function theory," Cambridge University Press, 1980.

[10] Kleene, S. and M. Beeson, "Introduction to Metamathematics," Ishi Press International, 2009.

[11] Lambek, J. and P. J. Scott, "Introduction to Higher Order Categorical Logic," Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1986, 7 edition.

[12] Longo, G. and E. Moggi, "Gödel numberings, principal morphisms, combinatory algebras," Springer Berlin Heidelberg, Berlin, Heidelberg, 1984 pp. 397–406.

[13] Manes, E. and M. Arbib, "Algebraic Approaches to Program Semantics," Monographs in Computer Science, Springer New York, 2012.

[14] O'Connor, R., "Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory," Ph.D. thesis, Institute for Computing and Information Science, Faculty of Science, Radboud University Nijmegen (2009).

[15] Paola, R. A. D. and A. Heller, *Dominical categories: Recursion theory without elements*, J. Symbolic Logic **52** (1987), pp. 594–635.

[16] Timany, A. and B. Jacobs, *Category theory in Coq 8.5*, in: *The 7th Coq Workshop*, Sophia Antipolis, France, 2015.
URL http://arxiv.org/abs/1505.06430

[17] Turing, A. M., *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London mathematical society **2** (1937), pp. 230–265.

[18] Vinogradova, P., "Investigating Structure in Turing Categories," Master's thesis, University of Ottawa (2012).

[19] Vinogradova, P., "Formalizing Abstract Computability: Turing Categories in Coq," Ph.D. thesis, University of Ottawa (2017).

[20] Zammit, V., *A mechanisation of computability theory in HOL*, in: *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96 (1996), pp. 431–446.
URL http://dl.acm.org/citation.cfm?id=646523.694703

[21] Zammit, V., *A proof of the S-m-n theorem in Coq*, Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK (1997), http://kar.kent.ac.uk/21524/.

Submitted to LSFA 2017

Confluence in Probabilistic Rewriting[1]

Alejandro Díaz-Caro[a,2] Guido Martínez[b]

[a] *Universidad Nacional de Quilmes & CONICET. Bernal, Buenos Aires, Argentina*
alejandro.diaz-caro@unq.edu.ar

[b] *Universidad Nacional de Rosario & CIFASIS-CONICET. Rosario, Santa Fe, Argentina*
martinez@cifasis-conicet.gov.ar

Abstract

Driven by the interest of reasoning about probabilistic programming languages, we set out to study a notion of unicity of normal forms for them. To provide a tractable proof method for it, we define a property of *distribution confluence* which is shown to imply the desired unicity (even for infinite sequences of reduction) and further properties. We then carry over several criteria from the classical case, such as Newman's lemma, to simplify proving confluence in concrete languages. Using these criteria, we obtain simple proofs of confluence for λ_1, an affine probabilistic λ-calculus, and for Q^*, a quantum programming language for which a related property has already been proven in the literature.

Keywords: abstract rewriting system, probabilistic rewriting, confluence

1 Introduction

In the formal study of programming languages, modelling execution via a small-step operational semantics is a popular choice. Such a semantics is given by an *abstract rewriting system* (ARS) which, mathematically, is no more than a binary relation on abstract terms specifying whether one term can rewrite to another. This relation is not required to be a function, and can thus allow for a program to rewrite in two different ways. In such a case, it is important that the different execution paths for a given program reach the same final value (if any); thus guaranteeing that any two determinizations of the semantics (e.g. execution machines) assign the same meaning to every program.

This correctness property is expressible at the level of relations, and is known as *unicity of normal forms* (UN): any two irreducible terms reachable from a common starting point must be equal. For non-trivial languages, such as the λ-calculus, it can be hard to prove this property directly. Fortunately, the property of *confluence* can serve as a proof method for it since UN is a trivial corollary of it while its proof

[1] This is basically the work done during the second author's *Licenciatura* thesis [17].

[2] Partially supported by projects STICAmSud 16STIC05 FoQCoSS and PICT-PRH 2015-1208.

tends to be more tractable. This was the approach followed by Church and Rosser in [6, Corollary 2], where UN and confluence were first proven for the λ-calculus in 1936. Nowadays, confluence is widely used to show the adequacy of operational semantics in many kinds of programming languages.

In the past decades there has been a growing interest in programming languages with probabilistic behaviours (e.g. [1,2,4,9,10,13,21,22]), which cannot be modelled as a mere relation between terms. Example features include a probabilistic choice operator [10] and quantum measurement [9]. In these settings, the same need of showing the correctness of the semantics is present.

At a first glance, it may seem as if neither UN nor confluence could ever hold in these cases, since the possible results of, say, rolling a die or measuring a qubit are irreconcilably distinct. The key observation is that, in a probabilistic language, a notion of equivalence of programs should be about distributions of values, and not about punctual values [16]. Indeed, a single program might evaluate to different values if it is run twice, but equality should certainly be reflexive. Thus, the expectation should be that the different reduction paths do not impact the final distribution of results. This is precisely the property we set out to study in this paper, developing an associated notion of confluence for it.

For a concrete example, take a hypothetical language for representing dice rolls, where $\square$ represents an unrolled die which can reduce to any element in $\{1,\dots,6\}$ with equal probability. For a pair of dice, represented by $(\square,\square)$, we should be allowed to choose which die to roll first. Rolling the first die can result in any term in the set $\{(i,\square)\}_{i=1,6}$ with equal probability; and similarly for the second. When continuing the rolls, both branches will end up in the same uniform distribution. However, this could be not so: consider the term $(\lambda x.(x,x))\ \square$. If the die is rolled before the abstraction is applied, only pairs with equal components can be obtained, which is not the case if we apply the abstraction first, obtaining $(\square,\square)$, which can reduce to any pair of results. We shall later come back to a similar language and provide conditions to avoid this divergence.

In Section 2 we introduce the problem of unicity of distributions for probabilistic rewriting. In Section 3 we define a rewriting system over distributions giving rise to our notion of distribution confluence and prove its adequacy. In Section 4 we give some criteria for proving distribution confluence, simplifying the burden of proof for concrete languages. In Section 5 we extend our unicity of distributions result to terms which are only asymptotically terminating. In Section 6 we prove confluence for two concrete programming languages: a simple, affine, probabilistic calculus, dubbed λ_1 and the quantum lambda calculus Q^* [9]. Finally, in Section 7 we conclude, give some insights on future directions, and analyse some related work.

2 Probabilistic Rewriting

2.1 Preliminaries

We assume familiarity with the study of abstract rewriting. We adopt the terminology from [6] and call a sequence of expansions followed by a sequence of reductions

$$[(p_1, a_1), (p_2, a_2)] \sim [(p_2, a_2), (p_1, a_1)] \qquad \text{(Flip)}$$
$$[(p_1, a), (p_2, a)] \sim [(p_1 + p_2, a)] \qquad \text{(Join)}$$
$$[(p_1 + p_2, a)] \sim [(p_1, a), (p_2, a)] \qquad \text{(Split)}$$

Fig. 1. Basic steps of the $\sim$ relation

a *peak*. When the order is reversed, such sequence is called a *valley*. The property of confluence can then be expressed as "there is a valley for every peak". When an ARS $\mathcal{A}$ is confluent, we note it by $\mathcal{A} \models \mathrm{CR}$, and similarly for UN and other properties. The relation R *modulo* E, noted as $R \,/\, E$, is defined for any equivalence relation E as $E \cdot R \cdot E$ [15, 20].

We denote by $\mathscr{L}(X)$ the type of finite lists with elements in X, where '$[\,]$', '$:$' and '$+\!\!+$' denote the empty list, the list constructor, and list concatenation, respectively. We also use the notation $[a, b, c]$ for $a : b : c : [\,]$.

We define $\mathscr{D}(A) = \mathscr{L}(\mathbb{R}^+ \times A)$, used as an explicit representation of (finitely-supported) distributions. There is no further restriction on $\mathscr{D}(A)$. In particular, any given element might appear more than once, as in $[(1/3, a), (1/2, b), (1/6, a)]$. On a node (p, a) of the distribution, p is called the *weight* and a the *element*. The weight of a distribution is defined to be the sum of all weights of its nodes, and can be any positive real number. When we require a normalised distribution, we use the type $\mathscr{D}_1(A)$, defined as the set of those $d \in \mathscr{D}(A)$ with unit weight. We abbreviate the distribution $[(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)]$ by $[(p_i, a_i)]_i$, where n should be clear from the context. We also write αD for the distribution obtained by scaling every weight in D by α; that is, $\alpha\,[(p_i, a_i)]_i = [(\alpha p_i, a_i)]_i$.

We often reason about *equivalence* of distributions. For that purpose, we define a relation '$\sim$' as the congruence closure of the rules in Fig. 1 (i.e. the smallest relation satisfying the rules and such that $D \sim D'$ implies $E_1 +\!\!+ D +\!\!+ E_2 \sim E_1 +\!\!+ D' +\!\!+ E_2$). We distinguish some subsets of '$\sim$' by limiting the rules that may be used. We note by S the congruence closure of Split, by (FJ) the congruence closure of both Flip and Join, and similarly for other subsets.

We call two distributions *equivalent* if they are related by the reflexive-transitive closure of $\sim$, noted $\approx$. Two distributions are equivalent, then, precisely when they assign the same total weight to every $a \in A$, regardless of order and duplication.

Arguably, using such a definition of distributions with lists is cumbersome and less clear than using a more semantic definition. However, we feel this is outweighed by the degree of rigurosity attained in later proofs (especially as we found some of them to be quite error-prone). As a secondary benefit, most of our development should be straightforwardly mechanizable.

A downside of the choice of lists is that we can only represent finitely-supported distributions. This restriction is present in other works as well (e.g. [11, 23]) and it seems to not be severe for modelling programming languages.

2.2 Probabilistic Abstract Rewriting Systems (PARS)

To model probabilistic rewriting, we need to move away from a simple relation between terms as used in ARSes. We shall then relate elements to *distributions*

of elements, which introduce probability. In favour of expressivity, we allow for a single element to be related to multiple distributions (or none)[3].

Definition 2.1 *A probabilistic abstract rewriting system (PARS) is a pair $(A, \mapsto)$ where A is a set (called the "carrier") and $\mapsto$ a relation of type $\mathcal{P}(A \times \mathscr{D}_1(A))$ (called the "pointwise evolution relation").*

It should be clear that every ARS is also a PARS by taking Dirac distributions (i.e. normalised distributions with a single element). We can provide a simple example of a PARS by extensionally listing $\mapsto$, as is commonly done for ARSes.

Example 2.2 Let $\mathcal{A}$ be the PARS given by

$$a \mapsto [(2/3, b), (1/3, c)] \qquad a \mapsto [(2/5, d), (3/5, e)]$$
$$b \mapsto [(1/2, c), (1/2, d)] \qquad c \mapsto [(1, d)]$$

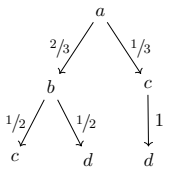
Here, a is the only non-deterministic element. We call d a *terminal* element since it has no successor distributions.

As more significant examples, in Section 6 we describe two probabilistic λ-calculi with operational semantics modellable by a PARS.

Execution in a PARS is a mixture of non-deterministic and probabilistic choices. The first kind, corresponding to the $\mathcal{P}$ operator, occur when the machine chooses a successor distribution for the current element. The second kind, corresponding to the $\mathscr{D}_1$ operator, is a random choice between the elements of the chosen successor distribution. To model such execution, we introduce the notion of *computation tree*.

Definition 2.3 *Given a PARS $(A, \mapsto)$, we define the set of its (finite) "computation trees" with root a (noted $\mathcal{T}(a)$) inductively by the following rules.*

$$\frac{}{a \in \mathcal{T}(a)} \qquad \frac{a \mapsto [(p_1, a_1), \ldots, (p_n, a_n)] \quad t_i \in \mathcal{T}(a_i)}{[a; (p_1, t_1); \ldots; (p_n, t_n)] \in \mathcal{T}(a)}$$



A graphical representation of an example tree for the PARS in Definition 2.2 is given in the right. We also sometimes consider infinite computation trees, by taking the coinductively defined set instead.

So, any tree $T \in \mathcal{T}(a)$ represents one possible (uncertain) evolution of the system after starting on a. There is no further assumption about trees: in particular, if an element a is expanded many times in a tree, different successor distributions may be used at each node[4].

When all the leaves of a tree are terminal elements, we call the tree *maximal* (as there is no proper supertree of it). A computation tree naturally assigns to each of its leaves a probability, taken as the product of the p_i in the path from the

[3] Note that limiting it to a single distribution would make it impossible to model the term $(\square, \square)$ without fixing a strategy.

[4] In some works, such as [3,4], it is assumed that execution is *Markovian*, meaning the choice of distribution depends only on the current state. We do not make such assumption and allow for a more general execution.

root to it. It is clear that collecting the leaves of a tree (along with their assigned probabilities) gives rise to a normalised list distribution. We call such distribution the *support* of T and note it as $\mathrm{supp}(T)$.

We can now state our property of interest.

Definition 2.4 (UTD) *A PARS $\mathcal{A}$ has "unique terminal distributions" when for every a and $T_1, T_2 \in \mathcal{T}(a)$ maximal, we have* $\mathrm{supp}(T_1) \approx \mathrm{supp}(T_2)$.

We said before that proving UN (for ARSes) directly is usually hard. Since PARSes subsume ARSes, the same difficulties arise for proving UTD directly. Therefore, we seek a property akin to confluence, providing a more tractable proof method.

One idea is to set up a rewriting over the supports of computation trees (expanding leaves as a reduction) and study its confluence. However, this is bound to be too rigid, as the behaviours of computation trees are not identified by equivalence of their support distributions, forbidding reasoning modulo equivalence. For example, a tree with two leaves $(1/2, a)$ can exhibit more terminal distributions than a tree with a single $(1, a)$ leaf. Therefore we would have to deem the following, morally confluent, system:

$$a \mapsto [(1/2, b), (1/2, c)] \qquad a \mapsto [(1, c)] \qquad b \mapsto [(1, c)]$$

as non-confluent, since $[(1/2, c), (1/2, c)] \neq [(1, c)]$.

In order to not reject cases like this, we could change the previous notion to allow for an equivalence when closing the diagram. In this case, it is hard to reason compositionally about the property.

3 Rewriting Distributions and Confluence

To arrive at a notion of confluence that avoids the previously mentioned drawbacks, we shall define a rewriting over distributions that is more liberal than that of computation trees (Definition 3.4).

Definition 3.1 *Given a PARS $\mathcal{A} = (A, \mapsto)$, we define the relation $\twoheadrightarrow_P$ (of type $\mathcal{P}(\mathscr{D}(A) \times \mathscr{D}(A))$) (called "parallel evolution") by the rules:*

$$\frac{a \mapsto A \quad ds \twoheadrightarrow_P ds'}{(p, a) : ds \twoheadrightarrow_P pA +\!\!+ ds'} \qquad \frac{ds \twoheadrightarrow_P ds'}{(p, a) : ds \twoheadrightarrow_P (p, a) : ds'} \qquad \frac{}{[\,] \twoheadrightarrow_P [\,]}$$

Note that without using the first rule, this is just the identity relation on distributions. We note the subset of this relation where the first rule must be used at least once in a step as $\twoheadrightarrow_P^1$, and call it *proper* evolution. Note that $\twoheadrightarrow_P$ is enough to simulate computation trees in this system, since it can be used to rewrite between their supports in the following sense.

Definition 3.2 *We call a relation $\to$ "compositional" when, if $D_1 \to E_1$ and $D_2 \to E_2$, then $\alpha D_1 +\!\!+ \beta D_2 \to \alpha E_1 +\!\!+ \beta E_2$ for all $\alpha, \beta \in \mathbb{R}^+$.*

Lemma 3.3 *If $T \in \mathcal{T}(a)$, then $[(1, a)] \twoheadrightarrow_P^* \mathrm{supp}(T)$*

Proof. First, note that $\twoheadrightarrow_P$ is compositional. The result then follows by induction

on T, using compositionality. □

We now define an ARS over distributions, combining both parallel evolution and equivalence steps. Our definition of confluence for a PARS $\mathcal{A}$ is then simply the usual confluence of that relation.

Definition 3.4 *Given a PARS $\mathcal{A} = (A, \mapsto)$, we define an associated ARS* $\mathrm{Det}(\mathcal{A})$ *(called the "determinisation" of $\mathcal{A}$) over the set $\mathscr{D}(A)$ by the relation $\twoheadrightarrow = (\twoheadrightarrow_P \cup \approx)$.*

Definition 3.5 (Distribution confluence) *We say a PARS $\mathcal{A}$ is "distribution confluent" (or simply "confluent") when* $\mathrm{Det}(\mathcal{A})$ *is confluent in the classical sense.*

Note that reduction in $\mathrm{Det}(\mathcal{A})$ is more liberal than the expansion of trees, since it allows for "partial" evolutions. Indeed, if $a \mapsto D$, we have $[(1, a)] \twoheadrightarrow [(1/2, a), (1/2, a)] \twoheadrightarrow 1/2 D + [(1/2, a)]$. Nevertheless, its confluence is adequate for proving UTD, as Lemma 3.7 shows.

Lemma 3.6 *If D_1 is terminal and $D_1 \twoheadrightarrow^* D_2$, then $D_1 \approx D_2$ and D_2 is terminal.*

Proof. It is clear, from the definition of $\twoheadrightarrow_P$, that if D_1 is terminal and $D_1 \twoheadrightarrow_P D'$, then $D_1 = D'$ (that is, exactly equal). The result then follows by induction on the number of steps, and the transitivity and reflexivity of $\approx$. □

Lemma 3.7 *If $\mathcal{A} \models \mathrm{CR}$, then $\mathcal{A} \models \mathrm{UTD}$.*

Proof. Take $T_1, T_2 \in \mathcal{T}(a)$ maximal. We know from Lemma 3.3 that $\mathrm{supp}(T_2) \twoheadleftarrow^* [(1, a)] \twoheadrightarrow^* \mathrm{supp}(T_1)$. By confluence, there must exist C such that $\mathrm{supp}(T_2) \twoheadrightarrow^* C \twoheadleftarrow^* \mathrm{supp}(T_1)$. Since T_1, T_2 are maximal, their supports are terminal. Then, from two applications of Lemma 3.6, we get that $\mathrm{supp}(T_2) \approx C \approx \mathrm{supp}(T_1)$, as needed. □

Furthermore, beyond UTD, distribution confluence implies that diverging computations (with no terminal distribution) can also be joined. As a consequence of that, confluence gives a neat method of proving the consistency of the equational theory induced by $\twoheadrightarrow$, as long as two distinct terminal elements exist.

Lemma 3.8 *If D_1, D_2 are terminal distributions, then $D_1 \twoheadleftarrow\twoheadrightarrow^* D_2$ if and only if $D_1 \approx D_2$.*

Proof. The way back is trivial, so we detail the way forward. From confluence (repeatedly), D_1 and D_2 must have a common reduct. The result then follows from Lemma 3.6. □

So, if a and b are distinct terminal elements, we know that $\twoheadrightarrow$-convertibility is a consistent theory as $[(1, a)] \not\approx [(1, b)]$. Summarizing, in a confluent PARS, reasoning about equivalence of programs is simplified and there is a strong consistency guarantee about convertibility, much like in the classical case.

4 Proving confluence

4.1 Introduction

In the previous section, we have introduced our definition of confluence and argued that it is correct and sufficient for studying programming languages. For the prop-

erty to be useful in practice, it should be also amenable to be proven. In this section we provide several simplified criteria for this task, obtaining analogues to many of the usual methods for classical confluence.

Since distribution confluence is no more than the classical confluence of $\twoheadrightarrow$, every existing classical criteria (such as the diamond property or Newman's lemma) is valid in this setting. However, very few of those are useful. Indeed, $\mathrm{Det}(\mathcal{A})$ is never strongly (or even weakly) normalising regardless of $\mathcal{A}$, and therefore Newman's lemma does not apply. With respect to the diamond property, consider a system with $a \mapsto D$, then the following reductions are possible:

$$[(1, a)] \twoheadleftarrow [(1/2, a), (1/2, a)] \twoheadrightarrow 1/2 D + [(1/2, a)]$$

and these two distributions cannot be joined in a single step (unless we make further assumptions on $\mathcal{A}$). Also, as evidenced by this example, we need to prove confluence for every distribution, not just for Dirac ones; and take into account equivalence steps as well.

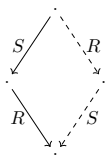
Thus, a priori, it would seem as if distribution confluence is hard to prove. To relieve that, we shall prove various syntactic lemmas about the relation $\twoheadrightarrow$, allowing us to decompose it into more manageable forms. We then show how we can limit our reasoning to Dirac distributions, ignore equivalence steps in the peaks and allow to use them freely in the valleys. Lastly, we carry over classical criteria for confluence into this setting, such as the aforementioned diamond property and Newman's lemma.

4.2 Syntactic lemmas about the relation $\twoheadrightarrow$

Since both $\twoheadrightarrow_P$ and $\approx$ are reflexive, we have $(\twoheadrightarrow_P \cup \approx)^* = (\twoheadrightarrow_P / \approx)^*$. Thus, since confluence is a property over the reflexive-transitive closure of a relation, it suffices to study the confluence of $\twoheadrightarrow_P / \approx$, where equivalence steps do not have a cost, but are pervasive.

Given the precise syntactic definition for both relations, we can prove by analysis on the reductions that any step of $\twoheadrightarrow_P / \approx$ can be made by splitting first, then evolving, and then joining back elements, as Lemma 4.3 states. We first introduce the following notion of commutation [5].

Definition 4.1 (Sequential commutation) *We say that a relation R "commutes over" S when $S \cdot R \subseteq R \cdot S$, and note it as $R \dashv S$. The property can be expressed by the diagram on the right.*



A key property of sequential commutation is that if $R \dashv S$, then $(R \cup S)^* = R^* \cdot S^*$. It is also preserved when taking the n-fold composition (i.e. "n steps") or reflexive-transitive closures on each side. We now prove some commutations relating evolution and equivalence steps (the last one needs some "administrative" steps).

Lemma 4.2 *We have $\twoheadrightarrow_P \dashv (FJ)^*$; $S \dashv (FJ)^*$ and $\twoheadrightarrow_P \cdot S \subseteq S \cdot \twoheadrightarrow_P \cdot (FJ)^*$.*

[5] Note that this is not the usual notion of commuting relations, defined as $S^{-1} \cdot R \subseteq R^{-1} \cdot S$, which is a symmetric property and could be called "parallel".

Proof. By induction on the shape of the reductions. □

Lemma 4.3 *The relations* $(\twoheadrightarrow_P / \approx)$ *and* $S^* \cdot \twoheadrightarrow_P \cdot (FJ)^*$ *coincide.*

Proof. The backwards inclusion is trivial, so we detail only the forward direction. By making use of the second commutation in Lemma 4.2 we get that $\approx = S^* \cdot (FJ)^*$. Thus, we need to show $S^* \cdot (FJ)^* \cdot \twoheadrightarrow_P \cdot S^* \cdot (FJ)^* \subseteq S^* \cdot \twoheadrightarrow_P \cdot (FJ)^*$. The proof then proceeds by using the other two commutations to reorder the relations. □

Furthermore, this equivalence extends to n-fold compositions.

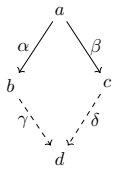
Lemma 4.4 *The relations* $(\twoheadrightarrow_P / \approx)^n$ *and* $S^* \cdot \twoheadrightarrow_P^n \cdot (FJ)^*$ *coincide.*

Proof. By induction on n, and using the previous lemma and commutations. □

4.3 Simplifying diagrams

With the previous decompositions, we can now prove a very generic result about diagram simplification with a specific root D, which then easily generalizes to the whole system.

Definition 4.5 *We say a pair of relations* (γ, δ) *"closes" another pair* (α, β) *"on* a*" if whenever* $b \leftarrow_\alpha a \rightarrow_\beta c$ *then there exists* d *such that* $b \rightarrow_\gamma d \leftarrow_\delta c$*. The diagram for the property can be seen on the right. When this occurs for all* a*, we simply say "*(γ, δ) *closes* (α, β)*". Note that* $\rightarrow$ *is confluent precisely when* $(\rightarrow^*, \rightarrow^*)$ *closes* $(\rightarrow^*, \rightarrow^*)$*.*



Definition 4.6 *We call a relation* $\rightarrow$ *"local", when if* $\alpha D_1 + \beta D_2 \rightarrow E$*, then there exist* E_1, E_2 *such that* $E = \alpha E_1 + \beta E_2$ *and* $D_i \rightarrow E_i$*. (Note that* $\twoheadrightarrow_P$ *and* S *are local).*

Theorem 4.7 *Let* α, β *be local relations and* γ, δ *compositional relations. If* $(\gamma / \approx, \delta / \approx)$ *closes* (α, β) *for the Dirac distributions of* D*, then* $(\gamma / \approx, \delta / \approx)$ *closes* $(S^* \cdot \alpha \cdot (FJ)^*, S^* \cdot \beta \cdot (FJ)^*)$ *for* D*.*

Proof. We give a sketch of the proof, for a more explanatory development please refer to [17]. We need to close $(S^* \cdot \alpha \cdot (FJ)^*, S^* \cdot \beta \cdot (FJ)^*)$. First, note that closing $(S^* \cdot \alpha, S^* \cdot \beta^*)$ is enough since we can revert the $(FJ)^*$ steps with $(FS)^*$ steps. Now, since α and β are local, $S^* \cdot \alpha$ and $S^* \cdot \beta$ are as well. Thus, we can limit ourselves to closing the Dirac distributions of D, and combine the reductions since γ, δ are compositional. We now need to close $(S^* \cdot \alpha, S^* \cdot \beta)$ when starting from some $[(1, a)]$. Note that the left (right) branch is then of the form $p_1 D_1 + \cdots + p_n D_n$ $(q_1 E_1 + \cdots + q_m E_m)$, where a reduces via α (β) to each D_i (E_j). We can apply our hypothesis to get a $C_{i,j}$ closing each D_i, E_j. By first splitting each branch appropriately, we can close them in $p_1 q_1 C_{1,1} + \ldots + p_1 q_m C_{1,m} + \ldots + p_n q_m C_{n,m}$, thus we conclude. □

From this theorem, we get as corollaries several simplified criteria for confluence, applicable at the level of a particular distribution or to the whole system.

Criterion 4.8 (Dirac confluence) *If for every element* a *of* D *and distributions* E, F *such that* $E \twoheadleftarrow_P^* [(1, a)] \twoheadrightarrow_P^* F$ *there is a* C *such that* $E \twoheadrightarrow^* C \twoheadleftarrow^* F$*, then* D

is confluent.

Proof. A corollary of Theorem 4.7, taking $\alpha = \beta = \gamma = \delta = \twoheadrightarrow_P^*$. □

Criterion 4.9 (Semi-confluence) *If for every element a of D and distributions E, F such that $a \mapsto E$ and $[(1, a)] \twoheadrightarrow_P^* F$ there is a C such that $E \twoheadrightarrow^* C \twoheadleftarrow^* F$, then D is semi-confluent for $\twoheadrightarrow_P / \approx$.*

Proof. A corollary of Theorem 4.7, taking $\alpha = \twoheadrightarrow_P$ and $\beta = \gamma = \delta = \twoheadrightarrow_P^*$. □

Criterion 4.10 (Diamond property) *If for every element a of D and distributions E, F such that $E \leftarrow\!\!\shortmid\, a \mapsto F$ there is a C such that $E \twoheadrightarrow_{P/\approx} C \twoheadleftarrow_{P/\approx} F$, then D has the diamond property for $\twoheadrightarrow_P / \approx$.*

Proof. A corollary of Theorem 4.7, taking $\alpha = \beta = \gamma = \delta = \twoheadrightarrow_P$. □

Note that in all these criteria, we need not consider any equivalence in the peak, and can use them freely in the valley, both before and after evolving. Also, proving any of these criteria for every element a entails the confluence of the system.

In the classical case, a common tool for proving confluence is switching to another rewriting relation with equal reflexive-transitive closure (and thus an equivalent confluence) but which might be easier to analyse. For distribution confluence, a similar switch is allowed, slightly simplified by Lemma 4.12.

Definition 4.11 *Given two PARS over the same carrier set A, with relations $\mapsto_1$ and $\mapsto_2$, if for every $a \mapsto_1 D$, we have $[(1, a)] \twoheadrightarrow_2^* D$ we say that $\mapsto_1$ is* simulated by *$\mapsto_2$.*

Lemma 4.12 *If $\mapsto_1$ is simulated by $\mapsto_2$, then $\twoheadrightarrow_1 \subseteq \twoheadrightarrow_2^*$. Hence, if both relations simulate each other, we have $\twoheadrightarrow_1^* = \twoheadrightarrow_2^*$, and their confluences are therefore equivalent.*

Proof. The first part follows by case analysis on the reduction $\twoheadrightarrow_1$. The second part is then trivial. □

4.4 Newman's lemma

Newman's lemma [18] states that, for a strongly normalising system, local confluence and confluence are equivalent properties, yet we have remarked previously that $\twoheadrightarrow_P$ is never a strongly normalising relation. To get an analogue to Newman's lemma, we thus provide a specialized notion of strong normalisation.

Definition 4.13 *A infinite sequence D_i such that $D_1 \to D_2 \to D_3 \to \cdots$ is called an "infinite $\to$-chain" (of root D_1).*

Definition 4.14 *We call a distribution D "strongly normalising" when there is no infinite $\twoheadrightarrow_P^1$-chain[6] of root D. We call a PARS strongly normalising when every distribution is strongly normalising.*

There are indeed systems which do satisfy this requirement, and it is intuitively what one would expect. Now a probabilistic analogue to Newman's lemma can be obtained, following a proof style very similar to that of [14].

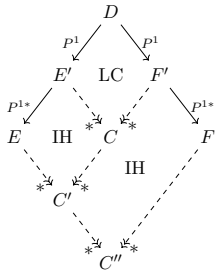
[6] Note that infinite $(\twoheadrightarrow_P^1 / \approx)$-chains always exist because of partial evolution.

Definition 4.15 *We say that a distribution D is "locally confluent" when $E \twoheadleftarrow_P^1 D \twoheadrightarrow_P^1 F$ implies that there exists C such that $E \twoheadrightarrow^* C \twoheadleftarrow^* F$.*

(Note that strong normalisation over Dirac distributions implies it for all distributions, and likewise for local confluence.)

Lemma 4.16 (Newman's) *If a PARS is locally confluent and strongly normalising, then it is confluent.*

Proof.
We shall prove, by well-founded induction over $\twoheadrightarrow_P^1$, that every distribution is confluent. For a particular distribution, it suffices to show that that any peak of proper evolutions can be closed by $\twoheadrightarrow^*$. Then, by Corollary 4.8 (and since $\twoheadrightarrow_P^* = \twoheadrightarrow_P^{1*}$), confluence follows. We want to close a diagram of shape $E \twoheadleftarrow_P^{1*} D \twoheadrightarrow_P^{1*} F$. If either of the branches is zero steps long, then we trivially conclude. If not, we can form the diagram on the right, completing the proof by local confluence and the induction hypotheses for E' and F'. □



5 Limit distributions

In classical abstract rewriting, an element can either be non-normalising, weakly normalising or strongly normalising (corresponding to the situations where it *will not*, *may*, and *will* normalise, respectively). In probabilistic rewriting, the story is not as simple. Consider the following PARS, where b is a terminal element.

$$a \mapsto [(1/2, a), (1/2, b)]$$

Is a normalising? One could say "no" since, indeed, it does not have a finite maximal computation tree, as there is always some probability for the system to be in the non-terminal a state. However, such a probability will be made arbitrarily small by taking sufficient steps, and the distribution $[(1, b)]$ is reached *in the limit*. In this case a is called *almost surely terminating* [3]. Certainly, a desirable fact is that almost-surely-terminating elements have a unique final distribution. We will prove that distribution confluence guarantees such unicity.

We first introduce a notion of distance between *mathematical distributions*, i.e. normalised functions of type $A \to [0, 1]$[7]. We note with $[\![D]\!]$ the mathematical distribution obtained from the list distribution D (with the expected definition). We also extend definitions over mathematical distributions to list distributions by applying $[\![-]\!]$ where appropriate.

Definition 5.1 *Given D, E mathematical distributions, we define the distance between them as $d(D, E) = \sum_{a \in A} |D(a) - E(a)|$.*

Definition 5.2 (Limit of a sequence) *Given an infinite sequence of mathematical distributions $D_0, D_1, \ldots$ we say L is a limit for the sequence if for every $\varepsilon > 0$,*

[7] We move away from list distributions to allow for infinitely supported distributions on the limit.

there exists $N > 0$ such that for any $i \geq N$, $d(D_i, L) < \varepsilon$.

Note that this distance is the L_1 distance and the definition of limit is the usual one for metric spaces. It is then well known that there is at most one limit for a given sequence. We are interested in limits composed of terminal elements, representing a distribution of values. For that, the following definition is useful.

Definition 5.3 *For a mathematical distribution D, we define its "liveness" as the sum of weights for non-terminal elements. That is,* $\mathrm{Liv}(D) = \sum_{a \in \mathrm{dom}(\mapsto)} D(a)$

Note that the liveness of a list distribution cannot increase by evolution, and that $\mathrm{Liv}(D) = 0$ iff D is terminal. Moreover, since the normalised part of a distribution cannot evolve, liveness provides an upper bound on the possible distance to be attained by evolution, as the following lemma states.

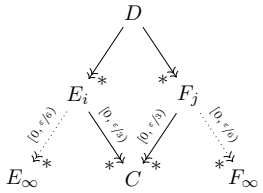
Lemma 5.4 *If $D \twoheadrightarrow^* E$, then $d(D, E) \leq 2 \cdot \mathrm{Liv}(D)$.*

Proof. By Lemma 4.4 there exist D' and E' such that $D \approx D' \twoheadrightarrow_P^* E' \approx E$. Because of the equivalences, it suffices to show the result for D' and E'. Assume, without loss of generality, that $D' = D_l + D_t$, where all elements of D_l are not terminal, and all those of D_t are. Since parallel evolution is local and terminal elements cannot evolve, we know that $E' = E'' + D_t$ for some E''. Then, $d(D', E')$ is simply $d(D_l, E'')$. Note that $\mathrm{Liv}(D')$ is the weight of D_l and of E''. Since distance is bounded by total weight, it follows that it is at most $2 \cdot \mathrm{Liv}(D') = 2 \cdot \mathrm{Liv}(D)$. □

Now, we can extend our notion of unicity of terminal distributions to limit distributions of terminal elements, accounting for an infinite sequence of reduction steps. We call this property *unicity of limit distributions* (ULD) and show it to be a consequence of distribution confluence.

Lemma 5.5 *If $\mathcal{A}$ is confluent, and a distribution D is the root of two infinite $\twoheadrightarrow$-chains E_i and F_j with respective limits E_∞ and F_∞ terminal distributions, then $E_\infty = F_\infty$.*

Proof.
Take $\varepsilon > 0$. By the definition of limit, we know there are i, j such that $d(E_i, E_\infty) < \varepsilon/6$ and $d(F_j, F_\infty) < \varepsilon/6$. Since E_∞ and F_∞ are terminal, $\mathrm{Liv}(E_i)$ and $\mathrm{Liv}(F_j)$ must be less than $\varepsilon/6$. The distributions E_i and F_j are reachable by a finite amount of $\twoheadrightarrow$ steps, so by confluence there exists a distribution C such that $E_i \twoheadrightarrow^* C \twoheadleftarrow^* F_j$. From Lemma 5.4, we get that $d(E_i, C) < \varepsilon/3$ and likewise for F_j. From these four bounds and the triangle inequality we get that $d(E_\infty, F_\infty) < \varepsilon$. Since this is the case for any positive ε, this distance must be exactly 0, and therefore $E_\infty = F_\infty$. □



$$\frac{}{(\lambda x.M)N \mapsto [(1, M[N/x])]}\ \text{R-}\beta \qquad \frac{}{(\lambda !x.M)!N \mapsto [(1, M[N/x])]}\ \text{R-}\beta!$$

$$\frac{}{M \oplus_p N \mapsto [(p, M), (1-p, N)]}\ \text{R-}\oplus$$

$$\frac{M \mapsto [(p_i, M_i)]_i}{M \oplus_p N \mapsto [(p_i, M_i \oplus_p N)]_i}\ \text{R-}\oplus\text{-L} \qquad \frac{N \mapsto [(p_i, N_i)]_i}{M \oplus_p N \mapsto [(p_i, M \oplus_p N_i)]_i}\ \text{R-}\oplus\text{-R}$$

$$\frac{M \mapsto [(p_i, M_i)]_i}{MN \mapsto [(p_i, M_i N)]_i}\ \text{R-AppL} \qquad \frac{N \mapsto [(p_i, N_i)]_i}{MN \mapsto [(p_i, MN_i)]_i}\ \text{R-AppR}$$

$$\frac{M \mapsto [(p_i, M_i)]_i}{\lambda x.M \mapsto [(p_i, \lambda x.M_i)]_i}\ \text{R-}\lambda \qquad \frac{M \mapsto [(p_i, M_i)]_i}{\lambda !x.M \mapsto [(p_i, \lambda !x.M_i)]_i}\ \text{R-}\lambda!$$

Fig. 2. Full semantics for λ_1

6 Case studies

6.1 A linear probabilistic λ-calculus: λ_1

In our introductory example, we used the term $(\lambda x.(x, x))\ \square$ as an example of a non-confluent computation. The tension apparently arises between "binding the result" (CBV) and "binding the computation" (CBN), which makes a difference in the probabilistic case when the binding is duplicated (as already pointed out in [10]). There seem to be three ingredients needed for this failure of confluence of a term $(\lambda x.M)N$: (1) x appears free more than once in M (2) N has a non-Dirac terminal distribution (3) both call-by-name and call-by-value reductions are possible.

In this section we define a probabilistic λ-calculus, dubbed λ_1, that prevents the combination of these three features by providing two kinds of abstractions, one restricting duplication and one restricting evaluation order [8]. We show λ_1 to be confluent (by a diamond property), giving evidence that little more than linearity of probabilistic arguments is required to achieve a confluent probabilistic programming language.

The calculus is heavily based on the one defined in [24]. The set of *pre*-terms is given by the following grammar

$$M, N ::= x \mid MN \mid \lambda x.M \mid \lambda !x.M \mid !M \mid M \oplus_p N$$

where the main novelty is the probabilistic choice operator $\oplus_p$, for any real number p in the open interval $(0, 1)$. Abstractions (λ) are affine, that is, in the scope of λx, there can be at most one free occurrence of x. Non-linear abstractions ($\lambda!$) have no such restriction. Affinity is enforced by a well-formedness judgment, whose definition is straightforward and which we thus omit. We work only with well-formed pre-terms, which form the set of terms.

The operational semantics is provided as a PARS in Fig. 2. A non-linear abstraction can only β-reduce with an argument of the form $!N$. Such terms cannot reduce, and are called *thunks*. This effectively implies that non-linear abstractions follow a fixed strategy (which is, morally, CBV until the argument is reduced to a thunk and CBN afterwards) [9].

[8] For more expressivity, the calculus could include another abstraction without neither restriction, but forbidding probabilistic arguments. We do not deem this as interesting for the scope of this paper.

[9] We are thus adopting "surface reduction" only since allowing for "internal reduction" [24] would give rise

To prove the diamond property for λ_1, we first need two substitution lemmas. When $D = [(p_i, a_i)]_i$, we write $D[M/x]$ for the distribution $[(p_i, a_i[M/x])]_i$. Similarly, $M[D/x]$ denotes $[(p_i, M[a_i/x])]_i$.

Lemma 6.1 *If $M \mapsto D$, then $M[N/x] \mapsto D[N/x]$.*

Proof. By induction on $M \mapsto D$. □

Lemma 6.2 *If $M \mapsto D$, and x is linear in N, then $N[M/x] \mapsto N[D/x]$.*

Proof. By induction on the well-formedness of N. □

Lemmas 6.1 and 6.2 are analogous to both statements of [24, Lemma 3.1]. Armed with both, we can prove the following theorem, which implies the diamond property.

Theorem 6.3 *If $D \leftarrow\!\!\shortmid M \mapsto E$ then there exist C, C' such that $D \twoheadrightarrow_P C$ and $E \twoheadrightarrow_P C'$ with $C \approx C'$.*

Proof. By induction on the shape of $M \mapsto D$ and $M \mapsto E$. □

By Corollary 4.10 we conclude that the calculus λ_1 is confluent, and thus enjoys both UTD and ULD.

6.2 Q^*

The Q^* calculus [9] is a quantum programming language with quantum measurement, an inherently probabilistic operation. Reduction occurs between *configurations*, which are terms coupled with a quantum state, and which we will not detail further. Its semantics does not fix a strategy and, as λ_1, is also based on [24]. Reduction steps are paired with a *label* indicating which type of reduction occurred (e.g. which qubit was measured). In Q^*, terms are linear and not affine: variables representing quantum data cannot be duplicated nor discarded, as per the no-cloning [26] and no-erasure [19] properties of quantum physics.

The authors prove a property called *strong confluence* which asserts that any two maximal (possibly infinite) computation trees with a common root have an equivalent support when restricted to normal forms, and, further, that any normal form appears in an equal amount of leaves on both trees.

To prove such property, the authors prove a crucial lemma called *quasi-one-step confluence*, which is morally a diamond property but with slightly different behaviours according to the reductions taken. The reductions are distinguished between two sets $\mathcal{N}$, $\mathcal{K}$ and those of the form $\mathbf{meas}_r$ (measurements). We will not describe these sets nor Q^*'s semantics (its full description can be found in [9]), and will only state the lemma about its reductions. The notation $C \to^p_\alpha D$ means "C reduces to D with probability p via the label α"; and $C \to^p_{\mathcal{N}} D$ means $C \to^p_\alpha D$ for some $\alpha \in \mathcal{N}$ (idem $\mathcal{K}$).

Lemma 6.4 (Quasi-one-step Confluence for Q^* [8, Proposition 4])
Let C, D, E be configurations and $C \to^p_\alpha D$, $C \to^s_\beta E$, then:

to the same non-confluence.

(i) *If* $\alpha \in \mathcal{K}$ *and* $\beta \in \mathcal{K}$*, then either* $D = E$ *or there is* F *with* $D \to^1_{\mathcal{K}} F$ *and* $E \to^1_{\mathcal{K}} F$*.*

(ii) *If* $\alpha \in \mathcal{K}$ *and* $\beta \in \mathcal{N}$*, then either* $D \to^1_{\mathcal{N}} E$ *or there is* F *with* $D \to^1_{\mathcal{N}} F$ *and* $E \to^1_{\mathcal{K}} F$*.*

(iii) *If* $\alpha \in \mathcal{K}$ *and* $\beta = \boldsymbol{meas}_r$*, then there is* F *with* $D \to^s_{\boldsymbol{meas}_r} F$ *and* $E \to^1_{\mathcal{K}} F$*.*

(iv) *If* $\alpha \in \mathcal{N}$ *and* $\beta \in \mathcal{N}$*, then either* $D = E$ *or there is* F *with* $D \to^1_{\mathcal{N}} F$ *and* $E \to^1_{\mathcal{N}} F$*.*

(v) *If* $\alpha \in \mathcal{N}$ *and* $\beta = \boldsymbol{meas}_r$*, then there is* F *with* $D \to^s_{\boldsymbol{meas}_r} F$ *and* $E \to^1_{\mathcal{N}} F$*.*

(vi) *If* $\alpha = \boldsymbol{meas}_r$ *and* $\beta = \boldsymbol{meas}_q$ *(with* $r \neq q$*), then there are* $t, u \in [0, 1]$ *and an* F *such that* $pt = su$*,* $D \to^t_{\boldsymbol{meas}_q} F$ *and* $E \to^u_{\boldsymbol{meas}_r} F$*.*

From this lemma, the fact that there are no infinite $\mathcal{K}$ sequences, and a "probabilistic strip lemma", the authors prove strong confluence [9, Theorem 5.4].

For distribution confluence, a simpler proof can be obtained. After modelling Q^* as a PARS (without labeled reductions, but sets of distributions instead) we can readily reinterpret Lemma 6.4 to prove the diamond property for it (by Corollary 4.10). From this result, distribution confluence follows, and therefore also unicity of both terminal and limit distributions. Notably, neither the normalisation requirement for $\mathcal{K}$ nor the "probabilistic strip lemma" are needed for this fact.

Our obtained distribution confluence is similar, but neither weaker nor stronger than strong confluence. It is not weaker as distribution confluence guarantees that divergences of computations without any normal form can be joined, which strong confluence does not. It is also not stronger as it implies nothing of limit distributions that are not terminal, while it follows from strong confluence that they must coincide in their normalized part. It also does not imply the equality between the amount of leaves on each tree [10].

7 Conclusions

We have studied the problem of showing that an operational semantics for a probabilistic language is not affected by the choice of strategy. For this purpose, we provided a definition of confluence for probabilistic systems by defining a classical relation over distributions. We showed our property of distribution confluence to be appropriate as, in particular, it implies a unicity of terminal distributions, both for finite and infinite reductions, and gives an equational consistency guarantee.

We believe this development demonstrates that distribution confluence provides a reasonable "sweet spot" for proving the correctness of probabilistic semantics, as it provides the expected guarantees about execution while allowing tractable proofs. Concretely, the provided proofs for λ_1 and Q^* are in line with what one would expect for linear calculi.

The proof about Q^* also partially answers the conjecture posed in [9, Section 8] ("any rewriting system enjoying properties like Proposition 4 [our Lemma 6.4]

[10] If needed, this can be recovered by removing Split and Join from the definition of $\sim$ and deriving criteria analogous to those of Section 4. One can then conclude that there is not only the same amount but that they are the same sequence of leaves reordered.

enjoys confluence in the same sense as the one used here") positively. The answer is partial since distribution confluence is not strictly equivalent.

Looking ahead, there are several interesting directions to explore. First, a study of confluence dealing with *terms* (and not just abstract elements) should provide more insights applicable to concrete languages, and we expect concepts such as orthogonality to have probabilistic analogues. As a generalization, it seems possible to take distribution weights from any mathematical field and not only the positive reals: even if interpreting such systems is not obvious, it seems most of our results would hold. Finally, a quantitative notion of confluence could also be explored, where a distribution is considered confluent if any divergence of it can be joined "up to ε"; in particular, obtaining useful simplified criteria for said property seems difficult.

7.1 Related work

In [7], similar definitions of rewriting of distributions and confluence are introduced. A key difference is that, in that work, equivalent distributions are identified and there is no partial evolution. In particular, this implies that the relation is not compositional, introducing a very subtle error in Lemma 10 (which, basically, states that a diamond property on Dirac distributions implies that of the whole system). The error is not severe for their development, but highlights the non-triviality of the matter.

In [12], a notion of confluence is defined and proven for an extension of λ_q [25] (a quantum λ-calculus) with measurements (thus introducing probabilistic behaviour). The proposed confluence is basically a confluence on computation trees, and the one we study in this paper is strictly weaker, yet sufficient for UTD and consistency.

In [9], already amply discussed, the notion of confluence introduced is a strong confluence over maximal trees (either finite or infinite) which is related, but neither weaker nor stronger than distribution confluence.

Finally, in [5], a property of confluence is defined and studied over probabilistic rewriting systems which do not contain any non-determinism (i.e. where $\mapsto$ is a partial function). This is a very different notion of confluence, speaking about punctual final results instead of distributions (indeed, distribution confluence trivially holds as there is no non-determinism).

References

[1] S. Andova. Process algebra with probabilistic choice. In J.-P. Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 111–129, 1999.

[2] J. Borgström, U. Dal Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 33–46, New York, NY, USA, 2016. ACM.

[3] O. Bournez and F. Garnier. Proving positive almost-sure termination. In J. Giesl, editor, *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 2005.

[4] O. Bournez and M. Hoyrup. Rewriting logic and probabilities. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 61–75, 2003.

[5] O. Bournez and C. Kirchner. Probabilistic Rewrite Strategies. Applications to ELAN. In S. Tison, editor, *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2002.

[6] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[7] U. Dal Lago, C. Faggian, B. Valiron, and A. Yoshimizu. The geometry of parallelism: Classical, probabilistic, and quantum effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 833–845, New York, NY, USA, 2017. ACM.

[8] U. Dal Lago, A. Masini, and M. Zorzi. Confluence results for a quantum lambda calculus with measurements. arXiv:0905.4567. Extended version of [9].

[9] U. Dal Lago, A. Masini, and M. Zorzi. Confluence results for a quantum lambda calculus with measurements. In B. Coecke, P. Panangaden, and P. Selinger, editors, *Proceedings of the 6th International Workshop on Quantum Physics and Logic (QPL'09)*, volume 270.2 of *Electronic Notes in Theoretical Computer Science*, pages 251–261. Elsevier, 2011.

[10] U. Dal Lago and M. Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO Theoretical Informatics and Applications*, 46(3):413–450, 2012.

[11] A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic λ-calculus and quantitative program analysis. *Journal of Logic and Computation*, 15(2):159–179, 2005.

[12] A. Díaz-Caro, P. Arrighi, M. Gadella, and J. Grattage. Measurements and confluence in quantum lambda calculi with explicit qubits. In B. Coecke, I. Mackie, P. Panangaden, and P. Selinger, editors, *Proceedings of the Joint 5th International Workshop on Quantum Physics and Logic and 4th Workshop on Developments in Computational Models (QPL/DCM'08)*, volume 270.1 of *Electronic Notes in Theoretical Computer Science*, pages 59–74. Elsevier, 2011.

[13] O. M. Herescu and C. Palamidessi. Probabilistic asynchronous π-calculus. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume 1784 of *Lecture Notes in Computer Science*, pages 146–160, 2000.

[14] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[15] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 83–92. ACM, 1984.

[16] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[17] G. Martínez. Confluencia en sistemas de reescritura probabilista. Master's thesis, Universidad Nacional de Rosario, Argentina, Mar. 27, 2017. Available at https://dcc.fceia.unr.edu.ar/es/lcc/tesinas-grado/tesinas/confluencia-en-sistemas-de-reescritura-probabilista.

[18] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[19] A. K. Pati and S. L. Braunstein. Impossibility of deleting an unknown quantum state. *Nature*, 404:164–165, 2000.

[20] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

[21] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 154–165, New York, NY, USA, 2002. ACM.

[22] D. S. Scott. Stochastic λ-calculi: An extended abstract. *Journal of Applied Logic*, 12(3):369–376, 2014.

[23] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In P. Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 354–368. Springer-Verlag, 2005.

[24] A. Simpson. Reduction in a linear lambda-calculus with applications to operational semantics. In J. Giesl, editor, *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 2005.

[25] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33:1109–1135, 2004.

[26] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982.

Machine-checked proof of the Church-Rosser theorem for the Lambda Calculus using the Barendregt Variable Convention in Constructive Type Theory

Ernesto Copello [1] Nora Szasz Álvaro Tasistro [2]

Universidad ORT Uruguay
Montevideo, Uruguay

Abstract

In this article we continue the work started in [3], deriving in Constructive Type Theory new induction principles for the λ-calculus, using (the historical) first order syntax with only one sort of names for both bound and free variables, and with α-conversion based upon name swapping. The principles provide a flexible framework for mimicking pen-and-paper proofs within the rigorous formal setting of a proof assistant. We here report on one successful application, namely a complete proof of the Church-Rosser theorem. The whole development has been machine-checked using the system Agda [5].

Keywords: Lambda Calculus, Formal Metatheory, Type Theory

1 Introduction

Let us consider the following definition of the Lambda Calculus terms:

$$M, N ::= x \mid MN \mid \lambda x.M$$

Fig. 1. Lambda Calculus syntax

This syntax has been considered too concrete a level on which to formally develop the metatheory of the calculus. The reason is that there is no significant difference between terms that differ only in the choice of the names of the bound variables, i.e., that are α-equivalent; and therefore it becomes natural to identify such terms already at the syntactic level. In the classical setting this amounts to working on α-equivalence classes of terms. One way to do this is to define functions and reason generally on terms by choosing adequate representatives of the classes in question.

[1] Partially supported by a scholarship granted by Agencia Nacional de Investigación e Innovación, Uruguay
[2] Email: copello,szasz,tasistro@ort.edu.uy

A prominent illustration of this practice is given in Barendregt's book [1], and the criterion followed to choose the representatives has thenceforth received as standard name *the Barendregt Variable Convention* (BVC).

In previous work [3], we have justified a form of the BVC in Constructive Type Theory by the following method: We introduced a recursion principle allowing to define *strongly α-compatible* functions on concrete terms, i.e. functions on the syntax introduced above that equate α-equivalent terms. This principle allows to specify the value of the function in the case of abstractions by considering only the case in which the bound variable does not belong to a given list of names. The principle is actually implemented by computing, for each given term, a canonical α-equivalent representative that satisfies such restriction. Then, the specified function actually operates on the computed representative. The α-conversion is implemented using the elementary operation of *name swapping* as in nominal techniques (see for instance [7] and [8]). Using this principle we have been able to define e.g. the substitution operation by considering, for abstractions, only the convenient case in which variable capture is avoided.

In addition, we also introduced a corresponding induction principle, which is presented in Figure 2. This principle requires the property being proved to be α-compatible, that is, preserved by α-conversion. Or, in other words, it must be a property of the abstract terms arising from the identification of α-equivalent concrete ones. As before, the induction principle allows one to prove the case of abstraction considering only the case in which the bound name does not belong to a given list of names:

$$\frac{\begin{array}{l} P\ \alpha\text{-compatible} \\ (\forall x)\ P(x) \\ (\forall M, N)\ (P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\exists xs, \forall M, \forall x \notin xs)(P(M) \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Fig. 2. Alpha induction principle

The principles referred to above were actually derived from the ordinary structural induction on concrete terms. As first applications, we developed, using the Agda proof assistant [5], some metatheoretical results concerning substitution that will be mentioned in the next section.

In this work we present strengthened α-induction principles on λ-terms that can be used to satisfactorily deal with relations whose definitions involve name swapping, and that allow to avoid finite lists of names in binding positions, not just in abstractions but in any term. In this way we can scale the metatheory of the λ-calculus up to the Church-Rosser theorem, formally reproducing a form of the BVC in the formalisation. The set of obtained principles provides a flexible framework quite able to pleasantly mimic pen-and-paper proofs within the rigorous formal setting of a proof assistant.

The work that stands closest to the present one is [10], where Urban and Norrish show how to emulate the BVC when performing induction on relations over λ-terms. They illustrate the use of the induction principles by proving the substitution lemma

for parallel reduction, and the weakening lemma of the typing relation. They present two induction principles on relations, one for the parallel reduction relation, and another one for the typing relation. When carrying out a proof by induction on these relations they are able to avoid a finite set of variable names as binders. To prove these strengthened induction principles they require that the relation definition rules satisfy the following preconditions: all functions and side conditions should be equivariant (i.e., preserved by name permutation), the side conditions must imply that all bound variables do not occur free in the conclusions, and all bound variables must be distinct. As a consequence, they have to modify the definition of the original relations to satisfy these preconditions in order to be able to prove the soundness of their induction principles.

All the definitions and proofs presented in the present article have been fully formalised in Constructive Type Theory [4] and machine-checked employing the system Agda [5]. The corresponding code is public, and it is available at:

https://github.com/ernius/formalmetatheory-nominal-Church-Rosser

In the subsequent text we give the proofs in English with a considerable level of detail so that they serve for making it clear how their formalisation was carried out.

The structure of the following sections is as follows: in Section 2 we recall the basic concepts of the λ-calculus, and some definitions and results from our previous work that are necessary for a better understanding of the material presented in this one. In Section 3 we introduce two new strengthened α-induction principles on λ-terms that will be useful to prove the main results of this work. Then, in Section 4 we present the notion of β-reduction and prove the Church-Rosser theorem by using the standard method due to Tait and Martin-Löf which involves the formulation and study of the parallel β-reduction. The overall conclusions are exposed in Section 5.

2 Preliminaries

Variables belong to a denumerable set of names, and terms are inductively defined as in Figure 1.

The *freshness* relation states that a variable does not occur free in a term:

Definition 2.1 [Freshness]

$$\frac{x \neq y}{x\#y} \qquad \frac{x\#M \qquad x\#N}{x\#MN} \qquad \frac{x\#M}{x\#\lambda y.M} \qquad x\#\lambda x.M$$

$x \notin_b M$ denotes that the variable x does not occur in a binding position in term M:

Definition 2.2 [$\notin_b$]

$$x \notin_b y \qquad \frac{x \notin_b M \qquad x \notin_b N}{x \notin_b MN} \qquad \frac{x \neq y \qquad x \notin_b M}{x \notin_b \lambda y.M}$$

Next comes the operation of *swapping* of names. A finite sequence (composition) of name swaps constitutes a finite *name permutation*, which is the renaming mechanism to be used on terms. The action of swapping is first defined on names themselves:

Definition 2.3 [Swapping]

$$(x\ y)\ z\ = \begin{cases} y \text{ if } & z = x \\ x \text{ if } & z = y \\ z \text{ if } & x \neq z \wedge y \neq z, \end{cases}$$

and then it is directly extended to terms by swapping all names occurring in a term, including abstraction positions.

The permutation operation is just defined as the sequential application of a list of swaps. We usually use π to denote permutations, and the application of a permutation π to a term M is written πM. $(x\ y)\pi$ denotes the permutation consisting of the swap $(x\ y)$ followed by the permutation π.

In the next definition we give a syntax-directed definition of α-conversion ($\sim_\alpha$) based on the swapping operation.

Definition 2.4 [Alpha equivalence relation]

$$(\sim_\alpha \mathrm{v})\ \frac{}{x \sim_\alpha x} \qquad (\sim_\alpha \mathrm{a})\ \frac{M \sim_\alpha M' \qquad N \sim_\alpha N'}{MN \sim_\alpha M'N'}$$

$$(\sim_\alpha \lambda)\ \frac{(\forall z \notin xs)\ \ (x\ z)M \sim_\alpha (y\ z)N}{\lambda x.M \sim_\alpha \lambda y.N}$$

In [3] we proved that this is an equivalence relation, preserved by the permutation operation (i.e., equivariant).

The substitution operation is defined using the α-compatible recursion principle mentioned in the previous section, and as a direct consequence of this, the following lemma is automatically derived.

Lemma 2.5 (α-compatibility of substitution)
$M \sim_\alpha M' \ \Rightarrow\ M[x{:=}N] = M'[x{:=}N]$.

The following results are successfully proved using the induction principles of [3].

Lemma 2.6 (Substitution preserves α-conversion)
$N \sim_\alpha N' \ \Rightarrow\ M[x{:=}N] \sim_\alpha M[x{:=}N']$.

Lemma 2.7 (Substitution under permutation)
$\pi\ (M[x{:=}N]) \sim_\alpha (\pi\ M)[(\pi\ x){:=}(\pi\ N)]$.

The next lemma shows that substitution commutes with abstraction up to α-conversion. This is so because the hypotheses ensure a fresh enough binder.

Lemma 2.8 (Substitution commutes with abstraction)
$x \neq y \wedge x\#N \;\Rightarrow\; (\lambda x.M)[y{:=}N] \sim_\alpha \lambda x.(M[y{:=}N])$.

Lemma 2.9 (Substitution composition)
$x \neq y \wedge x\#P \;\Rightarrow\; M[x{:=}N][y{:=}P] \sim_\alpha M[y{:=}P][x{:=}N[y{:=}P]]$.

The following result was not proved in our previous work, so we exhibit it in detail.

Lemma 2.10 (Swapping substitution variable)
$x\#M \;\Rightarrow\; ((x\ y)M)[x{:=}N] \sim_\alpha M[y{:=}N]$.

Proof. We use our α-induction principle in Figure 2. First, for arbitrary names x, y and term N we consider the following predicate on terms:

$$\Pi(M) \equiv\; x\#M \;\Rightarrow\; (x\ y)M[x{:=}N] \sim_\alpha M[y{:=}N].$$

We have to prove that Π is α-compatible, that is, if $M{\sim_\alpha}P$ and $\Pi(M)$, then $\Pi(P)$. Assume $\Pi(M)$ and $x\#P$. Then as freshness is preserved through $\sim_\alpha$, we have that $x\#M$. Then we proceed as follows:

$$\begin{array}{lcl} ((x\ y)P)[x{:=}N] & = & \{\sim_\alpha \text{ equivariance and Lemma 2.5}\} \\ ((x\ y)M)[x{:=}N] & \sim_\alpha & \{\Pi(M) \text{ and } x\#M\} \\ M[y{:=}N] & = & \{\text{Lemma 2.5}\} \\ P[y{:=}N]. & & \end{array}$$

Now we can proceed to the induction proper. We show the interesting case, namely the one of abstractions: We have $x\#\lambda z.M'$, where we choose $z \notin \{x, y\} \cup \mathrm{fv}(N)$. We need to prove $((x\ y)(\lambda z.M'))[x{:=}N] \sim_\alpha (\lambda z.M')[y{:=}N]$. As $x\#\lambda z.M'$ and $z \neq x$ we get $x\#M'$. Then we can reason as follows:

$$\begin{array}{lcl} ((x\ y)(\lambda z.M'))[x{:=}N] & = & \{\text{def. of swap}\} \\ (\lambda((x\ y)z).((x\ y)M'))[x{:=}N] & = & \{z \notin \{x, y\}\} \\ (\lambda z.((x\ y)M'))[x{:=}N] & \sim_\alpha & \{\text{Lemma 2.8}\} \\ \lambda z.(((x\ y)M')[x{:=}N]) & \sim_\alpha & \{\text{i.h.}\} \\ \lambda z.(M'[y{:=}N]) & \sim_\alpha & \{\text{Lemma 2.8}\} \\ (\lambda z.M')[y{:=}N] & & \end{array}$$

□

The preceding proof illustrates the usual pen-and-paper informal practice, which uses the BVC to assume binders fresh enough in some defined context, allowing us to apply substitution in a naive way without need of renaming.

The next result is a quite direct consequence of the previous lemma:

Lemma 2.11
$x\#\lambda y.M \;\Rightarrow\; ((x\ y)M)[x{:=}N] \sim_\alpha M[y{:=}N]$.

3 Alpha Induction Principles

In this section we introduce two new α-induction principles. The first one is presented in Figure 3.

It is a strengthened version of the one shown in Figure 2, where the induction

$$\frac{\begin{array}{l} P\ \alpha\text{-compatible} \\ (\forall x)\ P(x) \\ (\forall M, N)\ (P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\exists xs, \forall M, \forall x \notin xs)\ ((\forall \pi)\ P(\pi\ M) \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Fig. 3. Alpha induction principle with permutations

hypothesis of the abstraction case allows us to assume the property for all permutations of the body. This principle is useful to deal with relations which make use of the permutation operation in their definitions. We will show an example of this situation in the proof of Lemma 4.7 in the next section.

The α-induction principle with permutations shown in Figure 3 is proved using the one in Figure 4, which was derived in [3] from simple structural induction on λ-terms, in very much the same way as complete induction on natural numbers is derived from ordinary mathematical induction.

$$\frac{\begin{array}{l} (\forall x)\ P(x) \\ (\forall M, N)\ (P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\forall M, x)\ ((\forall \pi)\ P(\pi\ M)\ \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Fig. 4. Strong permutation induction principle

Proof. (Alpha induction principle with permutations).
The variable and application cases are direct. For the abstraction case, given any term M and variable x, we must prove $P(\lambda x.M)$ knowing:

$$(\forall \pi)\ P(\pi\ M) \tag{1a}$$
$$(\exists xs, \forall M', \forall y \notin xs)\ ((\forall \pi')\ P(\pi'\ M') \Rightarrow P(\lambda y.M')) \tag{1b}$$

Let xs be a list of names as in (1b). Let us further pick y not in xs and also fresh in $\lambda x.M$. Then for all M', π', $P(\pi'\ M') \Rightarrow P(\lambda y.M')$ holds. So taking $M' = (x\ y)M$ we have that $(\forall \pi')P(\pi'\ ((x\ y)M)) \Rightarrow P(\lambda y.(x\ y)M)$. Now, as $\pi'((x\ y)M) = ((x\ y)\pi')M$, we can use (1a) to get $P(\lambda y.(x\ y)M)$ from (1b), and finally $P(\lambda x.M)$ because P is α-compatible and $\lambda x.M \sim_\alpha \lambda y.(x\ y)M$. This last α-equivalence holds because we have chosen y fresh in $\lambda x.M$. □

The next induction principle (Figure 5) enables us to assume bound variables not in a given finite list of names xs through the entire induction, and not only for the abstraction case.

$$\frac{\begin{array}{l} P\ \alpha\text{-compatible} \\ (\forall x)\ P(x) \\ (\forall M, N)\ ((\forall y \in xs, y \notin_b MN) \wedge P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\forall M, x)\ ((\forall y \in xs, y \notin_b \lambda x.M) \wedge P(M) \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Fig. 5. Strengthened α-induction principle

Proof. (Strengthened α-induction principle).
To derive this principle we introduce a *rewrite* function such that, given a list of names xs and a term M, $\mathit{rewrite}(xs, M)$ returns a term α-convertible with M that does not contain any element of xs as binder.
To prove $P(M)$ for any term M, we proceed as follows. Given a list of names xs, an α-compatible predicate P, and the following hypotheses:

$$\begin{array}{l}(\forall x)\ P(x)\\ (\forall M, N)\ ((\forall y \in xs, y \notin_b MN) \wedge P(M) \wedge P(N) \Rightarrow P(MN))\\ (\forall M, x)\ ((\forall y \in xs, y \notin_b \lambda x.M) \wedge P(M) \Rightarrow P(\lambda x.M))\end{array} \tag{2}$$

we prove the following predicate Π by structural induction on terms:

$$\Pi(M) = ((\forall x \in xs)\ \Rightarrow x \notin_b M) \Rightarrow P(M) \tag{3}$$

Then, we use this predicate Π on the term $\mathit{rewrite}(xs, M)$, which has no bound variables in xs to obtain $P(\mathit{rewrite}(xs, M))$. Finally, as P is α-compatible and $\mathit{rewrite}(xs, M) \sim_\alpha M$ we get that $P(M)$ holds for any M.

In turn, the proof of $\Pi(M)$ by structural induction on M is straightforward because of the syntax directed definition of $\notin_b$:

- Variable case: Direct.
- Application case: We need to prove $\Pi(MN)$ for any M, N, such that $\Pi(M)$ and $\Pi(N)$ hold. That is, we have to prove $P(MN)$, given that any variable x in xs satisfies that $x \notin_b MN$. Then, by the syntax directed definition of $\notin_b$, we directly have that $x \notin_b M$ and $x \notin_b N$, and so we are able to use the induction hypothesis on M and N to get $P(M)$ and $P(N)$. So, we have all the premises in the second assertion in (2) hold, and hence its conclusion $P(MN)$.
- Abstraction case: We must prove $\Pi(\lambda y.M)$, that is, we need to prove $P(\lambda y.M)$ knowing that every variable x in xs satisfies that $x \notin_b \lambda y.M$. By the definition of $\notin_b$, we have that $x \neq y$ and $x \notin_b M$. We can apply the last result to the induction hypothesis $\Pi(M)$ to get $P(M)$. Finally, we get the desired result using the third assertion in (2).

□

4 Parallel Beta Reduction

The β-reduction relation ($\rightarrow_\beta$) is defined as the compatible (with the syntactic constructors) closure of the β-contraction $(\lambda x.M)N \rhd_\beta M[x{:=}N]$. The classical proof of confluence of β-reduction by Tait and Martin-Löf rests upon the property of confluence of the so-called parallel reduction, which can apply several β-contractions "in parallel" in one single step. We present our definition in Figure 6.

The first three rules have the same form as the ones defining the α-conversion relation presented in Definition 2.4, which evidences that we want this parallel reduction to be compatible with α-conversion, that is, if $M \rightrightarrows N$, $M \sim_\alpha M'$ and $N \sim_\alpha N'$ then $M' \rightrightarrows N'$. We will prove this property in lemmas 4.5 and 4.6. Finally, note that the β-rule has an extra premise involving α-conversion. The

$$(\rightrightarrows \mathrm{v})\ \frac{}{x \rightrightarrows x} \qquad (\rightrightarrows \mathrm{a})\ \frac{M \rightrightarrows M' \qquad N \rightrightarrows N'}{MN \rightrightarrows M'N'}$$

$$(\rightrightarrows \lambda)\ \frac{(\exists xs, \forall z \notin xs)\ (x\ z)M \rightrightarrows (y\ z)N}{\lambda x.M \rightrightarrows \lambda y.N}$$

$$(\rightrightarrows \beta)\ \frac{\lambda x.M \rightrightarrows \lambda y.P' \qquad N \rightrightarrows P'' \qquad P'[y{:=}P''] \sim_\alpha P}{(\lambda x.M)N \rightrightarrows P}$$

Fig. 6. Parallel reduction relation

reason for this is that our substitution operation modifies the bound names in terms as a consequence of being defined with our α-recursion principle. Without that additional premise we would not be able to prove that $\rightrightarrows$ is α-compatible on its right hand side.

We start by proving some basic properties:

Lemma 4.1 (Reflexivity of $\rightrightarrows$)
$M \rightrightarrows M$.

Proof. Direct application of the permutation induction principle in Figure 4. □

Lemma 4.2 (Equivariance of $\rightrightarrows$)
$M \rightrightarrows N \ \Rightarrow\ \pi M \rightrightarrows \pi N$.

Proof. By induction on the definition of $\rightrightarrows$.
The variable and application cases are direct. In the abstraction case, we have to prove $\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ N)$ from the premise of the rule $(\rightrightarrows\lambda)$ and the corresponding induction hypothesis. We can in addition exclude the variable z mentioned in the premise from the domain of the permutation π and reason as follows:

$$\begin{array}{ll}
(x\ z)M \rightrightarrows (y\ z)N & \Rightarrow\ \{\text{i.h.}\} \\
\pi((x\ z)M) \rightrightarrows \pi((y\ z)N) & \Rightarrow\ \{\text{def. of perm.}\} \\
((\pi\, x)\ (\pi\, z))(\pi\, M) \rightrightarrows ((\pi\, y)\ (\pi\, z))(\pi\, N) & \Rightarrow\ \{\text{as } z \notin \mathrm{dom}(\pi) \text{ then } (\pi\, z) = z\} \\
((\pi\ x)\ z)(\pi\ M) \rightrightarrows ((\pi\ y)\ z)(\pi\ N) & \Rightarrow\ \{(\rightrightarrows \lambda)\ \text{ rule}\} \\
\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ N). &
\end{array}$$

In the $(\rightrightarrows \beta)$ case we must prove $(\lambda(\pi\ x).(\pi\ M))(\pi\ N) \rightrightarrows \pi\ P$ from premises $\lambda x.M \rightrightarrows \lambda y.P'$, $N \rightrightarrows P''$ and $P \sim_\alpha P'[y{:=}P'']$. By direct application of the induction hypotheses corresponding to the first two premises we get:

$$\begin{array}{c}
\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ P') \\
\text{and } \pi\ N \rightrightarrows \pi\ P''
\end{array} \tag{4}$$

Then, using the third premise we can reason as follows:

$$\begin{array}{ll}
P \sim_\alpha P'[y{:=}P''] & \Rightarrow\ \{\sim_\alpha \text{ equivariance}\} \\
\pi\ P \sim_\alpha \pi\ (P'[y{:=}P'']) & \Rightarrow\ \{\text{Lemma 2.7}\} \\
\pi\ P \sim_\alpha (\pi\ P')[(\pi\ y){:=}(\pi\ P'')] & \Rightarrow\ \{\sim_\alpha \text{ symmetry}\} \\
(\pi\ P')[(\pi\ y){:=}(\pi\ P'')] \sim_\alpha \pi\ P &
\end{array}$$

We obtain the desired result using the $(\rightrightarrows\beta)$ rule with (4) and this last result as premises. □

As a direct consequence of the previous lemma we derive the following result:

Corollary 4.3 (Preservation of $\rightrightarrows$ under abstraction)
$M \rightrightarrows N \;\Rightarrow\; \lambda x.M \rightrightarrows \lambda x.N.$

The following lemmas state that our parallel reduction relation is preserved by α-equivalence. Both results are proved by easy inductions on the parallel reduction relation.

Lemma 4.4 (Right α-compatibility of $\rightrightarrows$)
$M \rightrightarrows N \wedge N \sim_\alpha P \;\Rightarrow\; M \rightrightarrows P.$

Lemma 4.5 (Left α-compatibility of $\rightrightarrows$)
$M \sim_\alpha N \wedge N \rightrightarrows P \;\Rightarrow\; M \rightrightarrows P.$

As $\rightrightarrows$ is reflexive, we can now prove in a direct manner that α-conversion is included in the parallel reduction.

Lemma 4.6
$\sim_\alpha \subseteq \rightrightarrows$.

Proof. Given $M \sim_\alpha N$, as $\rightrightarrows$ is reflexive by Lemma 4.1, we also know $M \rightrightarrows M$. Then using Lemma 4.4 we obtain the desired result.
□

As $\rightrightarrows$ basically applies β-contractions, no free variable should be introduced at any step, therefore freshness is preserved.

Lemma 4.7 ($\rightrightarrows$ preserves freshness)
$x\#M \wedge M \rightrightarrows N \;\Rightarrow\; x\#N.$

Proof. We use the α-induction principle with permutations (fig. 3) on the term M. In order to apply this principle we must prove, for any variable x, that the predicate

$$\Pi(M) \equiv \; (\forall N)\; (x\#M \wedge M \rightrightarrows N \;\Rightarrow\; x\#N).$$

is α-compatible, which follows from the α-compatibility of both freshness and parallel reduction. Now, for the main result, we only show the interesting abstraction case of the induction (i.e. for a term $\lambda y.M'$). We therefore have that $x\#\lambda y.M'$ and $\lambda y.M' \rightrightarrows \lambda z.N'$, and we must prove $x\#\lambda z.N'$. Now, $\lambda y.M' \rightrightarrows \lambda z.N'$ must be the result of an application of the $(\rightrightarrows\lambda)$ rule, so we get its premise $(\forall w \notin xs)\; (y\; w)M' \;\rightrightarrows\; (z\; w)N'$. The α-induction principle allows us to exclude some variables for the abstraction case, so we can also assume $y \neq x$. Using this inequality and the hypothesis $x\#\lambda y.M'$ we get by definition that $x\#M'$. Now let u be a variable such that $u\#N', u \notin xs$ and $u \neq x$; then $x\#(y\; u)M'$ because $x \neq y$, $x \neq u$ and $x\#M'$. We can apply the premise of the $(\rightrightarrows\lambda)$ rule with u, as $u \notin xs$, and we get $(y\; u)M' \rightrightarrows (z\; u)N'$. We use the induction hypothesis on M' and permutation $(y\; u)$ with the previous two results to get $x\; \#(z\; u)N'$. We also have that $\lambda u.(z\; u)N' \sim_\alpha \lambda z.N'$ because $u\; \#N'$. Then, as $\sim_\alpha$ preserves freshness, we get the desired result. □

We can now prove the following inversion lemmas, which state that the original definition of parallel reduction by Takahashi [9] (which we note $\rightrightarrows_T$ in the next definition) can be derived from ours. These lemmas will be useful in the proof of the diamond property of our relation $\rightrightarrows$.

$$\frac{}{x \rightrightarrows_T x} \qquad \frac{M \rightrightarrows_T M' \qquad N \rightrightarrows_T N'}{MN \rightrightarrows_T M'N'} \qquad \frac{M \rightrightarrows_T M'}{\lambda x.M \rightrightarrows_T \lambda x.M'}$$

$$\frac{M \rightrightarrows_T M' \qquad N \rightrightarrows_T N'}{(\lambda x.M)N \rightrightarrows_T M'[x{:=}N']}$$

Fig. 7. Takahashi's parallel reduction relation.

Lemma 4.8 ($\rightrightarrows$ λ-inversion)
$\lambda x.M \rightrightarrows M' \;\Rightarrow\; (\exists M'')\ (M \rightrightarrows M'' \;\wedge \lambda x.M \rightrightarrows \lambda x.M'' \wedge\; M' \sim_\alpha \lambda x.M'')$.

Proof. By definition of $\rightrightarrows$ it must be the case that $\lambda x.M \rightrightarrows M'$ is a result of an application of $(\rightrightarrows \lambda)$ rule; then we have that M' is in an abstraction $\lambda y.N$, and that there exists a list of variables xs such that $(\forall z \notin xs)\ (x\ z)M \rightrightarrows (y\ z)N$. We take $M'' = (x\ y)N$, and prove that M'' satisfies the three properties of the thesis.

- Let z be a variable such that $z \notin xs$ and $z\#\lambda y.M'$. By definition of $\#$, $x\#\lambda x.M$, and then, as parallel reduction preserves freshness, $x\#\lambda y.N$ also holds. So:

  $$\begin{array}{ll} (x\ z)M \rightrightarrows (y\ z)N & \Rightarrow \{\rightrightarrows \text{ equivariance}\} \\ (x\ z)(x\ z)M \rightrightarrows (x\ z)(y\ z)N & \Rightarrow \{\text{swap self inverse}\} \\ M \rightrightarrows (x\ z)(y\ z)N & \Rightarrow \{(*)\} \\ M \rightrightarrows (x\ y)N & \end{array}$$

  (*) Here we apply Lemma 4.4 with the premise $(x\ z)(y\ z)N \sim_\alpha (x\ y)N$. This swapping cancellation property requires z and x to be fresh enough, as it is the case.

- We apply Lemma 4.4 with $\lambda x.M \rightrightarrows \lambda y.N$ and the α-equivalence obtained above to prove $\lambda x.M \rightrightarrows \lambda x.(x\ y)N$.
- To prove $\lambda y.N \sim_\alpha \lambda x.(x\ y)N$, as x is fresh in $\lambda y.N$, we swap y with x in this term to get the α-equivalent term $\lambda x.(x\ y)N$ (Lemma 4.2).

□

Lemma 4.9 ($\rightrightarrows$ β-inversion)
If $(\lambda x.M)N \rightrightarrows P$ is obtained by application of the $(\rightrightarrows \beta)$ rule in the following way:

$$(\rightrightarrows\beta)\ \frac{\lambda x.M \rightrightarrows \lambda y.M' \qquad N \rightrightarrows N' \qquad M'[y{:=}N'] \sim_\alpha P}{(\lambda x.M)N \rightrightarrows P}$$

then, $(\exists M'')\ (\lambda x.M \rightrightarrows \lambda x.M'' \wedge\; M''[x{:=}N'] \sim_\alpha P)$.

Proof. We prove that $M'' = (y\ x)M'$ satisfies the thesis.

- $x\#\lambda x.M$ and $\lambda x.M \rightrightarrows \lambda y.M'$ so by Lemma 4.7 $x\#\lambda y.M'$. We can then swap y with x in the last term and obtain the α-equivalent term $\lambda x.(y\ x)M'$, using Lemma 4.2. Then, by Lemma 4.4 we get $\lambda x.M \rightrightarrows \lambda x.(y\ x)M'$.

- For the second condition we reason as follows:

$$\begin{array}{lll} ((y\ x)M')[x{:=}N'] & = & \{\text{swap commutativity}\} \\ ((x\ y)M'))[x{:=}N'] & \sim_\alpha & \{\text{corollary 2.11}\} \\ M'[y{:=}N'] & \sim_\alpha & \{\text{hypothesis}\} \\ P & & \end{array}$$

□

Theorem 4.10 ($\rightrightarrows$ substitution lemma)
$M \rightrightarrows M' \wedge N \rightrightarrows N' \ \Rightarrow\ M[x{:=}N] \rightrightarrows M'[x{:=}N']$.

The substitution lemma for parallel reduction is the crux of the Church-Rosser theorem, and the place in which our α-induction principles in [3] fail to capture the BVC. If we perform induction on the term M, the problem appears in the beta application case, specifically when the term is a redex. We then have $M = (\lambda y.P)Q$, and we need to prove $((\lambda y.P)Q)[x{:=}N] \rightrightarrows R[x{:=}N']$. But, as we are in the application case of the induction, the original α-induction principle gives no freshness information about the binder y. The use of the BVC would allow us to choose y different from x and fresh in N, and with those freshness conditions we could push the substitution inside the abstraction without any variable capture by the use of Lemma 2.8. We next use our strengthened α-induction principle presented in Figure 5 to prove this result.

Proof. Given terms N, N' such that $N \rightrightarrows N'$, and a variable x, we consider the following predicate on terms:

$$\Pi(M) \equiv\ (\forall M')\ (M \rightrightarrows M'\ \Rightarrow\ M[x{:=}N] \rightrightarrows M'[x{:=}N']).$$

Π is α-compatible, which is a direct consequence of both substitution and $\rightrightarrows$ being α-compatible (lemmas 2.5,4.4,4.5). Then we can use our strengthened α-induction principle to prove Π by induction on the term M, excluding the variable x, and the free variables in terms N and N' from the binders in M. We show the proof of the interesting application and abstraction cases.

- Application case: we prove $(\forall P, Q)\ ((\forall z \in \{x\} \cup \mathrm{fv}(N) \cup \mathrm{fv}(N'),\ z \notin_b P\ Q)\ \Pi(P) \wedge \Pi(Q) \Rightarrow \Pi(P\ Q))$. We have two subcases according to which rule is used to reduce the application $P\ Q$.
  - ($\rightrightarrows$a) rule subcase: we have that $P \rightrightarrows P'$ and $Q \rightrightarrows Q'$ and we need to prove that $(P\ Q)[x{:=}N] \rightrightarrows (P'\ Q')[x{:=}N']$. The proof is a direct application of the ($\rightrightarrows$a) rule to the induction hypotheses.
  - ($\rightrightarrows\beta$) rule subcase: given $(\lambda y.P)Q \rightrightarrows R$ we must prove $((\lambda y.P)Q)[x{:=}N] \rightrightarrows R[x{:=}N']$. We use the inversion Lemma 4.9 to obtain that there exists P'' such that $\lambda y.P \rightrightarrows \lambda y.P'' \wedge P''[y{:=}Q'] \sim_\alpha R$. Next, as we have assumed the binder y different from x and also fresh in N and N', we can reason as follows:

$$\begin{array}{lll} \lambda y.P \rightrightarrows \lambda y.P'' & \Rightarrow & \{\text{i.h.}\} \\ (\lambda y.P)[x{:=}N] \rightrightarrows (\lambda y.P'')[x{:=}N'] & \Rightarrow & \{\text{Lemma 2.8}\} \\ \lambda y.(P[x{:=}N]) \rightrightarrows \lambda y.(P''[x{:=}N']) & & \end{array}$$

By the induction hypothesis we know $Q[x{:=}N] \rightrightarrows Q'[x{:=}N']$. So if we prove:

$$P''[x{:=}N'][y{:=}Q'[x{:=}N']] \sim_\alpha R[x{:=}N'] \tag{5}$$

we will be able to apply the $(\rightrightarrows \beta)$ rule and get that $(\lambda y.(P[x{:=}N]))(Q[x{:=}N]) \rightrightarrows R[x{:=}N']$. Then, using the freshness premises, we can pull out the substitution operation on the left side of this parallel reduction, and using the Lemma 4.5, of α-compatibility of $\rightrightarrows$, we finally get the desired result.

It just remains to prove (5) to end the proof of this subcase. Again, here the classical informal proofs use the BVC. We can also mimic this practice in this case since our induction principle gives us a binder y distinct form x and fresh in N'. Then, we have the freshness premises to successfully apply the substitution composition Lemma 2.9 and conclude this proof in the following steps:

$$\begin{array}{lll} P''[x{:=}N'][y{:=}Q'[x{:=}N']] & \sim_\alpha & \{\text{Lemma 2.9}\} \\ P''[y{:=}Q'][x{:=}N'] & = & \{\text{Lemma 2.5 and } P''[y{:=}Q'] \sim_\alpha R\ \} \\ R[x{:=}N'] & & \end{array}$$

- Abstraction case: we have to prove $(\forall P, y)\ (\forall z \in \{x\} \cup \mathrm{fv}(N) \cup \mathrm{fv}(N'), z \notin_b \lambda y.P) \wedge \Pi(P) \Rightarrow \Pi(\lambda y.P)$. We apply the inversion Lemma 4.9 to the hypothesis $\lambda y.P \rightrightarrows Q$ to get that there exists Q' such that: $P \rightrightarrows Q'$, $\lambda y.P \rightrightarrows \lambda y.Q'$ and $Q \sim_\alpha \lambda y.Q'$. Then, we can conclude the proof in the following way:

$$\begin{array}{ll} P \rightrightarrows Q' & \Rightarrow \{\text{ind. hyp.}\} \\ P[x{:=}N] \rightrightarrows Q'[x{:=}N'] & \Rightarrow \{\rightrightarrows \text{ equivariance}\} \\ (y\ z)(P[x{:=}N]) \rightrightarrows (y\ z)(Q'[x{:=}N']) & \Rightarrow \{(\rightrightarrows\lambda) \text{ rule}\} \\ \lambda y.P[x{:=}N] \rightrightarrows \lambda y.Q'[x{:=}N'] & \Rightarrow \{\text{Lemma 2.8}\} \\ (\lambda y.P)[x{:=}N] \rightrightarrows (\lambda y.Q')[x{:=}N'] & \Rightarrow \{\text{Lemma 2.5 and } Q \sim_\alpha \lambda y.Q'\} \\ (\lambda y.P)[x{:=}N] \rightrightarrows Q[x{:=}N'] & \end{array}$$

□

In [10], the authors proceed by induction on the relation, so x, N, N' are universally quantified over the β-contraction rule definition, and they are forced to add the same freshness premises that we were able to assume in this proof –by the use of our strengthened α-induction principle– directly into the premises of their modified beta rule of the parallel relation. In contrast, we are performing induction on the term M to prove the predicate Π, and hence we are able to maintain those variables as a fixed context outside the definition of Π. Then by the use of our strengthened α-induction principle we are able to mimic the BVC also in the application case of the proof, specifically in the previously exposed $(\rightrightarrows\beta)$ rule subcase.

Finally, we can prove the diamond property of the parallel reduction. Instead of directly proving it by induction on terms (which can easily be done), we will follow the shorter method by Takahashi [9]. For this we first define the "star" operation (Figure 8), such that for any λ-term M, M^* is the result of contracting all the β-redexes existing in M simultaneously. Then we prove that for any terms M, N, if $M \rightrightarrows N$, then $N \rightrightarrows M^*$ (Lemma 4.11). Finally, the diamond property of $\rightrightarrows$ follows directly as a corollary of this result.

$$
\begin{array}{lcl}
x^* & = & x \\
(\lambda x.M)^* & = & \lambda x.M^* \\
(\quad x \quad M)^* & = & x\ M^* \\
((M_1 M_2)\ M_3)^* & = & (M_1 M_2)^* M_3^* \\
((\lambda x.M_1)\ M_2)^* & = & M_1^*[x := M_2^*]
\end{array}
$$

Fig. 8. Takahashi's star function

Lemma 4.11 (Star property)
$M \rightrightarrows N \ \Rightarrow\ N \rightrightarrows M^*$.

Proof. By structural induction on M. We show the interesting application and abstraction cases.

- Abstraction case: we have to prove that $N \rightrightarrows (\lambda x.M)^* = \lambda x.M^*$, knowing that $\lambda x.M \rightrightarrows N$ holds. We can use the inversion Lemma 4.8 on the latter to obtain the existence of the term N' such that: $N \sim_\alpha \lambda x.N'$ and $M \rightrightarrows N'$. We can now apply the induction hypothesis, and then corollary 4.3 to $M \rightrightarrows N'$, and obtain $\lambda x.N' \rightrightarrows \lambda x.(M^*)$. This last result directly gives us the desired result by Lemma 4.5 since we know that $N \sim_\alpha \lambda x.N'$.
- Application case: we have three subcases. The first two correspond to the third and fourth lines of the star operation definition, and are directly derived from the induction hypotheses.
  Finally, the redex case can be subdivided accordingly to which rule, ($\rightrightarrows$a) or ($\rightrightarrows\beta$), is used in the last step of its parallel reduction.
  · ($\rightrightarrows$a) rule subcase: we have that $\lambda x.M \rightrightarrows N$ and $M' \rightrightarrows N'$, and we need to prove that $NN' \rightrightarrows ((\lambda x.M)M')^* = M^*[x := M'^*]$. We begin applying the inversion lemma 4.8 to the hypothesis $\lambda x.M \rightrightarrows N$ to get that there exists N'' such that $N \sim_\alpha \lambda x.N''$ and $M \rightrightarrows N''$. We can now apply the induction hypothesis to the latter, and then the corollary 4.3 to conclude $\lambda x.N'' \rightrightarrows \lambda x.M^*$. Besides, we can also apply the induction hypothesis to the premise $M' \rightrightarrows N'$ to get $N' \rightrightarrows M'^*$. We can combine the last two inferred parallel reductions, using the ($\rightrightarrows\beta$) rule, and derive that $(\lambda x.N'')N' \rightrightarrows M^*[x := M'^*]$ holds. From this result we directly get the desired result just noticing that $N\,N' \sim_\alpha (\lambda x.N'')N'$, because $N \sim_\alpha \lambda x.N''$ and $N' \sim_\alpha N'$. Hence, by left α-compatibility of the parallel relation (Lemma 4.5) we finish this subproof case.
  · ($\rightrightarrows\beta$) rule subcase: we have the following hypotheses: $\lambda x.M \rightrightarrows \lambda y N$, $M' \rightrightarrows N'$ and $N[y := N'] \sim_\alpha P$, and we need to prove that $P \rightrightarrows M^*[x := M'^*]$ holds. We proceed analogously to the previous subcase and derive that there exists N'' such that $\lambda y.N \sim_\alpha \lambda x.N''$, $N'' \rightrightarrows M^*$ and $N' \rightrightarrows M'^*$. Then we apply the substitution lemma for $\rightrightarrows$ (Lemma 4.10) to obtain $N''[x := N'] \rightrightarrows M^*[x := M'^*]$. Finally, we can use left α-compatibility Lemma 4.5 to finish the proof if we prove that $P \sim_\alpha N''[x := N']$. Next we prove that this last alpha equivalence holds:

$$\begin{array}{lll} P & \sim_\alpha & \{\text{hypothesis}\} \\ N[y := N'] & \sim_\alpha & \{\text{by Lemma 2.11 as } x\#\lambda y.N\} \\ ((x\ y)N)[x := N'] & = & \{\text{by Lemma 2.5 as } (x\ y)N \sim_\alpha N''\} \\ N''[x := N'] & & \end{array}$$

In the previous derivation we used the freshness condition $x\#\lambda y.N$, which follows from $\lambda y.N \sim_\alpha \lambda x.N''$, $x\#\lambda x.N''$, and that freshness is preserved under α-conversion.

□

As a direct consequence of the previous lemma, we have the following result:

Lemma 4.12 (Diamond property of $\rightrightarrows$)
$M \rightrightarrows N \wedge M \rightrightarrows P \ \Rightarrow\ \exists Q, N \rightrightarrows Q \wedge P \rightrightarrows Q$

Definition 4.13 [Confluence] A relation is *confluent* if its reflexive and transitive closure has the diamond property.

We omit the proof details of the next results because they do no deal with λ-terms. They are proved in a direct way as in the classical literature.

Lemma 4.14 *If a relation R has the diamond property then it is confluent*

Lemma 4.15 *If a reduction relation R is confluent, then so is its reflexive and transitive closure R^*.*

As a direct application of the preceding two lemmas, we obtain:

Lemma 4.16 (Confluence of $\rightrightarrows$)
$\rightrightarrows^*$ is confluent.

If we now consider the β-reduction $\rightarrow_\beta$, we have:

Lemma 4.17
$(\rightarrow_\beta \cup \sim_\alpha)^* = \rightrightarrows^*$

Proof. We prove the double inclusion. To prove $(\rightarrow_\beta \cup \sim_\alpha)^* \subseteq \rightrightarrows^*$, it is enough to prove $(\rightarrow_\beta \cup \sim_\alpha) \subseteq \rightrightarrows$. By Lemma 4.6 we know $\sim_\alpha \subseteq \rightrightarrows$, and $\rightarrow_\beta \subseteq \rightrightarrows$ can be proved by a direct induction on the $\rightarrow_\beta$ reduction relation.
Finally, to prove $\rightrightarrows^* \subseteq (\rightarrow_\beta \cup \sim_\alpha)^*$ we first prove $\rightrightarrows \subseteq (\rightarrow_\beta \cup \sim_\alpha)^*$ by a direct induction on $\rightrightarrows$. Then, by monotonicity of * over $\subseteq$, we get $\rightrightarrows^* \subseteq ((\rightarrow_\beta \cup \sim_\alpha)^*)^*$, and the desired result follows from idempotence of *. □

Using the last two lemmas we finally arrive at the Church-Rosser theorem.

Theorem 4.18 (Church-Rosser)
The relation $(\rightarrow_\beta \cup \sim_\alpha)$ is confluent.

5 Conclusions

We have introduced principles of induction on terms of the Lambda Calculus that allow to reason on the abstract terms that arise by identifying α-equivalent concrete

ones. The principles work for α-compatible predicates, i.e. properties preserved by α-conversion, and allow to carry out the corresponding proofs by choosing convenient representatives of the abstract terms in question, namely by avoiding names in binding positions belonging to explicitly provided finite lists. We have derived these principles ultimately from the ordinary structural induction on (concrete) terms. Therefore we have provided a full justification of this form of the Barendregt Variable Convention (BVC). The whole work has been carried out in Constructive Type Theory —and machine-checked using the system Agda— without modifying the ordinary definitional equality of terms or formulating any kind of quotient construction. For the whole implementation to work it has proven essential to define α-conversion in terms of a fundamental operation of name swapping, as in Pitts and Gabbay's Nominal Techniques.

That the method of formalisation can be useful is maybe illustrated by our full formal proof in Agda of the Church-Rosser theorem. In this paper we have given a summarized description in English language of such proof, although showing the details we believe essential for the reconstruction of the completely formal version, which is publicly available at https://github.com/ernius/formalmetatheory-nominal-Church-Rosser.

In our development, the definition of the parallel reduction relation has to be formulated in such a way as to ensure that the relation is α-compatible. Because of this, it looks more concrete than the classical one, as presented by Barendregt [1] or Takahasi [9]. However, we are able to prove inversion lemmas that allow us to recover the original parallel reduction definition, and from them we are able to reproduce Takahashi's proof of the diamond property.

In a similar work, Urban and Norrish [10] also have to modify the parallel reduction relation in order to derive an ad-hoc induction principle on the parallel reduction to successfully prove the substitution lemma for this relation. However, they do not have to ensure the α-compatibility of the parallel relation because in their formalisation α-convertible terms are syntactically equal, since they work at the level of terms quotiented by α-equivalence, as explained in [6]. We believe our approach is more direct and general, since we derive an induction principle on simple terms, and not on the more complex relations over them. As shown in the beta subcase of the proof of the substitution theorem, we are able to derive the freshness conditions for the binders directly from our α-induction principle, as in the BVC, and not explicitly imposing them in the definition of the parallel reduction relation.

As another application of our method, we have also been able to (gently) formalise a proof of the Subject Reduction theorem for the system of assignment of simple types, which is also publicly available at the aforementioned site. Finally, we have generalised the techniques here exposed to a framework of regular trees with binders, thereby obtaining pleasant treatments of e.g. metatheory of the System F alongside that of the pure Lambda Calculus by way of instantiation. Report on this work can be found in the first author's upcoming PhD thesis [2].

References

[1] Hendrik Barendregt. *The λ-calculus Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, revised edition, 1984.

[2] Ernesto Copello. *On the Formalisation of the Metatheory of the Lambda Calculus and Languages with Binders*. PhD thesis, PEDECIBA Informática, Uruguay. Submitted for revision, June 2017.

[3] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the λ-calculus in constructive type theory. *Electronic Notes in Theoretical Computer Science*, 323:109 – 124, 2016.

[4] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[5] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[6] Michael Norrish. Mechanising λ-calculus using a classical first order theory of terms with permutations. *Higher-Order and Symbolic Computation*, 19(2):169–195, 2006.

[7] Andrew M. Pitts. Nominal Logic, a First 0rder Theory of Names and Binding. *Information and Computation*, 186(2):165–193, 2003.

[8] Andrew M. Pitts. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53(3):459–506, May 2006.

[9] M. Takahashi. Parallel Reductions in λ-Calculus. *Information and Computation*, 118(1):120 – 127, 1995.

[10] Christian Urban and Michael Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding*, MERLIN '05, pages 25–32, New York, NY, USA, 2005. ACM.

Counter-model Generation from Failed Proof Searches in Propositional Minimal Implicational Logic*

Jefferson de Barros Santos[1], Bruno Lopes Vieira[2], and Edward Hermann Haeusler[3]

[1] jefferson.santos@fgv.br FGV, Rio de Janeiro, Brazil
[2] bruno@ic.uff.br UFF, Niterói, Brazil
[3] hermann@inf.puc-rio.br PUC-Rio, Rio de Janeiro, Brazil

Abstract. This paper presents a new Termination and Completeness Theorems for the sequent calculus $\mathbf{LMT}^{\rightarrow}$ introduced in [15]. $\mathbf{LMT}^{\rightarrow}$ is aimed to be used for proof search in Propositional Minimal Implicational Logic ($\mathbf{M}^{\rightarrow}$), in a bottom-up approach. *Termination* of the calculus is guaranteed by a strategy of rule application that forces an ordered way to search for proofs such that all possible combinations are stressed. For an initial formula α, proofs in $\mathbf{LMT}^{\rightarrow}$ has an upper bound of $|\alpha| \cdot 2^{|\alpha|+1+2 \cdot log_2|\alpha|}$, which together with the system strategy ensure decidability. $\mathbf{LMT}^{\rightarrow}$ has the property to allow extractability of counter-models from failed proof searches (*bicompleteness*), i.e., the attempt proof tree of an expanded branch produces a Kripke model that falsifies the initial formula.

1 Introduction

Propositional Minimal Implicational Logic ($\mathbf{M}^{\rightarrow}$) is the fragment of the Propositional Minimal Logic containing only the logical connective $\rightarrow$.

In [15] we present the sequent calculus $\mathbf{LMT}^{\rightarrow}$ aimed to be used for bottom up proof search in $\mathbf{M}^{\rightarrow}$. Our main contribution here is a Termination Theorem for $\mathbf{LMT}^{\rightarrow}$ not covered in the original paper, and a new version of the Completeness Theorem initially showed there. $\mathbf{LMT}^{\rightarrow}$is based on a set of rules and in a general strategy for application of the rules in such a way that we can avoid the usage of *loop checkers* and mechanisms for *backtracking*. $\mathbf{LMT}^{\rightarrow}$ also avoids the necessity of working with different systems for provability and refutation, a very common approach to deal with this problem described in the literature. Counter-model generation (using Kripke semantics) is achieved as a consequence of the features of the system and the way the attempt proof tree (produced by a failed proof search) is constructed during a proof search process. We implemented $\mathbf{LMT}^{\rightarrow}$ as an interactive theorem prover in Lua. Its source code can be found at `https://github.com/jeffsantos/GraphProver`.

* The authors thank CNPq and CAPES for supporting this research

2 Minimal Implicational Logic

2.1 Syntax and Semantics for $\mathbf{M}^{\rightarrow}$

The syntax and semantics of $\mathbf{M}^{\rightarrow}$ is the intuitionistic syntax and semantics but restricted to $\rightarrow$ only. Thus, given a propositional language $\mathcal{L}$, a $\mathbf{M}^{\rightarrow}$ model is a structure $\langle U, \preceq, \mathcal{V}\rangle$, where U is a non-empty set (worlds), $\preceq$ is a partial order relation on U and $\mathcal{V}$ is a function from U into the power set of $\mathcal{L}$, such that if $i, j \in U$ and $i \preceq j$ then $\mathcal{V}(i) \subseteq \mathcal{V}(j)$. Given a model, the satisfaction relationship $\models$ between worlds in models and formulas is defined as in Intuitionistic Logic, namely:

- $\langle U, \preceq, \mathcal{V}\rangle \models_i p$, $p \in \mathcal{L}$, iff, $p \in \mathcal{V}(i)$
- $\langle U, \preceq, \mathcal{V}\rangle \models_i \alpha_1 \rightarrow \alpha_2$, iff, for every $j \in U$, such that $i \preceq j$, if $\langle U, \preceq, \mathcal{V}\rangle \models_j \alpha_1$ then $\langle U, \preceq, \mathcal{V}\rangle \models_j \alpha_2$.

As usual a formula α is valid in a model $\mathcal{M}$, namely $\mathcal{M} \models \alpha$, if and only if, it is satisfiable in every world i of the model, namely $\forall i \in U, \mathcal{M} \models_i \alpha$. A formula is a $\mathbf{M}^{\rightarrow}$ tautology, if and only if, it is valid in every model.

2.2 Proof Search and Counter-Model Generation in $\mathbf{M}^{\rightarrow}$

It is known that Prawitz's Natural Deduction System for Propositional Minimal Logic with only the $\rightarrow$-rules ($\rightarrow$-Elim and $\rightarrow$-Intro) is sound and complete for the $\mathbf{M}^{\rightarrow}$ regarding Kripke semantics. As a consequence of this, Gentzen's **LJ** system ([6]) containing only right and left $\rightarrow$-rules is also sound and complete. [6] also proved the decision problem for Propositional Intuitionistic Logic (**Int**), a case that includes the Propositional Minimal Logic (**Min**) and $\mathbf{M}^{\rightarrow}$. However, Gentzen's approach was not conceived to be a bottom-up proof search procedure.

A central aspect when considering mechanisms for proof search in $\mathbf{M}^{\rightarrow}$ (and also for **Int**) is the application of the $\rightarrow$-left rule. The **LK** system proposed by [6], the sequent calculus for Classical Logic, with some adaptations (e.g. [16]) can ensure that each rule, when applied in a bottom-up manner in the proof search, reduces the degree (the number of atomic symbols occurrences and connectives in a formula) of the main formula of the sequent (the formula to which the rule is applied) implying the termination of the system. However, the case for **Int** is more complicated. First, we have the "context-splitting" (using an expression from [3]) nature of $\rightarrow$-left, i.e., the formula on the right side of the conclusion sequent is lost in the left premise of the rule application. Second, as we can reuse a hypothesis in different parts of a proof, the main formula of the conclusion must be available to be used again by the generated premises. Thus, the $\rightarrow$-left rule has the repetition of the main formula in the premises, a scenario that allows the occurrence of loops in automatic procedures.

A common way to control the proof search procedure in $\mathbf{M}^{\rightarrow}$ (and in **Int**) is by the definition of routines for loop verification as proposed in [17]. Loop

checkers are very expensive procedures, although they are effective to guarantee termination in automatic provers for $\mathbf{M}^{\rightarrow}$ (and other logics with the same characteristic). The work in [9] and [11] are examples of techniques that can be used to minimize the performance problems that can arise with the use of such procedures.

To avoid the use of loop checkers, [2] proposed a terminating contraction-free sequent calculus for **Int**, named **LJT**, using a technique based on the work of [18] in the 50s. [14] extended this work showing a method to generate counter-examples in this system. They proposed two calculi, one for proof search and another for counter-model generation, forming a way to decide about the validity or not of formulas in **Int**. A characteristic of their systems is that the subformula property does not hold on them. In [5], a similar approach is presented using systems where the subformula property holds. They also proposed a single decision procedure for **Int** which guarantee minimal depth counter-model.

Focused sequent calculi appeared initially in the Andreoli's work on linear logic ([1]). The author identified a subset of proofs from Gentzen-style sequent calculus, which are complete and tractable. [13] proposed the focused sequent calculi **LJF** where they used a mapping of **Int** into linear logic and adapted the Andreoli's system to work with the image. [4] presented the focused system **LJQ** that work direct in **Int**. Focusing is used in their system as a way to implement restrictions in the $\rightarrow$-left rule as proposed by [18] and [12]. The work of [4] follows from the calculus with the same name presented in [8].

3 The Sequent Calculus $\mathbf{LMT}^{\rightarrow}$

The sequent calculus $\mathbf{LMT}^{\rightarrow}$ was first presented in [15]. Here, we presented a Termination Theorem for $\mathbf{LMT}^{\rightarrow}$ not covered in the original paper, and a new version of the Completeness Theorem initially showed there. A sequent in $\mathbf{LMT}^{\rightarrow}$ has the following general form:

$$\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, ..., \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, ..., p_n], \varphi \tag{1}$$

where φ is a formula in $\mathcal{L}$ and Δ, $\Upsilon_1^{p_1}, \Upsilon_2^{p_2}, ..., \Upsilon_n^{p_n}$ are bags[4] of formulas. Each $\Upsilon_i^{p_i}$ represents formulas associated with an atomic formula p_i.

A sequent has two *focus areas*, one in the left side (curly bracket)[5] and another on the right (square bracket). Curly brackets are used to control the application of the $\rightarrow$-left rule and square brackets are used to keep control of formulas that are related to a particular counter-model definition. Δ' is a set of formulas and $p_1, p_2, ..., p_n$ is a sequence that does not allow repetition. We call *context* of the sequent a pair (α, q), where $\alpha \in \Delta'$ and $\varphi = q$, where q is an atomic formula on the right side of the sequent.

[4] A bag (or a multiset) is a generalization of the concept of a set that, unlike a set, takes repetitions into account: a bag {A, A, B} is not the same as the bag {A, B}.

[5] Note that the symbols { and } here do not represent a set. They are used as an annotation in the sequent to determine the left side focused area. Therefore, Δ' instead is a set of formulas in the focused area.

The axioms and rules of $\mathbf{LMT}^{\rightarrow}$ are presented in Figure 1. In each rule, $\Delta' \subseteq \Delta$.

Rules are inspired by their backward application. In a $\rightarrow$-left rule application, the atomic formula, q, on the right side of the conclusion goes to the []-area in the left premise. Δ formulas in the conclusion are copied to the left premise and marked with a label relating each of them with q. The left premise also has a copy of Δ formulas without the q-label. This mechanism keeps track of proving attempts. In a restart rule application, the focused area of the left side of the conclusion sequent is cleaned on the premise. Also, the atomic formula, q, on the right side of the conclusion goes to the []-area in the premise and Δ formulas in the conclusion are copied to the premise and marked with a label relating each of them with q. In the premise, the p_i is removed from the []-area, becoming the right formula of the premise and each $\Upsilon_j^{p_j}$, for $j \leq i$ has their correspondent labels removed.

Axiom:

$$\frac{}{\{\Delta', q\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, \ldots, p_n], q}\ axiom$$

Focus:

$$\frac{\{\Delta', \alpha\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta, \alpha \Rightarrow [p_1, p_2, \ldots, p_n], \beta}{\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta, \alpha \Rightarrow [p_1, p_2, \ldots, p_n], \beta}\ \mathrm{f}_{\alpha}$$

Restart:

$$\frac{\{\}, \Upsilon_1, \Upsilon_2, \ldots, \Upsilon_i, \Upsilon_{i+1}^{p_{i+1}}, \ldots, \Upsilon_n^{p_n}, \Delta^q \Rightarrow [p_1, p_2, \ldots, p_{i+1}, \ldots, p_n, q], p_i}{\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_i^{p_i}, \Upsilon_{i+1}^{p_{i+1}}, \ldots, \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, \ldots, p_i, p_{i+1}, \ldots, p_n], q}\ \mathrm{r}_{p_i}$$

$\rightarrow$-Right

$$\frac{\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta, \alpha \Rightarrow [p_1, p_2, \ldots, p_n], \beta}{\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, \ldots, p_n], \alpha \rightarrow \beta}\ \rightarrow\text{-}\mathrm{r}_{\alpha \rightarrow \beta}$$

$\rightarrow$-Left

Considering $\overline{\Upsilon} = \bigcup_{i=1}^{n} \Upsilon_i^{p_i}$ and $\bar{p} = p_1, p_2, \ldots, p_n$, we have:

$$\frac{\{\alpha \rightarrow \beta, \Delta'\}, \overline{\Upsilon}, \Delta^q, \Delta \Rightarrow [\bar{p}, q], \alpha \qquad \{\alpha \rightarrow \beta, \Delta'\}, \overline{\Upsilon}, \Delta, \beta \Rightarrow [\bar{p}], q}{\{\alpha \rightarrow \beta, \Delta'\}\overline{\Upsilon}, \Delta \Rightarrow [\bar{p}], q}\ \rightarrow\text{-}\mathrm{l}_{(\alpha \rightarrow \beta, q)}$$

Fig. 1. Rules of $\mathbf{LMT}^{\rightarrow}$

3.1 A Proof Search Strategy

The following is a general strategy to be applied with the rules of $\mathbf{LMT}^{\rightarrow}$ to generate proofs from an input sequent (a sequent that is a candidate to be the conclusion of a proof), which is based on a bottom-up application of the rules. From the proposed strategy, we can then state a proposition about the termination of the proving process.

A *goal sequent* is a new sequent in the form of (1). It is a premise of one of the system's rules, generated by the application of this rule on an open branch during the proving process. If the goal sequent is an axiom, the branch where it is will stop. Otherwise, apply the first applicable rule in the following order:

1. Apply $\rightarrow$-right rule if it is possible, i.e., if the formula on the right side of the sequent, outside the []-area, is not atomic. The premise generated by this application is the new goal of this branch.
2. Choose one formula on the left side of the sequent, not labeled yet, i.e., a formula $\alpha \in \Delta$ that is not occurring in Δ', then apply the focus rule. The premise generated by this application is the new goal of this branch.
3. If all formulas on the left side have already been focused, choose a formula $\alpha \in \Delta'$ such that the context (α, q) was not yet tried since the last application of a restart rule. We say that a context (α, q) is already tried when a formula α on the left was expanded (by the application of $\rightarrow$-left rule) with q as the formula outside the []-area on the right side of the sequent. The premises generated by this application are new goals of the respective new branches.
4. Choose the leftmost formula inside the []-area that was not chosen before in this branch and apply the restart rule. The premise generated by this application is the new goal of the branch.

Observation 31 *From the proof strategy we can make the following observations about a tree generated during a proving process:*

(i) *A* top sequent *is the highest sequent of a branch in the tree.*
(ii) *In a top sequent of a branch on the form of sequent* (1)*, if $\varphi \in \Delta$ then the top sequent is an axiom and the branch is called a* closed *branch. Otherwise, we say that the branch is* open *and φ is an atomic formula.*
(iii) *In every sequent of the tree, $\Delta' \subseteq \Delta$.*
(iv) *For $i = 2, \ldots n$, $\Upsilon_{i-1}^{p_{i-1}} \subseteq \Upsilon_i^{p_i}$, since that in the $\rightarrow$-left rule application, that is when formulas are labeled, we always make a copy of each formula not labeled on the conclusion so that them become available to the next context of rule application.*

4 A Termination Strategy for $\mathbf{LMT}^{\rightarrow}$

In [10], Hirokawa presented an upper bound for the size of normal form Natural Deduction proofs of implicational formulas in **Int** (which includes $\mathbf{M}^{\rightarrow}$ formulas). The author showed that, for a formula $\alpha \in \mathbf{M}^{\rightarrow}$, this limit is $|\alpha| \cdot 2^{|\alpha|+1}$.

Our approach to establishing termination for $\mathbf{LMT}^{\rightarrow}$ proof search is to use the Hirokawa result to define a correspondent bound to it. To do that, we proposed two translation functions. The first one translates from normal proofs in Natural Deduction to a cut-free sequent calculus, following an $\mathbf{LJ}^{\rightarrow}$ system, adapted as discussed in Section 2.2, thus we can establish the limit for proof search in $\mathbf{LJ}^{\rightarrow}$ too. The second translates from $\mathbf{LJ}^{\rightarrow}$ to $\mathbf{LMT}^{\rightarrow}$, defining, then, the upper bound for this system. It is worth to note that this translation process can also be used to prove the completeness of $\mathbf{LMT}^{\rightarrow}$.

4.1 Translating Natural Deduction into Sequent Calculus

Figure 2 presents a recursively defined function[6] to translate Natural Deduction normal proofs of $\mathbf{M}^{\rightarrow}$ formulas into $\mathbf{LJ}^{\rightarrow}$ proofs (in a version of the system without the cut rule).

In this definition, c is a function that returns the conclusion (last sequent) of a $\mathbf{LJ}^{\rightarrow}$ demonstration as showed in (2). Also, $\rightarrow -lc$ is a function that receives two $\mathbf{LJ}^{\rightarrow}$ sequents and a formula to construct the conclusion of a $\rightarrow -left$ rule application, as defined in (3).

$$c\left(\begin{array}{c}\prod\\ \Gamma \Rightarrow \gamma\end{array}\right) = \Gamma \Rightarrow \gamma \tag{2}$$

$$\rightarrow -lc(\Gamma \Rightarrow \alpha; \beta, \Gamma \Rightarrow \gamma; \alpha \rightarrow \beta) = \Gamma, \alpha \rightarrow \beta \Rightarrow \gamma \tag{3}$$

Theorem 1. *The size of proofs in* $\mathbf{LJ}^{\rightarrow}$ *considering only implicational tautologies is the same of that in Natural Deduction, i.e. for an implicational formula* α*, a proof in* $\mathbf{LJ}^{\rightarrow}$ *has maximum height of* $|\alpha| \cdot 2^{|\alpha|+1}$.

Proof. This proof follows directly from the translation function aforementioned as each step in the Natural Deduction proof is translated into exactly one step in the $\mathbf{LJ}^{\rightarrow}$ resultant proof.

4.2 An Upper Bound for the Proof Search in $\mathbf{LMT}^{\rightarrow}$

We now propose a translation from $\mathbf{LJ}^{\rightarrow}$ proofs into the system $\mathbf{LMT}^{\rightarrow}$. The translation function needs to adapt a sequent in $\mathbf{LJ}^{\rightarrow}$ form to a sequent in $\mathbf{LMT}^{\rightarrow}$ form. Figure 3 presents the definition of the translation function[7].

[6] We use a semicolon to separate arguments of functions (in function definitions and function calls) instead of the most common approach to using commas. This change in convention aims to avoid confusion with the commas used to separate formulas and sets of formulas in sequent notation.

[7] Here, we also use semicolon to separate function arguments here

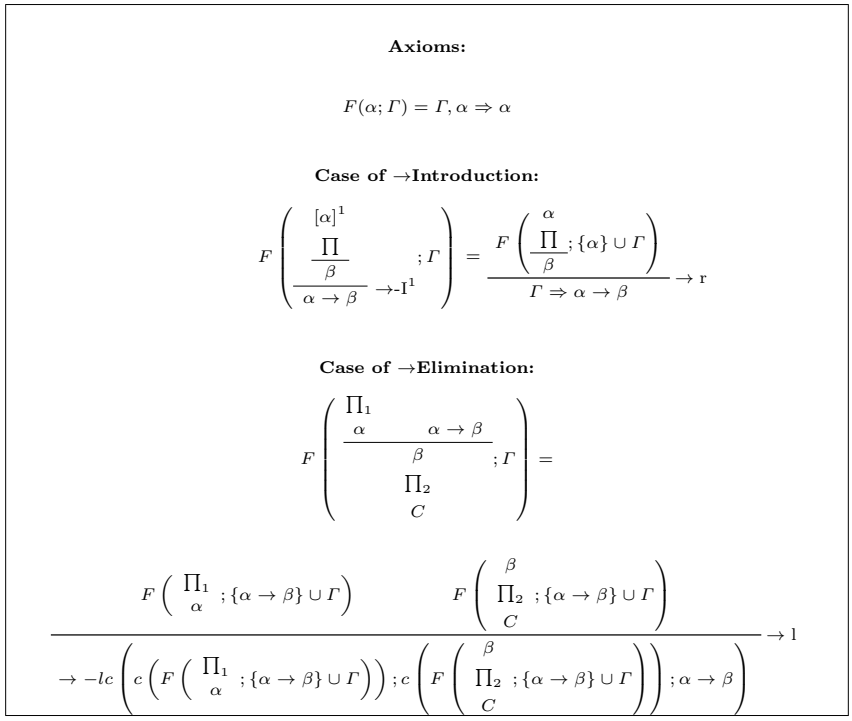


Fig. 2. A recursively defined function to translate Natural Deduction proofs into $\mathbf{LJ}^{\to}$

We use some abbreviations to shorten the function definition of Figure 3. We present them below.

$$\mathcal{D}'_1 = F'\left(\begin{array}{c}\mathcal{D}_1\\ \Gamma,\alpha\to\beta\Rightarrow\alpha\end{array};\Delta\cup\{\alpha\to\beta\};\Upsilon\cup\Gamma^q\cup\{(\alpha\to\beta)^q\};\Sigma\cup\{q\};\Pi'\right)$$

$$\mathcal{D}'_2 = F'\left(\begin{array}{c}\mathcal{D}_2\\ \Gamma,\beta\Rightarrow q\end{array};\Delta\cup\{\alpha\to\beta\};\Upsilon;\Sigma;\Pi'\right)$$

$$\Pi' = PROOFUNTIL\left(FOCUS\left(\begin{array}{c}\{\Delta\},\Upsilon,\Gamma,\alpha\to\beta\Rightarrow[\Sigma],q\\ \Pi\end{array}\right)\right)$$

Γ^q = means that all formulas of the set Γ are labeled with a reference to the atomic formula q.

$(\alpha\to\beta)^q$ = means the same for the individual formula $\alpha\to\beta$.

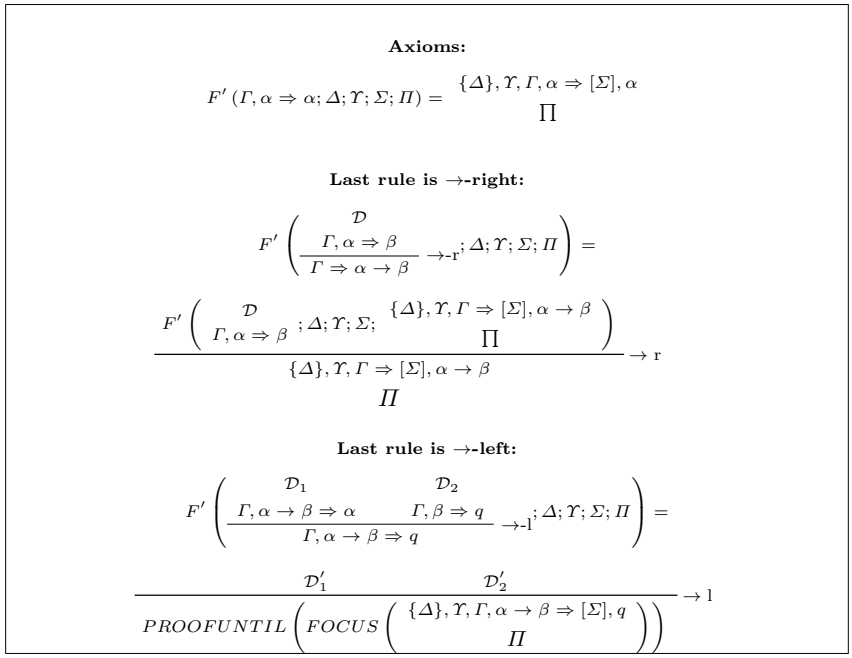


Fig. 3. A recursive function to translate $\mathbf{LJ}^{\to}$ into $\mathbf{LMT}^{\to}$

The complex case occurs when the function F' is applied to a proof fragment in which the last $\mathbf{LJ}^{\to}$ rule applied is an →-left. In this case, F' needs to inspect the proof fragment constructed until that point to identify whether the context $(\alpha \to \beta, q)$ was already used or not. This inspection has to be done since $\mathbf{LMT}^{\to}$ does not allow two or more applications of the same context between two applications of the *restart* rule. To deal with this, we use some auxiliary functions described below.

$FOCUS$ is a function that receives a fragment of proof in $\mathbf{LMT}^{\to}$ form and builds one application of the focus rule on the top of the proof fragment received in the case that the main formula of the rule is not already focused. The main formula is also an argument of the function. In the function definition (4), we have the constraint that $\alpha \in \Gamma$.

$$FOCUS\left(\begin{array}{c}\{\Delta\}, \Upsilon, \Gamma \Rightarrow [\Sigma], \beta \\ \prod \\ \{\} \Rightarrow [], \gamma\end{array}; \alpha\right) =$$

$$\left\{\begin{array}{ll} \dfrac{\{\Delta,\alpha\},\Upsilon,\Gamma\Rightarrow[\Sigma],\beta}{\{\Delta\},\Upsilon,\Gamma\Rightarrow[\Sigma],\beta}\ \text{focus} & \text{if } \alpha\notin\Delta \\ \prod \\ \{\}\Rightarrow[],\gamma & \\ \\ \{\Delta\},\Upsilon,\Gamma\Rightarrow[\Sigma],\beta & \\ \prod & \text{otherwise} \\ \{\}\Rightarrow[],\gamma & \end{array}\right\} \tag{4}$$

The function $PROOFUNTIL$ also receives a fragment of an $\mathbf{LMT}^{\rightarrow}$ proof where $(\alpha\rightarrow\beta,q)$ is one of the available contexts, applies the restart rule with an atomic formula p such that $p\in\Sigma$ in the top of this fragment of proof and, then, conducts a sequence of $\mathbf{LMT}^{\rightarrow}$ rule applications following the strategy aforementioned until the point that the context $(\alpha\rightarrow\beta,q)$ is available again. This mechanism has to be done in the case that the context $(\alpha\rightarrow\beta,q)$ is already applied in an $\rightarrow$-left application, some point after the last restart rule application in the proof fragment received as the argument Π. Otherwise, the proof fragment is returned unaltered. Function $PROOFUNTIL$ is described in the function definition (5).

$$PROOFUNTIL\left(\begin{array}{c}\{\Delta,\alpha\rightarrow\beta\},\Upsilon,\Gamma\Rightarrow[\Sigma],q\\ \prod\\ \{\}\Rightarrow[],\gamma\end{array}\right)=$$

$$\left\{\begin{array}{ll} \dfrac{\dfrac{\{\Delta'',\alpha\rightarrow\beta\},\Upsilon'',\Gamma'\Rightarrow[\Sigma''],q}{\vdots}}{\dfrac{\{\},\Upsilon',\Gamma^{q},\Gamma\Rightarrow[\Sigma'],p}{\{\Delta,\alpha\rightarrow\beta\},\Upsilon,\Gamma\Rightarrow[\Sigma],q}\ \text{restart}-p}\ \rightarrow\text{-left}(\alpha\rightarrow\beta,q)\in\prod \\ \prod \\ \{\}\Rightarrow[],\gamma & \\ \\ \{\Delta,\alpha\rightarrow\beta\},\Upsilon,\Gamma\Rightarrow[\Sigma],q & \\ \prod & \text{otherwise} \\ \{\}\Rightarrow[],\gamma & \end{array}\right\} \tag{5}$$

4.3 Termination

To control the end of the proof search procedure of $\mathbf{LMT}^{\rightarrow}$ our approach is to prove an upper bound to the size of its proof search tree. Then, we need to show that the $\mathbf{LMT}^{\rightarrow}$ strategy here proposed allows exploring all the possible ways to expand the proof tree until it reaches this size.

From Theorem 1, we know that the upper bound for cut-free proofs based on $\mathbf{LJ}^{\rightarrow}$ is $|\alpha| \cdot 2^{|\alpha|+1}$, where α is the initial formula that we want to prove. We use the translation presented in Figure 3 on the previous Section to find a similar limit for $\mathbf{LMT}^{\rightarrow}$proofs. We have to analyze three cases to establish an upper bound for $\mathbf{LMT}^{\rightarrow}$.

(i) Axioms of $\mathbf{LJ}^{\rightarrow}$ maps one to one with axioms of $\mathbf{LMT}^{\rightarrow}$;
(ii) $\rightarrow$-right applications of $\mathbf{LJ}^{\rightarrow}$ maps one to one with $\rightarrow$-right applications of $\mathbf{LMT}^{\rightarrow}$;
(iii) $\rightarrow$-left applications of $\mathbf{LJ}^{\rightarrow}$ maps to $\mathbf{LMT}^{\rightarrow}$ in three different possible subcases, according to the context $(\alpha \rightarrow \beta, q)$ in which the rule is being applied in $\mathbf{LJ}^{\rightarrow}$. We have to consider the fragment of $\mathbf{LMT}^{\rightarrow}$ already translated to decide the appropriate case.
   - If the context is **not yet focused neither expanded**, then, one application of $\rightarrow$-left in $\mathbf{LJ}^{\rightarrow}$ maps to two applications of rules in $\mathbf{LMT}^{\rightarrow}$: first, a focus application, then an $\rightarrow$-left application.
   - If the context is **already focused but not yet expanded**, then, one application of $\rightarrow$-left in $\mathbf{LJ}^{\rightarrow}$ maps to one application of $\rightarrow$-left in $\mathbf{LMT}^{\rightarrow}$.
   - If the context $(\alpha \rightarrow \beta, q)$ is **already focused and expanded**, then, one application of $\rightarrow$-left in $\mathbf{LJ}^{\rightarrow}$ maps to the height of the $\mathbf{LMT}^{\rightarrow}$ proof fragment produced by the execution of the $PROOFUNTIL$ function. Let this height be called h.

Lemma 1. *The height h that defines the size of the proof fragment returned by the function $PROOFUNTIL$ has a maximum limit of $2^{2log_2|\alpha|}$, where α is the main formula of the initial sequent of the proof in $\mathbf{LMT}^{\rightarrow}$.*

Proof. Consider a proof $\prod_{\mathbf{LJ}^{\rightarrow}}$ of an initial sequent in $\mathbf{LJ}^{\rightarrow}$ with the form $\Rightarrow \alpha$. The process of translating $\prod_{\mathbf{LJ}^{\rightarrow}}$ to $\mathbf{LMT}^{\rightarrow}$ produces a proof $\prod_{\mathbf{LMT}^{\rightarrow}}$ with the initial sequent in the form $\{\} \Rightarrow [], \alpha$. Consider that α has the form $\alpha_1 \rightarrow \alpha_2$. In some point of the translation to $\mathbf{LMT}^{\rightarrow}$, we reach a point where a context $(\psi \rightarrow \varphi, q)$ is already focused and expanded in the already translated part of the proof $\prod_{\mathbf{LMT}^{\rightarrow}}$. At this point, the function $PROOFUNTIL$ generates a fragment of the proof $\prod_{\mathbf{LMT}^{\rightarrow}}$, call it Σ of size h. The height h is bound by the number of applications of $\rightarrow$-left rules in Σ. This can be determined by the multiplication of the degree of the formula α_1 (that bounds the number of possible implicational formulas in the left side of a sequent in $\mathbf{LMT}^{\rightarrow}$) by the maximum number of atomic formulas (n) inside the []-area in the highest branch of Σ (each p_i inside the []-area allows one application of the restart rule). Thus we can formalize this in the following manner: $h = n \times |\alpha_1| \Rightarrow h = |\alpha| \times |\alpha| \Rightarrow h = |\alpha|^2 \Rightarrow |\alpha|^2 = 2^{2 \cdot log_2|\alpha|} \Rightarrow h = 2^{2 \cdot log_2|\alpha|}$.

Theorem 2. *The size of a proof for a formula $\alpha \in \mathbf{M}^{\rightarrow}$ in $\mathbf{LMT}^{\rightarrow}$ has an upper bound of $|\alpha| \cdot 2^{|\alpha|+1+2 \cdot log_2|\alpha|}$.*

Proof. Considering the size of proofs for a formula α using $\mathbf{LJ}^{\rightarrow}$and the Lemma 1, the proof follows directly.

Theorem 3. $\mathbf{LMT}^{\rightarrow}$ *eventually stops.*

Proof. To guarantee termination, we use the upper bound presented in Theorem 2 to limit the height of opened branches during the $\mathbf{LMT}^{\rightarrow}$ proof search process. The strategy presented in Section 4 forces an ordered application of rules that produces all possible combination of formulas to be expanded in the right and left sides of generated sequents. In other words, when the upper bound is reached the proof search had, for sure, stressed all possible expansions until that point.

5 Soundness

The soundness of $\mathbf{LMT}^{\rightarrow}$ is already proved in [15]. Here, we just present the main definitions and proposition regarding it.

Definition 1. *A sequent* $\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, \ldots, \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, \ldots, p_n], \varphi$ *is* valid, *if and only if,* $\Delta', \Delta \models \varphi$ *or* $\exists i(\bigcup_{k=1}^{i} \Upsilon_k^{p_k}) \models p_i$, *for* $i = 1, \ldots n$.

Definition 2. *We say that a rule is* sound, *if and only if, in the case of the premises of the sequent are valid sequents, then its conclusion also is.*

Proposition 1. *Considering validity of a sequent as defined in Definition 1,* $\mathbf{LMT}^{\rightarrow}$ *is sound.*

6 Completeness

By Observation 31.ii we know that a top sequent of an open branch in an attempt proof tree has the general form below, where q is an atomic formula:

$$\{\Delta'\}, \Upsilon_1^{p_1}, \Upsilon_2^{p_2}, ..., \Upsilon_n^{p_n}, \Delta \Rightarrow [p_1, p_2, ..., p_n], q$$

From Definition 1 and considering that $\Delta' \subseteq \Delta$ in any sequent of an attempt proof tree following our proposed strategy, we can define a invalid sequent as follows:

Definition 3. *A sequent is* invalid *if and only if* $\Delta \not\models q$ *and* $\forall i(\bigcup_{k=1}^{i} \Upsilon_k^{p_k}) \not\models p_i$, *for* $i = 1, \ldots, n$.

Our proof of the completeness starts with a definition about atomic formulas in the left and right side of a top sequent.

Definition 4. *We can construct a Kripke counter-model* $\mathcal{M}$ *that satisfies atomic formulas in the right side of a top sequent and that falsifies the atomic formula on the left. This construction can be done in the following way:*

1. *The model $\mathcal{M}$ has an initial world w_0.*
2. *By the proof strategy, we can conclude that, in any sequent of the proof tree, $\Upsilon_1^{p_1} \subseteq \Upsilon_2^{p_2} \subseteq \cdots \subseteq \Upsilon_n^{p_n} \subseteq \Delta$. We create a world in the model $\mathcal{M}$ corresponding for each one of these bags of formulas and, using the inclusion relation between them, we define a respective accessibility relation in the model $\mathcal{M}$ between such worlds. That is, we create worlds $w_{\Upsilon_1^{p_1}}, w_{\Upsilon_2^{p_2}}, \ldots, w_{\Upsilon_n^{p_n}}, w_\Delta$ related in the following form: $w_{\Upsilon_1^{p_1}} \preceq w_{\Upsilon_2^{p_2}} \preceq \cdots \preceq w_{\Upsilon_n^{p_n}} \preceq w_\Delta$. As w_0 is the first world of $\mathcal{M}$, it precedes $w_{\Upsilon_1^{p_1}}$, that is, $w_0 \preceq w_{\Upsilon_1^{p_1}}$ is also included in the accessibility relation. If $\Upsilon_i^{p_i} = \Upsilon_{i+1}^{p_{i+1}}$, for $i = 1, \ldots, n$, then the associated worlds that correspond to those sets have to be collapsed in a single world $w_{\Upsilon_i^{p_i} - \Upsilon_{i+1}^{p_{i+1}}}$. In this case, the previous relation $w_{\Upsilon_i^{p_i}} \preceq w_{\Upsilon_{i+1}^{p_{i+1}}}$ is removed from the $\preceq$ relation of the model $\mathcal{M}$ and the pairs $w_{\Upsilon_{i-1}^{p_{i-1}}} \preceq w_{\Upsilon_i^{p_i}}$ and $w_{\Upsilon_{i+1}^{p_{i+1}}} \preceq w_{\Upsilon_{i+2}^{p_{i+2}}}$ become respectively $w_{\Upsilon_{i-1}^{p_{i-1}}} \preceq w_{\Upsilon_i^{p_i} - \Upsilon_{i+1}^{p_{i+1}}}$ and $w_{\Upsilon_i^{p_i} - \Upsilon_{i+1}^{p_{i+1}}} \preceq w_{\Upsilon_{i+2}^{p_{i+2}}}$.*
3. *By Definition 3 of an invalid sequent, $\Delta \nvDash q$. The world w_Δ will be used to guarantee this. We set q false in w_Δ, i.e, $\mathcal{M} \nvDash_{w_\Delta} q$. We also set every atomic formula that is in Δ as true, i.e., $\forall p, p \in \Delta, \mathcal{M} \vDash_{w_\Delta} p$.*
4. *By Definition 3 of an invalid sequent, we also need that $\forall i (\bigcup_{k=1}^{i} \Upsilon_k^{p_k}) \nvDash p_i$, for $i = 1, \ldots n$. Thus, for each $i, i = 1, \ldots, n$ we set $\mathcal{M} \nvDash_{w_{\Upsilon_i^{p_i}}} p_i$ and $\forall p, p \in \Upsilon_i^{p_i}$, being p an atomic formula, $\mathcal{M} \vDash_{w_{\Upsilon_i^{p_i}}} p$. In the case of collapsed worlds, we keep the satisfaction relation of the previous individual worlds in the collapsed one.*
5. *In w_0 set every atomic formula inside the []-area (all of them are atomic) as false. That is, $\mathcal{M} \nvDash_{w_0} p_i$, for $i = 1, \ldots, n$. We also set the atomic formula outside the []-area false in this world: $\mathcal{M} \nvDash_{w_0} q$. Those definitions make w_0 consistent with the $\preceq$ relation of $\mathcal{M}$.*

The Figure 4 shows the general shape of counter-models following the steps enumerated above. This procedure to construct counter-model allows us to state the following lemma:

Lemma 2. *Let S be a top sequent of an open branch in an attempt proof tree generated by the strategy presented in Section 4. Then we can construct a Kripke model $\mathcal{M}$ with a world u where $\mathcal{M} \nvDash_u S$, using the aforementioned counter-model generation procedure.*

Proof. We can prove this by induction on the degree of formulas in Δ. From Definition 4, items 3 and 4 we know the value of each atomic formula in the worlds w_Δ and in each world $w_{\Upsilon_i^{p_i}}$. The inductive hypothesis is that every formula in Δ is true in w_Δ. Thus, as $\Upsilon_1^{p_1} \subseteq \Upsilon_2^{p_2} \subseteq \cdots \subseteq \Upsilon_n^{p_n} \subseteq \Delta$, every formula in $\Upsilon_i^{p_i}$ is true in $w_{\Upsilon_i^{p_i}}$, for $i = 1, \ldots, n$.

Thus, we have two cases to consider:

1. The top sequent is in the rightmost branch of the proof tree ([]-area is empty).

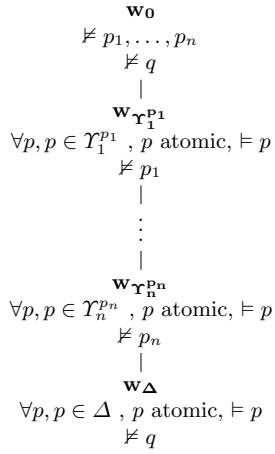


Fig. 4. General schema of counter-models

Let $\alpha \to \beta$ be a formula in $\mathbf{M}^{\to}$ that is in Δ. We show that $\mathcal{M} \vDash_{w_\Delta} \alpha \to \beta$. In this case, by the proof strategy, $\beta \equiv (\beta_1 \to (\beta_2 \to \cdots \to (\beta_m \to p)))$, where p is an atomic formula. By Definition 4.3 $\vDash_{w_\Delta} p$. As w_Δ has no accessible world from it (except for itself), $\vDash_{w_\Delta} \beta$. By the proof strategy, $\beta_m \to p, \beta_{m-1} \to \beta_m \to p, \ldots, \beta_2 \to \cdots \to \beta_{m-1} \to \beta_m \to p, \beta_1 \to \beta_2 \to \cdots \to \beta_{m-1} \to \beta_m \to p$ also are in Δ. The degree of each of these formulas is less than the degree of $\alpha \to \beta$ and, by the induction hypothesis, all of them are true in w_Δ. Thus $\vDash_{w_\Delta} \beta$ and $\vDash_{w_\Delta} \alpha \to \beta$.
As the []-area is empty, the sets $\Upsilon_i^{p_i}$ are also empty. The counter-model only has two words, w_0 and w_Δ, following the properties described in Definition 4.

2. The top sequent is in any other branch that is not the rightmost one ([]-area is not empty).
Let $\alpha \to \beta$ be a formula in $\mathbf{M}^{\to}$ that is in Δ. We show that $\mathcal{M} \vDash_{w_\Delta} \alpha \to \beta$. In this case, by the proof strategy, $\alpha \equiv (\alpha_1 \to (\alpha_2 \to \cdots \to (\alpha_m \to q)))$, where q is the atomic formula in the right side of the sequent, out of the []-area. By Definition 4.3 $\nvDash_{w_\Delta} q$. By the proof strategy, $\alpha_1, \alpha_2, \ldots, \alpha_m$ also are in Δ. The degree of each of these formulas is less than the degree of $\alpha \to \beta$ and, by the induction hypothesis, all of them are true in w_Δ. This ensures $\nvDash_{w_\Delta} \alpha$ and $\vDash_{w_\Delta} \alpha \to \beta$.
Considering now a formula $\alpha \to \beta$ from $\mathbf{M}^{\to}$ that is in $\Upsilon_i^{p_i}$. By Definition 4.2, $\alpha \to \beta$ also belongs to Δ. From the last paragraph, we show that, for any formula $\alpha \to \beta \in \Delta$, $\nvDash_{w_\Delta} \alpha$. As $\nvDash_{w_\Delta} \alpha$, by the accessibility relation of the Kripke model, $\nvDash_{w_{\Upsilon_i^{p_i}}} \alpha$, for each $i = 1, \ldots, n$. Thus, the value of $\alpha \to \beta$ is defined in any of these worlds by the value of $\alpha \to \beta$ in w_Δ, that we showed to be true. Thus, $\vDash_{w_{\Upsilon_i^{p_i}}} \alpha \to \beta$.

As stated in Definition 4.2, $\Upsilon_1^{p_1} \subseteq \Upsilon_2^{p_2} \subseteq \cdots \subseteq \Upsilon_n^{p_n} \subseteq \Delta$ and following the accessibility relation rule of the $\mathbf{M}^{\rightarrow}$ semantic (relations are reflexive and transitive) we conclude that:

$$\begin{aligned}\mathcal{M} \vDash_{w_0} & \Upsilon_1^{p_1}, \nvDash_{w_0} p_1 \\ \vDash_{w_0} & \Upsilon_2^{p_2}, \nvDash_{w_0} p_2 \\ & \vdots \\ \vDash_{w_0} & \Upsilon_n^{p_n}, \nvDash_{w_0} p_n \\ \vDash_{w_0} & \Delta, \nvDash_{w_0} q\end{aligned}$$

Definition 5. *A rule is said* invertible *or* double-sound *iff the validity of its conclusion implies the validity of its premises.*

By Definition 5 we know that a counter-model for a top sequent of a proof tree which can not be expanded anymore can be used to construct a counter-model to every sequent in the same branch of the tree until the conclusion (root sequent). For the $\rightarrow$-right rule, not just if the premise of the rule has a counter-model then so does the conclusion, but the same counter-model will do. [19] called rules with this property *preserving counter model*. Dyckhoff (personal communication, 2015) proposed to call this kind of rules of *strongly invertible* rules. For $\rightarrow$-left rule, this is the same when one of the premises is valid, but, considering the case that both premises are not valid, we need to mix the counter-models of both sides to construct the counter-model for the conclusion of the rule. This way to produce counter-models is what we call a *weakly invertible* rule (since it is well known that linear models are not complete for $\mathbf{M}^{\rightarrow}$). The third case in the following Lemma of Invertibility will solve this, proposing a way to mix counter-models that come from different branches in the attempt proof tree.

Lemma 3. *The rules of* $\mathbf{LMT}^{\rightarrow}$ *are invertible.*

Proof. Here we just show the case of $\rightarrow$-left. The other cases are already showed in [15]:

$\rightarrow$-left Considering that one of the premises of $\rightarrow$-left is not valid, the conclusion also is. We have to evaluate three cases:

1. **The right premise is invalid but the left premise is valid**. Then there is a Kripke model $\mathcal{M}$ where $\alpha \rightarrow \beta, \Delta', \Delta, \beta \nvDash q$ and $\forall i(\bigcup_{k=1}^{i} \Upsilon_k^{p_k}) \nvDash p_i$, for $i = 1, \ldots, n$ from a given world u. Thus, in the conclusion we have:
   - By the model $\mathcal{M}$, there have to be a world v, $u \preceq v$, in the model where $\alpha \rightarrow \beta, \Delta', \Delta, \beta$ are satisfied and where q is not.
   - By the model $\mathcal{M}$, for each i, exists a world v_i, $u \preceq v_i$, where $\vDash_{v_i} \Upsilon_i^{p_i}$ and $\nvDash_{v_i} p_i$.
   - Thus, the conclusion is invalid too.

2. **The left premise is invalid but the right premise is valid**. Then there is a Kripke model $\mathcal{M}$ where $\alpha \to \beta, \Delta', \Delta \nVdash \alpha$ and $\forall i(\bigcup_{k=1}^{i} \Upsilon_k^{p_k}) \nVdash p_i$, for $i = 1, \ldots, n$, and $\Delta^q \nVdash q$ from a given world u. Thus, in the conclusion we have:
   - By the model $\mathcal{M}$, there have to be a world v, $u \preceq v$, in the model where $\alpha \to \beta, \Delta', \Delta$ are satisfied and where α is not.
   - By the model $\mathcal{M}$, for each i, exists a world v_i, $u \preceq v_i$, where $\vDash_{v_i} \Upsilon_i^{p_i}$ and $\nVdash_{v_i} p_i$.
   - We also know by $\mathcal{M}$ that there is a world v_{Δ^q}, $u \preceq v_{\Delta^q}$, where $\vDash_{v_{\Delta^q}} \Delta^q$ and $\nVdash_{v_{\Delta^q}} q$. We also have that $\Delta^q = \Delta$ and that $\alpha \to \beta \in \Delta$. Therefore, $\vDash_{v_{\Delta^q}} \Delta'$ and $\vDash_{v_{\Delta^q}} \alpha \to \beta$.
   - Thus, the conclusion can not be valid.
3. **Both left and right premises are invalid**. Then there are two models $\mathcal{M}_1$ and $\mathcal{M}_2$, from the right and left premises respectively. In $\mathcal{M}_1$ there is a world u_1 that makes the right sequent invalid as described in item 1. In $\mathcal{M}_2$ there is a world u_2 that makes the sequent of the left premise invalid as described in item 2. Considering the way Kripke models are constructed based on Lemma 2, we know that u_1 and u_2 are root worlds of their respective counter-models. Thus, converting the two models into one, $\mathcal{M}_3$, by mixing u_1 and u_2 in the root of $\mathcal{M}_3$, called u_3, we have that in u_3:
   - $\alpha \to \beta, \Delta', \Delta$ are satisfied and α is not.
   - for $i = 1, \ldots, n$, we have that $\vDash_{u_3} \Upsilon_i^{p_i}$ and $\nVdash_{u_3} p_i$.
   - $\nVdash_{u_3} q$
   - Thus, the conclusion is also invalid.

Proposition 2. $\mathbf{LMT}^{\to}$ *is complete regarding the proof strategy presented in Section 4*

Proof. It follows direct from Theorem 3 ($\mathbf{LMT}^{\to}$ terminates) and Lemma 2 (we can construct a counter-model for a top sequent in a terminated open branch of $\mathbf{LMT}^{\to}$) and Lemma 3 (the rules of $\mathbf{LMT}^{\to}$ are invertible).

7 Conclusion and Future Work

We presented here a sequent calculus for proof search for $\mathbf{M}^{\to}$ called $\mathbf{LMT}^{\to}$ which aims to proceed the proof search of $\mathbf{M}^{\to}$ formulas in a bottom-up, forward-always approach. Termination of the proof search is achieved without using loop checkers. $\mathbf{LMT}^{\to}$ does not need an explicit backtracking mechanism to be complete. $\mathbf{LMT}^{\to}$ igenerates Kripke counter-models from search trees produced by unsuccess proving processes.

We can enumerate the following issues as future work: (a) **Define a precise upper bound for termination** (the upper bound used here for achieving termination in $\mathbf{LMT}^{\to}$ is a very high bound, many non-theorems can be identified in a small number of steps. Our labeling mechanism in conjunction with

the usage of the restart rule produces many repetitions in the proof tree); (b) **compression and sharing** (explore the techniques proposed by [7] to define approaches to shorten the size of proofs); (c) **minimal counter-models** (use references such as [2] and in [14] to improve $\mathbf{LMT}^{\rightarrow}$ counter-model generation).

References

1. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation 2(3), 297–347 (1992)
2. Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. The Journal of Symbolic Logic 57(03), 795–807 (1992)
3. Dyckhoff, R.: Intuitionistic decision procedures since gentzen. In: Advances in Proof Theory, pp. 245–267. Springer (2016)
4. Dyckhoff, R., Lengrand, S.: LJQ: a strongly focused calculus for intuitionistic logic. In: Logical Approaches to Computational Barriers, pp. 173–185. Springer (2006)
5. Ferrari, M., Fiorentini, C., Fiorino, G.: Contraction-free linear depth sequent calculi for intuitionistic propositional logic with the subformula property and minimal depth counter-models. Journal of automated reasoning 51(2), 129–149 (2013)
6. Gentzen, G.: Untersuchungen über das logische schließen. i. Mathematische zeitschrift 39(1), 176–210 (1935)
7. Gordeev, L., Haeusler, E.H.: NP vs PSPACE. arXiv preprint arXiv:1609.09562 (2016)
8. Herbelin, H.: A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In: Computer Science Logic. pp. 61–75. Springer (1995)
9. Heuerding, A., Seyfried, M., Zimmermann, H.: Efficient loop-check for backward proof search in some non-classical propositional logics. In: Theorem Proving with Analytic Tableaux and Related Methods, pp. 210–225. Springer (1996)
10. Hirokawa, S.: Number of proofs for implicational formulas. Introduction to Mathematical Analysis (in Japanese) 772, 72–74 (1991)
11. Howe, J.M.: Two loop detection mechanisms: a comparison. In: Automated Reasoning with Analytic Tableaux and Related Methods, pp. 188–200. Springer (1997)
12. Hudelmaier, J.: An O(n log n)-space decision procedure for intuitionistic propositional logic. Journal of Logic and Computation 3(1), 63–75 (1993)
13. Liang, C., Miller, D.: Focusing and polarization in intuitionistic logic. In: Computer Science Logic. pp. 451–465. Springer (2007)
14. Pinto, L., Dyckhoff, R.: Loop-free construction of counter-models for intuitionistic propositional logic. In: Symposia Gaussiana, Conf A. pp. 225–232. Walter de Gruyter & Co (Berlin) (1995)
15. Santos, J.d.B., Vieira, B.L., Haeusler, E.H.: A unified procedure for provability and counter-model generation in minimal implicational logic. Electronic Notes in Theoretical Computer Science 324, 165–179 (2016)
16. Seldin, J.P.: Manipulating proofs (1998)
17. Underwood, J.: A constructive completeness proof for intuitionistic propositional calculus. Tech. rep., Cornell University (1990)
18. Vorobev, N.N.: A new algorithm for derivability in the constructive propositional calculus. American Mathematical Society Translations 94(2), 37–71 (1970)
19. Weich, K.: Decision procedures for intuitionistic propositional logic by program extraction. In: Automated Reasoning with Analytic Tableaux and Related Methods, pp. 292–306. Springer (1998)

Algebraic semantics for Nelson's logic $\mathcal{S}$

Thiago N. Silva[1] Umberto Rivieccio[2]

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, Brazil

Abstract

Besides the better-known Nelson's logic and paraconsistent Nelson's logic, in *Negation and separation of concepts in constructive systems* (1959), David Nelson introduced a logic called $\mathcal{S}$ with motivations of arithmetic and constructibility. $\mathcal{S}$ was defined by means of a calculus (crucially lacking the contraction rule) having infinitely many rule schemata, and no semantics was provided. We look here at the propositional fragment of $\mathcal{S}$, showing that it is algebraizable (in fact, implicative) with respect to a class of involutive residuated lattices. We thus introduce the first (algebraic) semantics for $\mathcal{S}$ as well as a finite Hilbert-style calculus equivalent to Nelson's presentation.

Keywords: Nelson's Logic; Algebraic Logic; Strong negation; Constructive logic; Involutive residuated lattice.

1 Nelson's logic $\mathcal{S}$

In this section we give Nelson's original presentation of the propositional fragment of $\mathcal{S}$ [4].

Nelson's logic $\mathcal{S} = \langle \mathbf{Fm}, \vdash_{\mathcal{S}} \rangle$ is the sentential logic in the language $\langle \wedge, \vee, \to, \neg, \bot \rangle$ defined by the Hilbert-style calculus with the following axiom and rule schemata:

Axioms

(A1) $\varphi \to \varphi$

(A2) $\bot \to \varphi$

(A3) $\neg\varphi \to (\varphi \to \bot)$

(A4) $\neg\bot$

(A5) $(\varphi \to \psi) \leftrightarrow (\neg\psi \to \neg\varphi)$.

As usual, $\varphi \leftrightarrow \psi$ abbreviates $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

[1] Email: thiagnascsilva@gmail.com
[2] Email: urivieccio@dimap.ufrn.br

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Table 1
Rules

$$\frac{\Gamma \Rightarrow (\varphi \Rightarrow (\psi \Rightarrow \gamma))}{\Gamma \Rightarrow (\psi \Rightarrow (\varphi \Rightarrow \gamma))}\ (\mathrm{P}) \qquad \frac{\varphi \Rightarrow (\varphi \Rightarrow (\varphi \Rightarrow \gamma))}{\varphi \Rightarrow (\varphi \Rightarrow \gamma)}\ (\mathrm{C}) \qquad \frac{\Gamma \Rightarrow \varphi \quad \varphi \Rightarrow \gamma}{\Gamma \Rightarrow \gamma}\ (\mathrm{E})$$

$$\frac{\Gamma \Rightarrow \varphi \quad \psi \Rightarrow \gamma}{\Gamma \Rightarrow ((\varphi \Rightarrow \psi) \Rightarrow \gamma)}\ (\Rightarrow \mathrm{l}) \qquad \frac{\gamma}{\varphi \Rightarrow \gamma}\ (\Rightarrow \mathrm{r}) \qquad \frac{\varphi \Rightarrow \gamma}{(\varphi \wedge \psi) \Rightarrow \gamma}\ (\wedge \mathrm{l1})$$

$$\frac{\psi \Rightarrow \gamma}{(\varphi \wedge \psi) \Rightarrow \gamma}\ (\wedge \mathrm{l2}) \qquad \frac{\Gamma \Rightarrow \varphi \qquad \Gamma \Rightarrow \psi}{\Gamma \Rightarrow (\varphi \wedge \psi)}\ (\wedge \mathrm{r}) \qquad \frac{\varphi \Rightarrow \gamma \quad \psi \Rightarrow \gamma}{(\varphi \vee \psi) \Rightarrow \gamma}\ (\vee \mathrm{l1})$$

$$\frac{(\varphi \Rightarrow^2 \gamma) \qquad (\psi \Rightarrow^2 \gamma)}{((\varphi \vee \psi) \Rightarrow^2 \gamma)}\ (\vee \mathrm{l2}) \qquad \frac{\Gamma \Rightarrow \psi}{\Gamma \Rightarrow (\psi \vee \gamma)}\ (\vee \mathrm{r1}) \qquad \frac{\Gamma \Rightarrow \gamma}{\Gamma \Rightarrow (\psi \vee \gamma)}\ (\vee \mathrm{r2})$$

$$\frac{(\varphi \wedge \neg\psi) \Rightarrow \gamma}{\neg(\varphi \Rightarrow \psi) \Rightarrow \gamma}\ (\neg \Rightarrow \mathrm{l}) \qquad \frac{\Gamma \Rightarrow^2 (\varphi \wedge \neg\psi)}{\Gamma \Rightarrow^2 \neg(\varphi \Rightarrow \psi)}\ (\neg \Rightarrow \mathrm{r}) \qquad \frac{(\neg\varphi \vee \neg\psi) \Rightarrow \gamma}{\neg(\varphi \wedge \psi) \Rightarrow \gamma}\ (\neg \wedge \mathrm{l})$$

$$\frac{\Gamma \Rightarrow (\neg\varphi \vee \neg\psi)}{\Gamma \Rightarrow \neg(\varphi \wedge \psi)}\ (\neg \wedge \mathrm{r}) \qquad \frac{(\neg\varphi \wedge \neg\psi) \Rightarrow \gamma}{\neg(\varphi \vee \psi) \Rightarrow \gamma}\ (\neg \vee \mathrm{l}) \qquad \frac{\Gamma \Rightarrow (\neg\varphi \wedge \neg\psi)}{\Gamma \Rightarrow \neg(\varphi \vee \psi)}\ (\neg \vee \mathrm{r})$$

$$\frac{\varphi \Rightarrow \gamma}{\neg\neg\varphi \Rightarrow \gamma}\ (\neg\neg \mathrm{l}) \qquad \frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \neg\neg\varphi}\ (\neg\neg \mathrm{r})$$

Rules

Following Nelson's notation, in the Table 1 $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ is a finite set of formulas and the following abbreviations are used. $\Gamma \to \varphi := \varphi_1 \to (\varphi_2 \to (\ldots \to (\varphi_n \to \varphi)))$ and, if $\Gamma = \emptyset$, then $\Gamma \to \varphi := \varphi$. $\varphi \to^2 \psi := \varphi \to (\varphi \to \psi)$ and $\Gamma \to^2 \varphi := \varphi_1 \to^2 (\varphi_2 \to^2 \ldots (\to^2 (\varphi_n \to^2 \varphi)))$. Notice that we have fixed obvious typos in the rules ($\wedge$l), ($\wedge$r) and ($\neg \to$ l) as they appear in [4, p. 214-5]. The rule (C) above, called *weak condensation* by Nelson, replaces (and is indeed a weaker form of) the usual contraction rule:

$$\frac{\varphi \to (\varphi \to \psi)}{\varphi \to \psi}$$

2 $\mathcal{S}$ is algebraizable

In this section we prove that $\mathcal{S}$ is algebraizable in the sense of Blok and Pigozzi (and, in fact, it is *implicative* [1, Definition 2.3]), and we give two equivalent presentations for its equivalent algebraic semantics (called $\mathcal{S}$*-algebras*). The first one is obtained via the algorithm of [6, Theorem 2.17], while the second one is closer to the usual axiomatizations of classes of residuated lattices, which are the algebraic counterpart of many logics in the substructural family. The second presentation of $\mathcal{S}$-algebras

will allow us to see at a glance that they form an equational class, and also makes it easier to compare them with other known classes of algebras of substructural logics.

Definition 2.1 An *implicative logic* is a logic $\mathcal{L}$ in a language $\mathbf{L}$ with a binary term $\to$ such that the following conditions are satisfied:

IL1 $\vdash_{\mathcal{L}} \varphi \to \varphi$

IL2 $\varphi \to \psi, \psi \to \gamma \vdash_{\mathcal{L}} \varphi \to \gamma$

IL3 For each $\lambda \in L$, of arity $n > 0$, $\bigcup_{i=1}^{n}\{\varphi_i \to \psi_i, \psi_i \to \varphi_i\} \vdash_{\mathcal{L}} \lambda\varphi_i \cdots \varphi_n \to \lambda\psi_i \cdots \psi_n$

IL4 $\varphi, \varphi \to \psi \vdash_{\mathcal{L}} \psi$

IL5 $\varphi \vdash_{\mathcal{L}} \psi \to \varphi$

Let $\mathbf{Fm}$ be a set of formulas, henceforth the set the *equations* of the language $\mathbf{L}$ is denoted by $\mathsf{E}q$ and is defined as $\mathsf{E}q := \mathbf{Fm} \times \mathbf{Fm}$, we write $\varphi \approx \psi$ rather than (φ, ψ).

Theorem 2.2 *The calculus $\vdash_{\mathcal{S}}$ is implicative, and thus algebraizable, with translations $\tau\colon Fm \to Eq$ from formulas to equations and $\rho\colon Eq \to Fm$ from equations to formulas given by $\tau(\varphi) := \varphi \approx (\varphi \to \varphi)$ and $\rho(\varphi \approx \psi) := \{(\varphi \to \psi) \wedge (\psi \to \varphi)\}$.*

2.1 $\mathcal{S}$-algebras

By Blok-Pigozzi's algorithm ([6, Theorem 2.17], see also [1, Proposition 3.44]), we have that the equivalent algebraic semantics of $\mathcal{S}$ is a class of algebras given by Definition 2.3 below.

Definition 2.3 An $\mathcal{S}$-algebra is a structure $\mathbf{A} = \langle A, \wedge, \vee, \to, \neg, 0\rangle$ of type $\langle 2, 2, 2, 1, 0\rangle$ that satisfies the following equations and quasiequations:

(i) The equation $\mathsf{E}(\varphi)$ for any axiom φ of the calculus $\vdash_{\mathcal{S}}$, defined as follows:

$$\mathsf{E}(\varphi) \;:=\; \varphi \approx \neg 0.$$

(ii) The quasiequation $\mathsf{Q}(R)$ for any rule $R = \{\varphi_1, \ldots, \varphi_n, \varphi\}$ of the calculus $\vdash_{\mathcal{S}}$ (with premisses $\varphi_1, \ldots, \varphi_n$ and conclusion φ), defined as follows:

$$\mathsf{Q}(R) \;:=\; (\varphi_1 \approx \neg 0 \;\;\&\;\; \ldots \;\&\;\; \varphi_n \approx \neg 0) \;\;\Rightarrow\;\; \varphi \approx \neg 0.$$

We denote by $\mathsf{E}(\mathrm{A\,n})$ the equation that is the τ-translation of axiom $\mathrm{A\,n}$ (for $1 \leq \mathrm{n} \leq 5$), and by $\mathsf{Q}(R)$ the quasiequation that is the τ-translation of the rule R. From now on we shall also abbreviate $x * y := \neg(x \to \neg y)$, $1 := \neg 0$, $x^2 := x * x$ and $x^n := x * (x^{n-1})$ for $n > 2$.

Proposition 2.4 *Let A be an $\mathcal{S}$-algebra and $a, b, c \in A$. Then,*

(i) $a \to a = b \to b = 1$.

(ii) *The relation $\leq$ defined by $a \leq b$ iff $a \to b = 1$, is a partial order with maximum 1 and minimum 0.*

(iii) $a \to b = \neg b \to \neg a$.

(iv) $a \to (b \to c) = b \to (a \to c)$.

(v) $\neg\neg a = a$ *and* $a \to 0 = \neg a$.

(vi) $\langle A, *, 1\rangle$ *is a commutative monoid.*

(vii) $(a * b) \to c = a \to (b \to c)$.

(viii) *The pair* $\langle *, \to\rangle$ *is residuated, i.e.,* $a * b \leq c$ *iff* $b \leq a \to c$.

(ix) $a^2 \leq a^3$.

(x) $\langle A, \wedge, \vee\rangle$ *is a lattice with order* $\leq$.

(xi) $(a \vee b)^2 \leq a^2 \vee b^2$.

Proof. (i). Follows from the fact that $\mathcal{S}$ is implicative.
(ii). By $\mathsf{E}(\mathrm{A}\,2)$ we have that 0 is the minimum element with respect to the order $\leq$. The rest easily follows from the fact that $\mathcal{S}$ is implicative.
(iii). Follows from $\mathsf{E}(\mathrm{A}\,5)$ and item (ii) above.
(iv). By $\mathsf{Q}(\mathrm{P})$ and (ii) above, we have that $d \leq a \to (b \to c)$ implies $d \leq b \to (a \to c)$ for all $d \in A$. Then, taking $d = a \to (b \to c)$, we have $a \to (b \to c) \leq b \to (a \to c)$ which easily implies the desired result.
(v). $\neg\neg a = a$ follows from item (ii) above together with $Q(\neg\neg\mathrm{l})$ and $Q(\neg\neg\mathrm{r})$. By item (iii) above, $a \to 0 = \neg 0 \to \neg a = 1 \to \neg a = \neg a$. The last equality holds because, on the one hand, by $\mathsf{Q}(\to \mathrm{l})$ we have that $1 = 1$ and $\neg a \leq \neg a$ implies $1 \to \neg a \leq \neg a$. On the other, by item (i) we have $\neg a \to \neg a = 1$ and so we can apply $\mathsf{Q}(\to \mathrm{r})$ to obtain $1 \to (\neg a \to \neg a) = 1$. By item (iv), we have $1 \to (\neg a \to \neg a) = \neg a \to (1 \to \neg a)$, hence we conclude that $\neg a \to (1 \to \neg a) = 1$ and so, by item (ii), $\neg a \leq 1 \to \neg a$.
(vi). For commutativity, using item (iii) and (v) above, we have $a * b = \neg(a \to \neg b) = \neg(\neg\neg b \to \neg a) = \neg(b \to \neg a) = b * a$. For associativity, using (iii), (v), $\mathsf{Q}(\neg\neg\mathrm{r})$ and $\mathsf{Q}(\neg\neg\mathrm{l})$, we have $(a * b) * c = \neg(\neg(a \to \neg b) \to \neg c)) = \neg(\neg\neg c \to \neg\neg(a \to \neg b)) = \neg(c \to (a \to \neg b)) = \neg(a \to (c \to \neg b)) = \neg(a \to (b \to \neg c)) = \neg(a \to \neg\neg(b \to \neg c)) = a * (b * c)$. As to 1 being the neutral element, using item (v) above, we have $a * 1 = a * \neg 0 = \neg(a \to \neg\neg 0) = \neg(a \to 0) = \neg\neg a = a$.
(vii). Using items (ii), (iii), (v) and (vi) above, we have $(a * b) \to c = \neg(a \to \neg b) \to c = \neg c \to \neg\neg(a \to \neg b) = \neg c \to (a \to \neg b) = a \to (\neg c \to \neg b) = a \to (\neg\neg b \to \neg\neg c) = a \to (b \to c)$.
(viii). By item (ii) above, we have $a * b \leq c$ iff $(a * b) \to c = 1$ iff, by item (vii), $a \to (b \to c) = 1$ iff, by (vi), $b \to (a \to c) = 1$ iff, by (ii) again, $b \leq a \to c$.
(ix). By $\mathsf{Q}(\mathrm{C})$ we have that $a^3 \leq c$ implies $a^2 \leq c$ for all $c \in A$. Then, taking $c = a^3$, we have $a^2 \leq a^3$.
(x). We check that $a \wedge b$ is the infimum of $\{a, b\}$ with respect to $\leq$. Firstly, we have $a \wedge b \leq a$ and $a \wedge b \leq b$ by $\mathsf{Q}(\wedge\mathrm{l}1)$, $\mathsf{Q}(\wedge\mathrm{l}2)$ and item (ii) above. Then, assuming $c \leq a$ and $c \leq b$, we have $c \leq a \wedge b$ by $\mathsf{Q}(\wedge\mathrm{r})$. A similar reasoning, using $\mathsf{Q}(\vee\mathrm{r}1)$, $\mathsf{Q}(\vee\mathrm{r}2)$ and $\mathsf{Q}(\vee\mathrm{l}1)$, shows that $a \vee b$ is the supremum of $\{a, b\}$.
(xi). By (x) we have that $a^2 \leq a^2 \vee b^2$ and $b^2 \leq a^2 \vee b^2$. Hence, by item (viii), we have $a \leq a \to (a^2 \vee b^2)$ and $b \leq b \to (a^2 \vee b^2)$. By item (ii) we have then $a \to (a \to (a^2 \vee b^2) = b \to (b \to (a^2 \vee b^2) = 1$, hence we can use $\mathsf{Q}(\vee\mathrm{l}2)$ to obtain $(a \vee b) \to ((a \vee b) \to (a^2 \vee b^2) = 1$. Then item (ii) and (viii) give us $(a \vee b)^2 \leq a^2 \vee b^2$ as required. □

In the next section we introduce an equivalent presentation of $\mathcal{S}$-algebras which takes precisely the properties of Proposition 2.4 above as postulates.

2.2 Alternative presentation of $\mathcal{S}$-algebras

We start by recalling the following standard definition [3, p. 185].

Definition 2.5 A bounded integral residuated lattice is an algebra $\mathbf{A} = \langle A, \wedge, \vee, *, \to, 0, 1\rangle$ of type $\langle 2, 2, 2, 2, 0, 0\rangle$ such that:

(i) $\langle A, \wedge, \vee, 0, 1\rangle$ is a bounded lattice with ordering $\leq$, minimum element 0 and maximum 1.

(ii) $\langle A, *, 1\rangle$ is a commutative monoid.

(iii) $\langle *, \to\rangle$ form a residuated pair: $a * b \leq c$ iff $a \leq b \to c$ for all $a, b, c \in A$.

Letting $\neg x := x \Rightarrow 0$, we say that a residuated lattice is *involutive* [3, p. 186] when $\neg\neg a = a$ and $a \to b = \neg b \to \neg a$.

Definition 2.6 An $\mathcal{S}'$-algebra is a bounded involutive commutative integral residuated lattice that additionally satisfies the equations:

(i) $x^2 \leq x^3$

(ii) $(x \vee y)^2 \leq x^2 \vee y^2$.

Since all involutive residuated lattices form an equational class [3, Theorem 2.7], it is obvious that $\mathcal{S}'$-algebras are also an equational class. By Proposition 2.4, we immediately have the following:

Proposition 2.7 *Every $\mathcal{S}$-algebra is an $\mathcal{S}'$-algebra.*

For the converse inclusion, one needs to check that $\mathcal{S}'$-algebras satisfies all (quasi)equations introduced in Definition 2.3.

Proposition 2.8 *Every $\mathcal{S}'$-algebra is an $\mathcal{S}$-algebra.*

Besides what we said earlier, an advantage of the presentation given in Definition 2.6 is that it makes it straightforward to check that, for instance, the three-element MV-algebra [5] is a model of Nelson's logic $\mathcal{S}$. This in turn allows one to prove that the formulas which Nelson claims not to be derivable in $\mathcal{S}$ [4, p. 213] are actually not valid.

3 A finite Hilbert-style calculus for $\mathcal{S}$

In this section we introduce a finite Hilbert-style calculus (which is an extension of the *Full Lambek logic with exchange rule* FL_e of [3]) that is algebraizable with respect to the class of $\mathcal{S}'$-algebras. We will thus indirectly obtain an equivalence between our calculus and Nelson's.

The logic $\mathcal{S}' = \langle \mathbf{Fm}, \vdash_{\mathcal{S}'}\rangle$ is the sentential logic in the language $\langle \wedge, \vee, \to, *, \neg, \bot\rangle$ defined by the Hilbert-style calculus with the following axiom schemata and modus ponens (from φ and $\varphi \to \psi$, infer ψ) as the only rule:

(i) $\varphi \to \varphi$

(ii) $(\varphi \to \psi) \to ((\gamma \to \varphi) \to (\gamma \to \psi))$
(iii) $(\varphi \to (\psi \to \gamma)) \to (\psi \to (\varphi \to \gamma))$
(iv) $(\varphi * \psi) \to (\varphi \wedge \psi)$
(v) $(\varphi \wedge \psi) \to \varphi$
(vi) $(\varphi \wedge \psi) \to \psi$
(vii) $((\varphi \to \psi) \wedge (\varphi \to \gamma)) \to (\varphi \to (\psi \wedge \gamma))$
(viii) $\varphi \to (\varphi \vee \psi)$
(ix) $\psi \to (\varphi \vee \psi)$
(x) $((\varphi \to \psi) \wedge (\gamma \to \psi)) \to ((\varphi \vee \gamma) \to \psi)$
(xi) $\varphi \to (\psi \to (\psi * \varphi))$
(xii) $(\varphi \to (\psi \to \gamma)) \to ((\psi * \gamma) \to \gamma)$
(xiii) $\bot \to \varphi$
(xiv) $\varphi \to (\psi \to \varphi)$
(xv) $\varphi^2 \to \varphi^3$
(xvi) $(\varphi \vee \psi)^2 \to \varphi^2 \vee \psi^2$
(xvii) $(\neg\varphi \to \neg\psi) \leftrightarrow (\psi \to \varphi)$
(xviii) $\neg\neg\varphi \leftrightarrow \varphi$.

As before, $\varphi \leftrightarrow \psi$ abbreviates $(\varphi \to \psi) \wedge (\psi \to \varphi)$, while the connective $*$ is here taken as primitive.

Axioms 1–12 of our calculus are essentially those of FL_e as presented in [3, Figure 2.9] (with some simplifications due to the fact that we also have *weakening*, corresponding to integrality of the algebras), which is algebraizable with respect to the class of commutative residuated lattices [3, Theorem 2.27]. Algebraizability is preserved by axiomatic extensions, and since we have expanded FL_e only with a new constant $\bot$, checking that algebraizability is preserved by our expansion is trivial [1, p. 71]. Putting these considerations together we can state the following results.

Theorem 3.1 *The calculus $\mathcal{S}'$ is algebraizable (with the same translations as $\mathcal{S}$) with respect to the class of $\mathcal{S}'$-algebras.*

Corollary 3.2 *$\mathcal{S}$ and $\mathcal{S}'$ define the same logic.*

As mentioned in the abstract, Nelson introduced two other better-known logics, which are also algebraizable with respect to classes of residuated structures ($\mathcal{N}3$-lattices and $\mathcal{N}4$-lattices). The question then arises of what is precisely the relation between $\mathcal{S}$ and these other logics, or (equivalently) between $\mathcal{S}$-algebras and $\mathcal{N}3$ and $\mathcal{N}4$-lattices. Looking at the algebras, one immediately sees that $\mathcal{N}4 \not\subseteq \mathcal{S}$ and $\mathcal{N}3 \not\subseteq \mathcal{S}$, both are distributive with respect to $\{\wedge, \vee\}$, while $\mathcal{S}$ is not. Can be proved that $\mathcal{N}3$-algebras satisfies the definition 2.6, thus $\mathcal{S} \subseteq \mathcal{N}3$. Now, in $\mathcal{S}$-algebras the term $\varphi \to \varphi$ is an algebraic constant, in $\mathcal{N}4$ is not, like this $\mathcal{S} \not\subseteq \mathcal{N}4$.

References

[1] J. Font, "Abstract Algebraic Logic. An Introductory Textbook", Studies in Logic, College Publications, **60** (2016).

[2] N. Galatos and J. G. Raftery, *Adding involution to residuated structures*, Studia Logica, 77(2):181-207, (2004).

[3] N. Galatos, P. Jipsen, T. Kowalski, and H. Ono, "Residuated Lattices: an algebraic glimpse at substructural logics", Studies in Logic and the Foundations of Mathematics. Elsevier, Amsterdam, **151** (2007).

[4] N. Nelson, "Negation and separation of concepts in constructive systems, in "Constructivity in Mathematics. Proceedings of the colloquium held at Amsterdam, **39**, ed. Arend Heyting, 208-225, North-Holland, (1959).

[5] R. Cignoli, I. M. L. D'Ottaviano, and D. Mundici, "Algebraic foundations of many-valued reasoning", Trends in Logic **7**, Studia Logica Library. Kluwer Academic Publishers, Dordrecht, 2000.

[6] W. J. Blok and D. Pigozzi, "Algebraizable logics", Mem. Amer. Math. Soc. A.M.S., Providence, Jan. **396** (1989).

Rewriting Logic from a ρLog Point of View

Mauricio Ayala-Rincón

Department of Computer Science
University of Brasilia, Brazil

Besik Dundua

Ilia Vekua Institute of Applied Mathematics
Ivane Javakhishvili Tbilisi State University, Georgia

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria

Mircea Marin

Department of Computer Science
West University of Timişoara, Romania

Abstract

Rewriting logic is a well-known logic that emerged as an adequate logical and semantic framework for the specification of languages and systems. ρLog is a calculus for rule-based programming with labeled rules. Its expressive power stems from the usage of a fragment of higher-order logic (e.g., sequence variables, and function variables) to express atomic formulas. Its adequacy as a computational model for rule-based programming is derived from theoretical results concerning E-unification and E-matching in the fragment of logic adopted by ρLog.
In this paper we choose a fragment of the ρLog calculus and argue that it can be used to perform deduction in rewriting logic. More precisely, we define a mapping between the entailment systems of rewriting logic and ρLog for which the conservativity theorem holds. It implies that, like rewriting logic, ρLog also can be used as a logical and semantic framework.

1 Introduction

Rewriting logic [19] emerged as a simple computational logic based on the use of rewrite theories to represent with great generality (1) various models of computation (concurrency, programming languages, etc.), and (2) logical deduction. For computation, it represents states by equivalence classes in an equational theory, and local concurrent transitions by rewrite rules. For deductive purposes, it can represent formula-based data structures (e.g., sequents or sets of formulas) by terms, and the inference rules of the logic by conditional rewrite rules. The rewrite theories, which

are at the core of this logic, provide an adequate representation for a wide variety of applications, including automated deduction, software and hardware specification and verification, security, real-time and cyber-space systems, probabilistic systems, bioinformatics, and chemical systems. (See [19] for a convincing account.) Rewriting logic is the theoretical basis of Maude [3], a powerful reflective language with wide range of applications.

In [15], the authors described rewriting logic as a logical and semantic framework. They showed that it has a flexibility to represent in a natural way many other logics, maintaining the direct correspondence between proofs in object logics and proofs in rewriting logic (as the framework logic). This correspondence is often conservative, given by means of maps of logics, so that an implementation of the object logic is directly supported by an implementation of rewriting logic. Besides, the authors explored similarities of rewriting logic with Milner's CCS, concurrent object-oriented programming, and structural operational semantics.

An interesting refinement of rewriting logic, inspired by the use of rewriting logic as a logical framework for deduction, was to make a clear *separation of concerns* between the specification of the inference system and the heuristics which guide the way in which rules are applied. This point of view introduced the usage of strategy languages to define theory transformations parameterized by strategy modules [16].

ρLog [13,14] is a system for rule-based programming with labeled rules based on a calculus which makes all the ingredients of rewriting logic explicit. Terms, conditional rewrite rules, and strategies that specify the heuristics which guide the way in which rules are applied, can all be explicitly represented in its syntax. A novelty of ρLog is its definition in a fragment of logic with sequence variables, function variables, context variables, and membership constraints for their bindings in the rewrite process. These capabilities make the rule-based specifications of ρLog natural and concise. The calculus served as the basis for a strategy-based programming tool [12,7] and has found applications in constraint logic programming [5], XML transformation and Web reasoning [4], modeling rewriting strategies [6], in extraction of frequent patterns from data mining workflows [20], and for automatic derivation of multiscale models of arrays of micro- and nanosystems [2].

In this paper we choose a fragment of the ρLog calculus and argue that it can be used to perform deduction in rewriting logic. More precisely, we define a mapping between the entailment systems of rewriting logic and ρLog for which the conservativity theorem holds. It implies that, like rewriting logic, ρLog also can be used as a logical and semantic framework.

The paper is organized as follows: In Section 2 we review syntax and semantics of rewriting logic. ρLog is introduced in Section 3. Section 4 is the main part of the paper, where the mapping between the entailment systems of these two formalisms is defined. Section 5 concludes.

2 Syntax and Inference System of Rewriting Logic

In this section, we mainly follow the description of rewriting logic as it is given in [18]. The syntax of rewriting logic is given by *signatures*, which are pairs (F, E) of a set of ranked function symbols F and a set of equations E. Given a countable

set of variables V, terms over F and V are defined in the usual way:

$$\mathsf{t} ::= \mathsf{x} \mid \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where $\mathsf{x} \in \mathsf{V}$ and $\mathsf{f} \in \mathsf{F}$ is n-ary. The set of terms over F and V is denoted by $\mathcal{T}(\mathsf{F}, \mathsf{V})$. The letters $\mathsf{t}, \mathsf{r}, \mathsf{s}$ and u are used to denote its elements, while $\mathsf{x}, \mathsf{y}, \mathsf{z}$ stand for variables.

The equivalence class of a term t modulo E is denoted by $[\mathsf{t}]_{\mathsf{E}}$. The subscript E is usually omitted, when it causes no confusion. The set of E-equivalence classes of terms from $\mathcal{T}(\mathsf{F}, \mathsf{V})$ is denoted by $\mathcal{T}_{\mathsf{E}}(\mathsf{F}, \mathsf{V})$.

Given a signature (F, E), the considered *sentences* are sequents $[\mathsf{t}] \longrightarrow [\mathsf{r}]$.

A *substitution* σ is a mapping from variables to terms such that all but finitely many variables are mapped to themselves. Each substitution σ is represented as a finite set of pairs $\{\mathsf{x}_1 \mapsto \sigma(\mathsf{x}_1), \ldots, \mathsf{x}_n \mapsto \sigma(\mathsf{x}_n)\}$ where the x's are all the variables for which $\sigma(\mathsf{x}_i) \neq \mathsf{x}_i$.

A rewrite theory $\mathcal{R}$ is a 4-tuple $\mathcal{R} := (\mathsf{F}, \mathsf{E}, \mathsf{L}, \mathsf{R})$, where F and E are sets of function symbols, L is a set of labels, and R is a set of conditional rewrite rules. The latter are defined as pairs of a label and a nonempty sequence of pairs of E-equivalence classes of terms from $\mathcal{T}(\mathsf{F}, \mathsf{V})$. Usually, a rewrite rule of the form $(\mathsf{l}, ([\mathsf{t}_0], [\mathsf{r}_0])([\mathsf{t}_1], [\mathsf{r}_1]) \cdots ([\mathsf{t}_n], [\mathsf{r}_n]))$ is written as

$$\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0] \text{ if } [\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n],$$

where $[\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \ldots \wedge [\mathsf{t}_n] \to [\mathsf{t}_n]$ is called the condition of the rule.

A rewrite theory $\mathcal{R}$ *entails* a sequent $[\mathsf{t}] \longrightarrow [\mathsf{r}]$, written $\mathcal{R} \vdash [\mathsf{t}] \longrightarrow [\mathsf{r}]$, iff $[\mathsf{t}] \longrightarrow [\mathsf{r}]$ can be proved by finite application of the following four inference rules:

Reflexivity: For each $[\mathsf{t}] \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$,

$$\frac{}{[\mathsf{t}] \longrightarrow [\mathsf{t}]}.$$

Congruence: For each $\mathsf{t}_1, \ldots, \mathsf{t}_n, \mathsf{r}_1, \ldots, \mathsf{r}_n \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$ and $f \in \mathsf{F}$ with the arity $n \geqslant 1$,

$$\frac{[\mathsf{t}_1] \longrightarrow [\mathsf{r}_1] \quad \cdots \quad [\mathsf{t}_n] \longrightarrow [\mathsf{r}_n]}{[f(\mathsf{t}_1, \ldots, \mathsf{t}_n)] \longrightarrow [f(\mathsf{r}_1, \ldots, \mathsf{r}_n)]}.$$

Replacement: For each rule $\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0]$ if $[\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n] \in R$ and for each terms $\mathsf{s}_1, \ldots, \mathsf{s}_k, \mathsf{u}_1, \ldots, \mathsf{u}_k \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$.

$$\frac{[\mathsf{s}_1] \longrightarrow [\mathsf{u}_1] \quad \cdots \quad [\mathsf{s}_k] \longrightarrow [\mathsf{u}_k] \quad [\mathsf{t}_1\sigma] \longrightarrow [\mathsf{r}_1\sigma] \quad \cdots \quad [\mathsf{t}_n\sigma] \longrightarrow [\mathsf{r}_n\sigma]}{[\mathsf{t}_0\sigma] \longrightarrow [\mathsf{r}_0\vartheta]},$$

where the set of variables occurring in the rule is $\{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$, and the substitutions are $\sigma = \{\mathsf{x}_1 \mapsto \mathsf{s}_1, \ldots, \mathsf{x}_k \mapsto \mathsf{s}_k\}$ and $\vartheta = \{\mathsf{x}_1 \mapsto \mathsf{u}_1, \ldots, \mathsf{x}_k \mapsto \mathsf{u}_k\}$.

Transitivity: For each $[\mathsf{t}], [\mathsf{r}], [\mathsf{s}] \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$:

$$\frac{[\mathsf{t}] \longrightarrow [\mathsf{s}] \qquad [\mathsf{s}] \longrightarrow [\mathsf{r}]}{[\mathsf{t}] \longrightarrow [\mathsf{r}]}.$$

The entailment relation $\mathcal{R} \vdash [t] \longrightarrow [r]$ is defined to model the concurrent rewriting of $[t]$ to $[r]$, using the rewrite rules of $\mathcal{R}$. When $[t]$ is used to specify the rules of change in a concurrent system, an entailed sequent $[t] \longrightarrow [r]$ has the intended reading "$[t]$ becomes $[r]$." Concurrent rewriting, as it was shown in [18], actually coincides with deduction in rewriting logic.

The version of rewriting logic described above does not contain sorts for simplicity. What we are interested in this paper, however, is rewriting logic with ordered sorts. The notions defined above are easily transferred to the order-sorted case, provided that the signature satisfies a simple technical property called pre-regularity, which guarantees the existence of the least sort for each term. We briefly recall the main notions of ordered signatures and theories here, slightly adjusting them to our terminology. For details one can see, e.g., [8].

In order-sorted setting, in the role of F we have an alphabet, a triple $(B, \leqslant, S)$, where B is called the set of basic sorts, S is a $B^* \times B$-sorted set of sets of function symbols $\{F_{w,b} \mid w \in B^*, b \in B\}$, B is partially ordered by the ordering $\leqslant$, and the function symbols satisfy the following monotonicity condition:

$$f \in F_{w_1,b_1} \cap F_{w_2,b_2} \text{ and } w_1 \leqslant w_2 \text{ imply } b_1 \leqslant b_2.$$

When $f \in F_{w,b}$, we say that w is the arity of f and b is the result sort of f. When w is the empty word, then f is called a constant. The monotonicity condition excludes overloaded constants.

We assume that the set of variables is also sorted, which means that $V = \{V_b \mid b \in B\}$ is a family of disjoint sets V_b of variables for each $b \in B$. The set of order-sorted terms of sort $b \in B$ over $F = (B, \leqslant, S)$, denoted $\mathcal{T}_b(F, V)$, is defined as the least set satisfying the following properties:

- $V_b \subseteq \mathcal{T}_b(F, V)$.
- Let λ be the empty word of sorts. Then $F_{\lambda,b} \subseteq \mathcal{T}_b(F, V)$.
- $\mathcal{T}_{b'}(F, V) \subseteq \mathcal{T}_b(F, V)$ if $b' \leqslant b$.
- If $f \in F_{w,b}$ where $w = b_1 \cdots b_n \neq \lambda$ and $t_i \in \mathcal{T}_{b_i}(F, V)$ for all $1 \leqslant i \leqslant n$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_b(F, V)$.

The terms defined in this way might have different, even incomparable sorts. It has some unpleasant consequences (e.g., the generated term algebra is not initial, see [8]). However, with the above mentioned property of pre-regularity this problem disappears. The alphabet $(B, \leqslant, S)$ is called pre-regular iff the following property is satisfied: Let $w_0 \in B^*$. Then for any $w_1 \in B^*$ with $w_0 \leqslant w_1$ and $f \in F_{w_1,b_1}$, there is a least sort $b \in B$ such that $w_0 \leqslant w_1$ and $f \in F_{w,b}$ for some $w \in B^*$. Goguen and Meseguer in [8] proved that any term built over a pre-regular alphabet has the least sort. Sides of equalities are assumed to belong to a set of terms of the same sort.

We write $f : w \to b$ if $f \in F_{w,b}$.

Example 2.1 Let $\mathcal{R} = (F, E, L, R)$ be an order-sorted rewrite theory with two basic sorts Nat and Tree, ordered as Nat $<$ Tree. The signature F contains sorted function symbols $0 : \lambda \to \mathsf{Nat}$, $+ : \mathsf{Nat}\ \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{rev} : \mathsf{Tree} \to \mathsf{Tree}$, and $\bowtie$: $\mathsf{Tree}\ \mathsf{Tree} \to \mathsf{Tree}$. The set of equations E contains the commutativity axiom for $+$, In L there are the labels l_1, l_2, l_3, l_4, and the set R consists of the following four

rules:

$$\mathsf{l}_1 : [\mathsf{x}] + [0] \to [\mathsf{x}].$$
$$\mathsf{l}_2 : [\mathsf{suc}(\mathsf{x}) + \mathsf{suc}(\mathsf{y})] \to [\mathsf{suc}(\mathsf{suc}(\mathsf{x} + \mathsf{y}))].$$
$$\mathsf{l}_3 : [\mathsf{rev}(\mathsf{x})] \to [\mathsf{x}].$$
$$\mathsf{l}_4 : [\mathsf{rev}(\mathsf{x} \bowtie \mathsf{y})] \to [\mathsf{rev}(\mathsf{y}) \bowtie \mathsf{rev}(\mathsf{x})].$$

3 The ρLog Calculus

In this section we describe a fragment of ρLog calculus [13] that is relevant for our goal: to express the deduction system of rewriting logic.

The ρLog signature $\mathcal{F}$ consists of unranked function symbols. The symbols f, g, h, a, b, and c are used to denote them. The countably infinite set of variables $\mathcal{V}$ is split into three disjoint subsets: individual variables $\mathcal{V}_{\text{Ind}}$, whose elements are denoted by letters x, y, z; sequence variables $\mathcal{V}_{\text{Seq}}$, denoted by $\overline{x}, \overline{y}, \overline{z}$; and function variables $\mathcal{V}_{\text{Fun}}$, usually written as X, Y, Z. As usual, it is assumed that $\mathcal{F} \cap \mathcal{V} = \varnothing$.

Definition 3.1 The set of *terms over* $\mathcal{F}$ *and* $\mathcal{V}$, denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the set of *term sequences over* $\mathcal{F}$ *and* $\mathcal{V}$, denoted by $\mathcal{S}(\mathcal{F}, \mathcal{V})$, are the least sets satisfying the properties:

- $\mathcal{V}_{\text{Ind}} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$.
- $\epsilon \in \mathcal{S}(\mathcal{F}, \mathcal{V})$, where ϵ denotes the empty sequence of terms and sequence variables.
- $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$,[1] $n \geqslant 1$, if $s_i \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{V}_{\text{Seq}}$ for each $1 \leqslant i \leqslant n$.
- $f(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, if $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.
- $X(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, if $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.

Note that $\mathcal{T}(\mathcal{F}, \mathcal{V}) \subseteq \mathcal{S}(\mathcal{F}, \mathcal{V})$. In other words, we do not distinguish between a term and a singleton term sequence. Terms of the form $a(\epsilon)$ are abbreviated with a. For readability, we may write sequences within parentheses, usually when there is more than one element in the sequence.

We denote terms by t, r, terms or sequence variables by s, u, and sequences (of terms or sequence variables) by $\tilde{s}, \tilde{u}$.

If $\tilde{s} = (s_1, \ldots, s_n)$ and $\tilde{u} = (u_1, \ldots, u_m)$, $n, m \geqslant 0$, we slightly overload the comma, writing $(\tilde{s}, \tilde{u})$ for the sequence $(s_1, \ldots, s_n, u_1, \ldots, u_m)$. Obviously, when $n = 0$, i.e., when $\tilde{s} = \epsilon$, then $(\tilde{s}, \tilde{u}) = \tilde{u}$. Similarly, for $\tilde{u} = \epsilon$ we have $(\tilde{s}, \tilde{u}) = \tilde{s}$.

The set of variables of a sequence $\tilde{s}$ is denoted by $var(\tilde{s})$. We call $\tilde{s}$ ground if $var(\tilde{s}) = \varnothing$. These notions extend to sets of term sequences, etc.

3.1 Substitutions and Matching Problems

A *substitution* σ is a mapping $\sigma : \mathcal{V} \to \mathcal{S}(\mathcal{F}, \mathcal{V}) \cup \mathcal{F} \cup \mathcal{V}_{\text{Fun}}$ such that the following properties are satisfied:

[1] Note that $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$ means that the sequence $s_1, \ldots, s_n$ of terms and sequence variables belongs to $\mathcal{S}(\mathcal{F}, \mathcal{V})$. It should *not* be read as $s_1 \in \mathcal{S}(\mathcal{F}, \mathcal{V}), \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.

- for all $x \in \mathcal{V}_{\text{Ind}}$, $\sigma(x) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$,
- for all $\overline{x} \in \mathcal{V}_{\text{Seq}}$, $\sigma(\overline{x}) \in \mathcal{S}(\mathcal{F}, \mathcal{V})$,
- for all $X \in \mathcal{V}_{\text{Fun}}$, $\sigma(X) \in \mathcal{F} \cup \mathcal{V}_{\text{Fun}}$, and
- all but finitely many variables are mapped to themselves.

Substitutions are denoted by lowercase Greek letters σ, ϑ, and ε, where ε stands for the identity substitution. A substitution σ can apply to a term t or a sequence $\tilde{s}$ and result in the *instances* (under σ): $t\sigma$ of t and $\tilde{s}\sigma$ of $\tilde{s}$. They are defined as $x\sigma = \sigma(x)$, $f(\tilde{s})\sigma = f(\tilde{s}\sigma)$, $X(\tilde{s})\sigma = \sigma(X)(\tilde{s}\sigma)$, $\overline{x}\sigma = \sigma(\overline{x})$ and $(s_1, \ldots, s_n)\sigma = (s_1\sigma, \ldots, s_n\sigma)$. For instance, if $\sigma = \{\overline{x} \mapsto (g(a), \overline{y}), \overline{y} \mapsto \epsilon, z \mapsto a, X \mapsto f\}$, then $(\overline{x}, X(\overline{x}, z), b, \overline{y}, z)\sigma = (g(a), \overline{y}, f(g(a), \overline{y}, a), b, a)$.

The notion of substitution composition is defined in the standard way. (See, e.g., [1].) We use juxtaposition $\sigma\vartheta$ for composition of σ with ϑ

Matching with sequence variables is finitary, see, e.g., [9,10]. For instance, if $t = \{f(\overline{x}, b, \overline{y})$ and $r = f(b, f(a, c), b)\}$, then the complete set of matchers of t to r consists of the following two solutions: $\sigma_1 = \{\overline{x} \mapsto \epsilon, \overline{y} \mapsto (f(a, c), b)\}$ and $\sigma_2 = \{\overline{x} \mapsto (b, f(a, c)), \overline{y} \mapsto \epsilon\}$. If f is orderless (a generalization of commutativity for unranked function symbols), then we have more solutions: $\sigma_3 = \{\overline{x} \mapsto \epsilon, \overline{y} \mapsto (b, f(a, c))\}$, $\sigma_4 = \{\overline{x} \mapsto (f(a, c), b), \overline{y} \mapsto \epsilon\}$, $\sigma_5 = \{\overline{x} \mapsto b, \overline{y} \mapsto f(a, c)\}$, and $\sigma_6 = \{\overline{x} \mapsto f(a, c), \overline{y} \mapsto b\}$. These six substitutions form the complete set of matchers modulo the orderless theory for f.

3.2 Definite Fragment of ρLog

ρLog atoms are triples $(s, \tilde{t}, \tilde{r})$, usually written as a labeled rule $s :: \tilde{t} \Rightarrow \tilde{r}$, where s is called a strategy term, and $\tilde{t}$ and $\tilde{r}$ are term sequences. The intuition is that s denotes a transformation of $\tilde{t}$ into $\tilde{r}$.

In this paper we consider only definite ρLog programs (no negative literals involved) that are sets of nonnegative Horn clauses, constructed from ρLog atoms. The clauses are written as usual, e.g., $s_0 :: \tilde{t}_0 \Rightarrow \tilde{r}_0$ if $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$, $n \geqslant 0$. Goals are conjunctions of atoms, e.g., $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$. We often use the term *rule* when we refer to a ρLog clause. We also say that a clause defines the strategy in its head (i.e., the clause above defines the strategy s_0).

The inference system of our fragment of ρLog consists of two rules: one is resolution, and the other one is for the special strategy **id** (which, intuitively, denotes the identity transformation of a sequence to itself). Given a program $\mathcal{P}$ and a set of equations E, these rules are defined as follows (they should be read bottom up: To prove the query in the lower part, prove the query in the upper part.)

Resolution:

$$\frac{(s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n, \mathbf{id} :: \tilde{r}_0 \Rightarrow \tilde{u}, \mathcal{Q})\sigma}{s :: \tilde{t} \Rightarrow \tilde{u}, \mathcal{Q},}$$

where $s_0 :: \tilde{t}_0 \Rightarrow \tilde{r}_0$ if $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$ is a clause from $\mathcal{P}$, $s \neq \mathbf{id}$, and $s_0\sigma =_E s$, $\tilde{t}_0\sigma =_E \tilde{t}$.

Identity:

$$\frac{\mathcal{Q}\sigma}{\mathbf{id} :: \tilde{t} \Rightarrow \tilde{u},\ \mathcal{Q},}$$

where $\tilde{u}\sigma =_E \tilde{t}$.

Here we look at these inference rules as logical deduction rules.

There are some predefined ρLog strategies with fixed meaning, which are useful in the next section for the mapping from Rewriting Logic to ρLog:

- If s is a strategy term, then the strategy $\mathbf{map}(s) :: (t_1, \ldots, t_n) \Rightarrow (r_1, \ldots, r_n)$ succeeds iff each strategy $s :: t_i \Rightarrow r_i$, $1 \leqslant i \leqslant n$, succeeds.
- If $s_1, \ldots, s_n$ are strategy terms, then the strategy $\mathbf{choice}(s_1, \ldots, s_n) :: \tilde{t} \Rightarrow \tilde{r}$ succeeds iff at least one of the strategies $s_i :: \tilde{t} \Rightarrow \tilde{r}$, $1 \leqslant i \leqslant n$ succeeds.

Note that **map**, when realized as an extra inference rule for ρLog, can be used to perform transformations in parallel. It can also be specified within ρLog as a clause, doing transformations sequentially. Such an "internalization" of **map** is, in fact, pretty simple:

$\mathbf{map}(z) :: \epsilon \Rightarrow \epsilon.$
$\mathbf{map}(z) :: (x, \overline{x}) \Rightarrow (y, \overline{y})$ if $z :: x \Rightarrow y$, $\mathbf{map}(z) :: \overline{x} \Rightarrow \overline{y}$.

Similarly, it is also rather straightforward to specify the **choice** strategy as ρLog clauses:

$\mathbf{choice}(z, \overline{z}) :: \overline{x} \Rightarrow \overline{y}$ if $z :: \overline{x} \Rightarrow \overline{y}$.
$\mathbf{choice}(z, \overline{z}) :: \overline{x} \Rightarrow \overline{y}$ if $\mathbf{choice}(\overline{z}) :: \overline{x} \Rightarrow \overline{y}$.

The semantics of ρLog can be defined in the same way as it is done in logic programming, see, e.g., [11].

4 From Rewriting Logic to ρLog: Mapping Entailment Systems

The goal of this section is to illustrate that, via an appropriate mapping, deduction in rewriting logic can be modeled by deduction in ρLog. In other words, our goal is to define a mapping between what is called *entailment systems* [17] of rewriting logic and ρLog. By an entailment system of a logic one understands a triple consisting of the signature, set of sentences, and the entailment relation $\vdash$ that satisfies certain properties (reflexivity, monotonicity, transitivity, and $\vdash$-translation). The inference systems of both rewriting logic and the definite fragment of ρLog we consider here provide the entailment relation that satisfies those properties.

Hence, the goal is to define an *entailment system mapping* Φ from rewriting logic to ρLog such that the conservativity theorem holds. This theorem, formulated at the end of this section, states that a sequent is provable in rewriting logic with respect to a rewrite theory iff the image of the sequent under Φ is provable in ρLog with respect to a program obtained from the rewrite theory by Φ.

Hence, we start defining Φ for a rewrite theory $\mathcal{R} = (\mathsf{F}, \mathsf{E}, \mathsf{L}, \mathsf{R})$. We assume that $\mathcal{F}$ is split into five disjoint countable sets of symbols $\mathcal{F}_{\mathsf{F}}$, $\mathcal{F}_{\mathsf{V}}$, $\mathcal{F}_{\mathsf{S}}$, $\mathcal{F}_{\mathsf{L}}$, and $\mathcal{F}_{\rho}$, such that Φ for rewriting logic function symbols, variables, basic sorts, and labels is defined as follows:

- For each $\mathsf{f} \in \mathsf{F}$ we have a symbol $f \in \mathcal{F}_{\mathsf{F}}$, and

  $$\Phi(\mathsf{f}) = f.$$

- For each $\mathsf{x} \in \mathsf{V}$ there is a symbol $c_{\mathsf{x}} \in \mathcal{F}_{\mathsf{V}}$, and

  $$\Phi(\mathsf{x}) = \mathit{rlv}(c_{\mathsf{x}}),$$

  where rlv is a function symbol from $\mathcal{F}_{\rho}$. Hence, rewriting logic variables are mapped to ρLog ground terms tagged by the function symbol rlv.
- For each rewriting logic basic sort a there is a symbol $a \in \mathcal{F}_{\mathsf{S}}$, and

  $$\Phi(\mathsf{a}) = \mathit{rls}(a),$$

  where rls is a function symbol from $\mathcal{F}_{\rho}$. Hence, rewriting logic sort symbols are mapped to ρLog ground terms tagged by the function symbol rls.
- For each $\mathsf{l} \in \mathsf{L}$ there is a symbol $l \in \mathcal{F}_{\mathsf{L}}$, and

  $$\Phi(\mathsf{l}) = l.$$

The symbols from $\mathcal{F}_{\rho}$ will be also used in ρLog programs below.

Since ρLog is unsorted, we need to encode sort definitions and the subsort relation explicitly as clauses. This is done in the following way:

- For each pair of basic sorts a, b, related by the subsort relation $\leqslant$, Φ gives a clause

  $$\mathit{subsort\_basic} :: \mathit{rls}(a) \Rightarrow \mathit{rls}(b).$$

  Then the subsort relation is defined as follows:

  $$\begin{array}{l}\mathit{subsort} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(x).\\ \mathit{subsort} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(y) \text{ if}\\ \quad \mathit{subsort\_basic} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(z),\ \mathit{subsort} :: \mathit{rls}(z) \Rightarrow \mathit{rls}(y).\end{array}$$

- For each function symbol $\mathsf{f} : \mathsf{a}_1 \cdots \mathsf{a}_n \to \mathsf{b}$, Φ gives a clause

  $$\begin{array}{l}\mathit{sort\_def} :: f(x_1, \ldots, x_n) \Rightarrow \mathit{rls}(b) \text{ if}\\ \quad \mathit{sort} :: x_1 \Rightarrow \mathit{rls}(a_1), \ldots, \mathit{sort} :: x_n \Rightarrow \mathit{rls}(a_n),\end{array}$$

  where the strategy sort is defined as

  $$\begin{array}{l}\mathit{sort} :: x \Rightarrow \mathit{rls}(y) \text{ if}\\ \quad \mathit{sort\_def} :: x \Rightarrow \mathit{rls}(z),\ \mathit{subsort} :: \mathit{rls}(z) \Rightarrow \mathit{rls}(y).\end{array}$$

- We define a strategy *is_sorted* for terms:

$$is\_sorted :: x \Rightarrow true \text{ if } sort :: x \Rightarrow rls(y),$$

  where *true* is a function symbol. This clause says that a term is sorted if it has a sort.

Further, Φ is extended in a straightforward way to a mapping from $\mathcal{T}(\mathsf{F},\mathsf{V})$-terms, equations, and substitutions into $\mathcal{T}(\mathcal{F},\mathcal{V})$-terms, equations, and individual variable substitutions, respectively We define $\Phi([\mathsf{t}]) = \Phi(\mathsf{t})$.

For the set of rewrite rules R, the mapping Φ is defined as follows:

$$\begin{aligned}\Phi(\mathsf{R}) := \{&\Phi(rule) \mid rule \in \mathsf{R}\} \cup \\ &\{rwl\_choice :: \overline{x} \Rightarrow \overline{y} \text{ if } \mathbf{choice}(\Phi(\mathsf{l}_1),\ldots,\Phi(\mathsf{l}_m)) :: \overline{x} \Rightarrow \overline{y}\}.\end{aligned}$$

where $\mathsf{l}_1,\ldots,\mathsf{l}_m$ are all the labels of the rules in R, and Φ for the rules is defined below.

Before saying what the image of a rule $\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0] \text{ if } [\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n]$ under Φ is, we assume that $\Phi(\mathsf{l}) = l$, $\Phi([\mathsf{t}_i]) = t_i$, and $\Phi([\mathsf{r}_i]) = r_i$ for $0 \leqslant i \leqslant n$. Besides, let $\mathsf{x}_1,\ldots,\mathsf{x}_m$ be all rewriting logic variables in r_0 and for each $1 \leqslant i \leqslant m$, let c_{x_i} be the corresponding symbol from $\mathcal{F}_{\mathsf{V}}$. Then, by the definition of Φ, r_0 contains $rlv(c_{\mathsf{x}_i})$ in place of x_i. Let $c_1,\ldots,c_m$ be symbols from $\mathcal{F}_\rho$ that are fresh in the context, and denote by r_0' the term obtained from r_0 by replacing each $rlv(c_{\mathsf{x}_i})$, $1 \leqslant i \leqslant m$, by c_i. Then:

$$\begin{aligned}&\Phi(\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0] \text{ if } [\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n]) := \\ &\quad l :: y_0 \Rightarrow (c_1 \hookrightarrow s_1,\ldots,c_m \hookrightarrow s_m,\ r_0') \text{ if} \\ &\qquad match :: t_0 \ll y_0 \Rightarrow \overline{z}_0^\sigma, \\ &\qquad apply\_subst(\overline{z}_0^\sigma) :: t_1 \Rightarrow t_1',\ apply\_subst(\overline{z}_0^\sigma) :: r_1 \Rightarrow r_1', \\ &\qquad rwl\_inf :: t_1' \Rightarrow y_1, \\ &\qquad match :: r_1' \ll y_1 \Rightarrow \overline{z}_1^\sigma, \\ &\qquad apply\_subst(\overline{z}_0^\sigma,\overline{z}_1^\sigma) :: t_2 \Rightarrow t_2',\ apply\_subst(\overline{z}_0^\sigma,\overline{z}_1^\sigma) :: r_2 \Rightarrow r_2', \\ &\qquad \ldots, \\ &\qquad rwl\_inf :: t_n' \Rightarrow y_n, \\ &\qquad match :: r_n' \ll y_n \Rightarrow \overline{z}_n^\sigma, \\ &\qquad apply\_subst(\overline{z}_0^\sigma,\ldots,\overline{z}_n^\sigma) :: rlv(c_{\mathsf{x}_1}) \Rightarrow s_1, \\ &\qquad \ldots, \\ &\qquad apply\_subst(\overline{z}_0^\sigma,\ldots,\overline{z}_n^\sigma) :: rlv(c_{\mathsf{x}_m}) \Rightarrow s_m,\end{aligned}$$

where $\hookrightarrow$ is a function symbol from $\mathcal{F}_\rho$, used to model replacement pairs. Obviously, if r_0 is a ground term, then the sequence $c_1 \hookrightarrow rlv(c_{\mathsf{x}_1}),\ldots,c_n \hookrightarrow rlv(c_{\mathsf{x}_n})$ is empty and $r_0' = r_0$. *rwl_inf* is the strategy that corresponds to the inference in rewriting logic and is defined below.

The translation of r_0 into the sequence $(c_1 \hookrightarrow rlv(c_{\mathsf{x}_1}),\ldots,c_n \hookrightarrow rlv(c_{\mathsf{x}_n}),r_0')$ is a trick that will play its role with the Replacement inference rule, where the instances of variables in the right hand side of a rule are reduced. The intuition behind this

sequence is similar to the *let* construct in programming, and below (in the definition of the Replacement Rule) we will define a strategy that has a similar effect.

So far we have not taken into account the equational part of rewrite theories, i.e., the set E. Its translation can be dealt with in various ways. For instance, we can assume that it is incorporated into the matching mechanism of ρLog as a matching algorithm modulo E. This approach is feasible when matching modulo E is decidable and finitary. Or, if E induces a convergent rewrite system, then its image under Φ can be a set of ρLog rules, obtained in the similar way as $\Phi(\mathsf{R})$, but grouped under the name of one strategy (e.g., *reduce_by_equalities*) and in the inference step (i.e., in the strategy *rwl_inf* below) the terms before and after reduction by inference rules are brought to the normal form with respect to the strategy *reduce_by_equalities*. One can also consider a mixed variant (which is implemented in Maude), where matching modulo some equational theories are built-in, and the remaining set of equalities is convergent.

Basically, with this we have Φ defined for rewrite theories. In whatever way one deals with the rewriting logic equalities, we always need syntactic matching for the expressions translated in ρLog. Since the rewriting logic variables are mapped to ρLog ground terms, matching should be implemented explicitly:

$$
\begin{array}{l}
\mathit{match} :: (x \ll y) \Rightarrow \overline{x} \ \mathsf{if} \\
\quad \mathit{change\_tag}(\mathit{rlv} \mapsto \mathit{temp\_tag}) :: y \Rightarrow z, \\
\quad \mathbf{nf}(\mathbf{first\_one}(\mathit{finish}, \mathit{solved\_equation}, \mathit{variable\_elim}, \mathit{decomposition})) :: \\
\qquad (\mathit{mp}(x \ll z), \mathit{subst}) \Rightarrow \mathit{subst}(\overline{z}), \\
\quad \mathit{change\_tag}(\mathit{temp\_tag} \mapsto \mathit{rlv}) :: \mathit{subst}(\overline{z}) \Rightarrow \mathit{subst}(\overline{x}).
\end{array}
$$

The code above corresponds to the variant of matching algorithm when variables in the right hand side of the matching problem are replaced by temporary constants, then the matching rules are fired (first applicable, as long as possible), and in the computed matcher the introduced constants are mapped back to the original variables they replaced. The constructor function symbol for the matching problem is *mp*, and for the substitution *subst*.

The strategy *finish* below says that if the matching problem is empty, then the computed substitution should be returned. This corresponds to the success of matching. The other three strategies implement the standard matching rules.

$$
\begin{array}{l}
\mathit{finish} :: (\mathit{mp}, x) \Rightarrow x. \\
\mathit{solved\_equation} :: (\mathit{mp}(x \ll x, \overline{x}), \mathit{subst}(\overline{y})) \Rightarrow (\mathit{mp}(\overline{x}), \mathit{subst}(\overline{y})). \\
\mathit{variable\_elim} :: (\mathit{mp}(\mathit{rlv}(x) \ll y, \overline{x}), \mathit{subst}(\overline{y})) \Rightarrow \\
\quad (\mathit{mp}(\overline{x}_1), \mathit{subst}(\mathit{rlv}(x) \mapsto y, \overline{y}_1)) \ \mathsf{if} \\
\qquad \mathit{apply\_subst}(\mathit{rlv}(x) \mapsto y) :: \mathit{mp}(\overline{x}) \Rightarrow \mathit{mp}(\overline{x}_1), \\
\qquad \mathit{apply\_subst}(\mathit{rlv}(x) \mapsto y) :: \mathit{subst}(\overline{y}) \Rightarrow \mathit{subst}(\overline{y}_1). \\
\mathit{decomposition} :: (\mathit{mp}(F(\overline{x}_1) \ll F(\overline{x}_2), \overline{y}), x) \Rightarrow (\mathit{mp}(\overline{z}, \overline{y}), x) \ \mathsf{if} \\
\qquad \mathit{zip} :: (F(\overline{x}_1), F(\overline{x}_2)) \Rightarrow \overline{z}.
\end{array}
$$

The remaining strategies are auxiliary ones used in the rules or in the algorithm

control above:

$$zip := \mathbf{first\_one}(zip\_nonempty, zip\_empty).$$
$$zip\_nonempty :: (F(x_1, \overline{x}_1), F(x_2, \overline{x}_2)) \Rightarrow (x_1 \ll x_2, \overline{y}) \text{ if}$$
$$zip :: (F(\overline{x}_1), F(\overline{x}_2)) \Rightarrow \overline{y}.$$
$$zip\_empty :: (F, F) \Rightarrow \epsilon.$$

$$apply\_subst(\overline{x}) := \mathbf{first\_one}(apply\_subst\_basic(\overline{x}), apply\_subst\_rec(\overline{x})).$$
$$apply\_subst\_basic(\overline{x}, rlv(x) \mapsto y, \overline{y}) :: rlv(x) \Rightarrow y.$$
$$apply\_subst\_rec(\overline{x}) :: F(\overline{y}) \Rightarrow F(\overline{z}) \text{ if } \mathbf{map}(apply\_subst(\overline{x})) :: \overline{y} \Rightarrow \overline{z}.$$

$$change\_tag(F_1 \mapsto F_2) :=$$
$$\mathbf{first\_one}(change\_tag\_basic(F_1 \mapsto F_2), change\_tag\_rec(F_1 \mapsto F_2)).$$
$$change\_tag\_basic((F_1 \mapsto F_2)) :: F_1(x) \Rightarrow F_2(x).$$
$$change\_tag\_rec(F_1 \mapsto F_2) :: F(\overline{y}) \Rightarrow F(\overline{z}) \text{ if}$$
$$\mathbf{map}(change\_tag(F_1 \mapsto F_2)) :: \overline{y} \Rightarrow \overline{z}.$$

One could easily extend this algorithm to work, for instance, with commutative matching symbols. We would need to add only one rule, called commutative decomposition:

$$commutative\_decomposition :: (mp(F(x_1, y_1) \ll F(x_2, y_2), \overline{y}), x) \Rightarrow$$
$$(mp(x_1 \ll y_2, x_2 \ll y_1, \overline{y}), x) \text{ if}$$
$$is\_commutative :: F \Rightarrow true.$$

(The strategy *is_commutative* is assumed to be defined for each commutative function symbol, and it is a part of the translation of commutativity equations from E.) To make this rule work in the matching algorithm, we will need to replace the occurrence of *decomposition* in *match* above by the choice between *commutative_decomposition* and *decomposition*.

In a similar way, one could easily incorporate into ρLog equational matching algorithms in some other common theories, such as associativity, associativity-commutativity or their combinations with the unit element. (These are theories for which Maude also provides built-in matching algorithms.)

The final step in the construction of the mapping Φ is to define it for the inference rules of rewriting logic. They are translated into a set of ρLog as follows:

Reflexivity Rule in ρLog:

$$rwl\_refl :: x \Rightarrow x \text{ if } is\_sorted :: x \Rightarrow true.$$

Congruence Rule in ρLog:

$$rwl\_cong :: X(\overline{x}) \Rightarrow X(\overline{y}) \text{ if } \mathbf{map}(rwl\_inf) :: \overline{x} \Rightarrow \overline{y}.$$

Replacement Rule in ρLog:

$$rwl\_repl :: x \Rightarrow y \text{ if}$$

$rwl\_choice :: x \Rightarrow (\overline{y}, z)$,
$\mathbf{map}(reduce_{\hookrightarrow}) :: \overline{y} \Rightarrow \overline{z}$,
$let(\overline{z}) :: z \Rightarrow y$.

The strategy $reduce_{\hookrightarrow}$ is defined as

$reduce_{\hookrightarrow} :: x \hookrightarrow y \Rightarrow x \hookrightarrow z$ if $rwl\_inf :: y \Rightarrow z$.

The strategy *let* is defined as

$let(\overline{z}) :: x \Rightarrow y$ if $\mathbf{first\_one}(replace(\overline{z}), \mathbf{id}) :: x \Rightarrow y$.
$replace(\overline{z}_1, x \hookrightarrow y, \overline{z}_2) :: x \Rightarrow y$.
$replace(\overline{z}) :: X(\overline{x}) \Rightarrow X(\overline{y})$ if $\mathbf{map}(let(\overline{z})) :: \overline{x} \Rightarrow \overline{y}$.

Transitivity Rule in ρLog:

$rwl\_trans :: x \Rightarrow y$ if
$rwl\_inf :: x \Rightarrow z, rwl\_inf :: z \Rightarrow y$.

The main strategy is *rwl_inf*, which encodes the fact that an inference step in rewriting logic is made by the above mentioned inference rules (and guaranteeing well-sortedness of the involved terms):

Inference:

$rwl\_inf :: x \Rightarrow y$ if
$is\_sorted :: x \Rightarrow true$,
$\mathbf{choice}(rwl\_refl, rwl\_cong, rwl\_repl, rwl\_trans) :: x \Rightarrow y$,
$is\_sorted :: y \Rightarrow true$.

Hence, we constructed the translation mapping Φ from a rewriting logic theory $\mathcal{R}$ to the ρLog set of definite clauses $\Phi(\mathcal{R})$, and translated the inference rules of rewriting logic into ρLog definite clauses as well.

While the clauses for *rwl_refl*, *rwl_cong*, *rwl_trans* directly imitate the behavior of the corresponding inference rules of rewriting logic (and vice versa), the *rwl_repl* rule needs more explanation. For this purpose, we read the Replacement inference on page 3 bottom-up and see how proving the ρLog atom $rwl\_repl :: t_0\Phi(\sigma) \Rightarrow r_0\Phi(\vartheta)$ corresponds exactly to proving the rewriting logic sequent $[\mathsf{t}_0\sigma] \longrightarrow [\mathsf{r}_0\vartheta]$ by the Replacement inference, where σ and ϑ are substitutions from that rule.

Proving $rwl\_repl :: t_0\sigma \Rightarrow r_0\vartheta$ requires proving atoms in the body of the clause that defines *rwl_repl*. The first of them (call it **A1**), with the strategy *rwl_choice*, corresponds to finding a rule for the rewrite theory: It should be the one that has t_0 (or a term that equals t_0 modulo the set of equalities $\Phi(\mathsf{E})$) in its left hand side, and has the construction that corresponds to r_0 (or a term that is $\Phi(\mathsf{E})$-equal to r_0) in its right hand side. The construction consists of (i) a sequence of correspondences between fresh function symbols and $\Phi(\sigma)$-*instances* of variables of r_0 (this sequence is consumed by $\overline{y}$ in the clause and consists of terms of the form $c_i^x \hookrightarrow s_i$), and (ii) the term r_0' which is obtained from r_0 by replacing variables by those fresh atoms.

Note that proving **A1**, if the body of its clause is not empty, requires proving the $\Phi(\sigma)$-instance of that body, that is nothing else than the task of proving the sequents obtained from the σ-instance of the condition of the selected rule with the label l, i.e., those $[\mathsf{t}_i\sigma] \longrightarrow [\mathsf{r}_i\sigma]$ sequents in the upper part of the Replacement inference.

Next atom maps *rwl_inf* on the sequence $\overline{y}$, which means that $rwl\_inf :: c_i^x \hookrightarrow s_i \Rightarrow rhs_i$ should be proved for all elements $c_i^x \hookrightarrow s_i$ in the sequence. However, due to the fact that $\hookrightarrow$ does not appear in the left hand side of rules and c_i^x's were fresh, the rhs_i's should have a form $c_i^x \hookrightarrow u_i$ for some u_i's. But this corresponds to the proof of the sequent $[\mathsf{s}_i] \longrightarrow [\mathsf{u}_i]$ in the Replacement rule.

Finally, the strategy *let* puts each u_i in place of c_i^x in r_0', obtaining $r_0\Phi(\vartheta)$. It corresponds to the application of the substitution ϑ to r_0 in the Replacement rule.

The following result, called the conservativity theorem, connects deductions in rewriting logic to those in ρLog:

Theorem 4.1 *Given a rewrite theory $\mathcal{R}$, a sequent $[\mathsf{t}] \longrightarrow [\mathsf{r}]$ is provable in rewriting logic from $\mathcal{R}$ iff the atom $rwl\_inf :: \Phi(\mathsf{t}) \Rightarrow \Phi(\mathsf{r})$ is provable in ρLog from $\Phi(\mathcal{R})$.*

Proof. (Sketch) First, assume that the equational part of $\mathcal{R}$ is empty, i.e., we have the syntactic equality. We need to show that terms, sorts, subsort relation, equalities, rules, and inferences of rewriting logic are adequately represented in ρLog, but it follows directly from the construction of Φ. For instance, it can be immediately seen that b is a sort of a term t of rewriting logic iff $sort :: \Phi(\mathsf{t}) \Rightarrow \Phi(\mathsf{b})$ is proved in ρLog. The specification of matching in ρLog directly follows the rules of the algorithm. Based on the adequacy of sortedness and matching, we can see that adequacy is straightforward for the reflexivity, congruence, and transitivity inference rules. For the replacement rule, the proof is based on the reasoning above for *rwl_repl*.

As for non-empty equational theories, the result holds when equational matching can be effectively represented in ρLog. Essentially, it means that a terminating finitary algorithm should be available. In this case, we can reason similarly to the case when matching is syntactic. □

Example 4.2 At the end of this section, we see how the rewriting logic theory from Example 2.1 is translated into ρLog clauses. (The general part such as commutative matching and inference rules are the same as above.)

$$
\begin{aligned}
&l_1 :: z \Rightarrow (a \hookrightarrow s, a) \text{ if}\\
&\quad match :: rlv(a_{\mathsf{x}}) + 0 \ll z \Rightarrow \overline{z}^{\sigma},\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_{\mathsf{x}}) \Rightarrow s.\\
&l_2 :: z \Rightarrow (a_1 \hookrightarrow s_1, a_2 \hookrightarrow s_2, suc(suc(a_1 + a_2))) \text{ if}\\
&\quad match :: suc(rlv(a_x)) + suc(rlv(a_y)) \ll z \Rightarrow \overline{z}^{\sigma},\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_x) \Rightarrow s_1,\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_y) \Rightarrow s_2.\\
&l_3 :: z \Rightarrow (a \hookrightarrow s, a) \text{ if}\\
&\quad match :: rev(rlv(a_{\mathsf{x}})) \ll z \Rightarrow \overline{z}^{\sigma},
\end{aligned}
$$

$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_{\times}) \Rightarrow s.$

$l_4 :: z \Rightarrow (a_1 \hookrightarrow s_1, a_2 \hookrightarrow s_2, rev(a_2) \bowtie rev(a_1))$ if

$match :: rev(rlv(a_x) \bowtie rlv(a_y)) \ll z \Rightarrow \overline{z}^{\sigma},$

$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_x) \Rightarrow s_1,$

$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_y) \Rightarrow s_2.$

$subsort\_basic :: rls(a_N) \Rightarrow rls(a_T).$

$sort\_def :: x_1 + x_2 \Rightarrow rls(a_N)$ if

$sort :: x_1 \Rightarrow rls(a_N),$

$sort :: x_2 \Rightarrow rls(a_N).$

$sort\_def :: x_1 \bowtie x_2 \Rightarrow rls(a_T)$ if

$sort :: x_1 \Rightarrow rls(a_T),$

$sort :: x_2 \Rightarrow rls(a_T).$

$sort\_def :: suc(x) \Rightarrow rls(a_N)$ if

$sort :: x \Rightarrow rls(a_N).$

$sort\_def :: rev(x) \Rightarrow rls(a_T)$ if

$sort :: x \Rightarrow a_T.$

5 Conclusion

We showed how rewriting logic (RWL) and ρLog calculus can be related, defining a fragment of ρLog, into which rewriting logic can be encoded by a provability preserving mapping. The mapping, denoted by Φ, actually, relates entailment systems of these two formalisms. Given a theory of rewriting logic, consisting of an alphabet, equations, labels and rewrite rule, Φ maps

- each RWL constant to a constant in the language of ρLog,
- each RWL variable to a ground ρLog term, whose head indicates that it is an encoding of an RWL variable and the argument is a constant corresponding to the variable,
- each RWL basic sort to a ground ρLog term, whose head indicates that it is an encoding of an RWL sort and the argument is a constant corresponding to the sort,
- each RWL rule label into a ρLog constant,
- RWL sort definitions, subsort relation, rewrite rules, inference rules, inference control, and the matching mechanism into ρLog clauses.
- RWL sequents are mapped into ρLog atoms.

RWL equations are either considered to be represented in the implementation of equational matching of ρLog, or they are translated as rules if they induce a convergent rewrite system. Those rules then serve for normalization of terms before and after reduction. A mixed approach is also possible.

The range of Φ is the definite (negation-free) fragment of ρLog, but the rich strategy language of this formalism helps to imitate some kind of behavior which is

usually modeled with the help of negation-as-failure (or the cut) in logic programming. An example of such a strategy is **first_one**, which stops evaluation after one answer of the first applicable strategy is computed.

The important property of the mapping Φ is that it is conservative: provability of a rewriting logic sequent from a rewrite theory is equivalent to the provability of the Φ-image of the sequent from an Φ-image of the theory in ρLog. It shows the expressive power of ρLog: This formalism, like rewriting logic, can be used as a logical and semantic framework.

Acknowledgments

This research has been partially supported by the Brazilian National Council for Scientific and Technological Development CNPq under grant CsF/BJT 401319/2014-8, by Rustaveli National Science Foundation under the grants FR/508/4-120/14 and YS15 2.1.2 70, and by the Austrian Science Fund (FWF) under the project P 28789-N32.

References

[1] Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier BV., 2001.

[2] Walid Belkhir, Alain Giorgetti, and Michel Lenczner. A symbolic transformation language and its application to a multiscale method. *J. Symb. Comput.*, 65:49–78, 2014.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Jorge Coelho, Besik Dundua, Mário Florido, and Temur Kutsia. A rule-based approach to XML processing and Web reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR 2010*, volume 6333 of *LNCS*, pages 164–172. Springer, 2010.

[5] Besik Dundua. *Programming with Sequence and Context Variables: Foundations and Applications*. PhD thesis, Department of Computer Science, University of Porto, 2014.

[6] Besik Dundua, Temur Kutsia, and Mircea Marin. Strategies in PρLog. In Maribel Fernández, editor, *9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.

[7] Besik Dundua, Temur Kutsia, and Klaus Reisenberger-Hagmayer. An overview of PρLog. In Yuliya Lierler and Walid Taha, editors, *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, volume 10137 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2017.

[8] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[9] Temur Kutsia. Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols. RISC Report Series 02-09, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, May 2002. PhD Thesis.

[10] Temur Kutsia and Mircea Marin. Matching with regular constraints. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

[11] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[12] Mircea Marin and Temur Kutsia. On the implementation of a rule-based programming system and some of its applications. In Boris Konev and Renate Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03)*, pages 55–68, Almaty, Kazakhstan, 2003.

[13] Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

[14] Mircea Marin and Florina Piroi. Deduction and Presentation in ρLog. *ENTCS*, 93:161–182, 2004.

[15] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In Dov M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, 2002.

[16] Narciso Marti-Oliet, Jose Meseguer, and Alberto Verdejo. A Rewriting Semantics for Maude Strategies. *ENTCS*, 238(3):1–18, 2009.

[17] José Meseguer. General logics. *Studies in Logic and the Foundations of Mathematics*, 129:275–329, 1989.

[18] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[19] Jose Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012.

[20] Phong Nguyen. *Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows*. PhD thesis, Department of Computer Science, University of Geneva, 2015.